Dipl.-Ing. Wolfgang Raschke, BSc

A new Approach to Security Certification of Future Smart Card Systems

---

Dissertation

vorgelegt an der
Technischen Universität Graz

zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften
(Dr. techn.)

durchgeführt am Institut für Technische Informatik
Technische Universität Graz
Vorstand: Em. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Graz, im April 2015

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen / Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....
(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, the ....
(Signature)

Kurzfassung

Smart Cards sind kleine ressourenbeschränkte eingebettete Systeme mit harten Anforderungen an Informationssicherheit. Viele Anwendungsszenarien bedürfen unterschiedlicher Leistungsprofile, die sich an den unterschiedlichen Marktsegmenten orientieren, wie etwa: Banking-Anwendungen oder Anwendungen für mobile Kommunikationsgeräte. Jedes dieser Anwendungsszenarien hat unterschiedliche Anforderungen bezüglich Speicherbedarf, Energieeffizienz und Informationssicherheit. Die Smart Card Industrie wendet ihren Fokus mehr und mehr dem Kunden zu: Produkte werden auf die Kundenbedürfnisse maßgeschneidert. Damit gehen einige Herausforderungen einher. Für Standardprodukte war es bisher relativ einfach, Requirements, Design, Code und Tests konsistent zu halten. Diese genannten Entwicklungs-Artefakte sind Evidenz für eine Evaluierung bezüglich Informationssicherheit. Im Gegensatz zu den Standardprodukten haben individualisierte Produkte Erweiterungen, zusätzliche oder fehlende Features. Für solche Varianten von Standard-Produkten ist es oft notwendig, eine vollständige Zertifizierung der Informationssicherheit durchzuführen.

Diese Arbeit ist ein Schritt in Richtung effizienter Zertifizierungen von individualisierten Produkten bezüglich Informationssicherheit. Das Sekundärziel dieser Arbeit ist es, eine konsistente Code-Basis zu erreichen und zu managen. Dies ist eine wesentliche Vorbedingung für eine effiziente Zertifizierung. Die Code-Basis kann je nach Hardware-Plattform Unterschiede haben, da verschiedene Compiler eingesetzt werden. Außerdem kann der Befehlssatz unterschiedlich sein. Diese Arbeit liefert drei Beiträge zu diesem Sekundärziel: Richtlinien für plattformunabhängiges Codieren in Form sogenannter Patterns, ein Migrationsprozess und ein Defensive Virtual Machine (D-VM) Layer. Jedenfalls ist es unmöglich einen gemeinsamen Code für alle Produkte zu haben, wenn diese individualisiert werden. Daher ist ein Variantenmanagement notwendig, welches das Zusammenspiel zwischen gemeinsamen und unterschiedlichen Teilen unterstützt. Eine industrielle Fallstudie eines solchen Variantenmanagement-Systems wird beschrieben.

Das Primärziel dieser Arbeit ist es, die Wiederverwendung von Evidenz zu verbessern, die für eine Zertifizierung der Informationssicherheit benötigt wird. Eine Common Criteria Zertifizierung evaluiert immer eine spezifische Version und Konfiguration. Wenn sich die Konfiguration ändert, muss im schlechtesten Fall die gesamte Zertifizierung wiederholt werden. Jedenfalls existieren formelle und informelle Prozesse, um Evidenz wiederverwenden zu können. Diese Arbeit liefert eine Entscheidungshilfe, um den passendsten Prozess auswählen zu können. Das Hauptaugenmerk liegt auf einem Framework für eine inkrementelle Zertifizierung. Dieses ist wichtig, da viele Varianten von Smart Card Produkten mit geringen Unterschieden herausgegeben werden. Daher ist es sinnvoll, einen Zertifizierungsprozess zu wählen, der als Input eine Delta-Version des Inkrementes des Software-Systems verwendet. In dieser Arbeit wird ein derartiger inkrementeller Zertifizierungsprozess beschrieben. Außerdem werden die Konsequenzen diskutiert, sowie die spezifischen Anforderungen für die Unterstützung durch Software-Modelle erläutert. Schlussendlich werden relevante Veränderungen am Common Criteria Standard vorgeschlagen.

Abstract

Smart cards are small resource-constrained embedded systems that have to fulfill rigorous security requirements. Multiple application scenarios demand diverse product performance profiles which are targeted to markets such as banking applications and mobile applications. Each of these application scenarios has different requirements regarding memory footprint, runtime performance and power consumption and security. The smart card industry is currently shifting towards a customer-focused viewpoint: products are tailored to the customer's needs. This shift comes with several challenges. In a few standard products it is relatively easy to maintain consistency between requirements, design, implementation and tests. All these artifacts are evidence for a security evaluation. However, an individualized product sometimes has extensions, additional or missing features. For such variants of a standard product, it is necessary to perform a separate security certification.

This thesis provides a step towards more efficient security certifications for variants of a standard product. A minor goal for the reuse of security certification evidence is to achieve and manage a consistent code base. The code may vary from one hardware platform to another due to different compilers and a hardware-specific instruction set. This work provides three contributions to refactoring towards a common code for several hardware platforms: guidelines for platform independent coding in the form of patterns, a migration process and a Defensive Virtual Machine (D-VM) layer. However, it is impossible to have a common code for all products if they are individualized; and thus, a proper variant management has to support the interplay between common and variable parts. An industrial case of such a variant management is described.

The major goal is to improve the reuse of security certification evidence. A Common Criteria certification always evaluates a specific version and configuration of a software system. If the configuration alters, in the worst case, the whole certification has to be repeated. However, formal and informal processes for the reuse of certification evidence do exist. This work provides a decision framework for selecting an appropriate approach. The focus is on providing a framework for incremental certification. This is important because many variants of smart card products with few differences are issued. Thus, it makes sense to select a certification process which takes as input a delta version of the increment of a software system. In this work such incremental certification processes are described. Furthermore, the consequences are discussed and the specific requirements for a support by software models are stated. Finally, changes to the Common Criteria standard are proposed which would improve the flexibility of security certification processes.

Acknowledgements

This thesis is a research outcome of the DAVID project. This project has been conducted with the Institute for Technical Informatics and NXP Semiconductors Austria GmbH as research partners. Both institutions and their employers had a considerable impact on this thesis. However, unfortunately I am only able to mention a few of the contributors to this research. First of all, I want to thank Prof. Reinhold Weiß for supervising this thesis. Especially in the final phase his advice was a great help in composing the big picture of this thesis. Furthermore, I want to thank Christian Steger for supervising the project and advice for the scientific publishing process. Moreover, many thanks go to Christian Kreiner whose ability to ask the right question guided my research in a positive direction. The industry partner provided lots of hands-on experience and provided input for several experience reports and research ideas. I want to especially thank Johannes Loinig who provided scientific guidance in long discussions and Franz Krainer for support in the last months of this project. I really want to thank my collegues Massimiliano Zilli, Michael Lackner and Reinhard Berlach for long discussions of ideas, paper reviews and lots of motivation. My collegues Stefan Orehovec, Erik Gera-Fornwald and Andreas Sinnhofer deserve special thanks for being valuable co-authors. I would like to thank Jari Rauhamäki and Christian Beckers for shepherding publications. I want to thank Philip Baumgartner who contributed a lot during the work on his masters thesis. Moreover, this thesis would not have been possible without the support of pure::systems and Danilo Beuche. Finally, I would like to thank my family for their continuing support.

Graz, April 2015 Wolfgang Raschke

Extended Abstract

Smart cards are small resource-constrained embedded systems that have to fulfill rigorous security requirements. Multiple application scenarios demand diverse product performance profiles which are targeted to markets such as banking applications and mobile applications. Each of these application scenarios have different requirements regarding memory footprint, runtime performance and power consumption. The composition of such a diversified product is challenging since there is no explicit base of domain knowledge available. Currently, there is a trend in building more complex smart card products with better security mechanisms, more product features and also a larger memory footprint. Such complex smart cards are always increasing in memory footprint because they include all the features and artifacts which are part of a standard solution. So, it is relatively simple to maintain consistency among requirements, code, tests and documentation. The second trend is to decrease the complexity in order to have increase cost efficiency and only have appropriate security. In such smaller product configurations, some artifacts need to be opted out. The absence of certain artifacts may influence other parts of the software system. In order to fully assess the impact of opting artifacts or features out, it is imperative to trace all artifacts, such as security requirements, security code, countermeasures and the like. Manual tracing of such software systems means a high manual effort. Also defects are likely to be introduced due to the manual work required. Therefore, it is imperative to support the tracing and system configuration automatically with a method and appropriate tools.

As the smart card industry currently shifts from a (standard)-product-focused to a customer-focused view, it is common to apply iterative or so-called agile development processes. These processes take into account requirements changes from customers during the product development. In addition, new security requirements are introduced because attacks (such as the Heartbleed attack [1]) unveil vulnerabilities of secure systems. These agile methods have some drawbacks: First, the whole software system changes very quickly which makes the tracing of all software artifacts and their dependencies difficult. Therefore, a method for traceability and product configuration must address the evolution of the whole software system. Second, changing security requirements causes some problems for a fast and efficient security evaluation which is necessary for secure smart card systems. Summarizing, in this thesis, the challenges that are a consequence of building individualized smart cards with a smaller and appropriate feature set are addressed.

In Figure 1 the connections between the goals and the contributions are depicted. Basically it is shown that a consistent code base is a pre-condition for the reuse of certification evidence. It can be seen that some observations during the migration process have inspired a hypothesis for a cost model which can be applied to an evolving security certification. This cost model has revealed the need for several patterns of software modeling.

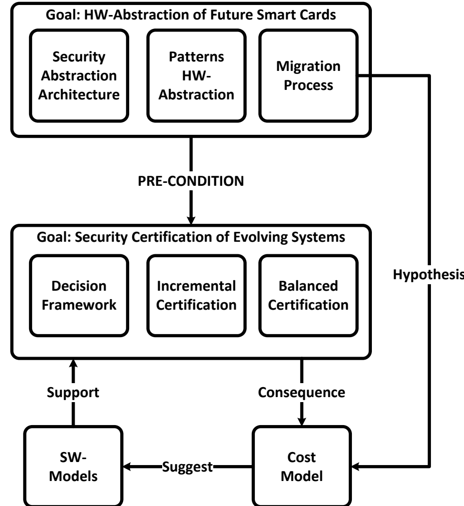


Figure 1: In order to achieve a hardware-abstracted code base, three strategies are applied: an architecture, guidelines or patterns for coding and a migration process. The abstracted code base increases the reusability of security certification evidence. The main goal is to support the security certifications of evolving systems. A decision framework helps to select an appropriate approach and incremental certification processes are detailed. Finally, the constraints and consequences of the proposed evaluation approaches are explained.

Minor Goal: Hardware Abstraction for Future Smart Card Architectures

A pre-condition for reuse of certification evidence is a hardware-abstracted code base (see Figure 1). For example, very small changes in the code base would shift code line numbers and make references to documentation useless. Therefore, it is important to unify the code base for several hardware platforms, as far as possible. A consistent hardware-abstracted code base is supported in this thesis by three contributions: an architectural solution to encapsulate security mechanisms, patterns for hardware-abstracted coding and a process for a migration to a hardware-abstracted code base. The architectural solution is to encapsulate security countermeasures in a dedicated Defensive-Virtual Machine (D-VM) layer[1]. This layer provides a unique interface to the upper virtual machine layer. Thus, exchanging security countermeasures in the D-VM layer does not impact the other parts of the system. Platform-specific key-words allow optimization for performance and memory footprint. It is important to also keep the possibility of optimization in the platform-abstracted code. Thus, four patterns are described which enable platform-independent coding without losing the possibility of optimization[2]. These patterns have been mined and applied in the industrial development of a secure smart card operating system. If this is not considered at the very beginning, the software is not written in a platform-independent manner. In this case, a systematic migration process to platform-independent code is necessary. This

[1] *A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards*, 7th Workshop on Information Security Theory and Practice (WISTP'13), Heraklion, Greece, 2013

[2] *Patterns for Hardware-Independent Development for Embedded Systems*, 20th European Conference on Pattern Languages of Programs (EuroPLoP'14), Irsee, Germany, 2014

thesis includes a report on the systematic approach of such a refactoring in the development of a secure smart card project[3]. It is described how test cases can be written platform-independently. The tests can then be used for a test-driven porting of the source code.
The architectural solution for a hardware-abstracted code base proposes a specific Defensive Virtual Machine (D-VM) layer. This layer provides a unique interface to the upper virtual machine layer. So the developers of the virtual machine do not have to worry about whether the security mechanisms are provided in hardware or in software. Thus, this layer helps to keep the code of the virtual machine consistent.

Major Goal: Certification of Evolving Security Systems

The major goal of this thesis is to improve the reuse of security certification evidence. Several patterns of security certification processes show that there are possibilities to apply a more modular certification under certain circumstances[4]. These patterns also help to select an appropriate certification process. Furthermore, this thesis focuses on an incremental security certification: generally, a security model can generate documentation for a Common Criteria security evaluation. It is shown that for agile software projects a model has to support model-evolution[5]. A means of supporting such a model evolution is the Change Detection Analysis (CDA). From this CDA two possible security evaluation processes can be derived. The two processes are compared. Furthermore, some indicators for selecting the appropriate approach are provided. Both processes are iterative but basically the frequency of iterations is different (weeks vs. years).
An incremental certification has the following consequences: first, there is a need for support from software modeling approaches. Second, a cost model shows how the certification processes affect the cost and third, future changes to the Common Criteria standard could improve the flexibility of certification. The software modeling approaches are described in the form of patterns in more detail[6]. A pattern is a mapping of a solution to a problem in a specific context. It is shown that a specific way of modeling software is typical for a specific context. Moreover, it is described how the patterns of software modeling change with the size and complexity of the software system. These patterns help to systematically select an appropriate method dependent on the industrial context.
The cost can be estimated with the help of a metric for software evolution[7]. Such a metric can be used as an indicator for selecting a software modeling pattern. For example, a high rate of software change would suggest the application of the RTE (Round-Trip Engineering) pattern for the modeling of the software. Moreover, this metric can help to select an appropriate security evaluation process.

[3] *Test-Driven Migration Towards a Hardware-Abstracted Platform*, 5th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS'15), Angers, France, 2015

[4] *Evaluation paradigm selection according to Common Criteria for an incremental product development*, International Workshop on MILS: Architecture and Assurance for Secure Systems (MILS'15), Amsterdam, Netherlands, 2015

[5] *Supporting evolving security models for an agile security evaluation*, In Evolving Security and Privacy Requirements Engineering (ESPRE'14), Karlskrona, Sweden, 2014

[6] *Patterns of Software Modeling*, In Fifth International Workshop on Information Systems in Distributed Environment (ISDE'14), Amantea, Italy, 2014

[7] *Where does all this waste come from?*, In 21st EuroSPI Conference (EuroSPI'14), Luxembourg, 2014

Future changes to the Common Criteria standard are proposed[8]. More specifically, this thesis proposes to view a security evaluation as space with the following two dimensions: functional assurance and process assurance. The proposal is to make the evaluation more flexible by balancing functional and process security assurance. Several examples make this approach more clear.

Contributions of this Thesis

Summarizing, this thesis provides contributions to the following fields:

1. **Hardware Abstraction for Future Smart Card Architectures:** A catalog of patterns to cope with cross-cutting hardware dependent code fragments is presented. These patterns discuss the context, the benefits, the drawbacks, and the consequences of specific solutions. The solutions allow the masking compiler specific code fragments and the ability to resolve them before compilation. In addition to these patterns, a test-driven process to migrate to a hardware independent code-base is described. In this process, after each code change (due to abstraction) all tests are executed. Thus, defects introduced by the porting activities can be detected, early. A defensive virtual machine (D-VM) layer abstracts possible hardware or software implementations of security checks from the virtual machine and provides a unique interface to it. So, security checks can be optimized for runtime performance or memory footprint without refactoring the virtual machine.

2. **Certification of Evolving Security Systems:** A decision framework helps to select an appropriate security certification process. This is difficult, because many of these processes are never explicitly stated in any standard. In order to improve the reuse of security certification evidence, an iterative security certification process helps to reuse evidence for small changes, such as a new configuration. Two iterative processes are discussed and their differences, benefits and drawbacks are compared. In order to better support a better iterative certification, changes to the Common Criteria standard are proposed. The main proposal is to balance process and product assurance to increase the flexibility of the certification. Many certification processes demand a specific modeling technique. These patterns of software modeling are discussed and a cost model helps to assess the consequences of a security certification process.

[8] *Balancing Product and Process Assurance for Evolving Security Systems*, In International Journal of Secure Software Engineering (IJSSE), vol. 6, no. 1, pp. 47-75, 2015

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
  - 1.1.1 Design Complexity Trend . . . 1
- 1.2 Design Flow for Low-End Java Card Operating Systems . . . 1
  - 1.2.1 The DAVID Project . . . 2
  - 1.2.2 Problem Statement . . . 3
  - 1.2.3 Contributions and Significance . . . 4
  - 1.2.4 Structure of the Work . . . 4

2 Related Work **6**
- 2.1 Security Modeling Notation Overview . . . 6
- 2.2 Security Modeling with Explicit Support of Evolving Systems . . . 7
- 2.3 Assurance Paradigms . . . 8
- 2.4 Hardware Abstraction . . . 10
- 2.5 Summary and Difference to State-of-the-art . . . 11

3 Method **13**
- 3.1 Overview . . . 13
- 3.2 Major Goal: Certification of Evolving Security Systems . . . 15
  - 3.2.1 Certification Processes . . . 15
  - 3.2.2 Cost Model . . . 22
  - 3.2.3 Patterns of Software Modeling . . . 23
- 3.3 Minor Goal: Hardware Abstraction for Future Smart Card Architectures . . . 24
  - 3.3.1 Patterns for Hardware Abstraction . . . 24
  - 3.3.2 Test-driven Porting . . . 25
  - 3.3.3 D-VM Layer . . . 26
  - 3.3.4 Product Line Migration . . . 27

4 Results and Evaluation **28**
- 4.1 Hardware Abstraction of the Source Code . . . 28
  - 4.1.1 Pilot Study . . . 28
  - 4.1.2 Case Study . . . 28
- 4.2 Scenario Analysis of the Security Processes . . . 29
  - 4.2.1 Case study: CeSeCore Architecture Overview . . . 29
  - 4.2.2 Case 1: Complete Evaluation . . . 30
  - 4.2.3 Case 2: Re-evaluation with Traceability Impact Analysis . . . 31
  - 4.2.4 Case 3: Re-evaluation with Experiential Impact Analysis . . . 31
  - 4.2.5 Relevance of Certification Processes for the Smart Card Industry . . . 31

5 Conclusion and Future Work **35**
5.1 Conclusion 35
5.2 Directions for Future Work 36
5.2.1 Security Certification 36
5.2.2 Security Requirements 36

6 Publications **38**
6.1 A Defensive Virtual Machine Layeer to Counteract Fault Attacks on Java Cards . 40
6.2 Patterns for Hardware-Independent Development for Embedded Systems 56
6.3 Test-Driven Migration Towards a Hardware-Abstracted Platform 64
6.4 Embedding research in the industrial field: a case of a transition to a software product line 71
6.5 Evaluation paradigm selection according to Common Criteria for an incremental product development 77
6.6 Supporting evolving security models for an agile security evaluation 82
6.7 Patterns of Software Modeling 88
6.8 Where does all this waste come from? 98
6.9 Balancing Product and Process Assurance for Evolving Security Systems 107

References **131**

List of Figures

1 Overview of the major goal, the minor goal and the corresponding publications. . . v

1.1 The growth of smart card security and the demand for variant management. 2
1.2 Trends in the smart card industry. 3

2.1 The incremental UMLsec approach. 8
2.2 Composition of assurance arguments. 9
2.3 Concept of a HAL. 10

3.1 Overview of the major goal, the minor goal and the corresponding publications. . . 14
3.2 Basic parts of the security model. 15
3.3 Decision framework for Common Criteria security certification processes. 16
3.4 Re-evaluation with Traceability Impact Analysis. 18
3.5 Re-evaluation with Experiential Impact Analysis. 18
3.6 Hard goal model of Common Criteria assurance components. 20
3.7 Soft goal model of Common Criteria assurance components. 21
3.8 Example of a model-based process assurance. 21
3.9 Defensive Virtual Machine layer. 25
3.10 Test-driven migration to a hardware-independent platform. 26
3.11 D-VM layer support for different levels of security. 26

4.1 Phases in a test-driven migration. 29
4.2 Number of certified product families. 33
4.3 Number of product versions vs. size of product family. 34

6.1 Overview of the major goal, the minor goal and the corresponding publications. . . 39

List of Tables

2.1 Related security modeling notations and their support for several desired objectives. 7
2.2 Assurance arguments for several certification standards from different domains. . . 9

3.1 Comparison of the processes with Traceability Impact Analysis and Experiential Impact Analysis. 19
3.2 Mapping of Common Criteria assurance classes to process and product assurance. 19
3.3 Terminology of the cost model. 22
3.4 Mapping of the certification processes to modeling patterns and their relevance. . . 23

4.1 Amount of modules which passed the first three steps of the test-driven migration process. 29
4.2 Direct impacts between modules of the *CeSeCore* 30
4.3 Direct and indirect impacts between modules of the *CeSeCore* 32

List of Abbreviations

AIS	Actual Impact Set
ALM	Application Lifecycle Management
CAP	Composed Assurance Package
CC	Common Criteria
CDA	Change Detection Analysis
CIS	Candidate Impact Set
DIS	Discovered Impact Set
D-VM	Defensive Virtual Machine
EAL	Evaluation Assurance Level
EIA	Experiential Impact Analysis
FPIS	False Positive Impact Set
HAL	Hardware Abstraction Layer
HW	Hardware
PC	Personal Computer
RTE	Round-Trip Engineering
SFR	Security Functional Requirement
SIS	Starting Impact Set
SYSML	Systems Modeling Language
TDD	Test-Driven Development
TIA	Traceability Impact Analysis
UML	Unified Modeling Language
VM	Virtual Machine

Glossary

Assurance Class
An assurance class is the top-level entity of the Common Criteria assurance argument.

Assurance Family
An assurance family is a refinement of an assurance class. Several assurance families are part of a single assurance class.

Assurance Component
An assurance component is part of an assurance family with a certain component leveling. Depending on the Evaluation Assurance Level (EAL) assurance components of a certain level must be fulfilled.

Certification Evidence
The certification evidence is produced by the Common Criteria evaluation facility. It is the result of the evaluation.

Developer Evidence
Developer evidence is documentation provided by the developers for the evaluation facility.

Evaluation Assurance Level
The Evaluation Assurance Level states the confidence that the security claim is fulfilled.

Security Functional Requirement
A Security Functional Requirement states a specific security function or security service of the device.

Static Certification
A static certification takes as input developer evidence which does not change anymore because the product development is already finished. Therefore, there is one iteration in a successful certification.

Incremental Certification
In an incremental certification there are several rounds of iteration. In each iteration the evaluation facility gets a delta version of the developer evidence.

Balanced Certification
A balanced certification takes into account process and product assurance. Both assurance arguments can be traded off to a certain degree.

Chapter 1

Introduction

1.1 Motivation

1.1.1 Design Complexity Trend

Smart cards are secure and small embedded devices which hold a secret. Such a secret allows the identification of whether a service may be used, such as a mobile phone service, a pay-tv service or a banking service. It can be seen that smart cards enable the billing of distributed services and thus it is obvious that they are sold in high numbers. An indicator for the use of smart cards is the number of mobile subscriptions. In 2014 there are about 7000 million subscriptions [2]. Since this number considers only one application scenario, it can be expected that the total number of smart cards exceeds this number: this also means, that each person uses several smart cards. In order to provide the most secure smart card products, the industry strives to implement as many security features as possible. In Figure 1.1 the green line indicates the increasing product security of a smart card standard product. However, security features consume a portion of memory and thus the price increases with the memory footprint and the level of security. In contrast, the red line indicates, that the customer's demand for security does not increase as fast as the security of standard products. This has the following implications: First, as long as the red line is below the green line, increasing the security of standard products is a good strategy for meeting the customer's demands. However, when the red line crosses the green line, standard products provide more security features than the customer requires. At this point in time the industry shifts from standard products to individualized products (see also [3]) because customers do not want to pay for obsolete security features. Such a tailoring of products can be accomplished with variant management. However, the individualization also means, that more products which resemble each other have to be certified for security. Since such certifications take a considerable amount of time and cost, they are a roadblock for the individualization of products.

1.2 Design Flow for Low-End Java Card Operating Systems

Since this thesis was written in the course of a research project, this chapter provides an overview of it. Furthermore, the problem statement and list of all the contributions which have been published are presented.

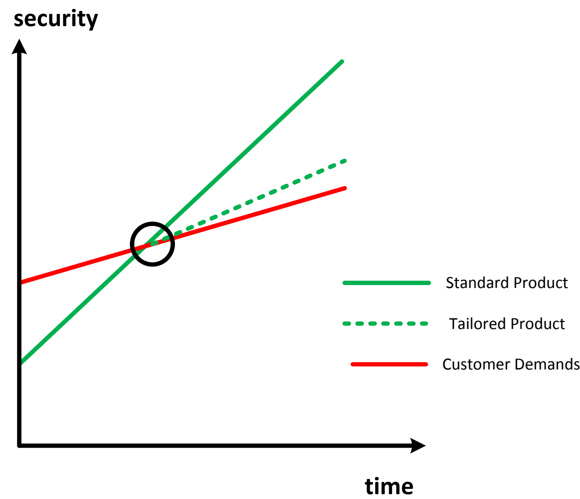


Figure 1.1: The security of standard smart card systems grows faster than the customer's demands. Therefore after a certain point in time, a tailoring of products with variant management becomes more and more important.

1.2.1 The DAVID Project

The DAVID project is a collaboration between the academic Institute for Technical Informatics and NXP Semiconductors Austria as industrial partner. This collaboration strives to set up a design flow for low-end smart cards. Figure 1.2 shows the two current trends in smart card operating systems. The DAVID project focuses on the trend of building smaller Java Card [4] systems with appropriate security mechanisms and runtime performance. Such smaller systems with a better price are attractive to several emerging market segments.

The DAVID project has two main approaches to minimizing the memory footprint of a Java Card operating system. First, compression methods (see [6][7][8]) strive to minimize the memory consumption. The so-called byte code compression compresses the code itself whereas the heap compression minimizes the memory consumption of Java Card objects and classes. For example, a code compression could summarize a sequence of frequently occurring bytecodes to a single new bytecode. A heap compression instantiates member variables of an object on demand and thus avoids memory consumption of never-used members. Second, leaving out unnecessary code and features is an approach to reduce the memory footprint of a Java Card operating system. However, leaving out some code pieces is not as simple, as it seems at a first glance. Each part of code is evidence of a severe security certification and is linked to higher-abstracted security objectives and requirements. Thus, the impact of omitting certain pieces of code or - artifacts - has to be known. Such knowledge is, at the moment, implicit knowledge of a few domain experts. However, in order to issue lots of individualized products, an explicit modeling of the knowledge is necessary.

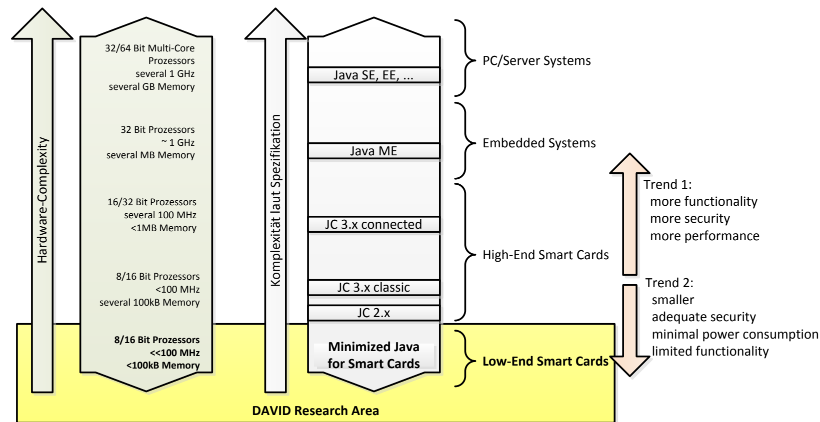


Figure 1.2: There are two trends for smart cards: first, there is a trend to increase security and functionality. The other trend is to decrease the functionality and security in order to minimizing the cost [5].

1.2.2 Problem Statement

Java Cards are resource-constrained devices with high security demands in very volatile markets. They have different areas of application, such as banking, mobile phones, pay tv and the like. There are always new application areas emerging and thus new market segments. In order to enter a new market segment, it is important to have a competitive price and the appropriate feature set. Sometimes, it is necessary to downsize the product and thus, the following challenges exist:

- New smaller hardware platforms.
- Products have to be tailored to the customer's needs.
- The tailored products have to be certified for security, as quickly as possible.

New smaller hardware platforms enable a competitive price for application areas with lower security and runtime performance demands. As new hardware platforms are introduced, the software has to be adapted to these new processors. If considered from the very beginning, a systematic platform-abstraction can save a lot of manual porting effort and ease the adaption for each new processor. Because of the shift towards a customer-focused market, standard products no longer sufficiently satisfy the customer's demands. A Common Criteria security certification is an integral part of the product development and release. Since there is now a focus on individualized products, the number of security certifications also skyrockets. Most of the certified products only differ in some parts. However, the Common Criteria standard does not provide a good support for an incremental certification. Another problem of the security certification is that it takes several months to certify a product after it is finished. Such a time lag is a huge drawback because time-to-market is still a very important success factor in the smart card industry. This

time lag could be reduced, if the certification is carried out in parallel with the product certification. Thus, an incremental certification also has to support evolving secure systems.

1.2.3 Contributions and Significance

Summarizing, this thesis provides contributions to the following fields:

1. **Hardware Abstraction for Future Smart Card Architectures:** A catalog of patterns to cope with cross-cutting hardware dependent code fragments is presented. These patterns discuss the context, the benefits, the drawbacks, and the consequences of specific solutions. The solutions allow the masking of compiler specific code fragments and their resolution before compilation. In addition to these patterns, a test-driven process to migrate to a hardware independent code-base is described. In this process, after each code change all tests are executed. Thus, defects introduced by the porting activities can be detected, early. A Defensive Virtual Machine (D-VM) layer abstracts possible hardware or software implementations of security checks from the virtual machine and provides a unique interface to it. So, security checks can be optimized for runtime performance or memory footprint without refactoring of the virtual machine.

2. **Certification of Evolving Security Systems:** A decision framework helps to select an appropriate security certification process. This is difficult, because many of these processes are never explicitly stated in any standard. An iterative security certification process helps to reuse evidence for small changes, such as a new configuration. Two iterative processes are discussed and their differences, benefits and drawbacks are compared. In order to better support an iterative certification, changes to the Common Criteria standard are proposed. The main proposal is to balance process and product assurance to increase the flexibility of the certification. Many certification processes demand for a specific modeling technique. These modeling techniques are discussed in form of patterns. Furthermore, a cost model helps to assess the consequences of a security certification process.

1.2.4 Structure of the Work

In chapter 2 the related work regarding the major and the minor goal is provided. First, an overview of the most important security modeling notations is given. A deeper look into a modeling notation with explicit support for model evolution is provided. Then, an overview of security assurance paradigms, such as product and process based assurance is presented. Several important security and safety standards are mapped to one or both of those paradigms. Thereafter, the related work of hardware abstraction methods is provided. Finally, this chapter compares the state-of-the-art with the findings of this thesis. In chapter 3, a method for an incremental security certification is provided. First, it makes sense to unify the source code as much as possible because this decreases the need for a re-certification for each hardware platform. In order to do this, patterns for hardware independent coding and a layered approach for encapsulating security mechanisms are presented. A test-driven migration process supports the transition towards such

a unified code base. Second, several processes for security certification are provided. A cost model helps the understanding of the consequences of each of these processes. It is shown which patterns of software modeling are necessary for specific security certification processes. Finally, an outlook for future enhancements of the Common Criteria standard is provided. Chapter 4 presents the results of a case study which has been conducted in collaboration with the academic institute together with the industrial research partner. In this case study parts of the source code have been refactored to be platform-independent. Respective results are listed. In addition, an analysis of possible scenarios of security certification processes and the potential reuse of secure software components is provided. Past certifications of the smart card industry are categorized to show how relevant the application of the proposed methods is. Chapter 5 concludes the work and gives an outlook for future work. The focus of future work is security certification, which can be improved, as well as methods and tools for it. In addition, an outlook for future work regarding security requirements engineering is given. In chapter 6 the papers which have been published in the course of this thesis are presented.

Chapter 2

Related Work

In this chapter an overview of established security modeling notations is provided. One notation is described in more detail because it supports most of the demanded requirements. Additionally, related work on hardware abstraction is discussed. Finally, difference to the state-of-the art is provided.

2.1 Security Modeling Notation Overview

An overview of security modeling notations is provided in Table 2.1. For each notation, information regarding several categories is provided. The approach states whether the notation is an encompassing modeling framework or which specific artifacts they model. The category *evolution support* states whether the framework provides explicit mechanisms for model evolution. Further, Table 2.1 provides information as to whether the model describes requirements or architecture. The category *process assurance* states whether the notation explicitly supports process assurance, such as code reviews and the like. A notation is iterative, if requirements drive architecture and the other way round. All these categories are beneficial in order to support the certification of evolving secure systems. As can be seen, the approach UMLsec supports most of the required attributes. Hence, this approach is described in the next section in more detail. However, in this section, a short overview of all the listed modeling notations is given.

Attack Tree An attack tree is a structured way to model attacks, attack goals, attacker motivation and attacker talent. An attack tree can be used for a high-level risk analysis and as a threat database [9][10][11].

Abuse Case An abuse case is a use case where the actor causes harm to the system by exploiting security vulnerabilities. Abuse cases are a simple and easy way to understand security analysis [12][13][14][15].

Secure Tropos Secure Tropos is a security system analysis framework which takes into account assets, actors and a (high-level) architecture. Goal models are used to refine the high-level goals to lower-level goals and architecture [16][17][18].

UMLsec UMLsec is an extension to UML that enables a model-based security engineering. UMLsec supports the evolution of models and checks, if the evolution keeps to

certain guidelines [19][20][21].

UML4pf UML4pf is an extension to UML with problem frames. A problem frame is a kind of pattern which describes problem class with its requirements, its context and its interfaces [22][23][24].

Security Requirements Framework (Haley) This framework consists of a structure which incorporates security goals, security functional requirements and system architecture. In addition to the structure, it describes an iterative process [25][26].

KAOS KAOS is a goal-oriented requirements engineering method. It describes a process which incorporates requirements, related objectives and a software system model. The process requires a review of the model and is thus an iterative process [27].

Table 2.1: Related security modeling notations and their support for several desired objectives.

Security Modeling Notation	Approach	Evolution Support	Software Requirements	Software Architecture	Process Assurance	Iterative
Attack Tree	Fault Model	N	Y	N	N	N
Abuse Case	Fault Model	N	N	Y	N	N
Secure Tropos	Framework	N	Y	Y	N	Y
UMLsec	Framework	Y	Y	Y	N	Y
UML4pf	Framework	N	Y	Y	N	Y
Security Requirements Framework	Framework	N	Y	Y	N	Y
KAOS	Goal Model	N	Y	N	N	N

2.2 Security Modeling with Explicit Support of Evolving Systems

The UMLsec tool[1] can check whether a UMLsec model conforms to specific security properties. Such a check is done on a static model. This means that the full model always

[1] https://www-secse.cs.tu-dortmund.de/jj/umlsectool/index.html

has to be checked against security properties, if it changes. Because such a static check is very resource consuming, it makes sense to only verify a delta version of all applied changes [28].

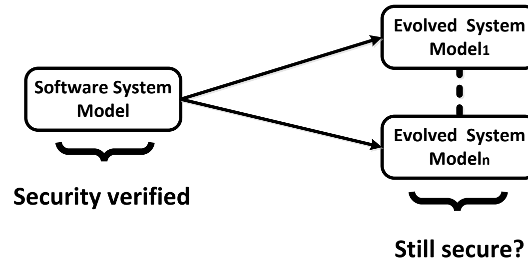


Figure 2.1: The incremental UMLsec approach checks if the delta of a security model is the outcome of a sequence of correct atomic changes [28].

However, a delta cannot be verified automatically, *per se*. It consists of a set of so-called *atomic* changes. An atomic change is a change that cannot be further divided into smaller changes. For such atomic changes it is possible to define rules and thus, they can be checked for correctness. In order to check a delta, it is necessary to check whether an evolution path of atomic changes can be constructed that fulfills the evolution rules. Since there can be several different evolution paths, the algorithm has to search for at least one possible evolution. If there is no possible correct evolution, the delta does not preserve the corresponding security property in this evolution step.

2.3 Assurance Paradigms

In order to allow a certification, each certification scheme requires arguments for assurance. As can be seen in Figure 2.2, the assurance argument can be composed of several sub-arguments (see also [29]). The certification standard determines the use of arguments for assurance. In product-based assurance the product, its related artifacts and documentation have to deliver arguments for assurance. Usually, the product has to fulfill functional requirements. The process-based assurance is established with sound development processes. If the right activities, such as requirements engineering, design, review and testing are done appropriately, this can serve as an argument for assurance. As can be seen in Figure 2.2, the competency of the resources is also part of the process argument, if further subdivided. In Table 2.2 several certification schemes in different domains are listed. Even if a certification is not relevant to security, the structure of the assurance argument can be similar.

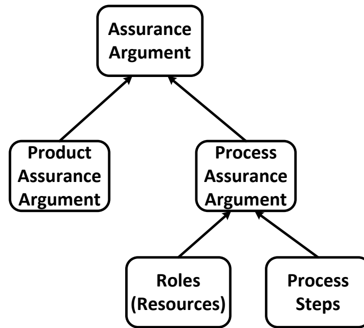


Figure 2.2: The assurance argument can be constructed from a product assurance and a process assurance argument. The process assurance argument incorporates roles and process steps.

Therefore, some standards are listed because it is possible to learn from them. In security, safety and process improvement, there exist standards which rely on a product argument, a process argument, or both. Moreover, certification standards for competency of resources are emerging.

Table 2.2: Assurance arguments for several certification standards from different domains.

Certification	**Purpose**	**Process**	**Product**	**Resource**
Common Criteria [30]	Security Certification	Y	Y	N
ISO 27001 [31]	Security Certification	Y	N	N
ISO 26262 [32]	Safety for Road Vehicles	Y	Y	N
IEC 61508 [33]	Safety for Electrical Systems	Y	Y	N
DO-178-B [34]	Safety of Avionic Software	Y	N	N
SPICE [35]	Software Process Improvement	Y	N	N
CMMI [36]	Software Process Improvement	Y	N	N
SACM [37]	Security focus and generic	N	N	Y
ECQA [38]	Safety focus and generic	N	N	Y

In the safety domain, a construction of an assurance argument from a product and a process argument is called a *safety case* [39][29]. The construction of arguments is based on Toulmin's work on arguments [40]. A product argument lacks confidence, if there is no process argument. For example a review of the product arguments increases confidence in it. However, in this example, it is not stated who (which role) conducts the review. This

provides a new perspective on a process-based assurance argument where the competency is a necessary part of this argument [41]. Since a sound process assurance argument can only be formed with appropriate competencies, it is necessary to also certify the competencies. As stated above, such certifications are an emerging topic in safety and security.

2.4 Hardware Abstraction

Hardware abstraction can be achieved via a layered system (see Figure 2.3). If there is no HAL (a) and (b), then the interface between processor and application depends on the hardware. Thus, the application software must be ported, if it is intended to be executed on a new processor. Since porting software is a costly task, it is better to implement a Hardware Abstraction Layer (HAL). The HAL has a unique interface to the upper application layer. Thus the application does not need to be ported.

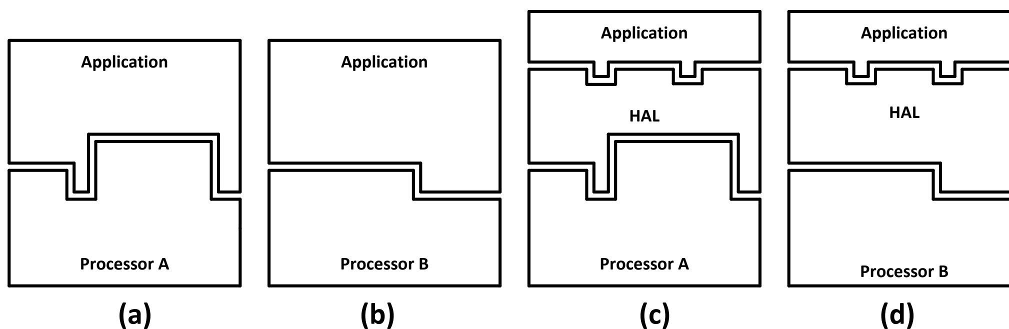


Figure 2.3: The interface of the application needs to be reworked for each hardware platform (see a and b). If an intermediate HAL exists (see c and d), the application can be reused on different hardware platforms without refactoring (see also [42]).

There are several refinements to the concept of a HAL. Handziski et al. [43] proposes a three layered HAL approach. The lowest layer is stateless and is only responsible for directly interfacing the hardware, such as initialization and register and port access. The middle layer is the core of the three layer architecture and may have states. It is responsible for complex resource management which may still be specific to the hardware in order to best exploit their capabilities, such as communication and memory. The highest layer provides an interface which is hardware independent.
Grenning [44] describes the challenges of hardware abstraction that come into being when Test-driven Development is applied to embedded systems. Basically, TDD is a design paradigm where first tests are written that reflect the specification of the unit under test [45]. Only after the relevant tests have been written, the corresponding functionality is implemented. All unit tests are repeated after each change of the unit. In this way, defects can be detected early. Test-driven development is an established method but it is rarely used in embedded systems for the following reasons: the build process may take several hours, the hardware is not (yet) available and the target hardware is expensive. Grenning introduces the embedded TDD cycle, where software is first written for a host PC compiler in order to test the functionality. If all tests pass, the next phase is to

switch to an embedded compiler. If the compilation works, the tests can be executed on an evaluation board and finally on the real target. In order to perform such cycles, the tests have to be written in a way that abstracts the hardware. For example the access to hardware units can be hidden behind so-called mock objects, or stubs.

2.5 Summary and Difference to State-of-the-art

This thesis improves the state-of-the art regarding the major and the minor goal:

- A cost model for software evolution of high-quality software is provided. It is shown that software evolution causes rework of quality activities, if they are accomplished in parallel. There exist models to quantify software evolution. However, there is no model which links the evolution to any cost. The hypothesis of our cost model has been empirically evaluated and thus provides evidence for its usefulness. This cost model helps to decide whether time-to-market or development and certification cost is more important.
- In contrast to the related work this thesis does not propose a security meta-model (a structure of security artifacts). A model-based framework can calculate a delta of two versions of a model. The contribution is the description of two possible processes for a Common Criteria certification which take this delta as input. The advantage of this approach is that the whole product does not have to be re-certified, which is very expensive. It is discussed under which circumstances the two processes are applicable.
- The described evaluation processes which are based on a delta are only applicable under certain constraints. As a contribution it is shown how the Common Criteria standard can be changed in order to better support reuse and an iterative certification. This thesis proposes to enable a balancing of product assurance and process assurance. Process assurance can be easily built in the model-evolution framework. For this purpose, for example, the framework has to log reviews of delta versions of the model.
- Since a model-based approach is proposed to support the security certification process, several patterns of software modeling are described. Such patterns allow the selection of an appropriate modeling approach. These approaches are categorized based on the information flow between different model entities. As opposed to the state-of-the-art, the use of a specific modeling framework (e.g. UML, SYSML) for a specific problem is not proposed. Rather, guidelines for selecting an appropriate approach are provided.
- There are several processes for Common Criteria certifications. Some of theses processes are formal: that means that they are explicitly mentioned in the Common Criteria standard. Other standards are informal: they are not mentioned in the standard and can be applied, as long as they do not contradict the standard. Four of these certification processes are discussed along with under which constraints they are useful and can be applied. Since these certification processes are hard to understand, a selection scheme is provided.

- Several patterns of hardware abstraction are provided. In the state of the art, hardware abstraction is achieved with a hardware abstraction layer. However, some cross-cutting issues cannot be encapsulated in a single layer. Such cross-cutting issues are type definitions, pointer definitions, structure alignments and byte endianness. Best practices regarding these issues are collected. As a contribution, not only technical solutions are presented. Moreover, an in depth discussion on the context in which a problem occurs is given. Within such a context, several solutions are possible. The alternative solutions can be traded-off via the so-called *forces* which are described in the patterns.
- The state-of-the art discusses Test-driven Development (TDD) in embedded systems. It is shown, how TDD can be applied to take existing code and refactor it to be platform independent.

Furthermore, in this thesis the following improvements are provided:

- It is shown how a software process improvement initiative between university and industry can be conducted via action research. Action research is a research method which strives to find *relevant* solutions from the field [46]. Experiences for an industry and academic collaboration are described and lessons learned which could help in a similar project are listed.

Chapter 3

Method

This chapter provides an overview of the publications regarding the major and minor goal. A cost model for assurance activities is a decision factor for selecting an appropriate software modeling pattern. The selection of the modeling pattern decides whether the software is certified after the product development or whether there is a certification in parallel. Further, several processes for security certification are described, as well as the context in which they can be conducted and a selection scheme for selecting an appropriate approach. Regarding the minor goal, several patterns for a hardware abstraction and a test-driven porting process are described. Moreover, experiences and lessons learned from a transition to an industrial variant management are provided.

3.1 Overview

Common Criteria [30] security evaluations for smart cards are expensive. There are several drawbacks to standardized security certification processes:

- Bad support for reuse of certification results.
- Bad support for evolution of certification evidence and thus bad support for an iterative security evaluation.
- Bad support for evolving security requirements.
- A long lag-time for certification after product is finished.

This thesis shows how these issues can be tackled. In publication 5 an overview of four different certification processes is given. It is described under which circumstances they can be applied. A selection scheme helps companies to select an appropriate approach. Therefore, several certification processes are described. It is shown in which context they can be applied. In publication 6 two processes for an incremental security certification are described which take as input an increment of a software system. However, the Common Criteria certification could be improved drastically in order to provide more flexibility for an evaluation. Thus, improvements of the standard are proposed in publication 7. Moreover, in publication 7 it is shown, how a specific software modeling pattern supports the extensions which have been proposed for the Common Criteria standard. Different

patterns of software modeling, which are explained in publication 8 support different security certification processes. A cost model which is described in publication 9, helps to trade-off a higher certification cost in a parallel certification versus a longer lag-time in a certification after the product is finished.

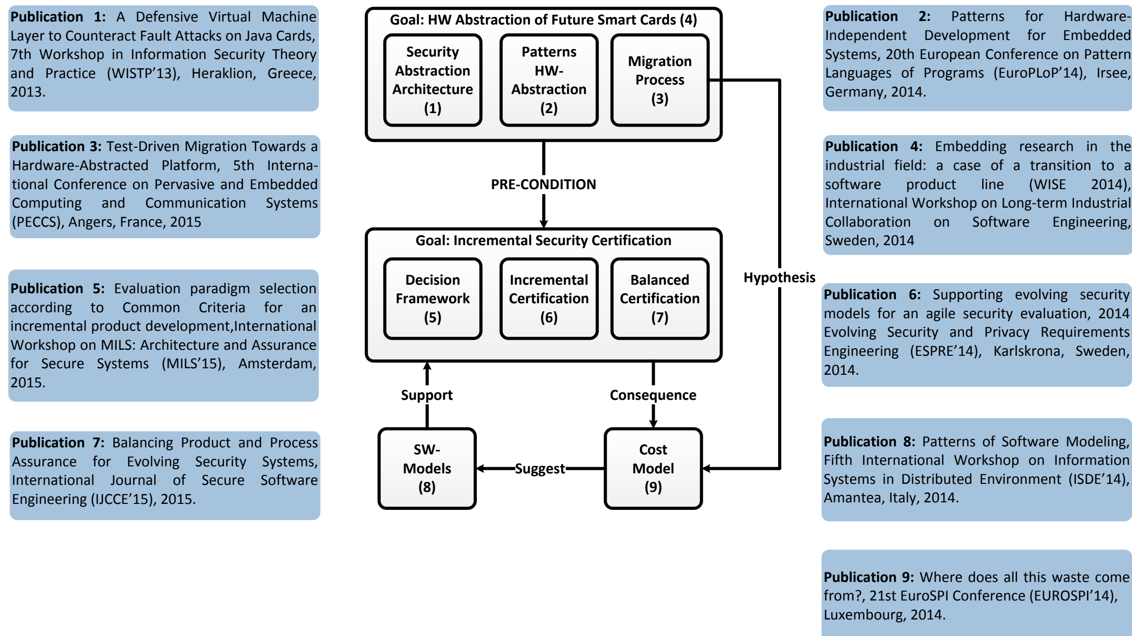


Figure 3.1: In order to achieve a hardware-abstracted code base, three strategies are applied: an architecture, guidelines or patterns for coding and a migration process. The abstracted code base increases the reusability of security certification evidence. The main goal is to support the security certifications of evolving systems. A decision framework helps to select an appropriate approach and incremental certification processes are detailed. Finally, the constraints and consequences of the proposed evaluation approaches are explained.

The diversification of smart cards leads to an increasing number of different products. However, products share commonalities and have differences. The productivity can be increased with a so-called product line engineering which strives to systematically reuse software artifacts. A product line approach for smart cards faces the following main challenges:

- Hardware-dependent code due to hardware-near programming.
- Portability issues due to the hardware dependent code.
- Hardware dependent tests. Also tests need to be ported.
- Many variants due to variations in hardware and optimization objectives.
- Variants in the security requirements of the software and complexity of the dependencies.
- Problems with reuse of security certifications.

A hardware-independent code can be more easily reused for a security certification than a hardware-dependent code. For this reason, in publication 1 an architectural solution for the hardware abstraction of security countermeasures is provided. Patterns for hardware independent coding are provided in publication 2. Moreover, porting activities for new hardware platforms are expensive. In publication 3, a systematic test-driven porting process shows how the transition to a hardware-independent platform can be kept relatively cheap. A process for a transition to a variant management is provided in publication 4. An overview of all publications is provided in Figure 3.1. The publications will be described in more detail in the following section.

3.2 Major Goal: Certification of Evolving Security Systems

Smart cards are certified for the security standard Common Criteria for a high level of assurance. For such a high Evaluation Assurance Level (EAL), it is necessary to provide evidence for assurance in the form of documents. The documents must contain the elements depicted in Figure 3.2. The security problem definition states the threats and the context of the secure device. Derived from the security problem definition, the security objectives are high-level security goals. These goals are refined by the Security Functional Requirements (SFR) which describe specific security functions. The functional specification shows how the SFR are fulfilled by the device. The design represents the security architecture and the implementation representation is the source code of the secure device.

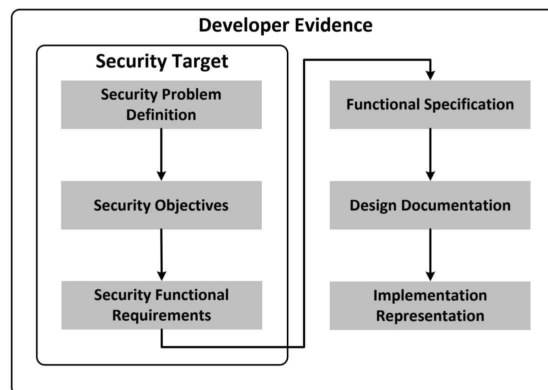


Figure 3.2: Basic parts of the security model. The arrows indicate a fulfillment operation. For example, the functional specification must fulfill the security functional requirements [47].

3.2.1 Certification Processes

This thesis describes several Common Criteria certification processes [48]:

Delta Evaluation The delta evaluation is a Common Criteria evaluation process. The evaluation facility has to make a delta analysis for each new version. For the delta, the evaluation facility has to make a change impact analysis, as well. For this reason, the evaluation facility needs all developer evidence, as well as the result of the previous certification.

Informal Modular In the case of a changed module or a new module, the evaluation facility can make an informal modular certification. An informal modular certification is a form of the delta evaluation. For this purpose, the evaluation facility needs all developer and certification evidence.

Formal Modular A formal modular certification is applied, if modules from different companies are certified and the evaluation facility does not have all the evidence. In this case, the composed evaluation or the composite evaluation can be performed.

Composed Evaluation This evaluation takes two certifications and applies the Composed Assurance package. However, this evaluation does not provide EAL (Evaluation Assurance Level) but a CAP level (Composed Assurance Package). Higher-level security assurance is not possible with this process.

Composite Evaluation This evaluation certifies a platform together with an application and is thus a layered approach for a certification. For this certification all EAL can be achieved.

The selection scheme is depicted in Figure 3.3 and takes as input the number of product developing companies, as well as the number of involved evaluation facilities. These factors are important because they determine whether developer evidence and certification evidence can be shared.

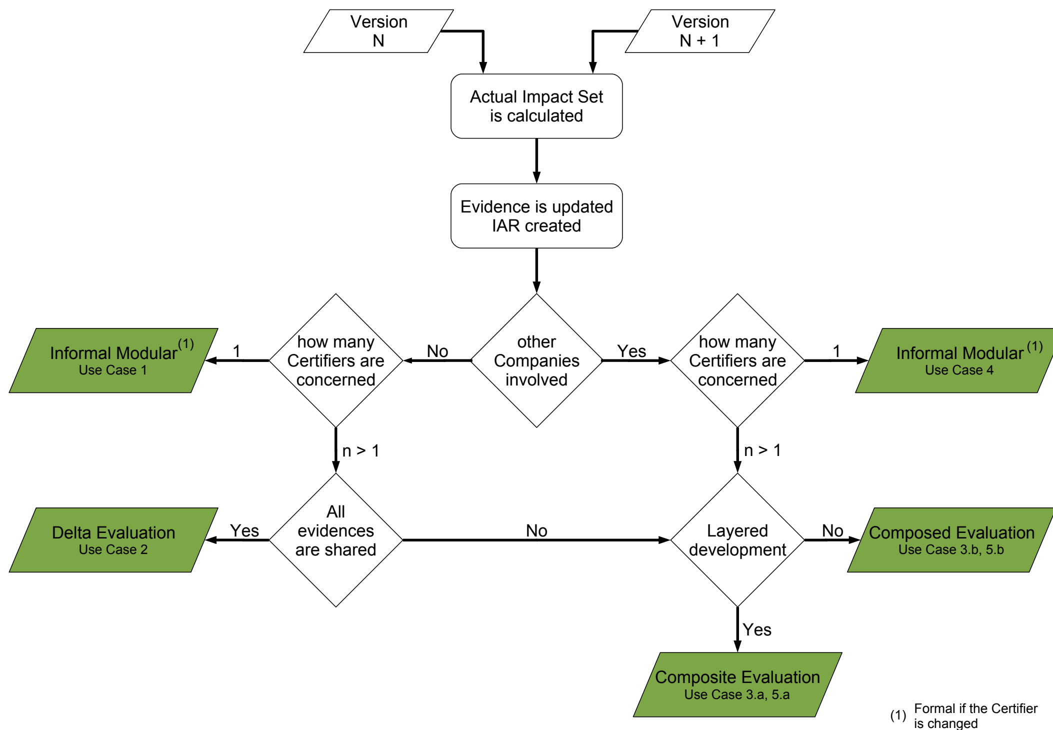


Figure 3.3: A decision framework helps to select an appropriate Common Criteria certification process [48].

The above stated selection scheme is applicable to a broad range of applications. In order to enable a better evaluation for smart cards, requirements for smart card certification are listed, below:

- The development process of smart cards is iterative because customers want to change the requirements during the development. Thus, the security requirements change in the course of a development.
- Early feedback from the evaluation facility enables an early validation of the security concepts. This early validation helps to minimize the risk of a complete late refactoring of the security architecture. The cost of such a late refactoring is very high.
- Better support for modular certification because a module can have several variants of its implementation. Moreover, it should be easy to add modules to the product without a whole re-evaluation of the system.

It is shown, how an incremental certification can meet these requirements [49]. However, in this publication the delta evaluation concept is refined and two certification processes are derived and refined. The context under which these processes can be applied is described in [47].

- Before the Common Criteria certification is started, the architecture must be close to completion. This seems to be a contradiction to the agile certification approach because in an incremental development process the architecture is said to be *evolving*. However, an implicit architecture sometimes exists and enables an iterative certification.
- The software system already exists and is evolving. For example, a new product often evolves from a base product. In this case there is always an existing architecture. Thus, an incremental certification is possible.

In this context the following two certification processes can be applied: The Traceability Impact Analysis (TIA) and the Experiential Impact Analysis (EIA). Both processes fulfill the Change Impact Analysis which is required for the incremental evaluation.

Traceability Impact Analysis (TIA): In the TIA process (see Figure 3.4), the Change Detection Analysis (CDA) first detects the delta between versions of the model. The result is the Starting Impact Set (SIS). This set is input for the TIA which calculates all artifacts which are possibly impacted. The result is the so-called Candidate Impact Set (CIS) which is the input for the manual re-evaluation. The outcome of the manual re-evaluation is three sets: the Discovered Impact Set (DIS), which contains manually detected impact, the False Positive Impact Set (FPIS), which contains not impacted artifacts of the CIS and the Actual Impact Set (AIS) which contains the detected actual impacts after the whole process.

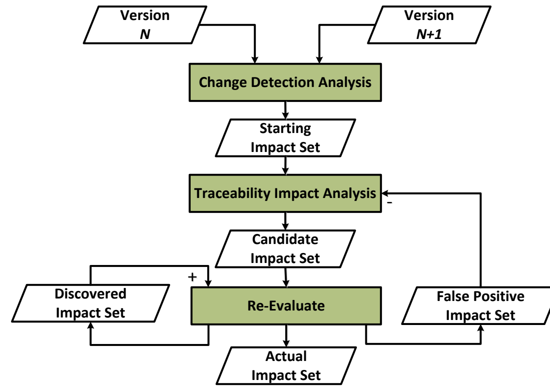


Figure 3.4: Re-evaluation process with Traceability Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts [47].

Experiential Impact Analysis (EIA): The EIA certification process (see Figure 3.5) is the similar to the TIA. Only the traceability impact analysis step is omitted. Thus, the Candidate Impact Set is the outcome of the CDA which detects the delta between two versions of the model *without* taking all possible impacts into account. Thusly, the input for the manual re-evaluation is smaller.

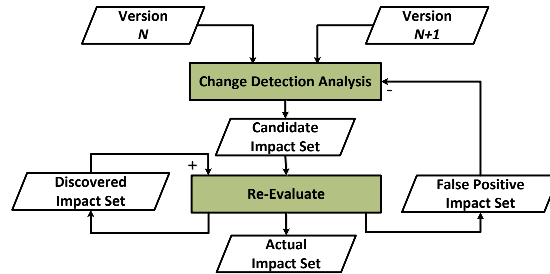


Figure 3.5: Re-evaluation process with Experiential Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts [47].

Both approaches are compared in Table 3.1. The TIA process fits better if there is a long time interval between two iterations: then the presumed experience of the evaluator regarding the software system is low. If the time interval between two iterations is low, the EIA process is more appropriate because the presumed experience of the evaluators is high.
However, the previously described certification processes can only be applied in the previously described context. If there is no implicit architecture because the product is developed for an unknown domain, the iterative certification approaches find their limits. For this purpose it is proposed that the Common Criteria standard certification should be made more flexible by composing the assurance argument from a product argument and a process argument [47].

Table 3.1: Comparison of the processes with Traceability Impact Analysis and Experiential Impact Analysis.

	TIA Process	**EIA Process**
DIS	SMALL	LARGE
FPIS	LARGE	SMALL
Presumed Experience	LOW	HIGH
Iteration Cycle	Months/Years	Weeks

Product Argument: The product argument provides evidence via features of the product, such as the architecture, the source code, the traceability from high-level objectives to evidence in the implementation.

Process Argument: The process argument provides assurance with certain properties of the development process, such as architecture reviews, testing, source code reviews and risk analysis.

In the following a short example is provided which illustrates how product assurance and process assurance can be balanced. The assurance class testing contains the assurance families *ATE_FUN*, *ATE_COV* and *ATE_DPT*. The Evaluation Assurance Level of the assurance class depends on the level of assurance of each family. In Table 3.2 a short description of each assurance family is provided and stated whether it is used as a product argument or a process argument.

Table 3.2: Mapping of Common Criteria assurance classes to process and product assurance.

CC Assurance Component	**Process Assurance**	**Product Assurance**	**Description of Assurance Component**
ATE_FUN	X		functional testing
ATE_COV		X	test coverage of functional specification
ATE_DPT		X	test coverage derived from design, architecture, ...

The current way in which Common Criteria determines whether the assurance class testing has EAL 3 is outlined in Figure 3.6. It shows a hard goal model [50] which states that for EAL 3 the following assurance components need to be verified: ATE_DPT.1, ATE_COV.1 and ATE_COV.2.

As can be seen in the current Common Criteria approach, there is no possible trade-off between process and product argument. Thus, it is proposed that the evaluation should

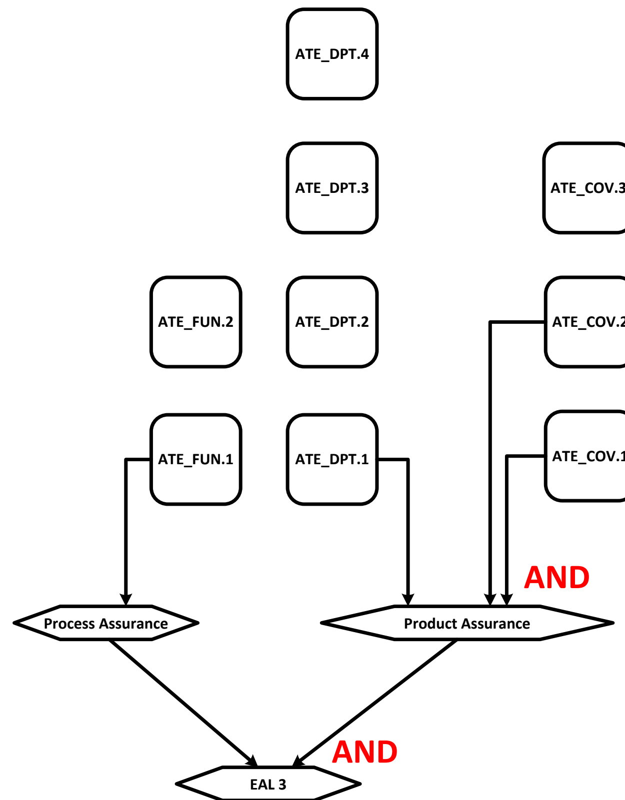


Figure 3.6: The current Common Criteria evaluation does not leave much room for flexibility. The hard goals are only fulfilled, if all (AND) inputs are true [47].

be made more flexible, as indicated in Figure 3.6. In comparison to a hard goal model, in a soft goal model [50], not all inputs into a goal have to be fulfilled. The inputs are a contribution to the soft goal and if the sum of all inputs is above a specified level, the output has a certain value. It can be seen that in this soft goal model, a higher level of process assurance can compensate for a lover level of product assurance and *vice versa*.
Moreover, in [47], it is shown how a model-based approach can be leveraged in order to better support process assurance. Figure 3.8 provides an example: in the local workspace, the developers may change the software and interfaces. In the repository, there should be code of high-quality, in this case this means reviewed code. So, if a developer changes an interface in the local workspace, a commit to repository operation would trigger the following process: then a model-difference engine would detect the changed interface and following a security architect has to review the interface change. If the change is ok, the model-merge engine merges the changes to the model in the repository. This approach combines a model-based difference analysis with a process argument (review) and a competency argument (security architect). All the review activities can be logged and serve as evidence for a certification.

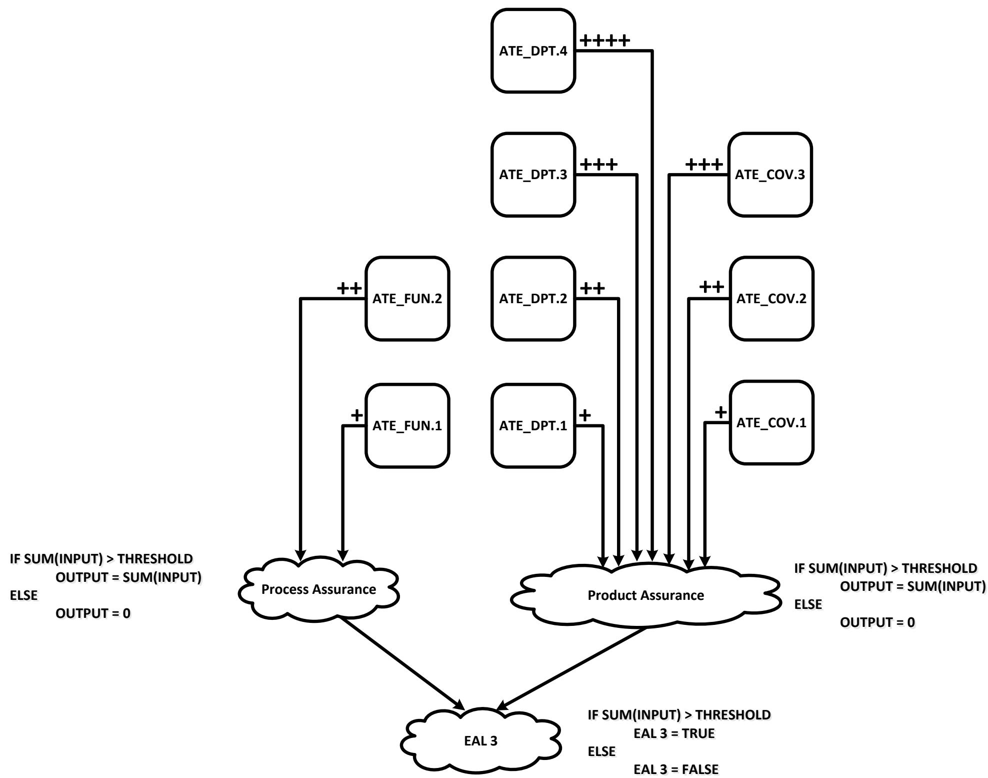


Figure 3.7: A more flexible evaluation of assurance levels can be modeled with soft goals. The plus signs next to the assurance components indicate the contribution to a soft goal. The output of the soft goals depends on the sum of its inputs. If this sum is below a specified threshold, the output is zero or FALSE. It the sum is above this threshold, the output is either the sum of all inputs or TRUE [47].

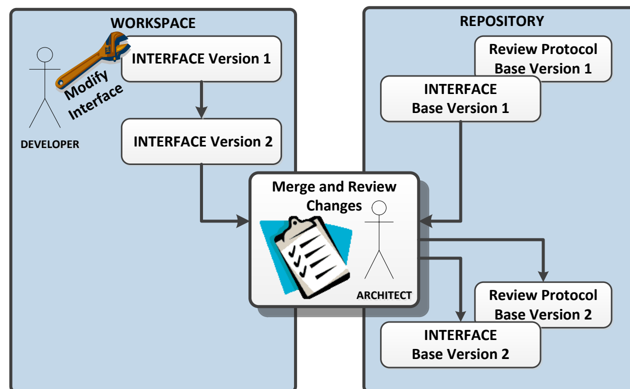


Figure 3.8: This use case illustrates how model merge and model diff contribute to an assurance argument. This argument is supported by the appropriate role (architect) who performs a process (review) [47].

3.2.2 Cost Model

There is a basic decision regarding the point in time of the certification. Basically, the certification can be done after the product development is finished or in parallel to it. In the following, a short description of both variants is provided:

Certification after the development is finished: The certification takes place, when the product development is finished. The benefit of this variant is that all evidence necessary for the certification is already in place. Moreover, the evidence does not alter because the development is finished. The drawback is that the certification is started late and thus there is a considerable increase in the time-to-market. In the smart card industry, time-to-market plays an important role, when new market segments with higher margins are entered.

Certification in parallel to the development: The certification is accomplished in parallel to the product development. This means that the evidence for certification changes when the product itself changes. Therefore, the certification has to be updated for each increment of changed evidence. Because of this continuous updating the cost of the certification is higher. The benefit is that the lag time between finishing the product and certification can be decreased. This can be a considerable advantage.

In order to quantify the cost of an incremental certification, a cost model for quality assurance has been developed. As explained above, the main factors for reworking the certification is the amount of evidence that has been changed. The following hypothesis is postulated [51]:

$$W = Q \cdot dA \tag{3.1}$$

Table 3.3: Terminology of the cost model.

Definition	Interpretation	Short
Architectural Evolution	Change of component interfaces	dA
Waste	Effort with no additional value	W
Quality assurance effort	Total effort for quality assurance	Q

This hypothesis suggests that unnecessary effort (waste) for quality assurance (security in this case) emerges when the interfaces of components are changed. In fact, this is very obvious: Every time the interfaces change, the design documentation, the functional specification and the implementation representation have to be altered. These parts have connections to other parts of the evidence, such as security functional requirements and higher level security goals. However, a detailed test of the hypothesis is provided in the corresponding publication [51].

3.2.3 Patterns of Software Modeling

Patterns of software modeling are listed, which will shortly be described, below [52]. Then it is shown, how they relate to the security evaluation processes.

Backward Engineering: In backwards engineering the software model is reconstructed from the software.

Forward Engineering: In forward engineering the model generates artifacts of the software, such as interfaces and thus the architecture.

Round-Trip Engineering: In round-trip engineering, the model can generate software artifacts. In addition the model can be updated from software artifacts. Moreover, *diff* and *merge* algorithms support information flow between model and code. Therefore, a model-evolution is supported.

Coordinative ALM: In coordinative ALM, the model-evolution is orchestrated with a process support. So, each delta-version of a model can trigger a process, such as a security review. The process steps are logged in order to deliver evidence for a process-based assurance.

Cooperative ALM: Cooperative ALM integrates knowledge-transfer tools, such as *wiki*, trackers, communication tools with model elements. The rationale behind this pattern is to provide better information sharing in the development of distributed software systems.

Table 3.4 shows how different modeling patterns are an enabler for different security certifications. It can be seen that the software modeling pattern can leverage the certification. The incremental certification is supported by round-trip engineering. A more sophisticated modeling pattern, such as coordinative ALM enables a balanced security certification which is the next logical step for improving security certifications.

Table 3.4: Mapping of the certification processes to modeling patterns and their relevance.

Modeling Pattern	**Static Certification**	**Incremental Certification**	**Balanced Certification**
Backward Engineering	x		
Forward Engineering	x		
Round-Trip Engineering	x	x	
Coordinative ALM	x	x	x
Coordinative ALM			
Relevance	past - present	present - future	future

3.3 Minor Goal: Hardware Abstraction for Future Smart Card Architectures

The source code has to be provided for a security certification and is called *implementation representation* in terms of Common Criteria. If a software module has to be ported to another processor family, the module has to be re-evaluated, if the implementation representation changes. Even if only a few lines of code are changed, this has to be done (a problem here is that all the line numbers change accordingly). However, if the implementation representation can be abstracted, so that it does not change due to a new processor or processor family, the certification can be reused. For this reason, a strategy is provided to deal with *processor* and *processor family* variability. Both types are explained in more detail, below:

Processor family variability: A processor family is a family of processors with the same architecture and instruction set, such as the 8051 architecture. A processor family may have several processors as derivates with the same architecture and the same instruction set. A processor family has a cross-cutting variability regarding the following issues [42]: endianness, structure alignment, type length and pointer optimizations. Additionally, there is a variability when hardware registers and resources are addressed: this variability is usually encapsulated within a HAL.

Processor variability: A processor is a derivate of a processor family. A processor can have additional instructions for security checks. Sometimes, security checks in hardware are faster and more reliable than software checks. Such additional security checks have to be addressed above the HAL.

Figure 3.9 shows a Java Card architecture: the lowest layer constitutes the hardware. The HAL is state-of-the-art and encapsulates register and hardware resource accesses behind a software interface. The cross-cutting issues, such as pointer optimizations and byte endianness are covered in publication [42]. A test-driven migration process is described in publication [53]. Additional instructions for security checks are handled by the D-VM layer: it is a unique interface for the virtual machine. So, the virtual machine developers do not need to directly access the hardware security checks. Rather, the D-VM provides exchangeable implementations of security checks: they can be implemented in hardware or in software. If runtime performance is an issue, the checks can be implemented in hardware and are therefore faster. If chip size is an issue, the checks can be implemented in software and are cheaper but less fast, as a consequence.

3.3.1 Patterns for Hardware Abstraction

Even if there is a defined HAL, some cross-cutting issues regarding hardware abstraction may remain. Such issues remain mainly because optimization above the HAL is still necessary. Typical optimization can be accomplished via specifying pointer locations (RAM, ROM, EEPROM). However, also these issues can be abstracted in the source code. Before the compilation, the hardware specific extensions are then injected. So, both benefits can be kept: a platform-independent code base and the possibility of some optimizations. The

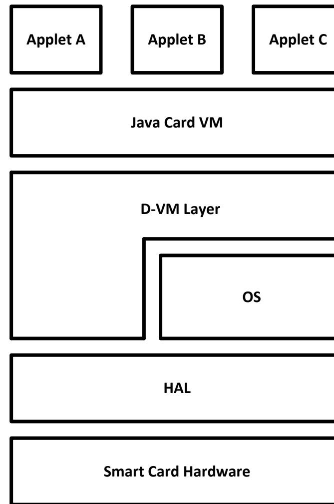


Figure 3.9: The proposed Java Card architecture contains a Hardware Abstraction Layer (HAL and also a Defensive-Virtual Machine Layer (D-VM) which encapsulates security mechanisms (see also [54]).

four patterns that have been mined in an industrial smart card development project, are described briefly below.

Pattern - *Visible Type Length*: This pattern states that the type length (number of bytes) shall be encoded in the type identifier. This avoids some problems with different hardware architectures (8-bit, 16-bit and 32-bit architecture).

Pattern - *Generic Pointer Types*: This pattern describes how hardware-specific optimization parameters can be masked for the hardware-independent code base. The hardware specific parameters are then resolved before compilation.

Pattern - *Endianness Abstraction*: Different hardware platforms may have a different endianness (byte order). When data is transmitted between different hardware platforms this may lead to errors. A data conversion function on each hardware platform avoids this problem.

Pattern - *Structure Alignment*: The memory representation of structures is usually optimized and not standardized. Thus, there may be problems, when data is transmitted between different hardware platforms. This pattern provides a solution to this problem.

3.3.2 Test-driven Porting

If a software is not written platform-independently, a transition to a platform-independent code-base can save much effort in future projects. However, such a transition is a hard task to tackle. In publication [53], a systematic process for such a refactoring is described (see Figure 3.10). This process leverages principles from test-driven development. So, first

the tests are refactored to be platform independent. In the second step, it is assured that during the refactoring of the test, no defects have been introduced. In a third step, they can be applied to the refactoring of the code-base. So, after each refactoring of the code base, the test runs provide feedback, if this operation has introduced a defect. Thus, the TDD approach can be applied for a low-risk and iterative refactoring to a platform-independent code base.

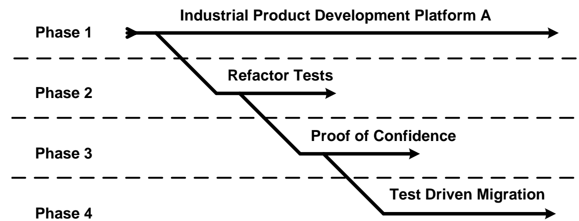


Figure 3.10: Test refactoring and proof of confidence are a precondition for a test-driven migration [53].

3.3.3 D-VM Layer

A Java Card virtual machine accesses security related assets, such as local variables, the operand stack and the like. These assets can be protected with several security mechanisms. However, the security mechanisms interact with the implementation of the virtual machine. Due to the dependencies between virtual machine and security mechanisms, exchanging the latter becomes difficult. Therefore, a novel Defensive Virtual Machine (D-VM) layer is described [54]. This layer provides a unique interface to the upper virtual machine implementation (see Figure 3.11).

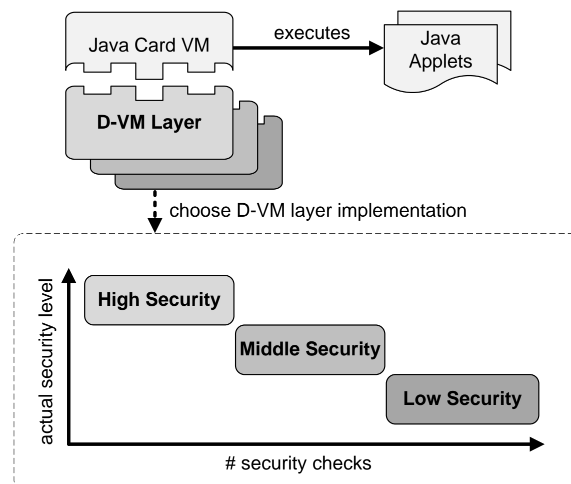


Figure 3.11: The D-VM layer provides a unique interface to the virtual machine. Thus, different levels of security countermeasure do not impede the functionality of the virtual machine (see also [54]).

Thus, exchanging the implementation of the D-VM layer versus another implementation becomes straightforward. As can be seen in Figure 3.11 there are implementations of this

layer in several levels of security. This approach supports the goal of delivering a tailored or good-enough security. Therefore, the customers only have to pay for the security level required.

3.3.4 Product Line Migration

The publication 4 is an industrial experience report on the transition of an industrial software system to a software product line [55]. It is reported, how the transition first starts with the academic partner who developed a first prototype. In a second iteration, both partners evaluate and refine the prototype. In the last iteration, the product line approach is fully integrated with the development process. Several lessons learned which have been experienced during this transition are listed.

Chapter 4

Results and Evaluation

This chapter first reports on an industrial case study of porting a secure Java Card operating system to a new processor family. This porting has been carried out with a test-driven development approach in order to minimize the risk of introducing defects to the industrial development process. Furthermore, a case study provides an impression under which constraints a beneficial security evaluation process can be selected. Depending on the evaluation process, certification evidence can be reused.

4.1 Hardware Abstraction of the Source Code

4.1.1 Pilot Study

In order to abstract the source code, a small pilot study has first been performed. The occurrences of pointer optimizations which are not platform-independent have been determined. In all software layers which are coded in C language, 443 total occurrences of problematic pointers were found and 416 of them were not commented out. In a first feasibility study, 27 of these occurrences were refactored successfully. This small success story was the motivation to start a larger case study of the approach.

4.1.2 Case Study

In this case study six developers were abstracting the code, so that compilers for two different processor families can compile the code. The code was under development for the 8051 processor family. The developers were abstracting the code, so that it can be compiled also for another processor family with a completely different instruction set. Figure 4.1 illustrates the situation: the industrial project develops source code on platform A. This development has a high priority and may not be interrupted by problems which could be introduced by the migration. Therefore, first the tests have been refactored to be platform-independent. In this first step, the test must compile for platform B. In a second step, the refactored tests have to pass on platform A, if they have passed before. In a third step, the unit itself was abstracted. When all tests had passed again on platform A, the changes have been committed to the repository. This process does not provide a fully functional porting of the software. However, it ensured that the code at least compiles on both platforms and that there is one single source code component, instead. Table 4.1

shows for each step, how many modules have passed it. This is a considerable achievement: only six developers have abstracted about 26 per cent of the modules in three months. It has to be taken into account that the industrial development team consists of about 100 developers. In addition to the migration many problems with the compiler and the tool chain for processor platform B have been detected and reported.

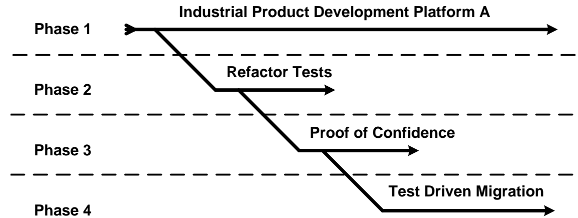


Figure 4.1: Test refactoring and proof of confidence are the precondition for a test-driven migration [53].

Table 4.1: Amount of modules which passed the first three steps of the test-driven migration process.

	Step 1	**Step 2**	**Step 3**
Number of modules	13	10	10
Percent of modules	33	26	26

4.2 Scenario Analysis of the Security Processes

Since the major goal of this thesis is to support an incremental security certification, in this evaluation several security evaluation processes are compared. The three main categories of a security evaluation are: static, incremental and balanced. Since only static and incremental certifications can be applied with the current version of the Common Criteria standard, the balanced certification is not part of this evaluation. A static certification is a certification of a whole and finished product with all developer evidence provided. An incremental certification takes two versions and a further change impact analysis of the differences is accomplished. Scenarios for these processes regarding reuse of certification evidence are described and evaluated.

4.2.1 Case study: CeSeCore Architecture Overview

In order to provide a case study, it is only possible to take an open source project as the subject of investigation. The open source project *CeSeCore*[1] is open source and certified for Common Criteria EAL 4+. All documents needed for a Common Criteria certification are available [56]. Basically, the CeSeCore is a set of security related functions bundled in a Java library. This library consists of 8 modules. These modules and their dependencies

[1] www.cesecore.eu/

are depicted in Table 4.2. In the first line of Table 4.2 can be seen (indicated by a 1) that the modules *Security Audit*, *Key Management* and *Access Control* depend on the module *Backup & Recovery*. In other words, the latter three modules are impacted by the module *Backup & Recovery*. However, this table only models direct impacts. Impacts can propagate over several modules. The direct and indirect impacts are modeled in Table 4.3.

Table 4.2: This matrix indicates the direct impacts between the modules. For example, a 1 in the first line indicates that the module *Backup & Recovery* provides a service to the corresponding module in the column.

	Backup & Recovery	*Certificate & Profile Management*	*Security Audit*	*Key Management*	*Access Control*	*Roles*	*Identification & Authentication*	*Trusted Time*
Backup & Recovery	0	0	1	1	1	0	0	0
Certificate & Profile Management	0	0	1	1	1	0	0	0
Security Audit	0	0	1	1	1	0	0	1
Key Management	0	0	1	0	1	0	0	0
Access Control	0	0	1	0	0	1	1	0
Roles	0	0	1	0	1	0	0	0
Identification & Authentication	0	0	1	0	0	0	0	0
Trusted Time	0	0	0	0	0	0	0	0

4.2.2 Case 1: Complete Evaluation

Context: The product is certified completely new, because none of the certification evidence can be reused. Several reasons may lead to this case:

- The product architecture is radically modified or completely new.
- Certification evidence is not fully available because the certification facility has been changed. Certification evidence may belong to the certification facility and is thus not always transferable to other facilities.

Reuse: 0 of 9 modules.

No reuse because the whole system has to be certified completely new. The certification can be done in a static or incremental way.

4.2.3 Case 2: Re-evaluation with Traceability Impact Analysis

Context: A new product version is certified and only one module has changed. Since the last version has been certified a long time ago, the certifiers are no longer familiar with the architecture and implementation of the product. For this reason, a full analysis of all impacts has to be accomplished. The traceability impact analysis calculates all possible (direct and indirect) impacts of a change. As can be seen in Table 4.3 only two columns do not contain any 1. This means that they are not impacted by any other modules. The last row does not contain any 1: this means that the module *Trusted Time* has no direct impact on other modules.

Reuse: 1, 2 or 7 of 9 modules.

If only one module is changed, the following modules have to be re-certified, depending on the changed module. The impacted modules which need to be re-certified are indicated by a 1 in the row of Table 4.3.

Changed module - Backup & Recovery OR Certificate and Profile Management: Reuse of evidence for 1 module. One module is not impacted. All other modules have to be re-certified.

Changed module - Trusted Time: Reuse of evidence for 7 modules. No module is impacted. So, only the module *Trusted Time* has to be re-certified.

Changed module - any other module: Reuse of evidence for 2 modules. The modules *Backup & Recovery* and *Certificate and Profile Management* are not impacted.

4.2.4 Case 3: Re-evaluation with Experiential Impact Analysis

Context: A new product version is certified and only one module has changed. The certifiers are familiar with the product and its architecture because frequently versions of this product are certified. For this reason, an Experiential Impact Analysis can be performed. This analysis starts at the changed module and impacts to other modules are evaluated manually by the certifiers. Only impacted modules are re-evaluated by the certifiers.

Reuse: 1 up to 8 of 9 modules.

At least the changed module has to be re-evaluated. Then it is evaluated whether the modules with a possible direct impact have to be re-evaluated. This process continues until no more modules have to be re-evaluated. The reuse factor is dependent on the impact of the change. However, this is the best case because only the modules that are actually impacted are evaluated, in the end.

4.2.5 Relevance of Certification Processes for the Smart Card Industry

It might be argued that the reuse approach with an incremental security certification is not relevant. In the following evaluation it will be shown that many products are incremental variants of other ones. Thus the security certification evidence could be reused, if

Table 4.3: This matrix indicates the direct and indirect impacts between the modules. For example, a 1 in the first line indicates that the module *Backup & Recovery* provides a service directly or indirectly to the corresponding module in the column.

	Backup & Recovery	*Certificate & Profile Management*	*Security Audit*	*Key Management*	*Access Control*	*Roles*	*Identification & Authentication*	*Trusted Time*
Backup & Recovery	0	0	1	1	1	1	1	1
Certificate & Profile Management	0	0	1	1	1	1	1	1
Security Audit	0	0	1	1	1	1	1	1
Key Management	0	0	1	1	1	1	1	1
Access Control	0	0	1	1	1	1	1	1
Roles	0	0	1	1	1	1	1	1
Identification & Authentication	0	0	1	1	1	1	1	1
Trusted Time	0	0	0	0	0	0	0	0

the appropriate process is applied. The protection profiles and the security target of all certified products are publicly available[2]. The security target describes the certified product and its security services. This information suffices to determine whether two products are incremental or not. This evaluation seeks to find product families which consist of incremental products. The number and size of product families allows a quantification of the reuse potential of the approach. Since the number of certified products is high and the majority of certifications are smart cards, this study is limited to smart card certifications. The time window of this evaluation is the year 2014. It is important to limit the scope of the study to a certain period, because it is a precondition for a reuse of certification evidence that the certifiers are familiar with the system. If this is not the case, the described reuse approaches cannot be applied.

Assumptions:

In order to assess whether two products are incremental several assumptions have to be taken because the security target does not contain all developer evidence.

[2]https://www.commoncriteriaportal.org/pps/

Assumption 1: Two products are incremental, if their basic functionality and their security services are similar.

Assumption 2: Two products can only be incremental, if they are based on the same processor family. The product can be a derivate of the processor family itself, or a software in combination with such a processor derivate.

Assumption 3: A processor family is a processor platform with the same architecture and instruction set.

Assumption 4: A processor derivate is an instance of a processor family. It may extend the basic architecture of the processor family by additional hardware modules for security and other purposes.

Results:

In Figure 4.2 the number of the product families dependent on their size is shown. It can be seen that there are 24 product families with a size of 1 which means that there are 24 products with no potential for reuse of certification evidence. The total number of product families with a size greater than 1 is 22. The number of products in each product family is shown in Figure 4.3. It can be seen that the overall number of reusable products is much higher than the number of non-reusable products. The total number of all reusable products is 69 as opposed to 24 non-reusable ones. It can be seen that in the smart card industry, reuse of certification evidence is applicable to the majority of all products.

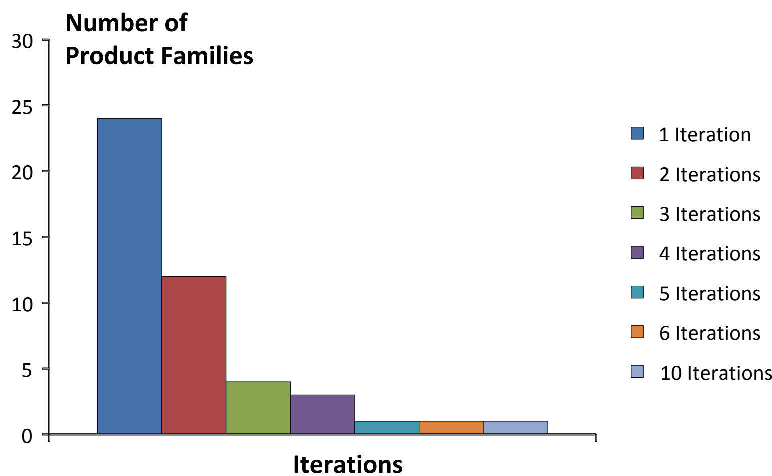


Figure 4.2: Number of product families depending on size of the product family (number of iterations). A product family is a product family and an iteration is a single version of this product family.

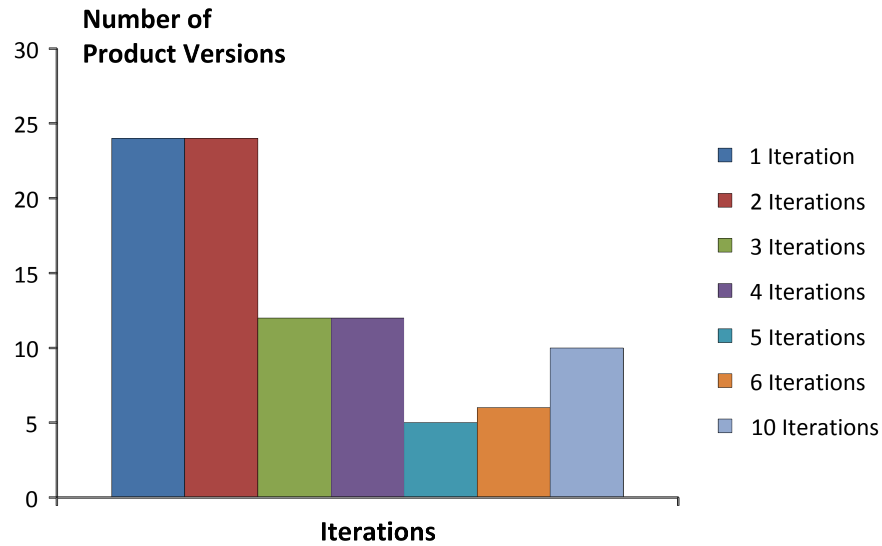


Figure 4.3: Number of product versions depending on size of the product family (number of iterations).

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis provides solutions for a major and a minor goal. The minor goal strives to unify software artifacts in order to reuse them and also to reuse the certification results of these artifacts. A layered solution and a cross-cutting solution provide a good abstraction for a secure embedded system. In addition a process for porting the code is described and evaluated. The major goal concerns the largest cost factor for the individualization of products: the security certification. The following improvements for Common Criteria certifications are described:

- Dedicated processes for iterative security evaluation and a corresponding change impact analysis are described. Moreover, it is shown how the software company interacts with the evaluation facility. This is actually the first time that a detailed process for an incremental Common Criteria certification has been published.
- The pre-conditions and the enabling context of each security certification process are described. A decision framework enables companies to find an appropriate certification process.
- Certain patterns of software modeling are required to support specific certification processes. For an iterative certification, a software model must support diff and merge operations. For a balanced certification the software model shall support a process integration.
- A cost model for security assurance helps to trade-off certification cost against time-to-market. It provides an additional guidance for selecting an appropriate certification process.
- The iterative security certification is of particularly high value for the reuse of evaluation results. Often, a new version of a product differs only in some configuration parameters. The Common Criteria requires a new certificate for such a new version.
- A detailed proposal for changing the Common Criteria standard in order to make it more flexible is provided. It is described how a specific pattern of software modeling is applied to leverage this approach.

Furthermore, this thesis shows methods that enable a hardware abstraction of the product family. The resulting consistent code base contributes to the reuse of certification results. A decision framework for selecting an appropriate certification process is of partiular value to companies which do not have much experience of Common Criteria certifications. However, the largest benefit is the described iterative certification: it enables reuse of certification results to a high degree and is a very common scenario.

5.2 Directions for Future Work

5.2.1 Security Certification

Directions for future work regarding security certification are provided in publication 7. This publication proposes to change the Common Criteria standard in order to balance product assurance and process assurance. In a first step, a mapping of Common Criteria assurance components to process and product assurance can be done. In a second step, an aggregation function which calculates the contribution of the assurance components to Evaluation Assurance Levels (EAL) can be designed. In a third step, this evaluation paradigm shall be evaluated for its appropriateness to properly evaluate the security. Detailed experience reports on the application of this method could successfully trigger a change of the Common Criteria standard.
At present, the security models support round-trip engineering which means advanced model evolution. It would be interesting to enhance the security models with a process integration. So, a high-level security model can be formally verified. Then, with the help of reviews and other process assurance techniques, the conformance of the software with this high-level model can be ascertained. This approach is not limited to Common Criteria certifications. It can be applied to many other standards.
The main goal of this thesis is to design a process which takes a product that is already certified and iterates towards a new product. This is already a great improvement. However, the approach can be improved, so that a certificate is not only issued for a single product but for a family of products or product configurations. This would be a tremendous improvement but it is still a challenge: the Common Criteria standard does not provide a certification of a family of products. However, there has been a considerable step in the right direction: it is now possible to certify a set of protection profiles with a limited set of configuration parameters. There is still some work to extend this approach to the certification of the product (which is much more complex than the protection profile).
This thesis provides different patterns of Common Criteria certifications. It would be interesting to apply this approach to other standards in the safety and security domain. The different standards can then be compared. The benefits of certain standards could be input for improving other standards. As an overall parenthesis, a pattern catalog could provide guidance for searching an appropriate certification pattern.

5.2.2 Security Requirements

Natural language requirements often incorporate ambiguities and are unclear. So, it would be interesting to investigate standards for the quality of their security requirements. Then, a semi-formal language, such as *EARS* [57] or *Planguage* [58] can help to improve the

quality of these requirements. It can be quantitatively evaluated how many security requirements can be expressed with such an approach. The benefits are clearer requirements and the possibility of using them for a formal verification. Such a formal verification is necessary for high-quality assurance of systems.
The security functional requirements of the Common Criteria standard could be expressed in such a semi-formal language. The next step would be linking the security requirements with evidence in the source code. A challenge is that it is very difficult to quantify the evidence which is distributed over the source code. Because it is so difficult to quantify the evidence, it is also a challenge to assess whether different variants fulfill the security requirements. Approaches to link requirements with evidence are goal models [50] and goal structuring notation [59] . Such a model shall also support a better means for a change impact analysis: for example it could determine the impact of taking another variant of a specific component. This impact analysis is important for the certification of product families.

Chapter 6

Publications

This chapter provides a selection of the publications that have been published in the course of this thesis. Publications 1,2 and 3 are solutions that help to achieve and keep a hardware-abstracted code base which also supports a reuse of certification evidence. Publication 4 reports on the introduction of a variant management system in an industrial project. Such a variant management is necessary to separate and compose common and variable parts of a software system. A model of a research-industry collaboration and its results are provided in this publication. Publications 5 and 6 describe security certification processes that are possible with the current version of the Common Criteria security standard. Future security certification processes are proposed in publication 7. The constraints of these processes are described in publication 8 and 9, namely as constraints regarding investment in setting up tools and a cost model for certification.

Publication 1:*A Defensive Virtual Machine Layeer to Counteract Fault Attacks on Java Cards*, 7th Workshop in Information Security Theory and Practice (WISTP'13), Heraklion, Greece, May, 28th – 30th 2013.

Publication 2:*Patterns for Hardware-Independent Development for Embedded Systems*, 20th European Conference on Pattern Languages of Programs (EuroPLoP'14), Irsee, Germany, July, 9th – 13th 2014.

Publication 3: *Test-Driven Migration Towards a Hardware-Abstracted Platform*, 5th International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS), Angers, France, February, 11th – 13th 2015.

Publication 4: *Embedding research in the industrial field: a case of a transition to a software product line*, 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering (WISE'14), Vasteras, Sweden, September, 16th 2014.

Publication 5: *Evaluation paradigm selection according to Common Criteria for an incremental product development*, International Workshop on MILS: Architecture and Assurance for Secure Systems (MILS'15), Amsterdam, January, 20th 2015.

Publication 6:*Supporting evolving security models for an agile security evaluation*, 2014 Evolving Security and Privacy Requirements Engineering (ESPRE'14), Karlskrona, Sweden, August, 25th 2014.

Publication 7: *Balancing Product and Process Assurance for Evolving Security Systems*, International Journal of Secure Software Engineering (IJCCE'15), in print.

Publication 8: *Patterns of Software Modeling*, Fifth International Workshop on Information Systems in Distributed Environment (ISDE'14), Amantea, Italy, October, 31th 2014.

Publication 9: *Where does all this waste come from?*, 21st EuroSPI Conference (EUROSPI'14), Luxembourg, Juni, 25th – 27th 2014.

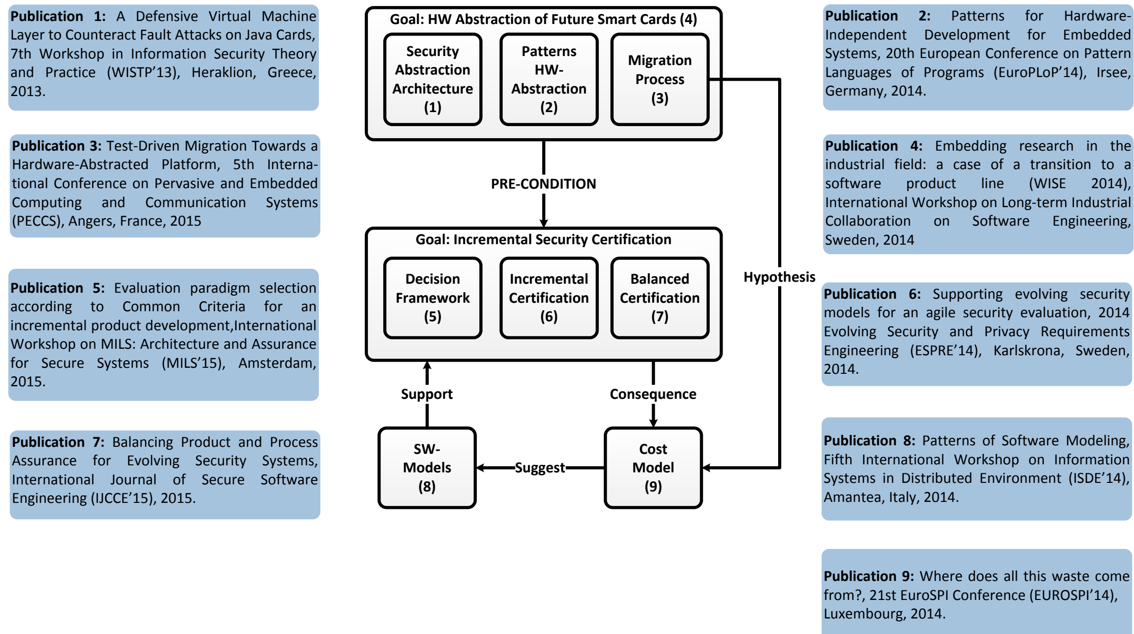


Figure 6.1: In order to achieve a hardware-abstracted code base, three strategies are applied: an architecture, guidelines or patterns for coding and a migration process. The abstracted code base increases the reusability of security certification evidence. The main goal is to support the security certifications of evolving systems. A decision framework helps to select an appropriate approach and incremental certification processes are detailed. Finally, the constraints and consequences of the proposed evaluation approaches are explained.

A Defensive Virtual Machine Layer to Counteract Fault Attacks on Java Cards

Michael Lackner, Reinhard Berlach, Wolfgang Raschke,
Reinhold Weiss, and Christian Steger

Institute for Technical Informatics,
Graz University of Technology, Graz, Austria
{michael.lackner,reinhard.berlach,wolfgang.raschke,
rweiss,steger}@tugraz.at

Abstract. The objective of Java Cards is to protect security-critical code and data against a hostile environment. Adversaries perform fault attacks on these cards to change the control and data flow of the Java Card Virtual Machine. These attacks confuse the Java type system, jump to forbidden code or remove run-time security checks. This work introduces a novel security layer for a defensive Java Card Virtual Machine to counteract fault attacks. The advantages of this layer from the security and design perspectives of the virtual machine are demonstrated. In a case study, we demonstrate three implementations of the abstraction layer running on a Java Card prototype. Two implementations use software checks that are optimized for either memory consumption or execution speed. The third implementation accelerates the run-time verification process by using the dedicated hardware protection units of the Java Card.

Keywords: Java Card, Defensive Virtual Machine, Countermeasure, Fault Attack

1 Introduction

A Java Card enables Java applets to run on a smart card. The primary purpose of using a Java Card is the write-once, run-everywhere approach and the ability of post-issuance installation of applets [21]. These cards are used in a wide range of applications (e.g., digital wallets and transport tickets) to store security-critical code, data and cryptographic keys. Currently, these cards are still very resource-constrained devices that include an 8- or 16-bit processor, 4kB of volatile memory and 128kB of non-volatile memory. To make a Java Card Virtual Machine run on such a constrained device, a subset of Java is used [19]. Furthermore, special Java Card security concepts, such as the Java Card firewall [18] and a verification process for every applet [15], were added. The Java Card firewall is a run-time security feature that protects an applet against illegal access from other applets. For every access to a field or method of an object, this check is performed. Unfortunately, the firewall security mechanism can be circumvented by applets

© 2013 WISTP. Reprinted, with permission, from Proceedings of 7th Workshop in Information Security Theory and Practice.

that do not comply with the Java Card specification. Such applets are called malicious applets.

To counteract malicious applets, a bytecode verification process is performed. This verification is performed either on-card or off-card for every applet [15]. Note that this bytecode verification is a static process and not performed during applet execution. The reasons for this static approach are the high resource needs of the verification process and the hardware constraints of the Java Card. This behavior is now abused by adversaries. They upload a valid applet onto the card and perform a fault attack (FA) during applet execution. Adversaries are now able to create a malicious applet out of a valid one [5].

A favorite time for performing a FA is during the fetching process. At this time, the virtual machine (VM) reads the next Java bytecode values from the memory. An adversary that performs an FA at this time can change the readout values. The VM then decodes the malicious bytecodes and executes them, which leads to a change in the control and data flow of the applet. A valid applet is mutated by such an FA to a malicious applet [5, 17, 11] and gains unauthorized access to secret code and data [16, 2].

To counteract an FA, a VM must perform run-time security checks to determine if the bytecode behaves correctly. In the literature, different countermeasures, such as control-flow checks [23], double checks [4], integrity checks [8] and method encryption [20], have been proposed. Barbu [3] proposed a dynamic attack countermeasure in which the VM executes either standard bytecodes or bytecodes with additional security checks.

All these works do not concentrate on the question of how these security mechanisms can be smoothly integrated into a Java Card VM. For this integration, we propose adding an additional security layer into the VM. This layer abstracts the access to internal VM resources and performs run-time security checks to counteract FAs. The primary contributions of this paper are the following:

- Introduction of a novel defensive VM (D-VM) layer to counteract FAs during run-time. Access to security-critical resources of the VM, such as the operand stack (OS), local variables (LV) and bytecode area (BA), is handled using this layer.
- Usage of the D-VM layer as a dynamic countermeasure. Based on the actual security level of the card, different implementations of the D-VM layer are used. For a low-security level, the D-VM implementation uses fewer checks than for a high-security level. The security level depends on the credibility of the currently executed applet and run-time information received by hardware or software modules.
- A case study of a defensive VM using three different D-VM layer implementations. The API of the D-VM layer is used by the Java Card VM to perform run-time checks on the currently executing bytecode.
- The defensive VMs are executed on a smart card prototype with specific HW security features to speed up the run-time verification process. The resulting

run-time and main memory consumption of all implemented D-VM layers are presented.

Section 2 provides an overview of attacks on Java Cards and the current countermeasures against them. Section 3 describes the novel D-VM layer presented in this work and its integration into the Java Card design. Furthermore, the method by which the D-VM layer enables the concept of dynamic countermeasures is presented. Section 4 presents implementation details regarding how the three D-VM implementations are inserted into the smart card prototype. Section 5 analyzes the additional costs for the D-VM implementations based on the execution and main memory overhead. Finally, the conclusions and future work are discussed in Section 6.

2 Related Work

In this section, the basics of the Java Card VM and work related to FA on Java Cards are presented. Then, an analysis of work regarding methods of counteracting FAs and securing the VM are presented. Finally, an FA example is presented to demonstrate the danger posed by such run-time attacks for the security of Java Cards.

2.1 Java Card Virtual Machine

A Java Card VM is software that is executed on a microprocessor. The VM itself can be considered a virtual computer that executes Java applets stored in the data area of the physical microprocessor. To be able to execute Java applets, the VM uses internal data structures, such as the OS or the LV, to store interim results of logical and combinatorial operations. All of these internal data structures are general objects for adversaries that attack the Java Card [4, 20, 24].

For every method invocation performed by the VM, a new Java frame [19] is created. This frame is pushed to the Java stack and removed from it when the method returns. In most VM implementations, this frame internally consists of three primary parts. These parts have static sizes during the execution of a method. The first frame part is the OS on which most Java operations are performed. The OS is the source and destination for most of the Java bytecodes. The second part is the LV memory region. The LV are used in the same manner as the registers on a standard CPU. The third part is the frame data, which holds all additional information needed by the VM and Java Card Runtime Environment (JCRE) [18]. This additional information includes, for example, return addresses and pointers to internal VM-related data structures.

2.2 Attacks on Java Cards

Loading an applet that does not conform to the specification defined in [19] onto a Java Card is a well-known problem called a logical attack (LA). After an

LA, different applets on the card are no longer protected by the so-called Java sandbox model. Through this sandbox, an applet is protected from illegal write and read operations of other applets. To perform an LA, an adversary must know the secret key to install applets. This key is known for development cards, but it is highly protected for industrial cards and only known by authorized companies and authorities. In conclusion, LAs are no longer security threats for current Java Cards.

Side-channel analyses are used to gather information about the currently executing method or instructions by measuring how the card changes environment parameters (e.g., power consumption and electromagnetic emission) during run-time. Integrated circuits influence the environment around them but can also be influenced by the environment. This influence is abused by an FA to change the normal control and data flow of the integrated circuit. Such FAs include glitch attacks on the power supply and laser attacks on the cards [2, 24]. By performing side-channel analyses and FAs in combination, it is possible to break cryptographic algorithms to receive secret data or keys [16].

In 2010, a new group of attacks called combined attacks (CA) was introduced. These CAs combine LAs and FAs to enable the execution of ill-formed code during run-time [5]. An example of a CA is the removal of the *checkcast* bytecode to cause type confusion during run-time. Then, an adversary is able to break the Java sandbox model and obtain access to secret data and code stored on the card [5, 17]. In this work work, we concentrate on countering FAs during the execution of an applet using our D-VM layer.

2.3 Countermeasures Against Java Card Attacks

Since approximately 2010, an increasing number of researchers have started concentrating on the question of what tasks must be performed to make a VM more robust against FAs and CAs. Several authors [22, 8] suggest adding an additional security component to the Java Card applet. In this component, they store checksums calculated over basic blocks of bytecodes. These checksums are calculated off-card in a static process and added to a new component of the applet. During run-time, the checksum of executed bytecodes is calculated using software and compared with the stored checksums. If these checksums are not the same, a security exception is thrown.

Another FA countermeasure is the use of control-flow graph information [23]. To enable this approach, a control-flow graph over basic blocks is calculated off-card and stored in an additional applet component. During run-time, the current control-flow graph is calculated and compared with the stored control graph.

In [20], the authors propose storing a countermeasure flag in a new applet component to indicate whether the method is encrypted. They perform this encryption using a secret key and the Java program counter for the bytecode of every method. Through this encryption, they are able to counteract attacks that change the control-flow of an applet to execute illegal code or data.

Another countermeasure against FAs that target the data stored on the OS is presented in [4]. In this work, integrity checks are performed when data are

pushed or popped onto the OS. Through this approach, the OS is protected against FAs that corrupt the OS data.

Another run-time check against FAs is proposed in [10, 14], in which they create separate OSes for each of the two data types, *integralValue* and *reference*. With this approach of splitting the OS, it is possible to counteract type-confusion attacks. A drawback is that in both works, the applet must be preprocessed.

In [3], the authors propose a dynamic countermeasure to counteract FAs. Bytecodes are implemented in different versions inside the VM, a standard version and an advanced version that performs additional security checks. The VM is now able to switch during run-time from the standard to the advanced version. By using unused Java bytecodes, an applet programmer can explicitly call the advanced bytecode versions.

The drawbacks of current FA countermeasures are that most of them add an additional security component to the applet or rely on preprocessing of the applet. This has different drawbacks, such as increased applet size or compatibility problems for VMs that do not support these new applet components. In this work, we propose a D-VM layer that performs checks on the currently executing bytecode. These checks are performed based on a run-time policy and do not require an off-card preprocessing step or an additional applet component.

2.4 EMAN4 Attack: Jump Outside the Bytecode Area

In 2011, the run-time attack EMAN4 was found [6]. In this work a laser was used to manipulate the read out values from the EEPROM to 0x00. By this laser attack an adversary is able to change the Java bytecode of post-issuance installed applets during their execution.

The target time of the attack is when the VM fetches the operands of the *goto_w* bytecode from the EEPROM. Generally the *goto_w* bytecode is used to perform a jump operation inside a method. The *goto_w* bytecode consists of the operand byte 0xa8 and two offset bytes for the branch destination [19]. This branch offset is added to the actual Java program counter to determine the next executing bytecode. An adversary which changes this offset is able to manipulate the control flow of the applet.

With the help of the EMAN4 attack it is possible to jump with the Java program counter outside the applet bytecode area (BA), as illustrated in Figure 1. This is done by changing the offset parameters of the *goto_w* bytecode from 0xFF20 to 0x0020 during the fetch process of the VM. The jump destination address of the EMAN4 attack is a data array outside the bytecode area. This data array was previously filled with adversary defined data. After the laser attack the VM executes the values of the data array. This execution of adversary definable data leads to considerably more critical security problems, such as memory dumps [7]. In this work we counteract the EMAN4 attack by our control flow policy. This policy only allows to fetch bytecodes which are inside the bytecode area.

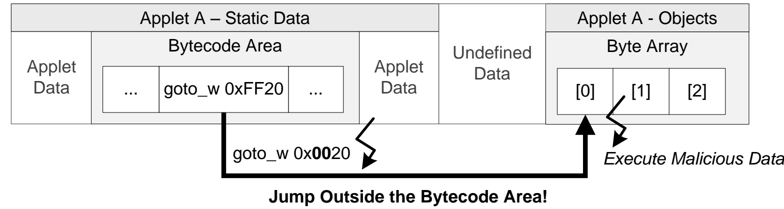


Fig. 1. The EMAN4 run-time attack changes the jump address 0xFF20 to 0x0020, which leads to the security threat of executing bytecode outside the defined BA of the current applet [6].

3 Defensive VM Layer

In this work, we propose adding a novel security layer to the Java Card. Through this layer, access to internal structures (e.g., OS, LV and BA) of the VM is handled. In reference to its defensive nature and its primary use for enabling a defensive VM, we name this layer the defensive VM (D-VM) layer. An overview of the D-VM layer and the D-VM API, which is used by the VM, is depicted in Figure 2 and is explained in detail below.

Functionalities offered by the D-VM API include, for example, pushing and popping data onto the OS, writing and reading from the LV and fetching Java bytecodes. It is possible for the VM to implement all Java bytecodes by using these API functions. The pseudo-code example in Listing 1.1 shows the process of fetching a bytecode and the implementation of the *sadd* bytecode using our D-VM API approach. The *sadd* bytecode pops two values of *integral data* type from the OS and pops the sum as an *integral data* type back onto the OS.

Listing 1.1. Pseudo-code of the VM using the API functions of the newly introduced D-VM layer.

```
//use the D-VM API to fetch the next bytecode from the BA
switch(dvm_fetch_bytecode())
{
  ...
  case sadd: //implementation of the sadd bytecode.
  {
    //use the D-VM API to obtain the two values from the OS
    result = dvm_pop_integralData() + dvm_pop_integralData();
    //use the D-VM API to write the sum back onto the OS
    dvm_push_integralData(result);
  }
  ...
}
```

The security mechanisms within the security layer intended to protect the VM from FAs are hidden from the VM programmer. A security architect, spe-

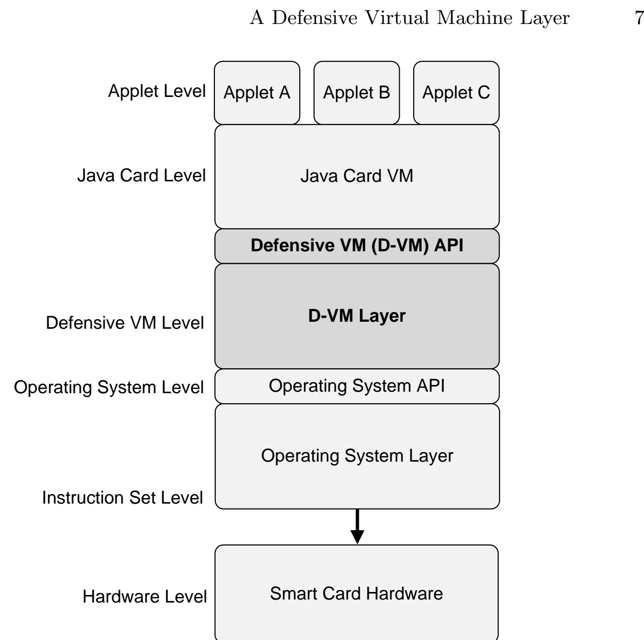


Fig. 2. The VM executes Java Card applets and uses the newly introduced D-VM layer to secure the Java Card against FAs.

cialized for VM security, is able to implement and choose the appropriate countermeasures within the D-VM layer. These countermeasures are based on state-of-the-art knowledge and the hardware constraints of the smart card architecture. Programmers implementing the VM do not need to know these security techniques in detail but rather just use the D-VM API functions.

If HW features are used, the D-VM layer communicates with these units and configures them through specific instructions. Through this approach, it is also very easy to alter the SW implementations by changing the D-VM layer implementation without changing specific Java bytecode implementations. It is possible to fulfill the same security policy on different smart card platforms where specific HW features are available.

On a code size-constrained smart card platform, an implementation that has a small code size but requires more main memory or execution time is used. The appropriate implementations of security features within the D-VM API are used without the need to change the entire VM.

Dynamic Countermeasures: The D-VM layer is also a further step to enable dynamic fault attack countermeasures such as that proposed by Barbu in [3]. In this work, he proposes a VM that uses different bytecode implementations

depending on the actual security level of the smart card. If an attack or malicious behavior is detected, the security level is decreased. This decreased security leads to an exchange of the implemented bytecodes with more secure versions. In these more secure bytecodes, different additional checks, such as double reads, are implemented, which leads to decreased run-time performance.

Our D-VM layer further advances this dynamic countermeasure concept. Depending on the actual security level, an appropriate D-VM layer implementation is used. Therefore, the entire bytecode implementation remains unchanged, but it is possible to dynamically add and change security checks during run-time. An overview of this dynamic approach is outlined in Figure 3.

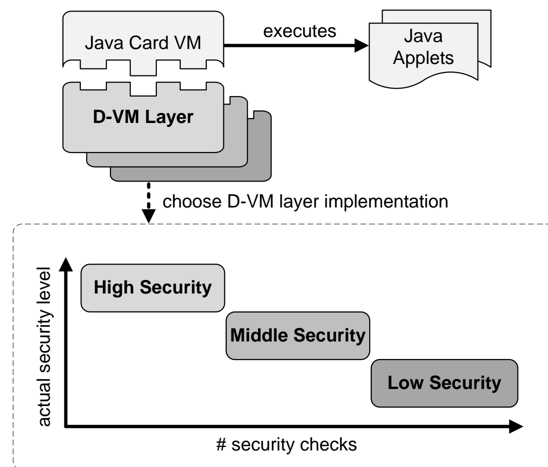


Fig. 3. Based on the current security level of the VM, an appropriate D-VM layer implementation is chosen.

The actual security level of the card is determined by HW sensors (e.g., brightness and supply voltage) and the behavior of the executing applet. For example, at a high security level, the D-VM layer can perform a read operation after pushing a value into the OS memory to detect an FA. At a lower security level, the D-VM layer performs additional bound, type and control-flow checks.

Security Context of an Applet: Another use case for the D-VM layer is the post-issuance installation of applets on the card. We focus on the user-centric ownership model (UCOM) [1] in which Java Card users are able to load their own applets onto the card. For the UCOM approach, each newly installed applet is assigned a defined security level at installation time. The security level depends

on how trustworthy the applet is. For example, the security level for an applet signed with a valid key from the service provider is quite high, which results in a high execution speed. Such an applet should be contrasted with an applet that has no valid signature and is loaded onto the card by the Java Card owner. This applet will run at a low security level with many run-time checks but a slower execution speed. Furthermore, access to internal resources and applets installed on the card could be restricted by the low security level.

3.1 Security Policy

This chapter introduces the three security policies used in this work. With the help of these policies, it is possible to counteract the most dangerous threats that jeopardize security-critical data on the card. The type and bound policies are taken from [14] and are augmented with a control-flow policy. The fulfillment of the three policies on every bytecode is checked by three different D-VM layer implementations using our D-VM API.

Control-Flow Policy: The VM is only allowed to fetch bytecodes that are within the borders of the currently active method's BA. Fetching of bytecodes that are outside of this area is not allowed. The actual valid method BA changes when a new method is invoked or a return statement is executed. Because of this policy, it is no longer possible for control-flow changing bytecodes (e.g., *goto_w* and *if_scmp_w*) to jump outside of the reserved bytecode memory area. This policy counters the EMAN4 attack [6] on the Java Card and all other attacks that rely on the execution of a data array or code of an-other applet that is not inside the current BA.

Type Policy: Java bytecodes are strongly typed in the VM specification [19]. This typing means that for every Java bytecode, the type of operand that the bytecode expects and the type of the result stored in the OS or LV are clearly defined. An example is the *sastore* bytecode, which stores a *short* value in an element of a *short* array object. The *sastore* bytecode uses the top three elements from the OS as operands. The first element is the address of the array object, which is of type *reference*. The second element is the index operand of the array, which must be of type *short*. The third element is the value, which is stored within the array element and is of type *short*.

Type confusion between values of integral data (*boolean*, *byte* or *short*) and object references (*byte[]*, *short[]* or *class A*, for example) is a serious problem for Java Cards [24, 17, 13, 25, 6, 11]. To counter these attacks, we divide all data types into the two main types, *integralData* and *reference*. Note that this policy does not prevent type confusion inside the main type *reference* between array and class types.

Bound Policy: Most Java Card bytecodes push and pop data onto the OS or read and write data into the LV, which can be considered similar to registers. The

OS is the main component for most Java bytecode operations. Similar to buffer overflow attacks in C programs [9], it is possible to overflow the reserved memory space for the OS and LV. An adversary is then able to set the return address of a method to any value. Such an attack was first found in 2011 by Bouffard [6, 7]. An overflow of the OS happens by pushing or popping too many values onto the OS. An LV overflow happens when an incorrect LV index is accessed. This index parameter is decoded as an operand for several LV-related bytecodes (e.g., *sstore*, *sload* and *sinc*). This operand is therefore stored permanently in the non-volatile memory. Thus, changing this operand through an FA gives an attacker access to memory regions outside the reserved LV memory region. These memory regions are created for every method invoked and are not changed during the method execution. Therefore in this work, we permit Java bytecodes to operate only within the reserved OS and LV memory regions.

4 Java Card Prototype Implementation

In this work three implementations of the D-VM layer are proposed to perform run-time security checks on the currently executing bytecode. Two implementations perform all checks in SW to ensure our security policies. One implementation uses dedicated HW protection units to accelerate the run-time verification process. The implementations of the D-VM layer were added into a Java Card VM and executed on a smart card prototype. This prototype is a cycle-accurate SystemC [12] model of an 8051 instruction set-compatible processor. All software components, such as the D-VM layer and the VM, are written in C and 8051 assembly language.

4.1 D-VM Layer Implementations

This section presents the implementation details for the three implemented D-VM layers used to create a defensive VM. All three implemented D-VM layers fulfill our security policy presented in Chapter 3 but differ from each other in the detailed manner in which the policies are satisfied. The key characteristic of the two SW D-VM implementations is that they use a different implementation of the type-storing approach to counteract type confusion. The run-time type information (*integralData* or *reference*) used to perform run-time checks can be stored either in a type bit-map (memory optimization) or in the actual word size of the microprocessor (speed optimization). The HW Accelerated D-VM uses a third approach and stores the type information in an additional bit of the main memory. Through this approach, the HW can easily store and check the type information for every OS and LV entry. An overview of how the type-storing policy is ensured by our D-VM implementations and a memory layout overview are shown in Figure 4 and explained in detail in the next sections.

Bit Storing D-VM: This D-VM layer implementation stores the type information for every element on the OS and LV in a type bitmap. The type information

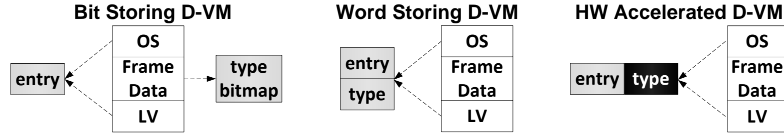


Fig. 4. The Bit Storing D-VM stores the type information for every OS and LV entry in a type bitmap. The Word Storing D-VM stores the type information below the value in the reserved OS and LV spaces. The HW Accelerated D-VM holds the type information as an additional type bit, which increases the memory size of a word from 8 bits to 9 bits.

for every entry of the OS and LV is now represented by a one-bit entry. A problem with this approach is that the run-time overhead is quite high because different shift and modulo operations must be performed to store and read the type information from the type bitmap. These operations (shift and modulo) are, for the 8051 architecture, computationally expensive operations and thus lead to longer execution times. An advantage of the bit-storing approach is the low memory overhead required to hold the type information in the type bitmap.

Word Storing D-VM: The run-time performance of the type storing and reading process is increased by storing the type information using the natural word size of the processor and data bus on which the memory for the OS and LV is located. Every element in the OS and LV is extended with a type element of a word size such that it can be processed very quickly by the architecture. By choosing this implementation, the memory consumption of the type-storing process increases compared with the previously introduced SW Bit Storing D-VM. Pseudo-codes for writing to the top of the stack of the OS for the bit- and word-storing approach are shown in Listings 1.2 and 1.3.

Listing 1.2. Operations needed to push an element on the OS by the Bit Storing D-VM.

```
dvm_push_integralData(value)
{
   //push value onto OS and
   //increase OS size
 OS[size++] = value;
   //store type information
   //into type bitmap,
   //INT->integralData type
 bitmap[size/8] = INT<<(size%8);
}
```

Listing 1.3. Operations needed to push an element on the OS by the Word Storing D-VM.

```
dvm_push_integralData(value)
{
   //push value onto OS
   //increase OS size
 OS[size++] = value;
   //store type information
   //into next memory word,
   //INT->integralData type
 OS[size++] = INT;
}
```

HW Accelerated D-VM: Performing type and bound checks in SW to fulfill our security policy consumes a lot of computational power. Types must be loaded, checked and stored for almost every bytecode. The bounds of the OS and LV must be checked such that no bytecode performs an overflow. The HW Accelerated D-VM layer uses specific HW protection units of the smart card to accelerate these security checks. New protection units (bound protection and type protection) are able to check if the current memory move (MOV) operation is operating in the correct memory bounds. The type information for the OS and LV entries is stored as an additional type bit for every main memory word. The information is decoded into new assembly instructions to specify which memory region (OS, LV or BA) and with which data type (*integralData* or *reference*) the MOV operation should write or read data. An overview of the HW Accelerated D-VM is shown in Figure 5. Depending on the assembly instruction, the HW protection units perform four security operations:

– Check if the Java opcode is fetched from the current active BA.
– Check if the destination address of the operation is within the memory area of the OS or LV. If the operation is not within these two bounded areas, a HW security exception is thrown.
– For every write operation write the type decoded in the CPU instruction into the accessed memory word.
– For every read operation, check if the stored type is equal to the type decoded in the CPU instruction. If they are not equal, throw a hardware security exception.

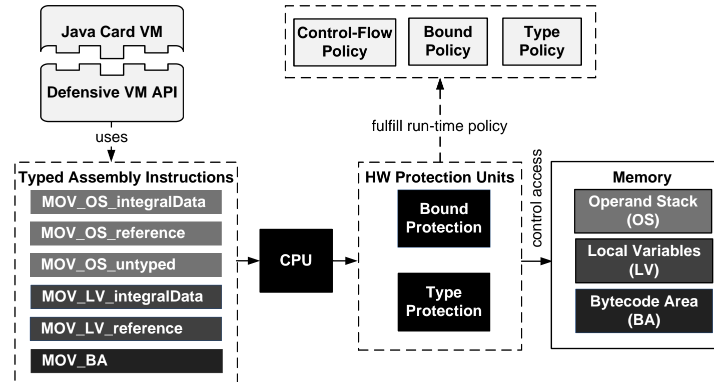


Fig. 5. Overview of the HW Accelerated D-VM implementation using new typed assembly instructions to access VM resources (OS, LV and BA). Malicious Java bytecodes violating our run-time policy will be detected by new introduced HW protection units.

5 Prototype Results

In this section, we present the overall computational overhead of the three implemented D-VM layers and their main memory consumption. All of them are compared with a VM implementation without the D-VM layer. The speed comparison is performed for different groups of bytecodes by self written micro-benchmarks where all bytecodes under test are measured. These test programs first perform an initialization phase where the needed operands for the bytecode under test are written into the OS or LV. After the execution of the bytecode under test the effects on the OS or LV are removed. Note that our smart card platform has no data or instruction cache. Therefore, no caching effects must be taken into account for all test programs.

5.1 Computational Overhead

Speed comparisons for specific bytecodes are shown in Figure 6. For example, the Java bytecode *sload* requires 148% more execution time for the Word Storing D-VM. For the Bit Storing D-VM, the execution overhead is 212%. The increased overhead is because of the expensive calculations used to store the type information in a bitmap. For the HW Accelerated D-VM, the execution speed decreases by only 4% because all type and bound checks are performed using HW. Additional run-time statistics for groups of bytecodes are listed in Table 1. As expected, the Bit Storing D-VM consumes the most overall run-time, with an increase of 208%. The Word Storing D-VM needs 142% more run-time. The HW Accelerated D-VM has only 6% more overhead.

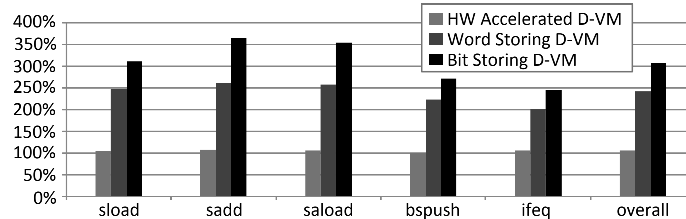


Fig. 6. Speed comparison of individual bytecodes for the different D-VM layer implementations proposed in this work. The results are compared with a VM without the D-VM layer.

5.2 Main Memory Consumption

The HW Accelerated D-VM requires one type bit per 8 bits of data to store the type information during run-time. This results in an overall main memory

Table 1. Speed comparison for different groups of bytecodes compared with a VM without the D-VM layer.

Bytecode Groups	HW Accelerated D-VM	Word Storing D-VM	Bit Storing D-VM
Arithmetic/Logic	+7%	+146%	+240%
LV Access	+5%	+185%	+243%
OS Manipulation	+5%	+151%	+231%
Control Transfer	+7%	+113%	+173%
Array Access	+5%	+130%	+166%
Overall	+6%	+142%	+208%

increase of 12.5%. The Word Storing D-VM requires in the worst case 33% more memoy because one type byte holds the type information for two data bytes. The Bit Storing D-VM requires approximately 6.25% more memory in the case in which the entire memory is filled with OS and LV data. This is because the Bit Storing D-VM requires one type bit per 16 bits of data.

6 Conclusions and Future Work

This work presents a novel security layer for the virtual machine (VM) on Java Cards. Because it is intended to defend against fault attacks (FAs), it is called the defensive VM (D-VM) layer. This layer provides access to security-critical resources of the VM, such as the operand stack, local variables and the bytecode area. Inside this layer, security checks, such as type checking, bound checking and control-flow checks, are performed to protect the card against FAs. These FAs are executed during run-time to change the control and data flow of the currently executing bytecode.

By storing different implementations of the D-VM layer on the card, it is possible to choose the appropriate security implementation based on the actual security level of the card. Through this approach, the number of security checks can be increased during run-time by switching among different D-VM implementations. Furthermore, it is possible to assign a trustworthy applet a low security level, which results in high execution performance, and vice versa. One D-VM layer implementation can be, for example, low security with high execution speed or high security with low execution speed. Another advantage is the concentration of the security checks inside the layer.

To demonstrate this novel security concept, we implemented three D-VM layers on a smart card prototype. All three layers fulfill the same security policy (control-flow, type and bound) for bytecodes but differ in their implementation details. Two D-VM layer implementations are fully implemented in software but differ in the manner in which the type information is stored. The Bit Storing D-VM has the highest run-time overhead, 208%, but the lowest memory increase, 6.25%. The Word Storing D-VM decreases the run-time overhead to 142% but consumes approximately 33% more memory. The HW Accelerated D-VM uses dedicated Java Card HW to accelerate the run-time verification process and has an execution overhead of only 6% and a memory increase of 12.5%.

In future work, we will focus on the question of which sensor data should be used to increase the internal security of the Java Card. Another question is how many security states are required and how much they differ in their security needs.

Acknowledgments The authors would like to thank the Austrian Federal Ministry for Transport, Innovation, and Technology, which funded the CoCoon project under the FIT-IT contract FFG 830601. We would also like to thank our project partner NXP Semiconductors Austria GmbH.

References

1. Akram, R., Markantonakis, K., Mayes, K.: A Paradigm Shift in Smart Card Ownership Model. In: Computational Science and Its Applications (ICCSA), 2010 International Conference on. pp. 191 –200 (march 2010)
2. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proceedings of the IEEE 94(2), 370 –382 (2006)
3. Barbu, G., Andouard, P., Giraud, C.: Dynamic Fault Injection Countermeasure. In: Mangard, S. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7771, pp. 16–30. Springer Berlin Heidelberg (2013)
4. Barbu, G., Duc, G., Hoogvorst, P.: Java Card Operand Stack:Fault Attacks, Combined Attacks and Countermeasures. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 297–313. Springer Berlin Heidelberg (2011)
5. Barbu, G., Thiebeauld, H., Guerin, V.: Attacks on Java Card 3.0 Combining Fault and Logical Attacks. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 148–163. Springer Berlin Heidelberg (2010)
6. Bouffard, G., Iguchi-Cartigny, J., Lanet, J.L.: Combined Software and Hardware Attacks on the Java Card Control Flow. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 283–296. Springer Berlin Heidelberg (2011)
7. Bouffard, G., Lanet, J.L.: The Next Smart Card Nightmare. In: Naccache, D. (ed.) Cryptography and Security: From Theory to Applications, Lecture Notes in Computer Science, vol. 6805, pp. 405–424. Springer Berlin / Heidelberg (2012)
8. Bouffard, G., Lanet, J.L., Machemie, J.B., Poichotte, J.Y., Wary, J.P.: Evaluation of the Ability to Transform SIM Applications into Hostile Applications. In: Prouff, E. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7079, pp. 1–17. Springer Berlin / Heidelberg (2011)
9. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]. pp. 227 – 237 (2003)
10. Dubreuil, J., Bouffard, G., Lanet, J.L., Cartigny, J.: Type Classification against Fault Enabled Mutant in Java Based Smart Card. In: Availability, Reliability and Security (ARES), 2012 Seventh International Conference on. pp. 551 –556 (aug 2012)

11. Hamadouche, S., Bouffard, G., Lanet, J.L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: Subverting Byte Code Linker service to characterize Java Card API. pp. 122–128. Proceedings of the 7th Conference on Network and Information Systems Security (SAR-SSI) (2012)
12. IEEE: Open SystemC Language Reference Manual IEEE Std 1666-2005, IEEE
13. Iguchi-Cartigny, J., Lanet, J.L.: Developing a Trojan applets in a smart card. Journal in Computer Virology 6, 343–351 (2010)
14. Lackner, M., Berlach, R., Loinig, J., Weiss, R., Steger, C.: Towards the Hardware Accelerated Defensive Virtual Machine - Type and Bound Protection. In: Mangard, S. (ed.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 7771, pp. 1–15. Springer Berlin Heidelberg (2013)
15. Leroy, X.: Bytecode verification on Java smart cards. Software: Practice and Experience 32(4), 319–340 (2002)
16. Markantonakis, K., Mayes, K., Tunstall, M., Sauveron, D., Piper, F.: Smart card security. In: Nedjah, N., Abraham, A., Mourelle, L. (eds.) Computational Intelligence in Information Assurance and Security, Studies in Computational Intelligence, vol. 57, pp. 201–233. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-71078-3_8
17. Mostowski, W., Poll, E.: Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In: Grimaud, G., Standaert, F.X. (eds.) Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, vol. 5189, pp. 1–16. Springer Berlin / Heidelberg (2008)
18. Oracle: Runtime Environment Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
19. Oracle: Virtual Machine Specification. Java Card Platform, Version 3.0.4, Classic Edition (2011)
20. Razafindralambo, T., Bouffard, G., Thampi, B., Lanet, J.L.: A Dynamic Syntax Interpretation for Java Based Smart Card to Mitigate Logical Attacks. In: Thampi, S., Zomaya, A., Strufe, T., Alcaraz Calero, J., Thomas, T. (eds.) Recent Trends in Computer Networks and Distributed Systems Security, Communications in Computer and Information Science, vol. 335, pp. 185–194. Springer Berlin Heidelberg (2012)
21. Sauveron, D.: Multiapplication smart card: Towards an open smart card? Information Security Technical Report 14(2), 70 – 78 (2009), Smart Card Applications and Security
22. Sere, A., Iguchi-Cartigny, J., Lanet, J.L.: Checking the Paths to Identify Mutant Application on Embedded Systems. In: Kim, T.h., Lee, Y.h., Kang, B.H., Slezak, D. (eds.) Future Generation Information Technology, Lecture Notes in Computer Science, vol. 6485, pp. 459–468. Springer Berlin / Heidelberg (2010)
23. Sere, A., Iguchi-Cartigny, J., Lanet, J.L.: Evaluation of Countermeasures Against Fault Attacks on Smart Cards. International Journal of Security and Its Applications, Vol.5 No.2 pp. 49–61 (2011)
24. Vertanen, O.: Java Type Confusion and Fault Attacks. In: Breveglieri, L., Koren, I., Naccache, D., Seifert, J.P. (eds.) Fault Diagnosis and Tolerance in Cryptography, Lecture Notes in Computer Science, vol. 4236, pp. 237–251. Springer Berlin / Heidelberg (2006)
25. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In: Gollmann, D., Lanet, J.L., Iguchi-Cartigny, J. (eds.) Smart Card Research and Advanced Application, Lecture Notes in Computer Science, vol. 6035, pp. 133–147. Springer Berlin Heidelberg (2010)

Patterns for Hardware-Independent Development for Embedded Systems

Wolfgang Raschke
Institute for Technical Informatics
Graz University of Technology
wolfgang.raschke@tugraz.at

Massimiliano Zilli
Institute for Technical Informatics
Graz University of Technology
massimiliano.zilli@tugraz.at

Stefan Orehovec
Institute for Technical Informatics
Graz University of Technology
stefan.orehovec@tugraz.at

Erik Gera-Fornwald
Institute for Technical Informatics
Graz University of Technology
erik.gera-fornwald@tugraz.at

Johannes Loinig
NXP Semiconductors Austria
johannes.loinig@nxp.com

Christian Steger
Institute for Technical Informatics
Graz University of Technology
steger@tugraz.at

Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
christian.kreiner@tugraz.at

ABSTRACT

In embedded systems, different hardware architectures require additional effort for writing portable software. Generally, a Hardware Abstraction Layer (HAL) provides an abstraction for portable software. However, a HAL is a layer and does not support an abstraction of cross-cutting issues, such as data types. Such cross-cutting issues provide often some possibility for optimization parameters. In this paper we provide some patterns which help to mask such platform-dependent parameters. In this way, there is a defined point, where these platform-dependent parameters are inserted. The rest of the code is then abstracted and coded platform-independently. The platform-dependent parameters are then resolved during pre-processing and compilation. The described patterns have been mined and also applied in a large-scale industrial embedded software project. They have shown to cover most of the concerns that appear when platform-dependent optimizations shall be combined with platform-independent coding.

Categories and Subject Descriptors

B.1.4 [**Microprogram Design Aids**]: Firmware engineering; D.2.11 [**Software Architectures**]: Patterns; D.2.13 [**Reusable Software**]: Reuse models

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EuroPLoP'14 , July 09 - 13, 2014, Irsee, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3416-7/14/07…$15.00
http://dx.doi.org/10.1145/2721956.2721959 .

General Terms

Design, Languages

1. INTRODUCTION

The development of software for embedded systems is fundamentally different from that of classical PC systems. The development of such an embedded system has a strong connection between software (SW) and hardware (HW): embedded systems are designed usually with performance issues in mind. Thus, the software exploits the features which the hardware provides to speed up performance or to optimize for power and the like. Such hardware features can be for example: efficient memory handling (DMA controller), asynchronous communication (UART) and so on. Performance is important in embedded systems and thus, compilers optimize memory representation of data (e.g. structure alignment). The memory representation is also an issue for standard types, such as: int, long, and double. When data is exchanged between hardware platforms, also the byte order comes into play (big endian, little endian). Moreover, compilers often allow to specify the location of a pointer, and it's pointed object. All these issues are cross-cutting: they may occur in each layer of an embedded system's software, even above a Hardware Abstraction Layer (HAL). In this paper, we will further examine these cross-cutting issues and present patterns which show how to cope with them. The target audience are experienced embedded system developers who want to achieve portability and also optimize the code for performance. Beginners are not the audience because many of the described patterns are not relevant if they

© 2014 EUROPLOP. Reprinted, with permission, from Proceedings of 20th European Conference on Pattern Languages of Programs.

do not have a focus on optimization issues. Because the patterns can be used for programming close to hardware, we assume a profound knowledge of the C programming language.

2. HARDWARE ABSTRACTION

Hardware abstractions are sets of routines in software that abstract details of a specific hardware platform. A hardware platform is defined as follows[1]: "Each hardware platform, or CPU family, has a unique machine language. All software presented to the computer for execution must be in the binary coded machine language of that CPU". Typical hardware abstractions are file operations: they are hardware-independent functions to access a file. These hardware-independent file operations need to be glued to the hardware which may be a flash memory or an EEPROM memory. The HAL is this intermediate glue logic. Other examples for HAL (Hardware Abstraction Layer) functions are timer/clock handling, communication and I/O handling. To control these hardware units, often specific hardware control registers need to be manipulated. These registers (and their location) differ from one platform to another. The HAL is an intermediate software layer between the software application and the hardware platform. The idea behind a HAL is to provide a standardized interface to the upper software application (see Figure 1c and Figure 1d). This avoids a major rework of the software application for each processor as indicated in Figure 1a and Figure 1b. Nevertheless, the HAL has to be designed individually for each processor.

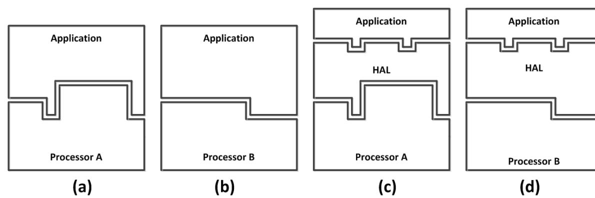


Figure 1: The interface of the application needs to be reworked for each hardware platform (see a and b). If an intermediate HAL-layer exists (see c and d), the application can be reused on different hardware platforms without refactoring.

The Hardware Abstraction Layer (HAL) provides the following benefits:

- Facilitating portability because the HAL always provides the same interface to the application.
- Allowing application developers to extract performance out of the hardware devices: the HAL gives options for different modes of operation. For example, depending on the application, symmetric or asymmetric communication mechanisms have a better performance.
- Enabling application development independently of the hardware architecture.

To ensure portability, the application software should be written in a high-level language, without using CPU-specific instructions. A high-level language can be compiled to different CPU's, which does apparently not work with CPU-specific assembler code. Thus, HAL-functions should be used. This is an important part of the abstraction to make an application platform-independent. The lower layers (Processor A and B) have unique interfaces. The HAL is an abstraction which provides a unified interface to the upper layer. However, there remain some cross-cutting issues which cannot be resolved by the HAL layer alone. Partly, these issues remain because some optimizations are also necessary and desirable above the HAL layer. These optimizations are usually concerned with runtime performance. Thus, they are indispensable in embedded systems. Also, some of the mentioned issues are due to the different properties of a hardware architecture: byte order and structure alignment. When data are transmitted from one device to another, it is necessary to consider the memory representation. This paper is not about the HAL itself. It deals with these cross-cutting issues. The cross-cutting issues, which we discuss in this paper are:

- Different datatype sizes
- Different pointer sizes and locations
- Endianness
- Struct Alignment

In this paper, we address these issues with appropriate patterns which we will explain in the following sections.

3. PATTERN MAP

In Figure 2, the patterns and their relationships are shown. The *Visible Type Length* pattern is appropriate to abstract simple types, such as int, short int and long int. The *Generic Pointer Type* pattern helps to pass optimization parameters to pointers in a platform-independent way. For complex types the *Structure Alignment* pattern helps to deal with different byte orders and different memory representation of complex types. The *Endianness Abstraction* pattern helps to deal with communication between systems with different byte endianness.

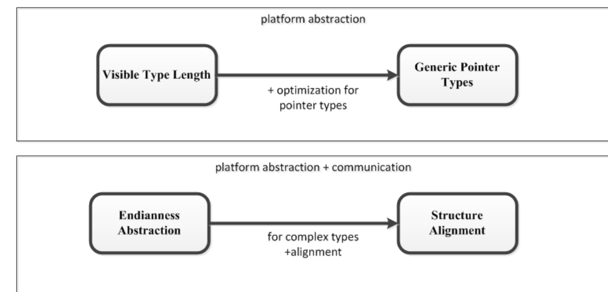


Figure 2: Pattern map - relation between the introduced patterns.

[1] http://www.pcmag.com/encyclopedia/term/44113/hardware-platforms

4. PATTERNS

4.1 Pattern Name: Visible Type Length

Context: You are developing software to be deployed on platforms of different architecture. Historically, different platforms have used different sizes of integer. It was (and still is) general practice to define the native size as the size of an int type. So, on an 8-bit architecture, the int type has 8 bit. On a 16-bit architecture, the int has a size of 16 bit. So, the definition of an int is always relative and not absolute. This pattern is applicable to the following types (and their extensions): int, float and double (see [3]).

Problem: I want to use the appropriate data type on each platform without worrying about native data type definitions.

Forces: On the one hand, using the encoded type name comes with some additional overhead: the type has to be defined in a header file for each hardware platform. Moreover, the application of encoded type names makes only sense, if there is some means of control, that the alternative types are not used. Confusion would be the result. On the other hand, using native type definitions (not the encoded types) limits the portability of the source code. A dangerous source for defects is introduced: native types may still be compiling on another platform, but the type length can be different.

Solution: The size of the type name shall be encoded in the type name. When the size of type name is appropriate, the definition of the relevant types should be given in a separate file (see Figure 3). This file maps all native types to the platform-independent types. In the source code then, the appropriate definition file is inserted depending on the hardware platform.

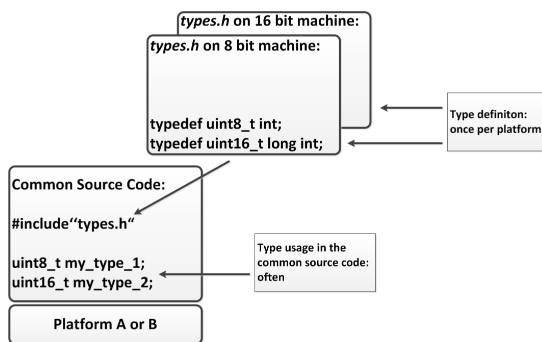


Figure 3: Definition and usage of platform-independent types. The appropriate type definition file *types.h* has to be inserted dependent on the hardware platform.

Consequences:

Pro:

- Reuse of type definitions within the source code on several hardware platforms.
- The size of the type is clearly communicated.
- If considered from the beginning, the overhead is only the definition of the type definition files.

Con:

- Works only if the type length is supported by the compiler. If a type of a certain length cannot be defined by a combination of a type and its qualifiers (e.g. long long long int), the pattern does not work for this type.
- Int, char and other integral types may still be used.
- Encoding meta-information into the type identifier.

Examples: As example we show a naming scheme which is used very often in the C programming language. The type definitions are usually defined in a file called stdint.h (see [2]). Figure 4 shows a state machine which allows the generation of type names for signed and unsigned integers with a different size.

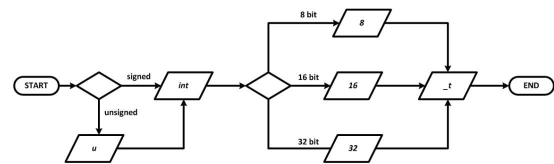


Figure 4: The state machine allows the generation of signed and unsigned integer types with a different number of bits.

Table 1 lists some of the platform-independent types and their corresponding native types. It can be seen that the native types have a different size on different hardware platforms.

Table 1: Platform-independent types and their native counterparts on two different hardware platforms.

	ARM Cortex M3 (32-bit)	unsigned char
uint8_t	unsigned char	unsiged char
uint16_t	unsigned short	unsigned int
uint32_t	unsigned int	unsigned long

4.2 Pattern Name: Generic Pointer Types

Context: Compilers for embedded systems provide the possibility to add information to pointer types which allows optimization. This additional information can indicate the location of the pointer and the pointed object.

Problem: Compiler-specific key-words for optimizing the pointer location thwart portability of the source code. The application of type qualifiers allows many different combinations of optimization parameters. The high number of different combinations confuses programmers.

Forces:

- Optimization parameters for variables which are directly encoded within the source code are a problem for reuse on several hardware platforms.

- Programmers in a huge software project tend to define their own types. Thus, a static source code analysis shall control the usage and conformance of the types. The resulting smaller set of types increases the maintainability of the source code.
- Actually, if we assume 3 possible different locations in an embedded system (RAM/ ROM/ FLASH) this results in many possible combinations of pointer and pointed object locations. The problem here is that too many possible types will definitely confuse the programmers (see also previous force). It makes sense to reduce the number of possible types to a pre-defined selection.

Solution: Mask compiler-specific optimization parameters (qualifiers) by platform-independent key-words. Define a new pointer type and include the necessary information in this definition. The newly defined type shall follow a naming convention that indicates the optimization parameters. The naming convention described in the following. Figure 5 shows a state machine which indicates the construction of a type name. The leading p indicates that the type is a pointer. Then follows: the name of the pointed object type and after this the pointed object alias. The pointed object alias (see also Table 2 and 3) indicates the location of the pointer independently of the hardware platform: small, medium or large.

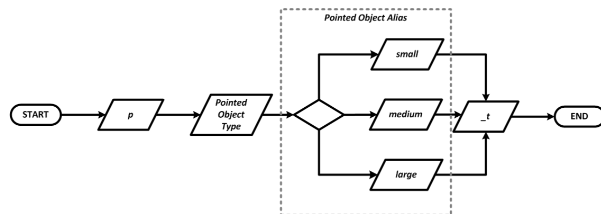


Figure 5: The state machine allows the generation of type names with one optimization parameter.

In the following we will show some possible combinations of pointer parameters. We will select one of these combinations and explain why this is an appropriate choice.

Fixed pointer location: Figure 6 shows addresses and values of the different memory locations of a 8051 processor. The DATA memory section is located in the internal RAM and consists of 256 bytes. Thus, the pointer is only 1 byte long. The XDATA is located in the external RAM and consists of 64K memory. The pointer size is here 2 bytes. The FAR pointer type can access extended memory and can address 3 byte addresses. The example in Figure 6 shows the definition of a pointer type where the pointer is located in the FAR memory section. The location of the pointer is indicated by the FAR key-word after the asterisk. As mentioned before, the pointed object is not specified here and in principle can be located in any of the 3 memory sections. In the example, the FAR pointer has a size of 3 bytes and thus, it can address data in all memory locations. The pointer in the DATA section is only 1 byte long and can address 256 bytes of data. Thus, it can only reference memory in the DATA section.

pointer type	DATA: 1 byte		XDATA: 2 byte		FAR: 3 byte	
	adress	value	adress	value	adress	value
uint8_t * far my_ptr	0x0000	0x000x	0x1000	0x0001	0x2000	0x0001

Figure 6: The pointer location is fixed here and located in the FAR memory region. The location of the pointed object is not specified in the pointer type definition.

Fixed pointed object location: In Figure 7 a pointer type definition is shown, where the pointed object location is specified as DATA. The pointed object location is indicated by the DATA key-word before the asterisk. As can be seen, just the pointed object location (address 0x0001) is fixed. The pointer can be either in the DATA, XDATA or FAR memory region.

pointer type	DATA: 1 byte		XDATA: 2 byte		FAR: 3 byte	
	adress	value	adress	value	adress	value
uint8_t data * my_ptr	0x0000	0x0001	0x1000	0x0001	0x2000	0x0001
	0x0001	0x1010	0x1001		0x2001	

? ?

Figure 7: The pointed object location is fixed here and located in the data memory region. The location of the pointer is not specified in the pointer type definition.

pointer type	DATA: 1 byte	
	adress	value
uint8_t data * data my_ptr	0x0000	0x0001
	0x0001	0x1010

Figure 8: The location of the pointer and the pointed object are specified in the pointer type definition. Both are located in the DATA memory region.

Fixed pointer location and fixed pointed object location: Figure 8 shows a pointer type definition where the pointer location and the pointed object location are specified. The pointer location is specified by the DATA keyword after the asterisk. The pointed object's location is specified by the DATA key-word before the asterisk.

Information encoded in the type:

1. Pointer type: it is the result of (2), (3) and (4)
2. Pointed object type
3. Pointed object location
4. Pointer location

The optimization parameters are:

- Pointed object location
- Pointer location

As a rule of thumb the following guideline makes sense:

- Let the programmer decide the pointed object location. An analysis for optimizing the pointed object location is too difficult for a compiler.
- Let the compiler decide pointer location. The optimization analysis of the compiler here is sufficient.

Thus, as a manually selectable optimization parameter remains the location of the pointed object. Mask this optimization parameter in order to be platform-independent. Masking the pointed object location: the above stated pointed object locations (DATA/ XDATA/ FAR) are compiler specific: only the keil compiler [1] can understand them. Thus, it is necessary to mask these key-words with a type definition. Part of this type definition is the so-called pointed object alias: it is part of the type name and indicates the pointed object location in a platform independent way. Table 2 shows a possible mapping between the pointed object alias and the keil-compiler key-words indicating a memory location.

Table 2: Summary of all lessons learned.

POINTED OBJECT ALIAS	POINTED OBJECT LOCATION ON KEIL
small	DATA
medium	XDATA
large	FAR

Consequences:
Pro:

- Optimization still possible, because a masked parameter can be passed to the compiler.
- Without this pattern, there are so many combinations of pointed object locations and pointer locations to consider. This is confusing for developers, who are not very experienced in embedded systems. The predefined standard types tell the programmers which types shall be used.
- Types can be reused on several hardware platforms

Con:

- The number of possible optimization parameters needs to be limited
- The usage of correct types has to be controlled somehow or no one will keep to them
- Developers need to keep to this types
- Effort to formulate the header files. The definition of a common set of defined types is not straightforward and will be the point of several discussions. It is not easy to achieve an agreement upon types in a large software project.

Examples: In the following paragraph, we show how the pointer types listed in Table 3 can be defined in the C language. The compiler-specific key-words are used only once in a separate header file for each platform. The resulting type name does not contain any compiler specific key-words and is then used in the source code. Thus, for migration to another platform, only the header file has to be changed.

```
//pointed object is in DATA (SMALL)
// pointer anywhere
typedef uint8_t data * puint8_small_t;
typedef uint16_t data * puint16_small_t;
typedef uint32_t data * puint32_small_t;

//pointed object is in XDATA (MEDIUM)
//pointer anywhere
typedef uint8_t xdata * puint8_medium_t;
typedef uint16_t xdata * puint16_medium_t;
typedef uint32_t xdata * puint32_medium_t;
```

Table 3 indicates some possible constructions for a pointer type which is consistent with the naming convention shown in Figure 5.

4.3 Pattern Name: Endianness Abstraction

Context: You are writing software which communicates via a network. You don't know on which hardware platforms my software will be deployed. A typical example for this type of software is an embedded communication protocol stack. The different communicating platforms are likely to have different data presentation in terms of endianness, which defines the order of data in memory. Data is typically stored in bytes. The order of data is then called endianness. There are two basic types: little endianness and big endianness (see Figure 9). In addition to the basic types there exist also some arbitrary types. The pattern can be applied to all types of endianness.

Problem: A problem is communication with other systems. In this case, the byte order matters because the bytes leave the closed system and interact with the outside world where the byte order may be different.

Problem Example: Figure 9 shows an integer value and its memory representations. In a system with little endianness, the first byte has the value 0x78. With big endianness, the first byte has the value 0x12. Apparently, this is a problem for a platform-independent source code. Usually, one would expect the same value for the same expression.

Forces:

- Conversion is not necessary, if the network endianness and all the hardware architectures have the same byte order. In this case, the conversion function adds runtime overhead because it is encapsulated in a function call.
- If the conversion is omitted, the result is a decreased portability of the source code.
- Location of single bits may inherit information, such as error flags of packed signals within 1 communication frame.

Solution: For data which is used also outside of the hardware platform, use standard functions for the conversion of the byte order. Figure 10 shows the communication between two hardware platforms with different byte endianness. The network endianness is per definition big endian. When data

Table 3: Sample type definitions with different *oiinted object locations* for the Keil compiler (see [1]).

Pointer Type	Pointed Object Type	Pointer Locaton	Pointed Object Alias	Pointed Object Location
puint8_small_t	uint8_t	undefined	small	DATA
puint16_small_t	uint16_t	undefined	small	DATA
puint32_small_t	uint32_t	undefined	small	DATA
puint8_medium_t	uint8_t	undefined	medium	XDATA
puint16_medium_t	uint16_t	undefined	medium	XDATA
puint32_medium_t	uint32_t	undefined	medium	XDATA

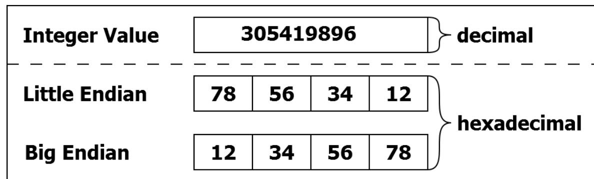


Figure 9: A value can have different memory representations: little endian and big endian.

is sent from Platform A to the Network, the endianness does not change. Thus, the conversion function hton (host to network) simply pipes through the data (see example). The situation is different, when the data arrives at Platform B which has a different endianness than the network. Here the conversion function ntoh (network to host) changes the byte endianness of the data from big endian to little endian.

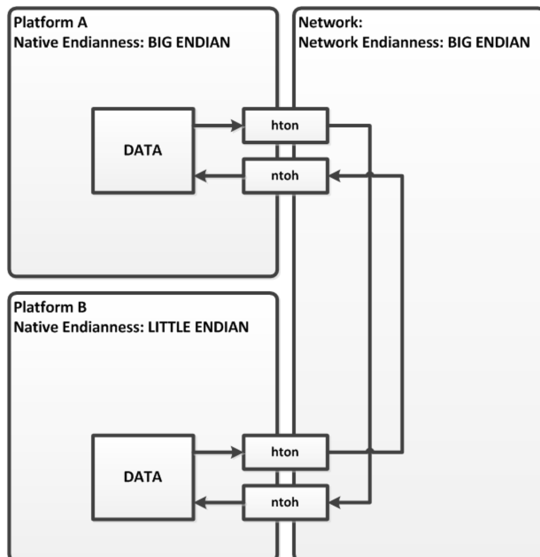


Figure 10: The exchange of data between platforms with different endianness requires the utilization of conversion functions.

Consequences:
Pro:

- This pattern makes communication between platforms with different endianness possible.
- Standard names of the conversion functions can be remembered easily.

Con:

- Runtime overhead because the conversion is encapsulated within a function call (see example).
- Endianness of all included participants needs to be known.
- Conversion needs to be done for each data type.

Examples: In the following we show a source code example which converts the endianness from the host to the network. If the byte order is big endian, the function simply returns the data. If the byte order is little endian, the order of the four bytes is reversed and the returned.

```
001  uint32_t  htonl (x)
002  {
003  #if BYTE_ORDER == BIG_ENDIAN
004      return x;
005  #elif  BYTE_ORDER == LITTLE_ENDIAN
006      return  = (0x000F & x) << 24 |
007                (0x00F0 & x) << 16 |
008                (0x0F00 & x) >> 16 |
009                (0xF000 & x) >> 24;
010  #endif
011  }
```

Known Uses: In Table 4 some standard conversion functions are listed[2]. The implementation of these functions is similar to the above listed example conversion function.

Table 4: Conversion functions for different data sizes.

Conversion function	Conversion direction	Data size
htonl	host to network	32 bit
htons	host to network	16 bit
ntohl	network to host	32 bit
ntohs	network to host	16 bit

4.4 Pattern Name: Structure Alignment

Context: More and more embedded devices are interconnected and data are exchanged between the devices. The problem example we give below shows a struct for a measurement system. Such a system typically incorporates many different interconnected devices with different hardware architectures. Regarding the memory representation of data there are two main points: byte endianness (see also Endianness Abstraction Pattern) and structure alignment. The memory representation of structures is not standardized.

[2]http://lwip.wikia.com/wiki/IPv4

Thus, the compilers strive to optimize the memory representation for runtime performance. Depending on the hardware architecture (8-bit, 16-bit or 32-bit), the members of a struct are aligned to addresses with the power of two (16-bit) or the power of four (32-bit). Most compilers allow the definition of packed structures. Such a structure stores all its members one after another without inserting padding bytes. Unfortunately, the syntax for packing structures is not standardized and is different for each compiler. The application of this pattern can be omitted, if only one specific hardware architecture with equal alignment and byte order is part of the communication network.

Problem: The memory representation of complex types (e.g. structs) depends on the hardware platform. Issues arise when corresponding data is exchanged between platforms with different memory representations of structures.

Problem Example: The following code listing shows how a struct for sensor data could look like: the type indicates the kind of measured value, the *id* ensures an appropriate chronological order. The *num_bytes* define the size of the sensor data.

```
001   struct header
002   {
003       uint8_t type;
004       uint32_t id;
005       uint8_t num_bytes;
006   }
```

The memory representation of the struct may look like shown in Figure 11 on different processors[3].

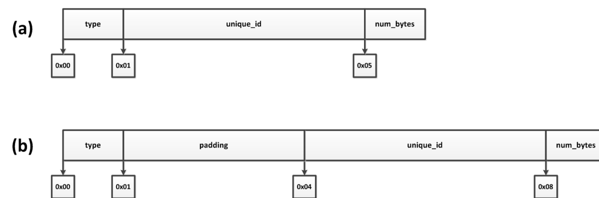


Figure 11: Memory representation of the struct sample_header. A packed representation is shown in (a). Padding bytes are inserted in (b).

As can be seen in Figure 11, a struct can have a different memory representation on different hardware. In Figure 11 (a) the memory representation is packed: the members of the struct are stored one after another in the memory. In Figure 11 (b) alignment is used: this means that uint16_t types are aligned at addresses with the power of two (0x02, 0x04, ...). The type uint32_t is aligned at addresses with the power of four (0x04, 0x08, ...). This alignment is used for optimization purposes.

Forces: If different hardware architectures are involved in communication, the de-facto solution is to pack the structs in order to ensure a consistent data exchange. Unfortunately, the packing of structs comes with a price: the runtime optimization performed by the compiler is lost. The packing has a benefit which may outweigh the lost runtime optimization: the memory footprint is lower because no meaningless padding bytes are inserted. A solution may be feasible without packing but we did not find a related known usage. Thus, such a solution without packing is out of the scope of this pattern.

[3]http://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures

Solution: Force the compiler to pack (see [1]) the structs which are involved in communication. Convert the endianness of each struct member when the struct is transmitted over the network. Most compilers provide an option to pack a struct. This means that no padding bytes are inserted in the memory representation. The syntax of the pack option is compiler-dependent. Thus, the structure definition shall be done in a hardware-specific header file (see Figure 12). Convert each member with the appropriate conversion function (hton or ntoh).

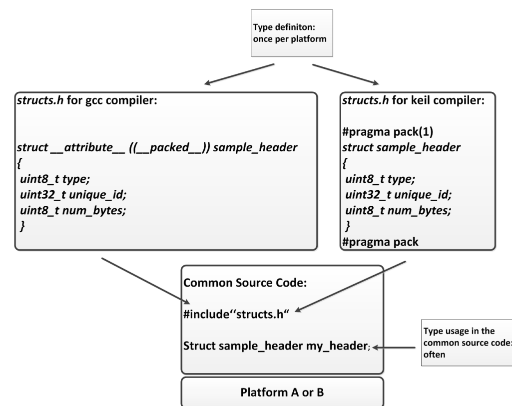


Figure 12: The compiler-dependent structs shall be defined in a platform-specific header file.

Example: This code listing demonstrates the appropriate application of this pattern. The function transmit_header transmits the sample data (see problem example). In line 5 no conversion of the member type is performed, because it has only one byte length. Principally, this line does nothing. It is in this example for a better understandability. In line 8 the member id is converted and stored again. Here the conversion is necessary because this member has a size of four bytes. In line 11 the statement is also just for a better readability of the example. In line 13, the sample is transmitted. It has now network endianness and no padding bytes.

```
01 void transmit_header(header sample)
02  {
03  //no conversion necessary
04  //because only 1 byte
05  sample.type = sample.type;
06
07  //conversion necessary
08  sample.id   = htonl(sample.id);
09
10  //no conversion necessary
11  sample.num_bytes = sample.num_bytes;
12
13  transmit(&sample, sizeof(header));
14  }
```

Consequences:
Pro:

- Smaller memory consumption of the stored struct.
- The transmit function is platform-independent.

Con:

- Works only, if the compiler supports packing of structs.
- The compiler-specific (runtime) optimization of structs is lost.
- The conversion functions ntoh and hton have a runtime overhead.
- The transmit function must be coded for each structure, separately.
- The pattern demands the application of compiler specific key-words for struct packing. However, these specifics are located in a single header file which can be substituted by another, easily.

Known uses: IP-stack implementations in C. See for example: Lightweight TCP/IP stack[4].

5. CONCLUSIONS

In this paper we describe four patterns for hardware abstraction. We mined these patterns in a real industrial software project. We found that the application of these four patterns is sufficient for a proper hardware abstraction. These patterns are tailored for embedded systems and thus, we present examples in the C programming language. Since this language is used in a majority of embedded systems, this focus helps the respective professionals to understand rapidly the application of the patterns. If applied consequently, these patterns support a reuse of software on several hardware platforms.

6. ACKNOWLEDGMENTS

First, I want to thank my shepherd Jari Rauhamäki, who did a great job in all his reviews. Thanks go also to all participants of my writers workshop group. They all did a great job. Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure::systems GmbH for support.

7. REFERENCES

[1] ARM. Cx51 user's guide.
[2] S. Loosemore. The gnu c library reference manual.
[3] T. Rothwell. The gnu c reference manual.

[4]http://git.savannah.gnu.org/cgit/lwip.git

Test-Driven Migration Towards a Hardware-Abstracted Platform

Wolfgang Raschke[1], Massimiliano Zilli[1], Johannes Loinig[2], Reinhold Weiss[1], Christian Steger[1], and Christian Kreiner
[1]*Institute for Technical Informatics, Graz University of Technology, Inffeldgasse 16/I, Graz, Austria*
[2]*Business Unit Identification, NXP Semiconductors Austria GmbH, Gratkorn, Austria*
{*wolfgang.raschke, massimiliano.zilli, rweiss, steger, christian.kreiner*}*@tugraz.at, johannes.loinig@npx.com*

Keywords: Software Reusability, Test-Driven Development

Abstract: Platform-based development is one of the most successful paradigms in software engineering. In embedded systems, the reuse of software on several processor families is often abandoned due to the multitude of compilers, processor architectures and instruction sets. In practice, we experienced that a lack of hardware abstraction leads to non-reusable test cases. We will demonstrate a re-engineering process that follows test-driven development practices which fits perfectly for migration activities. Moreover, we will introduce a process that provides trust for the test cases on a new hardware.

1 INTRODUCTION

Engineering in the field of Smart Card development faces several challenges, such as the demand for a high level of security (Mostowski and Poll, 2008), low memory footprint, power consumption and runtime performance (Rankl and Effing, 2003). All these requirements are interrelated and in fact, the multitude of dependencies hinders Smart Card suppliers and issuers from deploying a great deal of diversified customizable products. Rather, there are only a few standard products available on the market. These do not meet the needs of today's customers who are increasingly demanding tailor-made products.
Principles of platform-based development and Software Product Line Engineering (SPLE) (Pohl et al., 2005)(Clements and Northrop, 2002) are a successful paradigm in software engineering. SPLE aims at systematic reuse where possible and provides a conceptual framework for the diversification of products. In the *domain engineering* process the purpose is not to develop single products but to develop a base of related systems in respect to the product family. Dedicated rules of composition are defined. In the *application engineering* process, the engineers assemble a product out of the product familythat corresponds to these rules of composition.
Products are based on several processor families (PF) which have different implications regarding compilers, byte endianness and architecture. Test cases are not platform-independent *per se*, even if they are written in Junit[1], a Java based unit test framework.
The migration to a product line has to be accomplished during operation: a Smart Card system is under construction on PF A. The plan to transition towards a hardware-abstracted software is to track the development of the Smart Card system on a second PF B as a proof of concept. The way of working is as follows: first, take existing test cases and abstract them from PF A. Second, build confidence for platform-abstracted test cases. Third, use test cases on PF B in order to port the software in a test-driven development process.

2 Requirements

2.1 Requirements in the Industrial Context

It is intended to port as many software components as possible to several PF. Initially, it was deemed appropriate to elicit a set of coding guidelines in order to keep the source code platform-abstracted. Once a pilot project had been conducted to validate our approach, it became apparent that the tests, in particular, create a bottleneck. Porting the source code has worked without much refactoring of the code. Unfortunately, it turned out that most of the

[1]http://junit.org/

© 2015 Scitepress. Reprinted, with permission, from Proceedings of 5th International Conference on Pervasive and Embedded Computing and Communication Systems.

test cases needed to be refactored manually. This was not acceptable for the following reasons: first, the Software Product Line is intended to run on several PF. Industrial embedded systems are usually tested by several thousands of test cases. So, porting test cases manually is not economically feasible in the long run.
Second, manual porting activities are a source of possible defects. The test cases inhibit a high amount of memory dumps. Manually processing this data is error-prone to some degree. Thus, the test cases introduce an additional risk for each migration to a new hardware.
Third, in order to achieve platform-abstraction, we decided to allow *no code change* for different PF, *neither* in source code *nor* in test code. A code change for a specific PF would significantly decrease the reusability of the code.

2.2 Requirements Due to Variability Drivers

The Java Card (Oracle, 2011b)(Oracle, 2011a) operating system under analysis is basically built up as depicted in Figure 1. At the bottom of the system, the variability stems from the utilization of the different PF A or B. Both of them introduce several facets of variability: first, each PF enforces the utilization of a separate tool chain which usually includes compiler, linker and simulator. This facet of variability propagates to the hardware abstraction layer. A considerable portion of it has to be written in assembler and code which is not ANSI-C compliant.
Second, each PF may have a different byte endianness and pointer size. For instance, a 32 bit pointer on processor A corresponds to a 16 bit pointer on processor B. We experienced these factors as the major impediments for test case reuse over several PF. These drivers affect the process in all layers except for the Java Card application layer.

2.3 Platform Lifecycle: Product 1 - Reengineering - Platform, Product 2

The major constraint of the refactoring process is that the industrial product development for PF A may not be disturbed. The continuing industrial development is shown as phase 1 in Figure 2. The pilot study is intended to demonstrate and prove the feasibility of a platform-abstracted SPL. If the study is a success we will transform the development to a platform. In order to fullfill the dedicated requirements, the process is structured as follows:

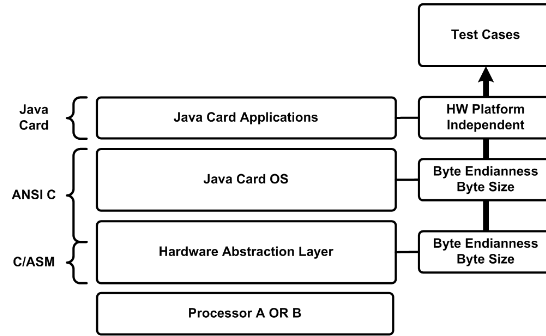


Figure 1: Variability Drivers for Test Cases

Phase 1 is the ongoing industrial product development process which may not be disturbed.
Phase 2 is intended to refactor the tests to be hardware-abstracted.
Phase 3 is a dedicated phase where confidence of the tests on all PF must be demonstrated.
Phase 4 uses the abstracted tests to refactor the code base. Daily test runs keep the feedback cycle accurate for early detection of defects. This will help to mitigate the influence of the hardware abstraction activities on the industrial product development process.

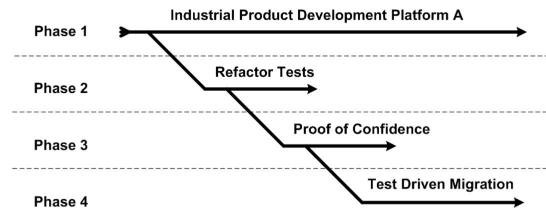


Figure 2: Test Refactoring and Proof of Confidence are the Precondition for Test-Driven Migration

3 Technical Background

The existing test infrastructure is basically split up into two parts: *off-card* and *on-card*. Off-card, Junit is used as a test framework.
Junit launches a test case which then has the responsibility to serialize the test data within a *transmit buffer*. This buffer is then transmitted via packets to the on-card side.

On-card, the In System Test Framework (ISTF) stores the test data within a *receive buffer*. The dispatcher then analyzes the address which denotes the intended caller stub. Then, the *caller stub* is launched. It has access to the receive buffer. The test data has to be de-serialized, which means that it is retrieved from

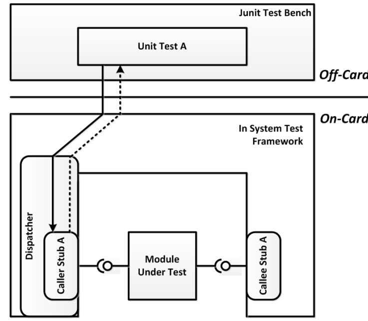


Figure 3: The Test System is Contains Two Parts: The *Junit Test Bench* and the *In System Test Framework (ISTF)*

the buffer and stored in variables. These variables are used as parameters for calling functions of the *Module Under Test* (MUT). The response is collected by the stub module and propagated to the ISTF and then to the Junit test bench. At that point, the test response is evaluated and a corresponding test report is generated. The *callee stub* is not connected directly to the Junit test bench. It substitutes the other modules the MUT usually calls but which are not present in unit testing. This stub has to provide the MUT with the appropriate responses.

4 Method

The method of handling hardware dependencies within tests is similar to that of Model-Based Testing (MBT) (Pretschner and Philipps, 2005). The latter methodology aims to abstract the system to a model in order to generate abstract tests and test specifications. Nevertheless, the goal of MBT is different to Test-Driven Development (TDD). In TDD, tests basically represent pure functional requirements.
A reasonable synthesis between MBT and TDD is to raise the level of *functional* test cases to an abstraction where the following conditions are fullfilled: first, developers are easily able to formulate the test cases without the need of a formal model. Second, the tests need to abstract all hardware specifics that impede portability.

4.1 Abstraction Method

Abstract test cases need to abstract two issues: first, the endianness of the target PF has to be abstracted. This is accomplished by defining big endianness as a rule for implementing abstract test cases. Fortunately, the Java byte endianness is big endian, by definition. Second, the data (usually a memory dump) has to be abstracted in order to be reusable for several PF. Complementing the data with data types is a reasonable abstraction of a binary representation and meets the previously stated requirements.
The basic methodology of test case abstraction is shown in Figure 4. The methodology constitutes of 4 states and 3 transitions.

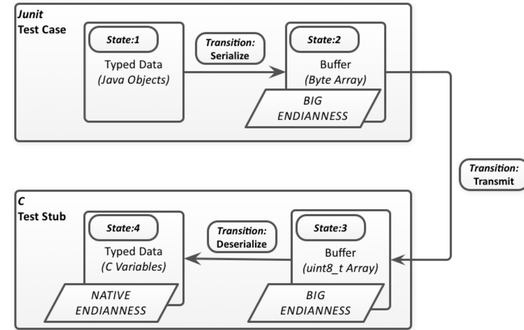


Figure 4: The Method of Handling Tests on Different Processors Contains 4 *States* and 3 *Transitions*

The process starts at *State 1*, where the data is bound to types. These types are Java classes. When they are instantiated by objects they are initialized with the test data. In order to send these objects to the Java Card, they have to be serialized which is shown in Figure 4 as a transition. Afterwards, the data is stored without any type information in a Java Byte Array in *State 2*. The serialized data is organized in the buffer with *big endianness*. In the next transition the data is transmitted from the Junit framework to the Java Card. Thereafter, in *State 3* the data is available in a buffer. Here, the byte order is still *big endian*, regardless of the processor. In the following transition the data has to be de-serialized. Finally, in *State 4* the endianness is resolved and the data is stored in variables. These variables are usually parameters for calling the MUT. The test case is now determined and executable.

4.2 Traditional TDD Process

The traditional *red-green-refactor process* (Beck, 2002) for TDD is shown in Figure 5. Basically, a test is first written which covers a certain functional feature. In the *red* step, this test is executed without the implementation of this feature. This then results in the test failing, which shall be demonstrated in this step. If the test does not fail, it indicates that there is something wrong. Then follows the *green* step where code is written in order to allow the test to pass. If this is achieved, the next phase is *refactoring*. Here

the code is rewritten until it also meets the defined non-functional requirements, such as maintainability, reliability and the like.

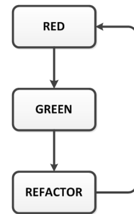


Figure 5: Red-Green-Refactor Process of Test-Driven Development (Beck, 2002)

4.3 Inverted U Process

Due to the high number of test cases, it is not economically feasible to review each and every test case. Thus, we developed a dedicated process (Figure 6) for providing confidence in the abstracted test cases.

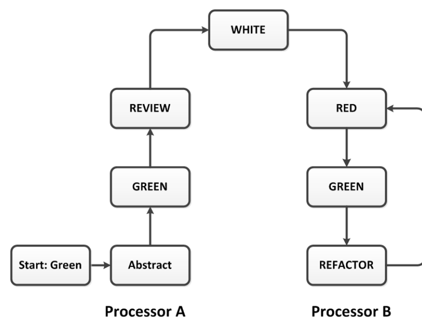


Figure 6: Inverted U Process to Provide Confidence for Abstract Test Cases on PF A *and* B

The inverted U process is used to port the tests and provide confidence in them. Because this process embraces two processors, it has a right and a left branch. The process has a dedicated starting point on the left hand side. A green (passed) test case on processor A is the starting condition. If it has passed once, it can be abstracted, as described previously. After the abstraction, the test case needs to pass again to show that no flaws have been introduced. There is now evidence that the test case is correct on processor A. Still there needs to be assurance that it also works on processor B. Thus, the next step is a *review*, where the abstracted test case is investigated for potential pitfalls, such as the utilization of pointers.
In the *white* step the test case passes on processor A and there is some confidence for it on processor B. Nevertheless, the level of confidence here is not high enough, so the process continues on the right hand side.
Here, the test first enters the *red* phase. Until the test passes, there remains uncertainty about its correctness. So, in the *green* step, the code is ported to run on processor B. If the test did not go green before, consideration is given to investigating and rewriting the test. After the test has turned green for the first time, there is confidence that the test makes sense on processor B. Nevertheless, it should always be checked that the test is compatible with other hardware platforms.
From now on, on the right branch the process is in the *traditional* TDD refactoring loop, as described previously.

5 Implementation

In the following, we will explain the implementation of our method with a sample test case at hand. This test case sets the Java Card program counter *pc* to a certain value. For the explanantion, we will follow the abstraction reference process which we defined in Section 4.1 and in Figure 4.

5.1 Implementation of the Test Case and the Test Stub

The previously mentioned reference process starts in the *Junit* test case implementation which is shown in Figure 7 and continues in the *C* test stub (see Figure 8). The states and transitions of the reference process are indicated within the comments. In the following, we will describe the implementation of the states and transitions.

State 1: In State 1 (see line 3-8 in Figure 7) first, a new object of the type *pointer* is created. In line 5 the pointer address is set to this object. In the next step, the buffer is allocated with the correct length (see line 7).

Transition - Serialize: In line 10-11 the object is serialized and stored within the buffer. Each class which can be serialized has to implement its own *serialize* function.

State 2: After the serialization, the objects data is now stored within the buffer in big endianness. This state is indicated by comments in line 13-15.

Transition - Transmit: In line 18 the *send* function transmits the buffer to the Java Card.

```
01 public void setPC(address addr)
02 {
03 //State 1
04 pointer pc = new pointer();
05 pc.setAddress(addr);
07 ByteBuffer txBuffer = ByteBuffer
08 .allocate(pc.getLength());
09
10 //Transition: Serialize
11 txBuffer.put(pc.serialize());
12
13 //State 2
14 //serialized data are now stored
15 //off-card in the buffer
16
17 //Transition: Transmit
18 send(txBuffer.array());
19 ...
20 }
```

Figure 7: *Junit* Test Case: It Sets the Java Card Program Counter (pc) to a Value

State 3: In State 3 (see line 3-5 in Figure 8), the In System Test Framework has already stored the transmitted data in a buffer, which is located on-card.

Transition - De-Serialize: In line 8 a de-serializion macro is used to retrieve the pointer and store it in a variable. The de-serialization method will be explained in more detail in Section 5.2.

State 4: In State 4, the buffer index *idx* is incremented by the pointer size (line 16). Finally the function of the MUT is called with the currently calculated pointer. The MUT returns a value which can be used to evaluate the test response.

5.2 Implementation of the De-serialization Macro

The de-serialization macro is called in line 8 in Figure 8. For each PF, there is a separate implementation of it. Figure 9 is a variant for an architecture with 3 byte integer pointers and big endianness. First, in line 0, the constant PRT_INT_LEN is set to 3 according to the pointer size. The pointer is reconstructed by shifting and concatenating the bytes with an *or* in the appropriate order. It can be seen that the first byte is not shifted. So, the first byte is the *smallest* which is the case for little endian byte order. The following bytes are then shifted by increments of 8. Finally, the resulting value has to be casted to the relevant pointer.

```
00 static ErrorCode StubJvmSetPc(void)
01 {
02 ...
03 //State 3
04 //serialized data are now stored
05 //on-card in the buffer
06
07 //Transition: De-Serialize
08 pc = GET_INT_PTR
09
10 //State 4
11 //Data are now stored
12 // in the variable pc
13
14 //increment buffer index
15 // by pointer length
16 idx += PTR_INT_LEN;
17
18 // call module function
19 returnCode = setJvmPC(pc);
20 ...
21 }
```

Figure 8: *C* Test Stub: It Sets the Java Card Program Counter (pc) to a Value

```
00 #define  PTR_INT_LEN 3
01
02 //Get Little Endian Pointer
03 #define GET_INT_PTR (uint8_t *)(
04 (uint32_t)(rxBuffer[0 + idx])  |
05 (uint32_t)(rxBuffer[1 + idx]) << 8 |
06 (uint32_t)(rxBuffer[2 + idx]) << 16);
```

Figure 9: De-Serialization Macro for Resolving 3 Byte Pointers From the Buffer

In Figure 10 the same principle is applied for a processor with 4 byte integer pointers and big endianness. The principle is the same but in contrast, in line 0 the PRT_INT_LEN is set to 4. Regarding the reconstruction of the pointer, the lowest byte is shifted by 24 bits. Thus, the first byte is the *highest* which is true for big endian byte order. The next bytes are shifted by increments of -8.

```
00 #define  PTR_INT_LEN 4
01
02 //Get Big Endian Pointer
03 #define GET_INT_PTR (uint8_t *)(
04 (uint32_t)(rxBuffer[0 + idx]) << 24 |
05 (uint32_t)(rxBuffer[1 + idx]) << 16 |
06 (uint32_t)(rxBuffer[2 + idx]) << 8 |
07 (uint32_t)(rxBuffer[3 + idx]));
```

Figure 10: De-Serialization Macro for Resolving 4 Byte Pointers From the Buffer

6 Results

6.1 Identification of Variability Within Junit Tests

When we started the porting of the software, we were not aware that Junit tests incorporate that high a degree of variability. Most of the tests have not been written with portability in mind which, in the beginning, made our approach difficult.

6.2 Initiative Came From a Programmer

The initiative for changing the legacy testing system came from a programmer who was involved in the porting of the software. For those who were involved in these activities it was initially almost impossible to keep up with the porting of the test cases. The guidelines for writing test cases which we developed helped a lot.

6.3 There is a Need for Training

We experienced that the appropriate coding of test cases requires the training of programmers. Otherwise they are not aware of the problems and do not create platform-independent tests. We created a set of guidelines and provided training to the programmers. The resulting awareness mitigated many problems during the porting.

6.4 Low Overhead

The overhead of the methodology is low, if it is adjusted at the start of a project. The overhead is then limited to training programmers and keeping to the coding guidelines. If the methodology is introduced at a later stage, the additional work is higher because all tests have to be refactored and it usually takes some time for people to become familiar with it.

6.5 High Benefit

If the embedded software is used on more than one hardware platform, the benefit of the methodology is high. In embedded systems, there are usually several hundred or thousand tests that could be reused. With the number of supported PF the benefit also increases with comparably little overhead.

7 Related Work

Software Product Line Engineering (Pohl et al., 2005)(Clements and Northrop, 2002) aims at systematically reusing software. For this purpose, a base of reusable software components, the *product family* is maintained. Rules of aggregation are explicitly defined for code and tests.

Platform-based methodologies for embedded systems are given in (Sangiovanni-Vincentelli and Martin, 2001). The authors discuss the specific reuqirements of a reuse strategy for embedded systems, including hardware and software. They provide a vision and a conceptual framework for platform-based software which starts with a high-level system description. This description is then refined incrementally.

TDD is part of agile development practices (Cockburn, 2006) which are lightweight processes that make use of feedback methodologies. Greene (Greene, 2004) gives insight to the application and requirements of agile practices on embedded systems development. He discusses several facets of XP and Scrum and their adoption of embedded systems design. He concludes that there is a positive effect of most of the applied practices.

Grenning (Grenning, 2007) describes special challenges of TDD in embedded systems. These challenges are addressed with the *embedded TDD* cycle that embraces several stages of testing which are applied with different frequency. The five stages range from testing on a workstation to manual testing in the target. This approach has the benefit that the most simple testing approaches are applied most frequently. Finally, Greening discusses issues regarding compiler compatibility and hardware dependencies and possible solutions regarding TDD.

Karlesky et al. (Karlesky et al., 2007) present the so-called *Model-Conductor-Hardware* design pattern in order to facilitate testing in hardware-dependent software. This design pattern is adopted from the Model-View-Presenter (MVP) and the Model-View-Controller (MVC) patterns. Both patterns address issues regarding the development and interaction with Graphical User Interfaces (GUI). A GUI has similar challenges for programming and testing as hardware (event-handling, asynchronous communication and accessibility). Furthermore, a four-tier testing strategy is presented which deals with issues in automation, hardware and communication testing.

In (Bohnet and Meszaros, 2005) a case study of porting software using TDD is presented. The legacy application is a business software which was ported to adapt to a new database system. In order to port the system, the test cases served as a template and

specification for the required functionality. It turned out that the application of TDD resulted in less code on the target platform because unused code was not ported. Moreover, the authors showed that defect test cases are a severe problem because the process suggests searching for problems within the code and not within the tests.

8 Conclusion and Future Work

We challenged the problem of porting a legacy system to a new hardware platform. In order to do this economically, a high number of tests had to be rewritten to be platform-independent. We experienced that this is possible with a relatively low overhead. A major problem of the proposed test-driven migration process is that the correctness of the tests on the new hardware needs to be shown. A dedicated process helps to establish the necesary confidence. The challenges can only be addressed successfully, if the technical realization of the porting parallels the proposed process. Additional training and guidelines for programmers are neccessary.

For further work, it would be interesting to add type-specific information to the serialized data which can be reused during the de-serialization. Going further, a domain-specific language for testing would allow the generation of both, the Junit tests and the C test stubs from one description of a test case.

ACKNOWLEDGEMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

REFERENCES

Beck, K. (2002). *Test-driven Development*. Addison-Wesley Professional.

Bohnet, R. and Meszaros, G. (2005). Test-Driven Porting. In *AGILE*, pages 259–266. IEEE Computer Society.

Clements, P. C. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

Cockburn, A. (2006). *Agile Software Development*. Pearson Education.

Greene, B. (2004). Agile methods applied to embedded firmware development. In *Agile Development Conference*, pages 71–77.

Grenning, J. (2007). Applying test driven development to embedded software. *Instrumentation & Measurement Magazine, IEEE*, 10(6):20–25.

Karlesky, M., Williams, G., Bereza, W., and Fletcher, M. (2007). Mocking the embedded world: Test-driven development, continuous integration, and design patterns. In *Proc. Emb. Systems Conf, CA, USA*.

Mostowski, W. and Poll, E. (2008). Malicious Code on Java Card Smartcards: Attacks and Countermeasures. pages 1–16. Springer.

Oracle (2011a). *Runtime Environment Specification*. Java Card Platform, Version 3.0.4, Classic Edition.

Oracle (2011b). *Virtual Machine Specification*. Java Card Platform, Version 3.0.4, Classic Edition.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

Pretschner, A. and Philipps, J. (2005). 10 Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291. Springer.

Rankl, W. and Effing, W. (2003). *Smart Card Handbook*. John Wiley & Sons, Inc., 3 edition.

Sangiovanni-Vincentelli, A. and Martin, G. (2001). Platform-based design and software design methodology for embedded systems. *Design Test of Computers, IEEE*, 18(6):23–33.

Embedding Research in the Industrial Field: A Case of a Transition to a Software Product Line

Wolfgang Raschke
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
wolfgang.raschke@tugraz.at

Massimiliano Zilli
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
massimiliano.zilli@tugraz.at

Johannes Loinig
NXP Semiconductors Austria GmbH
Gratkorn, Austria
johannes.loinig@nxp.com

Reinhold Weiss
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
rweiss@tugraz.at

Christian Steger
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
steger@tugraz.at

Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
christian.kreiner@tugraz.at

ABSTRACT

Java Cards [4][5] are small resource-constrained embedded systems that have to fulfill rigorous security requirements. Multiple application scenarios demand diverse product performance profiles which are targeted towards markets such as banking applications and mobile applications. In order to tailor the products to the customer's needs we implemented a Software Product Line (SPL). This paper reports on the industrial case of an adoption to a SPL during the development of a highly-secure software system. In order to provide a scientific method which allows the description of research in the field, we apply Action Research (AR). The rationale of AR is to foster the transition of knowledge from a mature research field to practical problems encountered in the daily routine. Thus, AR is capable of providing insights which might be overlooked in a traditional research approach. In this paper we follow the iterative AR process, and report on the successful transfer of knowledge from a research project to a real industrial application.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.13 [**Reusable Software**]: Domain Engineering; K.6.1 [**Project and People Management**]: Systems analysis and design; K.6.1 [**Project and People Management**]: Systems development

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WISE'14, September 16, 2014, Vasteras, Sweden.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3045-9/14/09 ...$15.00.
http://dx.doi.org/10.1145/2647648.2647649 .

Keywords

Knowledge Transfer, Action Research, Software Reuse

1. INTRODUCTION

As the field of Software Product Line Engineering [3][6] is now maturing, it is of growing importance to provide industrial cases. In the industrial context, many challenges exist in the establishment of a SPL which are not obvious to academia. Nevertheless, it is important to provide such an experience to an audience who is interested in establishing a SPL but has no prior real-life experience. For such an audience, it is beneficial to draw on a catalog of documented experiences. In this paper we strive to provide such a catalog.

In a matured research field, when the scientific results are about to be transfered to real applications, it is important to leave the proverbial research lab and apply research in the field. The classical scientific model requests to first state a problem, solve it, then evaluate the solution. In an industrial context, the evaluation of a process improvement is often not bound to a single and consistent problem. Rather, an evaluation is more oriented towards improving an existing solution. Action Research (AR) is an established research method which can be applied in such a context [2][7]. In order to assess the improvement a specific action has to a solution, this research method is cyclic and iterative. At the end of each iteration, the outcome and experiences are digested into lessons learned. In addition we describe how AR helps to elicit field experience in a knowledge transfer project. We show that the applied method can provide different insights to the traditional research approach.

2. RESEARCH METHOD

The more a science field grows and matures, the more it loses relevance for practitioners. This difficulty has been ob-

© 2014 ACM. Reprinted, with permission, from Proceedings of 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering (WISE'14).

served for different fields [7]. Action research is an approach that bridges a highly developed research field and industrial practice. Thusly, as action researchers we are interested in the way a working system was established and how problems have been resolved. The research is accomplished en route [7]:

> "objectives, the problem, and the method of the research must be generated from the process itself, and that the consequences of selected actions cannot be fully known ahead of time".

We are interested in the process of the transition itself, and thus the iterative AR process is beneficial.

2.1 The Process of Action Research

As can be seen in Fig. 1, the AR process is iterative. Each iteration incorporates the following process steps [7]: diagnosing, action planning, action taking, evaluating and specifying learning.

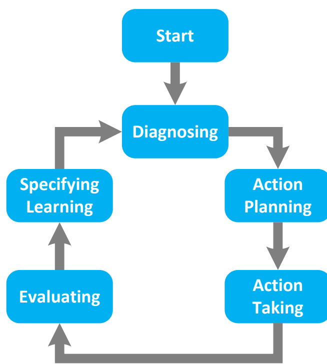


Figure 1: The Action Research process [7] is iterative and encompasses the following steps: *Diagnosing*, *Action Planning*, *Action Taking*, *Evaluating* and *Specifying Learning*. Each iteration starts with the *Diagnosing* process step.

- **Diagnosing:** An iteration starts with a problem. The problem is further elaborated and described. It states the requirements (research question) for the current iteration.
- **Action Planning:** In this phase the possible solutions for the diagnosed problem are investigated. If there are alternative solutions, they need to be compared with each other. In the end, a specific action has to be selected.
- **Action Taking:** Action needs to be taken. In the scope of this report this an increment of a prototype towards a running solution.
- **Evaluation:** The actions are evaluated against the diagnosed problems.
- **Specifying Learning:** Comparing the actions and the evaluation leads to lessons learned (LL). Moreover, reflecting on the lessons learned leads to newly diagnosed industrial problems and research questions.

3. FIRST ITERATION: DESIGN

The first iteration was accomplished mainly by the research partner. The industrial partner was inolved via interviews. In this phase we strived to create an initial big picture of a Software Product Line approach applied to the existing software system.

3.1 Diagnosing

In the first iteration the diagnosing step is a general elicitation of requirements for a Software Product Line. The initial adoption of a SPL is expensive because appropriate tools are needed. Moreover, there is a large amount of effort required for refactoring. Thusly, a decision needs to be taken whether the SPL shall be implemented on an industrial scale or not. For this reason, we started with a research project with the intention of gradually increasing the industrial participation. We can then also regard each iteration of the AR cycle as an evaluation if the transition to a SPL shall be continued. Basically, the problem set by the first iteration is: The SPL approach is usually not known to everyone who is involved in the software development. Before starting a big implementation (which is expensive) the responsible people need to be convinced that the approach works. Moreover, there is no way to alter the software system to the needs of an SPL. It has to be minimally invasive in the sense that the existing software system shall continue working as it is without much refactoring.

3.2 Action Planning

For an initial action plan we listed the most problematic pain points in the design and configuration process:

- **Requirements consistency:** There are many interrelations between requirements (product features and security features). Requirements may demand the inclusion or exclusion of other requirements. Due to the high number of requirements and corresponding dependencies, it is hard to maintain the consistency across the selected requirements of a certain product.
- **Mapping features to source code:** Features and source code are two distinct worlds with different roles involved: product managers, security engineers, test engineers and developers. It is important to bridge these two worlds with a mapping between the high-level features to existing source code and tests.
- **Build configuration:** The build configuration shall be consistent with the selected product and security features. This is only possible, if there is an automatic derivation of build information from the features mentioned.
- **Test selection:** The test selection shall be consistent with the product and security features. Again, the test selection shall be derived automatically from the high-level information.

For providing a feasibility study, we sought to rapidly create a first prototype. The design of this prototype is given in Fig. 2. The design states that the inputs for a configuration are the product and security features. Based on their selection, the appropriate set of components is calculated. Since unit tests and integration tests are bound to components, the component set also determines the set of these tests.

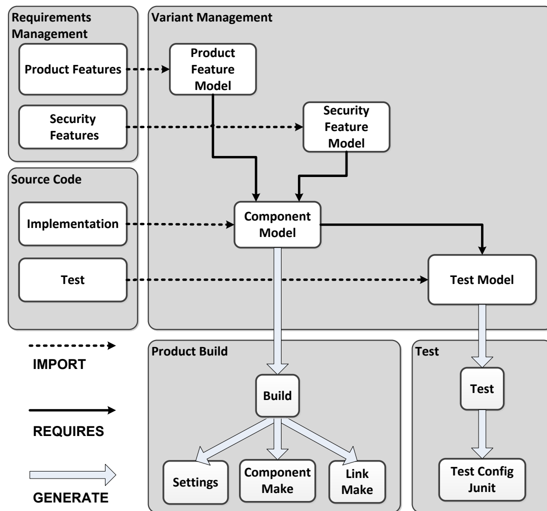


Figure 2: Variability models are imported from a requirements management system and from the source code. The user selects a set of product and security features. The resulting configuration is calculated. As a result, a test and a product build configuration is generated.

3.3 Action Taking

As a proof of concept, we implemented a first prototype of the design. The first prototype is not an appropriate implementation of the industrial problem. The industrial implementation would take lots of expert knowledge for the exact mapping of: features to components, and features to system tests (which has been omitted in the prototype).

3.4 Evaluation

We have modeled four main artifacts: product features, security features [1], components and tests. Thus, it is necessary to demonstrate the approach involving all of them. Moreover, all pain points shall be covered by the evaluation. We designed a use case for this purpose: First, select a specific feature and generate the build and test configuration. Demonstrate that the test passes in this case. Second, deselect a specific feature and generate build and test configuration. Demonstrate that the test is not executed. All the other previously executed tests should deliver the same results as before. Although the use case seems trivial at a first glance, it is in fact a real improvement in an industrial setting and has made considerable impression.

3.5 Specifying Learning

LL1: Don't start with a big implementation. It is not feasible to start with a big implementation for several reasons: First, the SPL approach is usually not commonly known within a company. Before starting a big implementation (which is expensive) the responsible people need to be convinced of the feasibility of the approach. This is one of the most important issues, because a transition of real-world source code is extremely expensive. Moreover, a first prototype can be demonstrated and encourages reflection processes within and in-between individuals and influences the requirements for the SPL. In the following, we provide a short example: Variant management in an industrial context is usually associated with source code configuration. People are not aware that more artifacts come into play, such as documentation and tests. After the demonstration of the prototype, people regarded the test selection and configuration as the most valuable asset of such an SPL.

LL2: It is not apparent where the product features come from. The first apparent source of features are high-level requirements in requirements management systems and tables from product management. Nevertheless, the number of features from such sources are in the range of several hundred. A consistent mapping of so many features to source code artifacts would require too much effort from domain experts. It is hard to justify so much effort for building a prototype solution. Although, it makes sense to address these points later in the transition, there needs to be a smaller feature set.

LL3: We learned also, that the mining of component dependencies is not facile. For the first prototype, we retrieved these dependencies from *include* statements. However, this approach is just an approximation: includes may mask other inclusions of dependencies. Such nested *ifdef* constructs occurred frequently but could be avoided with adhere coding guidelines and appropriate refactoring.

LL4: One of the most surprising lessons learned was that the complexity of the Software Product Line was underestimated. We, as research partners did underestimate the size of the software, the complexity of the configuration and the dynamics of an industrial project. Moreover, the complexity of the problems that come with a SPL were underestimated. This is due to the reason that some mechanisms for variant management were still implemented in the software project. These mechanisms worked fine but with the rising number of configuration switches they grew too complex.

4. SECOND ITERATION: BUILD

The second iteration was accomplished by the research partner and the industrial partner. As a research partner we participated in the daily routine of the industrial software project. Together, we sought to create an applicable solution for a sophisticated build management.

4.1 Diagnosing

The purpose of the second iteration was the improvement of the existing build system for variant management. At the time of diagnosing, variant management was still accomplished to a certain degree. The problems with the existing approach were the following: First, nested *ifdef* constructs were becoming increasingly unreadable. There were no concise coding guidelines for such constructs. Second, the granularity was not consistently defined: sometimes makefiles were split into more files and sometimes not. There was no evident splitting criterion. Third, configuration switches were spread over the code: they exist in makefiles, scripts, xml files, source code and the like.

4.2 Action Planning

The rationale of this iteration is to show that a considerable number of features can be managed systematically. However, the outcome of this iteration shall not only be a feasibility study, but also the development of appropriate tools, guidelines and practices. It is not possible to evaluate these improvements in an active project because the risk of

interruption and the resulting cost is too high. Thus, we decided to implement the build iteration in a small team.

4.3 Action Taking

We started with refactoring the existing makefiles and writing a proposal for the respective makefile coding guidelines. The refactoring of the makefiles was only possible to a very limited degree: each change of a makefile had to be tested for several product configurations. This diligent testing was necessary, because the refactoring was considered risky for the product development. Before the refactored makefile was tested fully, the makefile had been changed by the industrial team and needed adjustments again. Summarizing, a makefile refactoring is only feasible, if the entire team focuses on this objective.

The definition of a common vocabulary was a major issue. A variant management existed beforehand with different vocabulary to that used in academia. This was the source of several misconceptions. Another issue was the development of a naming scheme for compiler flags and constant definitions.

It was important to discuss the configuration of the high-level switches (features). The existing set of several hundred product features was considered as too complex to be linked to source code artifacts. This is not a technical limitation. It is a limitation of complexity. In an industrial context it is nearly impossible to map several hundred features to source code, because the required domain expertise is rare. Thusly, we decided to limit the number of product features to 30. Nevertheless, handling such a number of high-level switches in a complex industrial project is a challenge and deemed appropriate as a starting point.

In order to connect the existing makefiles with the software product line tooling, it was necessary to find the existing switches within makefiles and source code files. During this examination we found that there are several high-level switches available in the makefiles.We planned to use them for the first product feature set because their number did not exceed the mentioned 30. With these rare existing product features it was feasible to build another increment of the variant management prototype and to demonstrate the feasibility of our approach.

4.4 Evaluation

The most relevant achievement was the agreement on the concept of mapping features, makefiles and compiler flags. We decided to start with up to 30 features in order to keep complexity low. In fact, controlling up to 30 features in an industrial environment is a challenge: the knowledge of the mapping between features and source code artifacts is not apparent. It is spread over many developers and domain experts whose time is limited. For this purpose we first mined this knowledge from the source code and makefiles. We have shown the feasibility of the approach when the number of features is up to 30. We managed to agree on a makefile concept that satisfies all parties such as developers, configuration managers and the like. We could elicit a set of coding guidelines for makefiles. Unfortunately, the makefiles could not be refactored in the active project, because of the frequently altering dependencies.

We agreed on a concept of storing all features in a string. This string is then parsed in the makefiles and conditions are evaluated according to the feature set in the string. The makefiles are then responsible for further configuration of the source code.

4.5 Specifying Learning

LL5: Configuration switches were scattered across many different locations. A goal of this transition is to centralize the logic of the configuration. As a rule of thumb, avoiding nested ifdefs is a good coding practice because this is an indicator of there being no hidden configuration knowledge.

LL6: Regarding the number of product features there is a difference between technical feasibility and the actual feasibilty (which is limited by complexity and effort). At the beginning of a SPL establishment it is necessary to demonstrate the feasibility with a reduced feature set.

LL7: The sources of product features are often not so apparent. Initially, we mined them in tables from product management and in Requirement Management Systems (RMS). Later we found, that makefiles and source code are a better starting point for retrieving features.

LL8: Building up a common vocabulary is difficult. Because there were many involved parties (industry, tool vendor, academia), many different phrases were used to denote the same meaning. Since this vocabulary was not consistent, it lead to many misunderstandings. It took a long time and many discussions to build up a vocabulary that was consequently used by all involved parties.

5. THIRD ITERATION: INDUSTRIAL DEVELOPMENT

Before the start of the third iteration much knowledge was transfered to the industrial partner who was then excellently prepared for establishing a SPL. Moreover, a tool vendor was involved in order to tailor the existing tool to the specific needs.

5.1 Diagnosing

The main issues are: build and test configuration on an industrial scale. The approach must be minimally invasive: the tooling must create configuration artifacts which can be used without the tool. This is due to the fact that many software developers work on the project: First, not everyone shall be able to change the variability model. We agreed that it is better to a few well trained developers working on the model and generating configuration artifacts with the tool. These configuration artifacts are then submitted to the version management system. Second, the transition to a product line with dedicated tooling is costly in terms of software licenses and resources. Thus, a possible termination of the transition shall not result in a system which no longer works.

As mentioned above, the makefiles will be controlled by configuration artifacts. As an addition, it is necessary to refactor the makefiles. Such a refactoring has failed in the previous iteration. Therefore, co-ordination needs to be planned more effectively.

Again,we want to address the problem of test configuration. There are actually two dimensions of test case selection: First, tests are selected regarding their functionality, which means that the tests shall cover the functionality defined by the product feature set. The respective tests are unit tests, integration tests and system tests. Second, the tests are also

selected regarding the degree of coverage: smoke tests are very few tests which shall detect possible errors. The full test set strives to cover each line of code. Moreover, the test selection also depends on the test platform: simulator, emulator, real hardware.
Currently, we utilize a test selection mechanism, based on JUnit[1] *pre-conditions*. However, these pre-conditions shall be removed and deployed to the variant model.
At present, there is no single point of test selection. In the underlying industrial context, this is the major argument for the introduction of a variant management tool.

5.2 Action Planning

Initially, there was a phase of requirements engineering and many interactions between the industrial partner and the tool vendor. This phase was intended to gather information and demands regarding the SPL project.
We decided to first start with the build configuration and proceed with the configuration of unit and integration tests. Afterwards, we intend to manage the selection of system tests. For each of these objectives, we defined three different phases: a development phase, a pilot phase and a transfer phase. In the development phase, the software and the tool are developed in order to meet the requirements. It is not used in the active software project. In the pilot phase, the finished solution is evaluated in order to find its flaws and possible corrections. If the solution is regarded as mature enough, it passes to the transfer phase. In the transfer phase the solution is used in the industrial project and developers are trained to apply it.

5.3 Action Taking

The new variant model is similar to the model of the first iteration (see Fig. 2). It contains product features, components and tests. It has been taken over by the industrial partner, for the bigger part. Also, the mapping of unit tests and integration tests to components is similar to the first prototype.
System tests cause the test selection to be problematic: several thousand of them cannot be mapped to components. Thus, they need to be mapped to features. This can only be done by experienced domain experts whose time is limited. Thus, we still strive to find the dependencies in a different way.

5.4 Evaluation

The industrial project was successful, so far. With the limited feature set, the mapping between features and source code is feasible. At the moment, the pilot phase of the solution seems to work as desired. What is missing is experience from the broad transfer of the solution to the daily routine. academic partner's previous work could be reused in the industrial setup. So, it can be concluded that the knowledge transfer was successful to a considerable degree.

5.5 Specifying Learning

LL9: What we especially learned is that such a transition to a SPL can only be successful, if there is one *insider* responsible for it. It is usually much easier for such an insider to get information and support. This is especially the case, when an academic project starts to become industrial.

[1] http://junit.org/

LL10: A very important point is the following: such a project must deliver some early successes to get enough support. For this reason and to minimize risk, the industrial partner fostered a staged transition to a SPL. After each stage the solution is evaluated with the possibility of the project being canceled.

6. CONCLUSION

In Table 1 the lessons learned are summarized. As can be seen we gathered 10 lessons learned in 3 iterations. In the third iteration we could only identify 2 lessons learned: this is due to the fact that this iteration is not yet fully completed. We rated the experience of each lesson learned regarding the degree of technical and field experience. The more the experience is field-related the less it can be reproduced in a traditional research lab setting. Field-related experience may have a technical facet but usually involves more aspects. The relation between pure technical experience and field experience shows to which degree action research can enhance traditional research methods. We rated each dimension on a scale between zero and three plus: No related experience results in a zero rating. A rating of three plus denotes a highly related experience.

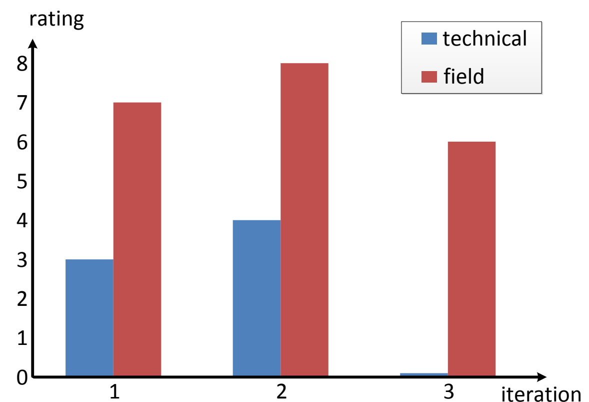


Figure 3: The rated technical and field experience of the lessons learned for all three iterations.

Fig. 3 shows the evolution of the learning experience over the three iterations. In the first iteration, the focus of the SPL transition was on the research partner. Following the rated experience is lower than in the second iteration. There, the research partner and the industrial partner were closely working together on a solution. In the third iteration the rated experience is lower than in the previous iterations. That is because this iteration is not yet fully completed. It is interesting to see that the field experience is always rated higher than the technical experience. This is due to the fact that we were participating in a large and dynamic software project. In such a setting, the research conditions are different to an isolated research project. However, the research method also helps to gather insights which are not possible to elicit with a traditional research method. Such a method would only reveal the same technical experience, in the best case.

Table 1: Summary of all lessons learned.

Lesson Learned	Description	Iteration	Technical Experience	Field Experience
LL 1	don't start big	1		+++
LL 2	unclear source of features	1	+	+
LL 3	mining component dependencies	1	++	+
LL 4	underestimated complexity	1		++
LL 5	avoid nested *ifdefs*	2	++	+
LL 6	limit number of features	2		+++
LL 7	retrieve features from source	2	++	+
LL 8	build up a common vocabulary	2		+++
LL 9	insider responsibility	3		+++
LL 10	early successes	3		+++

In this paper we reported on a successful knowledge transfer from academia to industry. We described this transfer with an established research method which fosters the transfer of science to real and practical applications. This research method has enabled the extraction of several lessons learned which are likely to appear in a similar setting. The lessons learned have demonstrated that the action research approach is capable of providing more insight to field experiences than the traditional research method which focuses solely on problem solving.

7. ACKNOWLEDGMENTS

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

8. REFERENCES

[1] *Common Criteria. Java Card Protection Profile - Open Configuration. Version 3.0 (May 2012).*

[2] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen. Action research. *Communications of the ACM*, 42(1):94–97.

[3] P. C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, 2002.

[4] Oracle. *Runtime Environment Specification.* Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[5] Oracle. *Virtual Machine Specification.* Java Card Platform, Version 3.0.4, Classic Edition, 2011.

[6] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[7] G. Susman and R. Evered. An assessment of the scientific merits of action research. *Administrative science quarterly*, Jan. 1978.

Evaluation paradigm selection according to Common Criteria for an incremental product development

Andreas Daniel Sinnhofer
Institute for Technical Informatics
Graz University of Technology, Austria
a.sinnhofer@tugraz.at

Wolfgang Raschke
Institute for Technical Informatics
Graz University of Technology, Austria
wolfgang.raschke@tugraz.at

Christian Steger
Institute for Technical Informatics
Graz University of Technology, Austria
steger@tugraz.at

Christian Kreiner
Institute for Technical Informatics
Graz University of Technology, Austria
christian.kreiner@tugraz.at

ABSTRACT
Today, agile product development techniques are widely used providing a rapidly and steadily progression of incremental product improvements. Traditionally, a product certification is issued in a late stage of the development process, although some Common Criteria evaluation paradigm would exists to support an agile or modular development process. The usage of such a paradigm would result in a beneficial certification process, since the evaluator gains experience through the maturing product. To provide a systematic way to integrate the evaluation process into the development process — and thus saving money and time — we have identified use case scenarios with the according evaluation paradigm, providing a selection scheme for the right paradigm.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**]: Reusable Software—*Reuse models*

General Terms
Design, Security

Keywords
Common Criteria, Security Evaluation

1. INTRODUCTION
Today, agile product development techniques are widely used providing a rapidly and steadily progression of incremental product improvements, based on common parts and a modular product architecture [7]. This leads principally to a faster time to market and enables the ability to survive and compete in a competitive market. A problem with this flexible and adaptive development paradigm comes up when a certification of the product should be issued, since — traditionally — agile methods are already not used for the development and evaluation process of secure products.
At present, a common approach is to start the certification process of a product in a very late phase of the development, which can result in huge costs when the evaluation facility gives a negative attestation, because a redesigned must be issued. As identified by Boehm [9], the later changes are introduced in the development process, the higher the costs are.
Another problem with such an approach is the long period of time an evaluation process can take, even when the certification of the product is positive. E.g. the certification process of Microsoft Windows 7 took one year and eight months[1]. This can lead to a delayed release if a certificate is a condition for the disposal of a product (e.g. the CE certificate for resale within the EU) or a big gap between the date of release and the date a certificate is issued. Either way, both situations can potentially result in a loss of customers when a competitor is already selling a certified product.
To overcome these drawbacks, Raschke et al. [14] introduced two processes capable for a modular or agile product development, where the certification process is started in parallel. Furthermore he provides a method to automatically detect the actual impact set, so that only those modules are re-evaluated which have an effect to the security assurance of the product. This approach has the key benefit that the evaluator is integrated since the early stages of the process. In fact, the evaluator is gaining experience with the maturing system. Moreover, the feedback of the evaluator can be directly integrated in the next iteration step leading to lower redesign costs [8] [6]. The Common Criteria certification process itself is not further specified, which means that any possible paradigm can be chosen, such as the assurance

[1] see the 14th International Common Criteria Conference (ICCC) `https://www.commoncriteriaportal.org/iccc/ICCC_arc/presentations/T2_D2_2_30pm_Grimm_Evaluating_Windows.pdf`

© 2014 ACM. Reprinted, with permission, from Proceedings of International Workshop on MILS: Architecture and Assurance for Secure Systems (MILS'15).

continuity, a compositional evaluation or a delta evaluation, depending on the current development environment regarding the number of involved developing companies and the number of involved certification facilities.
The contribution of our paper is the identification of the appropriate evaluation scheme for a Common Criteria certification for an agile or modular product development which is applicable in combination with the processes from Raschke et al. [14]. The proposed selection scheme is also applicable for products which are based on previously certified products or modules (e.g. for bug-fix releases).
Section 2 gives a short introduction into the evaluation paradigms according to Common Criteria and the processes identified by Raschke et al. [14]. Section 3 gives an overview over the use case scenarios, providing further information on the according evaluation paradigm and Section 4 summarizes the findings from the use case scenarios in the proposed selection scheme. Finally the results of this paper are summarized and related work is presented.

2. BACKGROUND

2.1 Assurance Continuity

As proposed in Common Criteria Assurance Continuity [1], an evaluation paradigm for the maintenance and re - evaluation of already Common Criteria certified products exists. The flow chart of this approach is illustrated in Fig. 1. It can be seen that based on an impact analysis report (IAR) a decision is made whether the changes to the target of evaluation (TOE) is minor (does not affect the assurance baseline) or major. In the case of minor changes, the previously issued certificate is updated with a maintenance addendum and a maintenance report. The Common Criteria Assurance Continuity [1] states, that *"Maintenance may, in general, continue for up to two years beyond the certification date"*. Due to the fact, that we only consider major changes, the maintenance process is not further contemplated.

In case of major changes, a re-evaluation needs to be performed regarding all affected parts and a new certificate is issued. This can be achieved using an informal modular evaluation scheme (i.e. Delta-Evaluation) through re-evaluation of only the changed and affected modules as stated in the Common Criteria Information Statement on the reuse of evaluation results (see Section 2.2).
The drawback of the assurance continuity approach is, that it is only applicable in those situations, where the evaluation facility is not changed and where a certificate was already issued. Therefore, this approach is intended to be used for bug-fix releases/revisions of old products.

2.2 Delta Evaluation

As stated in the "Common Criteria Information Statement on the reuse of evaluation results" [2] the following evidences must be shared to reuse previously created evidences:

- Product and supporting documentation
- New security target(s)
- Original security target(s)
- Original evaluation technical report(s)

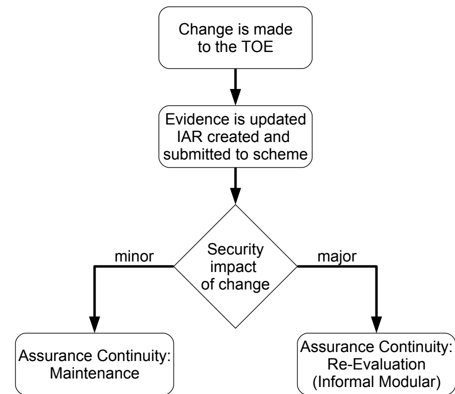


Figure 1: Common Criteria Assurance Continuity flow chart

- Original certification/validation report(s)
- Original Common Criteria certificate(s)
- Original evaluation work packages (if available)

It is specified that

> *"... the evaluation facility conducting the current evaluation should not have to repeat analysis previously conducted where requirements have not changed nor been impacted by changes in other requirements ..."*

where such changes are identified through a so called delta analysis:

> *"... The evaluation facility would be required to perform a delta analysis between the new security target and the original security target(s) to determine the impact of changes on the analysis and evidence from the original evaluation(s) ..."*

which is similar to an impact analysis.
As a result, a product re-evaluation can be performed by an analysis of the impacts of changes and through evaluation of only the changed and affected modules. Unaffected modules need not be reconsidered for the overall evaluation process. Drawback of this approach is that the evaluation technical report is typically generated by the evaluation facility and thus, in some cases, is considered as proprietary to that facility, which makes the interchange of evidences between different certification facilities difficult.

2.3 Composite evaluation

As stated in the Common Criteria Mandatory Technical Document on the composite product evaluation for smart cards and similar devices [3] a composite evaluation can be performed for all kind of products where

> *"... an independently evaluated product is part of a final composite product to be evaluated ..."*

and hence is not limited to smart cards only, but with the limitation that

> *"... The composite product is a product consisting of at least two different parts, whereby one of them represents a single product having already been evaluated and certified ... The underlying platform is the part of the composite product having already been evaluated ..."*

Thus it is applicable for example for an embedded system whereas an application runs on a certified OS, respectively the OS is running on a certified hardware. I. e. a layers pattern is used for the product, whereby trust is established through each layer. The lowest EAL of all components is the limiting factor of the composite product.

2.4 Composed evaluation

As stated in the Common Criteria part 3 (see [4]), the composed evaluation is intended for situations, where independently certified (or going through an independent certification process) products/modules are assembled to a new product which should be certified. It is applicable, where a composite evaluation is not suitable and a delta evaluation cannot be performed due to missing evidences (proprietary documents are not shared). At present, a composed evaluation for higher assurance levels (higher than CAP-C[2] is not supported through the composed scheme and hence a re-evaluation of the whole product is necessary. Due to this, composed evaluations have been performed much less successful than composite evaluations.

2.5 Informal: Identification of the impact set

Due to the fact that it is not necessary to perform unaffected evidences twice, it is meaningful to use change detection analysis to determine the actual affected modules so that only these modules need to be reconsidered in the evaluation. It is important to understand, that modules can interact with each other and hence not only the directly changed module but all other interacting modules need to be reconsidered. This can be achieved through the use of the change impact analysis process proposed by Bohner [10] or the refined processes by Raschke et al. [14]. Our work only mentions the processes proposed by Raschke et al. since he also describes a tool for an automatic change detection analysis, which is well-suited for an partially automatic generation of the Impact Analysis Report (respectively delta analysis), but every other approach is also applicable.
The change detection analysis is based on the so-called *Security Model*, which describes the properties and relationships of the developer evidences, based on the security target, the design documentation, the implementation and the tests (see Figure 2 explanatory graphical representation). Therefore it is applicable to trace and detect all dependencies between each module.

[2]Attack potential "Enhanced Basic"; approximately comparable with EAL-4 (see[4] pages 38 and 47)

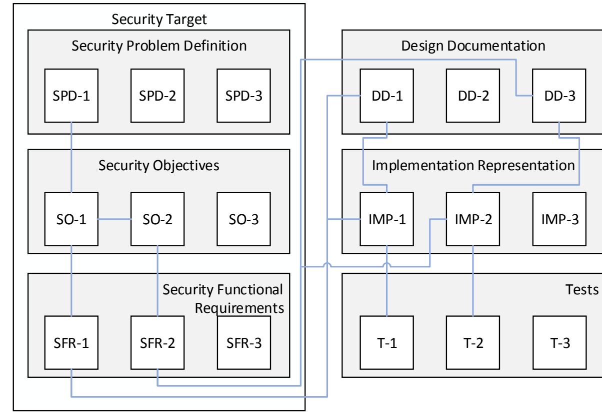


Figure 2: Explanatory Security Model, showing some exemplary artefacts and traces

3. PROPOSED USE CASES AND ACCORDING EVALUATION PARADIGM

Overall situation: Aforementioned, we consider an agile or modular product development process, where the (final) certified product is assembled using a number of modules. In each development iteration new modules can be added or old modules can be changed or removed. Various companies can be involved in the development process of the product and any number of evaluation facilities can be integrated in the certification process.
The selection scheme is applicable for the following scenarios:

- *Use case 1:* One company develops a number of modules which are all evaluated at the same evaluation facility. Since the evaluation facility has full access to all modules and all related evidences, an evaluation can be achieved by a simple informal modular evaluation. If during the development process the evaluation facility is changed, a formal modular paradigm would need to be chosen.
- *Use case 2:* One company develops a number of modules, whereas a number of evaluation facilities ($n > 1$) are involved in the certification process, interchanging all kind of evidences. Therefore, a delta evaluation can be issued.
- *Use case 3:* One company develops a number of modules, whereas a number of evaluation facilities ($n > 1$) are involved in the certification process, but unfortunately they do not interchange evidences. Depending on the architecture of the developed product a composite (Use case 3.a) evaluation or an composed (Use case 3.b) evaluation can be issued.
- *Use case 4:* Several companies are involved in the development process of the product, but one central evaluation facility is used. In this scenario an informal modular evaluation can be used since the certification facility has direct access to every contribution of every

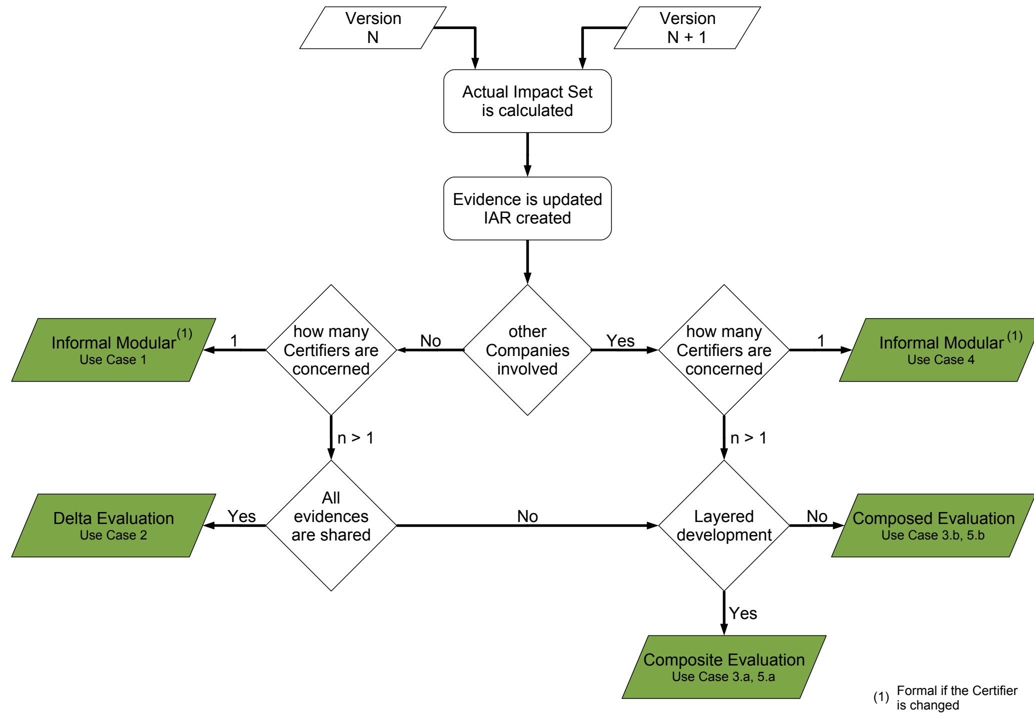


Figure 3: Proposed paradigm selection scheme

company. If during the development process the certification facility is changed, a formal modular scheme would need to be chosen.

- *Use case 5:* Several companies are involved in the development process of the product and any number of evaluation facilities ($n > 1$) are included into the certification process (e.g. each company consults a different evaluation facility). A delta evaluation is possible if the different evaluation facilities interchange all kind of evidences, which can be a problem since the evaluation facilities would need to provide information on their evaluation process and their used methods. In practice, a composite (Use case 5.a) or composed (Use case 5.b) evaluation scheme is used, depending on the used architecture.

4. PARADIGM SELECTION SCHEME

Based on the activities during the assurance continuity process [1], a selection scheme for the presented use cases was created. The first steps towards the reuse of any evidence is the analysis of the impacts on the assurance of the current Target of Evaluation which is intended to be done by one of the processes proposed by Raschke et al. [14]. The selection scheme is split-up into two main leafs, where one is applicable if a product is developed from a single company and the other one for a product which is developed from many companies. As identified in the use cases, another factor which must be considered is the number of certification facilities and the fact if these certification facilities do interchange all needed evidences so that the evaluation results can be reused efficiently. Another criterion which needs to be reconsidered is derived from the composed evaluation scheme, whereby the developed product is structured in a layered approach. The lowest layer must be already certified.

Generally spoken an informal approach can be used if certification facilities do interchange evidences or a single certification facility is issuing the product evaluation and formal approaches must be chosen in all other cases which are usually more time and money intensive.

The next enumeration provides a short description of the according paradigms:

- *Informal Modular:* The certification facility has full access to all modules and evidences, therefore only the affected modules are re-evaluated (Delta Evaluation).
- *Formal Modular:* The certification facility was changed and hence, all modules need to be reconsidered in the evaluation process. Previously created evidences (e.g. certified modules) can be reused, if all needed information is available.
- *Composed Evaluation:* This evaluation is based on the Composed Assurance Package (CAP) of the Common Criteria part 3 (see Section 2.4). Drawback is that the highest achievable CAP level is CAP-C, which is comparable to EAL-4. Higher levels of assurance are only possible through a complete re-evaluation of the assembled product.
- *Composite Evaluation:* This evaluation paradigm is based on a layered product development, where trust is gained through the combination of all layers. In difference to the composed evaluation, the composite product is the final product for which an EAL level

certification is issued. This allows a direct comparison with similar products certified after a single evaluation. [3]

- *Delta Evaluation:* This is the delta evaluation as described in Section 2.2. A concrete process for the certification is not provided through the Common Criteria standard and thus the according certification facility needs to be consulted.

5. RELATED WORK

Klohs [12] provides observations and thoughts on the modularisation concepts for the development of a smart card operating system according to Common Criteria. He points out that the JIL document [5] on the security architecture requirements for smart cards and similar devices, establishes a first starting point for the reuse of software components, based on a description of the security interface and the implemented security mechanism which is implemented from the component independent of a concrete security target.

The Assert4SOA[3] project focuses on the development of methods for the certification of service oriented architectures (SOAs), reusing existing certification processes to overcome the challenging tasks for an evolving software ecosystem. The project itself does not focus on the Common Criteria scheme, but provides a guidance to integrate the Common Criteria certification scheme into a service oriented architecture in [13].

The Euro-MILS[4] project focuses on providing a framework for trustworthiness by design and high assurance based on *Multiple Independent Levels of Security (MILS)* [11]. In fact, assurance of the whole product is gained through the composition of assurance arguments of its components and the system's security architecture. The developed framework is based on the Common Criteria evaluation schemes.

6. CONCLUSION

Today's industry is embossed through fast changing requirements regarding functional and security needs. These circumstances are tried to be solved through the usage of agile or incremental manufacturing techniques. We have identified a scheme for the selection of the appropriate evaluation paradigm to support an agile or modular development processes regarding the security certification to reduce the time shift between the successful certification and the time the product development finished. Furthermore the costs for re-evaluating the developed product/modules can be kept as low as possible since the most suitable paradigm is chosen, maximizing the reuse of already evaluated modules and providing a direct integration of the evaluation facility in the process so that the feedback is directly integrated into the next development iteration.

7. ACKNOWLEDGEMENT

Project partners are NXP Semiconductor Austria GmbH and the Technical University of Graz. The project is funded by the Austrian Research Promotion Agency (FFG).

[3]www.assert4soa.eu

[4]http://www.euromils.eu

8. REFERENCES

[1] Common Criteria. Assurance Continuity CCRA Requirements. Version 2.1 (June 2012).

[2] Common Criteria Information Statement. Reuse of Evaluation Results and Evidence. (October 2002).

[3] Common Criteria Supporting Document Mandatory Technical Document - Composite product evaluation for Smart Cards and similar devices. Version 1.2 (April 2012).

[4] Common Criteria for Information Technology Security Evaluation. Part 3 Security assurance compnents. Version 3.1 Revision 4 (September 2012).

[5] Common Criteria Supporting Document Guidance - Security Architecture requirements (ADV_ARC) for smart cards and similar devices. Version 2.0 (April 2012).

[6] S. Ambler. *The Object Primer: Agile Model-Driven Development with UML 2.0 - Third Edition.* Cambridge University Press, 2004.

[7] D. Anderson. *Agile Product Development for Mass Customization: How to Develop and Deliver Products for Mass Customization, Niche Markets, Jit, Build-To-Order and Flexible Manufacturing.* Irwin Professional Pub., 1997.

[8] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition).* Addison-Wesley Professional, 2004.

[9] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[10] S. A. Bohner. Extending software change impact analysis into cots components. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)*, SEW '02, pages 175–, Washington, DC, USA, 2002. IEEE Computer Society.

[11] H. Blasum, S. Tverdyshev, B. Langenstein, J. Maebe, B. De Sutter, B. Leconte, B. Triquet, K. Müller, M. Paulitsch, A. Söding- Freiherr von Blomberg, A. Tillequin. Secure European Virtualisation for Trustworthy Applications in Critical Domains - MILS Architecture, 2014.

[12] D. K. Klohs. Software modularisation and the common criteria - a smartcard developer's perspective.

[13] M. B. Samuel Paul Kaluvuri and Y. Roudier. Bringing common criteria certification to web services.

[14] W. Raschke, M. Zilli, P. Baumgartner, J. Loinig, C. Steger and C. Kreiner. Supporting evolving security models for an agile security evaluation, 2014.

Supporting Evolving Security Models for an Agile Security Evaluation

Wolfgang Raschke*, Massimiliano Zilli*, Philip Baumgartner†
Johannes Loinig†
Christian Steger*, and Christian Kreiner*

*Institute for Technical Informatics, Graz University of Technology, Graz, Austria
{wolfgang.raschke, massimiliano.zilli, steger, christian.kreiner}@tugraz.com
†NXP Semiconductors Austria GmhH, Gratkorn, Austria
{philip.baumgartner, johannes.loinig}@nxp.com

***Abstract*—At present, security-related engineering usually requires a big up-front design (BUFD) regarding security requirements and security design. In addition to the BUFD, at the end of the development, a security evaluation process can take up to several months. In today's volatile markets customers want to influence the software design during the development process. Agile processes have proven to support these demands. Nevertheless, there is a clash with traditional security design and evaluation processes. In this paper, we propose an agile security evaluation method for the Common Criteria standard. This method is complemented by an implementation of a change detection analysis for model-based security requirements. This system facilitates the agile security evaluation process to a high degree.**

I. Introduction

Traditional security engineering requires a big up-front design (BUFD) which includes the following engineering tasks: threat analysis, security requirements elicitation, security design and (security) architecture. Reviews ensure the consistency of these artifacts. In highly secure systems, the consistency of the security requirements and the security architecture is formally verified. Altogether, this constitutes a huge effort, before the implementation is even started upon. After the security design, the system is implemented according to requirements and design. Thereafter, evidence for security has to be provided in the form of documentation. This documentation is then delivered to the evaluation facility. The evaluation may take several months. If the feedback from the evaluator is negative, the system has to be re-designed. If this is the case, then a large amount of time and effort would be required. Moreover, the pressure to deliver the product to market is high. Summarizing, the challenges for security engineering are: early validation and time-to-market.
The software industry came up with the trend of agile development practices (see [2]), such as XP, Scrum, and Test-driven development. Agile methods focus on customer interaction and incremental software development. Feedback loops, such as customer on-site, pair programming and refactoring tend to minimize errors. Generally, agile processes react flexibly to changing customer requirements. In contrast, traditional processes tend to fulfill contractually specified requirements and are inflexible, by nature. Agile processes have demonstrated a high success rate in several software projects [3].
Unfortunately, agile methods are not well studied for high-level security engineering and evaluation. Case studies and success stories are lacking, especially for the Common Criteria standard [1]. Despite the fact that traditional security engineering seems to counter agile practices (see [4]), we think that a synthesis of both is not contradictory, if well considered.
Our contribution is the analysis of two processes which are possibly suitable for an agile security evaluation. Both processes are compared and one of them is then proposed for an agile security evaluation. In an agile evaluation process, all security properties of the system are kept in a model which is capable of creating documentation for an evaluation round. In the agile evaluation process only the increment of the model has to be reviewed. This has several benefits: the entire system does not have to be reviewed in each iteration. The evaluator starts with a small system and gains experience with the maturing of the system. Early feedback enables an early and thus more inexpensive correction of the system. With elaborated algorithms we are able to create a difference model for all changes during an increment. Additionally, we demonstrate the feasibility of automation by customizing existing open source software.

II. Background

A. Requirements for the Security-Relevant Agile Development Process

We developed this method mainly in order to gain improvements regarding time-to-market and early validation. In addition we strive to improve the semi-automation of the document creation with the model-based approach.

1) Time-to-Market: is one of the key success factors in the high-security business. In a traditional certification, the evaluation is accomplished after the implementation is finished. The time-to-market can significantly be reduced, if the evaluation parallels the development.

© 2014 IEEE. Reprinted, with permission, from Proceedings of 2014 Evolving Security and Privacy Requirements Engineering (ESPRE'14).

2) Early Validation: In the traditional certification approach, the evaluation starts late, when the product is already finished. If there is a negative evaluation result, the refactoring is expensive and seriously delays the completion of the product.

3) Semi Automation: The agile way of working states that communication shall be over documentation. Frequent interaction with the customer and building and adapting quickly a running system is imperative. Regarding the mentioned issues of a quick and flexible development, a documentation effort is counterproductive. In principle, the documentation could start right after the implementation: one would not have to care about documentation during the agile development but the benefit of time-to-market is lost. In this case the documentation and certification may last several months. In a highly competitive business this is a considerable delay. Moreover, also the advantage of early validation is lost.
It can be seen that the described security-relevant agile development process is a trade-off. However, much of the documentation effort can be automated: the model can be synchronized with the source code, the tests, and other relevant artifacts. A considerable degree of the documentation can then be created by a code-generator from the model.

B. Change Detection Analysis

A Change Detection Analysis (CDA) computes all changes between two versions. The CDA is more commonly known under the name *diff* analyis: diff[1] is a software that compares two versions of a text and marks the differences between them. It is a standard tool for software versioning and configuration management. The CDA for models is similar but does not make a line-based comparison of the text. Such a line-based comparison between two models is not sufficient because it does not take into account the structure of a model. In contrast, the CDA recognizes the type of the changed artifact. For example, the CDA recognizes, if and interface has changed in the source code. In this case, more related artifacts need to be reviewed: the regarding design, the tests and the like. If a change is not part of an interface, much less artifacts need to be taken into account. Textual diff tools do not have these capabilities. Basically, the CDA detects three different kinds of changes [5]:

- **Add:** Add a model element. A model element is a node in a model. It may have parent and child nodes.
- **Delete:** Delete an element.
- **Update:** Change of an element property. A property (attribute) belongs to an element.

C. Traceability Impact Analysis

The Traceability Impact Analysis (TIA) is an extension of the CDA. It takes all changed elements and finds all elements which are possibly affected by them. For this purpose, the TIA has to follow all dependencies (traces) of the changed elements. Thus, an element can be directly affected (by a changed element) or indirectly via several intermediate dependencies.

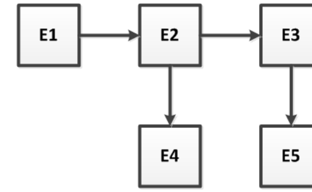


Fig. 1. Sample dependency graph. E4 is impacted directly by E2 and intermediatly by E1 (via E2).

1) Dependency Matrix: In order to perform a TIA, the dependency matrix has to be constructed. In (1) the matrix *D* represents the dependency graph shown in Fig. 1. In the matrix, each line lists the depending elements of a certain element. So, line 1 states that element E2 depends on element E1.

$$D = \begin{array}{c} \text{Element} \\ E1 \\ E2 \\ E3 \\ E4 \\ E5 \end{array} \begin{array}{c} \begin{array}{ccccc} E1 & E2 & E3 & E4 & E5 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array} \quad (1)$$

2) Reachability Matrix: The reachability matrix (2) can be computed from the dependency matrix. It states all direct and indirect dependencies between two elements. The reachability matrix can be used to calculate all impacts of a changed element.

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \quad (2)$$

For example, in (3), the reachability matrix is multiplied with a vector that indicates that element E2 has changed (the *1* in the second row). The outcome of the multiplication is the vector *IS* that lists all impacted elements. So, if element E2 changes, element E3, E4 and E5 are impacted.

$$IS = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (3)$$

[1]http://www.gnu.org/software/diffutils/

D. Delta Evaluation

The Common Criteria information statement on *Reuse of Evaluation Results and Evidence* [6] requires for a reuse of evaluation results the following evidence:

- Product and supporting documentation
- New Security Target
- Original Security Target
- Original Evaluation Technical Report
- Original Common Criteria Certificate
- Original Evaluation Work Packages

The document states:

> "The evaluation facility would be required to perform a delta analysis between the new security target and the original security target to determine the impact of changes on the analysis and evidence from the original evaluations."

and:

> "However, the evaluation facility conducting the current evaluation should not have to repeat analysis previously conducted where requirements have not changed nor been impacted by changes in other requirements."

Summarizing, this approach states the necessary documentation and the need for a Change Impact Analysis. A more detailed process for this evaluation is not given and has to be defined together with the evaluation facility.

E. Security Model

The security model describes the properties and relations of the developer evidence. Basically, the developer evidence contains the *Security Target* (ST), the design documentation, the implementation representation and tests. Such a model for a Common Criteria evaluation is very complex due to the very hierarchical composition of the Common Criteria artifacts. Therefore, we describe only the basic properties of such a model which are necessary for understanding the proposed evaluation approach (see Fig. 2).

The Security Target is the security claim and describes the operating context and how the TOE establishes its security. The ST is published and thus contains no detailed description of the design and implementation. The model of the ST contains the following main parts:

- *The Security Problem Definition* describes the environment of the TOE: threats, assumptions about its operation and its security relevant assets.
- *The Security Objectives* are high-level security goals which are then refined by the Security Functional Requirements.
- *The Security Functional Requirements* are a set of well understood requirements in a security domain and have to be fulfilled by the TOE.

The remaining developer evidence basically contains the design documentation, the implementation representation and

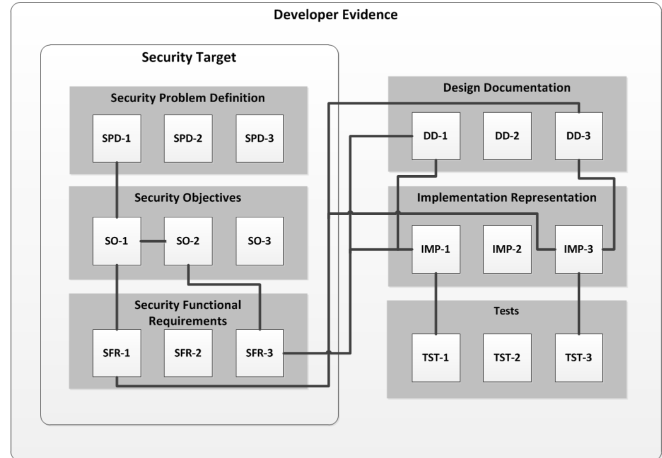


Fig. 2. Artifacts of the security model and traces between these artifacts.

tests. The evaluation has to prove that the TOE fulfills the security claim of the ST. For this reason, mainly the following developer evidence is necessary:

- *The Design Documentation* contains the functional specification and the design specification.
- *The Implementation Representation* is the source code of the implementation.
- *Tests:* This part contains of the source code of the tests and also a corresponding test documentation.

III. Process Design

In this section we will explain and compare two possible processes for a security evaluation. The first one is more formal and suits a delta certification approach. The second is more informal and more appropriate for an agile security evaluation. What is the major difference between a delta and an agile evaluation process? The main difference is the frequency of the evaluation: the delta evaluation is based on a previous evaluation result which has been ascertained some time ago. Basically two product versions are compared. In the agile approach the evaluator becomes update documentation in each agile iteration (which lasts several weeks). Nevertheless, the certificate is only ascertained once, at the end of the product development lifecycle. The following processes are based on the change impact analysis process described by Bohner [7].

A. Process with Traceability Impact Analysis

In the following, we will describe the TIA process with an example. The resulting sets are listed in Table I. The graph in Fig. 1 shows the structure of the two versions N and N+1. The structure of the model does not alter during this increment. For the example, we assume, that the content of the node E1 has changed. The CDA detects all model differences between both versions. The output is the Starting Impact Set (SIA). In this case the SIA is the changed element E1: SIA = $\{E1\}$.

The TIA resolves then all traces and dependencies of the SIA and computes the so-called Candidate Impact Set (CIS). The CIS is the set of all possibly impacted artifacts. In this

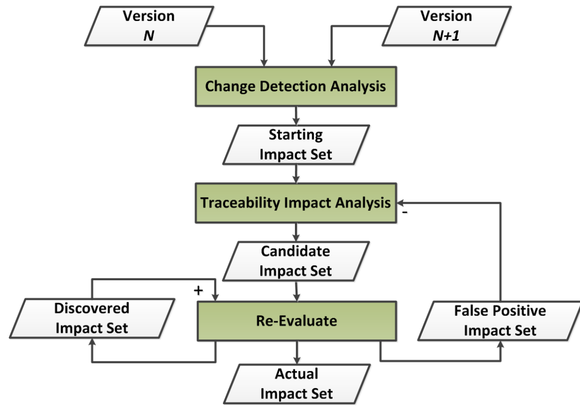


Fig. 3. Re-evaluation process with Traceability Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts.

case, the CIS is: CIS = {E1,E2,E3,E4,E5}. Nevertheless, the CIS has to be manually re-evaluated:
(a) If the candidate has a possible impact, the corresponding evaluation artifacts need to be updated. The set of all re-evaluated artifacts form the Actual Impact Set (AIS). The Actual Impact Set in this example is: AIS ={E1,E2}.
(b) During the re-evaluation, new dependencies may be found: the Discovered Impact Set. Also the newly discovered impacts (DIS) have to be re-evaluated, again. In this example, the DIS is empty.
(c) If the candidate has no impact, it is part of the False Positive Impact Set (FPIS). In this case, a further security evaluation of the corresponding artifacts can be omitted. The FPIS in this example is: FPIS = {E3,E4,E5}.
Basically, the TIA predicts the AIS. This prediction is of course partly wrong, because new impacts are discovered (DIS) and some predicted impacts actually have no impact (FPIS). In (4) the relationships between the mentioned sets are stated:

$$AIS = CIS + DIS - FPIS \tag{4}$$

TABLE I
SAMPLE SETS FOR THE PROCESS WITH TRACEABILITY IMPACT ANALYSIS. THE CORRESPONDING EXAMPLE IS DESCRIBED IN SECTION III-A.

	Set	**Output Of**
Starting Impact Set	{E1}	CDA
Candidate Impact Set	{E1,E2,E3,E4,E5}	TIA
Discovered Impact Set	{}	re-evaluation
False Positive Impact Set	{E3,E4,E5}	re-evaluation
Actual Impact Set	{E1,E2}	re-evaluation

B. *Process with Experiential Impact Analysis*

We will describe the EIA process with an example. The resulting sets are listed in Table II. The graph in Fig. 1 shows the structure of the two versions N and N+1. As shown in Fig. 4, the process with Experiential Impact Analysis (EIA) takes as input two versions (N, N+1) of a model.

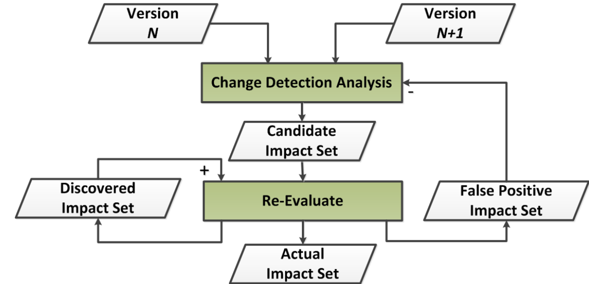


Fig. 4. Re-evaluation process with Experiential Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts.

The CDA detects all model differences between both versions. In the example, the structure of the model between both version remained unchanged. Only the the content of node E1 has changed. The resulting Candidate Impact Set (CIS) is then: CIS = {E1}. As can be seen, in this process the TIA is omitted: the re-evaluation is accomplished directly after the CDA. This leads to two consequences: first, the number of false positives is much lower (compare Table I and Table II), because the CIS is smaller without a TIA, which also takes into account intermediate dependencies. Second, the DIS is larger because more impacts need to be detected during the re-evaluation (compare Table I and Table II). In this example, the manually detected impacts are: DIS = {E1,E2}. This evaluation relies on Experiential Impact Analysis (EIA) which has been described by Kilpinen [8]. The EIA is based on expert design knowledge and review techniques: code inspections and walkthroughs. In any case, these techniques have to be performed during a security evaluation [1].

TABLE II
SAMPLE SETS FOR THE PROCESS WITH EXPERIENTIAL IMPACT ANALYSIS. THE CORRESPONDING EXAMPLE IS DESCRIBED IN SECTION III-B.

	Set	**Output Of**
Candidate Impact Set	{E1}	CDA
Discovered Impact Set	{E2}	re-evaluation
False Positive Impact Set	{}	re-evaluation
Actual Impact Set	{E1,E2}	re-evaluation

It can be argued that the effort for EIA is expensive because it has to be conducted manually. Thus, a major issue in this process is to keep the cost of detecting the DIS low. This cost can be kept low, if the evaluators are involved early and

regularly. Thus, they can build up knowledge of the possible impact between software artifacts. This ensures that the EIA can be accomplished efficiently if each evaluator works on the same but narrow set of software modules.

C. Comparison of the Processes

It is an important observation here that the DIS and the FPIS are both discovered manually, whereas the CIS can be detected automatically with tools. As shown in (5) the total effort of both processes calculates:

$$\mathit{Effort}_{total} = \mathit{Effort}(DIS) + \mathit{Effort}(FPIS) \quad (5)$$

As can be seen, the number of DIS and FPIS are a major factor for selecting an appropriate process. Nevertheless, there are more parameters which influence such a selection. The most important factors are listed in Table III.

TABLE III
COMPARISON OF THE PROCESSES WITH TRACEABILITY IMPACT ANALYSIS AND EXPERIENTIAL IMPACT ANALYSIS.

	TIA Process	**EIA Process**
DIS	SMALL	LARGE
FPIS	LARGE	SMALL
Presumed Experience	LOW	HIGH
Iteration Cycle	Months/Years	Weeks

The process with Traceability Impact Analysis (TIA) calculates a large Candidate Impact Set because the number of traces and dependencies is usually high (see Table I). Thus, the number of direct plus indirect possible impacts soars. On the one hand, a large CIS reduces the likelihood of manually detected impacts because all direct and indirect traces have still been resolved by the TIA (see Table I). On the other hand, a large CIS will contain many false positives (see Table I and also [7]) which need to be detected manually. For human beings it is easier to falsify a visible impact than to detect an invisible impact, if no experience exists. For long time intervals between two iterations, usually no experience of the software under evaluation can be presumed. Thus, a process which detects many candidate impacts is more appropriate in this case.
The process with EIA detects more impacts via the re-evaluation process (DIS, see Table II). Fewer false positives are detected, because the number of candidate impacts is smaller without a TIA (see Table II). This process fits better for an iterative evaluation with short intervals between the re-evaluation activities. Its strength is to build up knowledge of the discovered impact set which makes the re-evaluation efficient. Although there are several iterations of re-evaluation, only one certificate is issued, in the end. In the following we will show a tooling that can assist a process with EIA.

IV. Tool Support for Automation of the Change Detection Analysis

In order to support an agile evaluation process with tooling we implemented a Change Detection Analysis. The CDA for the security model takes as input two versions and delivers a list of all differences. For example, the CDA lists all changed source code and test entities. This has the benefit that the unchanged parts do not have to be taken into account for a re-evaluation. Basically, such a CDA for models is not new. Our contribution here is to show how such a tool can be built with existing open source software. Moreover, we faced some issues because we work with large models. These issues suggest a careful design of the model structure: each node within a model shall be tagged with a unique identifier.

A. Basic Requirements for the Tooling

First of all, the *diff engine* shall work as a standalone and independently of other proprietary tools. Unfortunately, many modeling tools do not support sophisticated model-diff engines, at present. Such a diff engine shall accept a standard format as input. The produced a diff report shall be in a format which can be easily read by humans and software. Second, we must handle large models with several thousands of artifacts. Comparing them with an inefficient algorithm is not feasible. Thus, we strive to utilize a generic diff engine which can be extended by customizable algorithms. Third, the diff engine shall be able to deal with different meta-models without much refactoring of the algorithms.

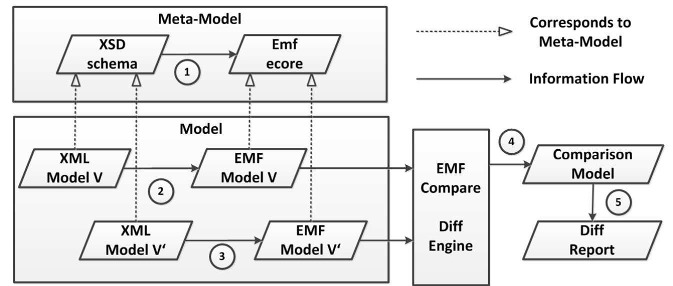


Fig. 5. Process and tooling for a Change Detection Analysis.

B. Basic Tool and Format Overview

The technical process for the CDA is shown in Fig. 5. First (1), the XSD schema[2] is imported to the EMF[3] framework and is then an EMF *ecore* meta-model. Such a meta-model describes the structure and properties of an EMF model. Then (2 and 3) the XML models are imported to the EMF framework and are then represented as EMF models. The EMF compare engine takes the two models as input and computes (4) a *comparison model*. The comparison model can be used to generate a customized diff report. This diff report is then stored in a file and contains all *add*, *delete* and *update* operations and the unique identifiers of the changed elements. The diff report can be accessed from other tools and GUI's.

[2]http://www.w3.org/XML/Schema.html
[3]http://www.eclipse.org/modeling/emf/

C. EMF Compare Customization

We identified *EMF compare*[4] as a highly customizable and extensible tool. It is open souce and can be embedded in java applications. EMF compare can be used without Eclipse in a standalone fashion. For this purpose only the appropriate *EMF compare* and *EMF ecore* libraries need to be included to a java application. Actually, we experienced some runtime issues because we were differencing large models with many differences. This is due to the standard match algorithm of EMF compare. A match engine is part of the difference engine and determines which elements of the old and new model correspond to each other. Basically, for this purpose, a similarity metric is computed for each pair of elements. The pair with the highest similarity metric is then a matched pair. Apparently, this algorithm performs poorly for models with a high number of elements. Although some optimizations of the pairwise comparison have been implemented (see [9][10]) they perform well under the assumption that not many differences exist between two versions of a model.

This assumption is not true in our case because we compare large models at a time interval of several weeks (an agile iteration). Thus, we decided to implement a match algorithm based on a comparison of unique identifiers which is much faster because no pairwise comparison is needed. EMF compare provides the possibility to utilize such a matching algorithm.

V. Related Work

The Assert4SOA[5] project is concerned with the issue of certifying software which is composed of several services. It is difficult to ascertain trust in heterogeneous software in a software ecosystem. The traditional certifying approaches do not take into account such heterogeneous systems. This project deals mainly with the issue of composing evidence for security for such systems. An approach for Common Criteria Certification of such systems is taken into account.

The EURO-MILS[6] project strives to exploit virtualization techniques in order to enable a separation between different levels of security for networked embedded systems. If an appropriate separation between highly secure software and low security software is possible, only the highly secure software needs to be certified which makes the approach more cost effective. In addition to the virtualization also the communication between the separated software entities needs to be taken into account.

The SecureChange[7] project deals with supporting security evaluation during the evolution of the software system at all levels of the software development process. A focus of this project is to focus on the delta between software releases in order to concentrate only on the changed software artifacts. Tools and processes have been developed in order to meet the mentioned objectives.

Jürjens [11] extends the UMLSec (UML) profile with annotations, so that model evolutions can be registered in the model. The original UMLSec extension can verify the model for specific security properties. The UMLSecCh can use change annotated models and compute a difference model from it. The verification is then applied to the difference model. It has been shown that such an incremental verification of security properties is much more efficient (in terms of calculation time) than a full model verification.

[4] http://www.eclipse.org/emf/compare/
[5] http://www.assert4soa.eu/
[6] http://www.euromils.eu/
[7] http://www.securechange.eu/

VI. Conclusion

Today's software systems are developed with changing requirements regarding functionality and security. Moreover, fast delivery is an important issue which is impacted by the duration of development and security evaluation. We deduced a conceptual background for an agile security evaluation which allows the management of changing requirements and fast time-to-market constraints. Moreover, this evaluation approach provides early feedback regarding the security concept of a product and thus avoids late and costly refactoring. We described a process which utilizes Change Detection Analysis and Experiential Change Impact analysis to improve such an iterative approach. In addition, we described an implementation of a change detection analysis for model-based security requirements and design.

Acknowledgment

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

References

[1] *Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final (July 2009).*
[2] A. Cockburn, *Agile Software Development*. Pearson Education, Oct. 2006.
[3] M. Cohn, *Succeeding with Agile*. Pearson Education, Oct. 2009.
[4] K. Beznosov and P. Kruchten, "Towards agile security assurance," in *NSPW*, 2004, pp. 47–54.
[5] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, "Why model versioning research is needed!? an experience report," in *Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS*, 2009.
[6] *Common Criteria. Reuse of Evaluation Results and Evidenc (October 2002).*
[7] S. A. Bohner, "Extending software change impact analysis into COTS components," in *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, 2002, pp. 175–182.
[8] M. S. Kilpinen, C. M. Eckert, and P. J. Clarkson, "The emergence of change at the interface of system and embedded software design," in *Conference on Systems Engineering Research, Hoboeken, NJ*, 2007.
[9] S. Wenzel, "Unique identification of elements in evolving software models," *Software & Systems Modeling*, vol. 13, no. 2, pp. 679–711, 2014.
[10] C. Brun and A. Pierantonio, "Model differences in the eclipse modeling framework," *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29–34, 2008.
[11] J. Jürjens, L. Marchal, M. Ochoa, and H. Schmidt, "Incremental security verification for evolving UMLsec models," *Modelling Foundations and*, pp. 1–17, Jan. 2011.

Patterns of Software Modeling

Wolfgang Raschke[1], Massimiliano Zilli[1], Johannes Loinig[2], Reinhold Weiss[1], Christian Steger[1], and Christian Kreiner[1]

[1] Institute for Technical Informatics
Graz University of Technology, Graz, Austria
{wolfgang.raschke,massimiliano.zilli,rweiss,steger,christian.kreiner}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria
johannes.loinig@nxp.com

Abstract. Software systems start small and grow in complexity and size. The larger a software system is, the more it is distributed over organizational and geographical confines. Thus, the modeling of software systems is necessary at a certain level of complexity because it can be used for communication, documentation, configuration and certification purposes. We came to the conclusion that several patterns of software modeling exist. The existence of such patterns is dependent on the history and the evolution of the system under consideration. We will show that a software system in its lifecycle has to face several crises. Such a crisis is a watershed in the application of new patterns. We provide an evolutionary view of software systems and models which helps understanding of current problems and prospective solutions.

Keywords: Model-Based Software Development, Collaborative Software Development, Application Lifecycle Management, Software Process Improvement.

1 Introduction

Typically, software systems are always increasing in complexity. This impacts development teams, inter-organizational software development, and geographical distribution. The distribution of the development process complicates communication and sharing of knowledge. Software modeling is a successful approach to mitigate these issues. However, there are currently a plethora of different approaches with regards to the modeling of software, requirements and specifications, such as requirements engineering, domain-specific modeling (DSM), software product lines (SPL), agile requirements engineering and application lifecycle management (ALM). It is hard to select an appropriate approach, because the success of it heavily depends on the context of the organizational situation. Thus, we provide a conceptual framework that helps to characterize the different approaches. This framework aims to facilitate the selection of a suitable software modeling method. We define several modeling patterns which we name and explain. Such a pattern is an abstraction and is kept simple in order to make a clear difference between the basic characteristics of today's software modeling

© 2014 Springer. Reprinted, with permission, from Proceedings of Fifth International Workshop on Information Systems in Distributed Environment (ISDE'14).

approaches. We outline the observation that during the lifecycle of software, the dynamics cause a change in its context. Within the new context, solutions often turn into problems. This effect results in a time of change and turmoil, a so-called *revolution* phase. We introduce the concept of Greiner [1] which regards the learning and development of systems as a sequence of evolutions which are interrupted by revolutions. Regarding this viewpoint, we derive an explanation and categorization of software modeling patterns. We show that the emergence of software modeling trends follow a generic pattern.

2 Conceptual Background

Usually, we apply known solutions to problems. If a solution has succeeded many times in the past, we are tempted to see this as a universal solution. Unfortunately, this attitude overlooks the fact that solutions always work in a specific situation, the so-called *context*. Fig. 1 shows the concept of a pattern: the problem is located in the context-free *problem space*. The solution is part of the *solution space* and it is constrained by the context. In the traditional view of a design pattern, the context is fixed and does not alter [2, 3].

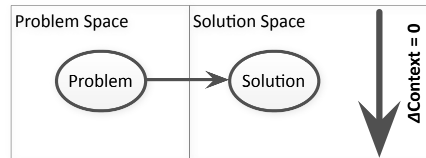


Fig. 1. Problem and solution in a fixed context

Dynamics arise when the context changes. Fig. 2 demonstrates that solutions turn into problems within the new context. The new problem has to be solved again in a constrained environment. This process may perpetuate for several iterations. These kinds of dynamics can be observed very often [4]. We use it as a reference for the development of modeling archetypes. For example, regarding transportation the dynamics are as follows: The petrol powered vehicle has been the solution to the problem of transportation over long distances. This solution has changed its environment because of pollution and the decreasing amount of petrol ultimately available. These new problems may one day be addressed by the electrical vehicle. This new solution will then itself cause new problems.

Greiner [1] describes a model where evolutionary development is interrupted by revolutions. These revolutions are caused by the changing context of size and time. The solutions no longer work within the altered context. Moreover,

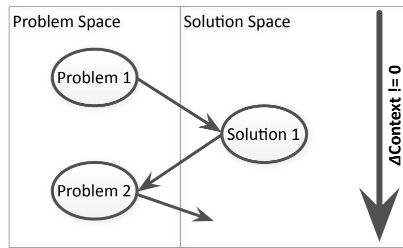


Fig. 2. The dynamics of problems and solutions in a changing context

they are then the cause of the new issues. He describes five phases of organizational development, their crises and the ensuing solutions. Although the phases of Greiner's model are different to the archetypes we state here, they have served as inspiration.

3 Software Modeling Patterns

We gathered experiences from the development of two industrial RMS. Moreover, in order to build up a catalog of patterns we interviewed experts from the following domains: automotive software, logistics software, secure software and financial software. Each of the patterns occurred in at least one of the software projects that was studied. The sample is broad and representative enough to formulate basic patterns within a changing context.
In Fig. 3 the continuous lines represent evolutionary phases with no change of the modeling pattern. These evolutionary phases are disrupted by revolutionary phases which are indicated by the dashed lines. Such a phase denotes the change from one pattern to the next and always involves a learning phase. This learning phase includes recognizing the problems and searching for new solutions. It takes some time to pass through this phase and proceed further.

In Fig. 3 it can be seen that the two influencing factors (the context) here are *complexity* and *time*. Complexity increases with size and mainly influences the style of modeling. The second factor is time: it is needed to adapt to new circumstances. Thus, it is mainly an influencing factor during a time of crisis. Then, neither size nor complexity grow rapidly. In the following we describe the patterns in more detail.
The patterns are structured as follows: The first patterns begins with a starting point which describes the situation at the beginning of a small software system. The problem section is part of each pattern: it depicts the established working practices. After the solution, we provide the context section. The context describes the environment and the situation. The solution must work within the context. If the context changes, the solution turns into a problem, as a conse-

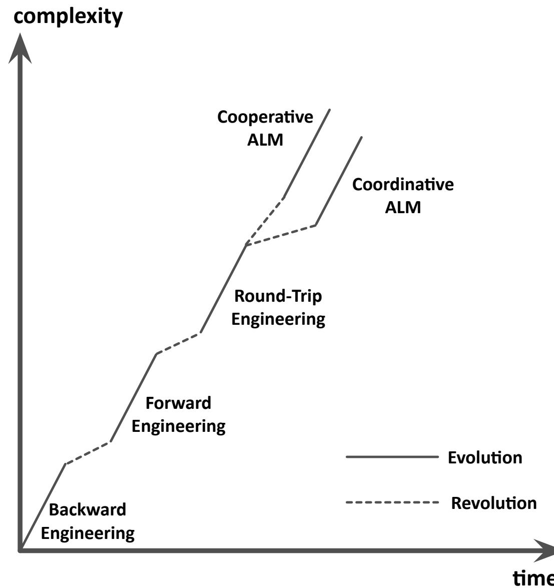


Fig. 3. The development of software modeling patterns within the context of complexity and time

quence. The consequences of a pattern characterize the problem of the following pattern.

3.1 Backward Engineering (BE)

Starting Point Initially, the software is small and only a few developers work on it. There is no clear division of functionality between people, so each developer works on the whole system. Thus, it can be assumed, that everyone has knowledge of the complete system. Hence, there is no need for explicit modeling of the software.

Solution The first pattern is outlined in Fig. 4. The sub-types *code-alone*, *code-parallels-model* and *code-to-model* are similar. We do not see a major difference in the three depicted types regarding the effect on the development process. Thus, they are collected within one single pattern. In *code-alone*, code is written and compiled to a finished product. This is the simplest model that can exist. Very soon, a mental model emerges which parallels the code development (*code-parallels-model*). This model can be written documentation or a domain vocabulary. The main point is that it is not explicitly derived from the code. It emerges on an *ad hoc* basis. The next step is *code-to-model*: to derive the

model from the source code. Such a model may be documentation. It can be generated automatically via document generators. This method can be refined to a certain degree. It is simple and can applied without a high overhead, even in small software projects.

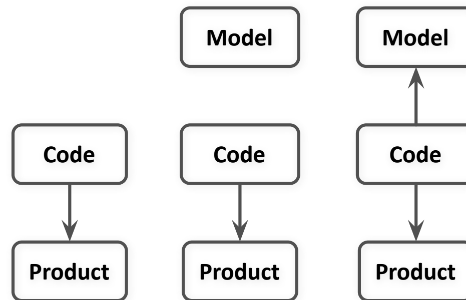


Fig. 4. The three sub-types of backward engineering are: *code-alone*, *code-parallels-model* and *code-to-model* [5]

Context Change and Consequences Successful software systems tend to grow in complexity and size. Thus, the number of software developers also grows. At a certain point, the whole system can no longer be understood by each developer. Communication becomes more difficult because of the increasing number of developers.

3.2 Forward Engineering (FE)

Solution Forward engineering is a top-down approach. In Fig. 5, *Model A* is created and maintained manually. It is used to generate certain artifacts, such as parts of the documentation, configuration and source code. The generation of redundant code is efficient and works well with an explicit model. Moreover, documentation artifacts are generated which go beyond the functionality of a general-purpose document generator. The architecture is crafted in the model and the source code artifacts (header files, class skeletons) are generated. So, the generated architecture is consistent with the model. UML[3] is a prominent example for such an approach. However, there is still a very high manual effort in maintaining these models.
In Fig. 5 it can also be seen that *Model B* is created from the source code. This is not a contradiction to the concept of FE because each arrow only points in one direction. Thus, no synchronization is provided.

[3] www.uml.org/

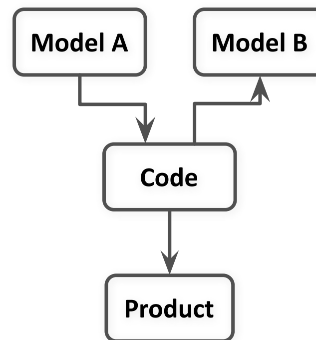


Fig. 5. In forward engineering the information flow is either from model to code or from code to model

Context Change and Consequences The forward engineering principle works adequately for software of small and middle complexity with a relatively low system dynamic. In this context, this method works perfectly. Such a model grows with the complexity and the size of the software. It requires increasingly more effort to maintain this model and at one point there are several thousand changing information items. It is then hard to keep the model consistent with the fast changing software. There are no synchronization methods available. Thus, concurrent work on the models is virtually impossible. All these ingredients form a bottleneck, so that the model is not able to follow the source code. At this point it is likely that maintenance of the model is stopped.

3.3 Round-Trip-Engineering (RTE)

Solution Round-trip engineering incorporates forward engineering, backward engineering and the synchronization of both. Fig. 6 shows that the information flow between model and code goes in both directions. So, the high amount of manual work required for synchronization can be leveraged by recovering the model from the code. Changes in the code can be synchronized with the model. Moreover, it can be seen in Fig. 6 that synchronization between models is also possible. Hence, these approaches need advanced synchronization mechanisms, such as *model-to-model*, *text-to-model* and *model-to-text* transformations. For the synchronization of different versions of such a model, two techniques are needed: *merge* and *diff*. If all these algorithms are supported, most of the artifacts can be synchronized automatically. Still, such a synchronization has to be controlled by responsible roles.

Context Change and Consequences Round-trip engineering is still, at the moment, a high-end solution and expensive to establish. Thus, it can be ex-

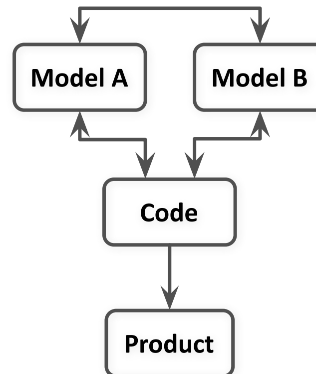


Fig. 6. Round-trip engineering enables synchronization between models and between model and code

pected that it is present only in medium and large software. The bottleneck of the previous pattern is now resolved. In any case, the level of information flow escalates. There are now several versions of the software and the model. It becomes increasingly difficult to coordinate these different versions. Anyone can integrate results with the model. As a consequence, the introduction of flaws is more likely. Moreover, there are no clear responsibilities and access rights.
First, the size of the software leads to the effect that the domain experts (who are rare) are more involved in the specification of the system. Programming activities can be outsourced more easily. As a consequence the software may be constructed by several software suppliers. Such a *software ecosystem* [6, 7] exists for example in the automotive industry. There, tier 1 and tier 2 suppliers work together with the car manufacturer. However, such a software ecosystem comes with many risks and uncertainties because the software development can no longer be controlled as a whole [8].
As we can see, the crisis stems from two reasons: first, the increased information flow that is caused by the high number of different versions (synchronization). Second, the uncertainty and risks within a software ecosystem remains a problem. Both issues are addressed in the following patterns: *Coordinative Application Lifecycle Management* and *Cooperative Application Lifecycle Management*.

3.4 Coordinative Application Lifecycle Management

Solution Until now, the evolution of model-based systems only incorporated structural models. Fig. 7 shows a workflow engine controlling information flow as an addition to RTE. The next logical step is such a workflow (process) integration with models. This means, that all steps within a process shall be controlled, documented and delegated to the appropriate roles. Thus, artifacts within a

model have states (started, analyzed, finished) and roles (accountability, responsibility, access rights). Examples for such workflow management systems are: support of reviews, project plans, allocation of roles to tasks and the like. For certification issues, all these processes have to be documented. We expect tool chains with a tight integration of test results, bug reports, bug resolving, reviews, project plans and so on. Thus, a seamless documentation to provide evidence for process maturity (e.g. SPICE [9], CMMI [10] and dependability (e.g. security [11], safety [12]) is possible from such a model. This phase puts emphasis on coordination which is, generally speaking, the formal performing of actions.

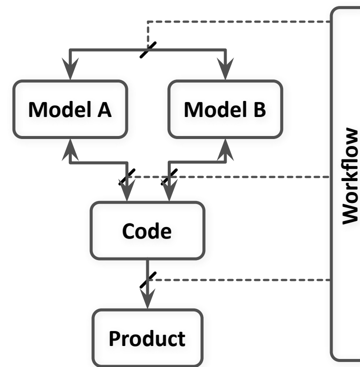


Fig. 7. In coordinative ALM the information flow is coordinated by an integrated workflow management

Context and Consequences It is very difficult to ascertain a higher level of process maturity. There is a high cost overhead for establishing these processes. This high cost overhead makes it difficult for smaller companies to exist within such a supply chain. This issue can be mitigated through highly integrated tool chains and outsourcing of process knowledge. So, developers can concentrate on domain expertise. We expect a high degree of dependency of these firms in the supply chain on external consulting and tool providers. At the moment, the software industry comes up with so-called ALM tools. They extend traditional requirements engineering with a process integration.

3.5 Cooperative Application Lifecycle Management

Solution In Cooperative ALM, communication needs to be bound to artifacts within a software model. Such artifacts can be: activity-based (tasks, change requests) or structural (architecture, requirements). Moreover, synchronous and

asynchronous communication mechanisms need to be implemented. Respective tools enable a notification based on the state of the artifacts. For instance, the completion of a task is an event which triggers a notification message. High-end RMS integrate these mechanisms. Prominent examples are: Attlassian JIRA[4] and CollabNet TeamForge[5].

Context and Consequences Cooperative ALM seems to parallel the Coordinative ALM. The objective of Coordinative ALM is to provide evidence for trust through formal processes. Whereas, the goal of Cooperative ALM is to foster and enable the sharing of information. Unfortunately, this cannot be done efficiently via formal channels. So, the principle is based on collaboration instead of coordination. A corresponding trend in software engineering is agile working. Information interchange has become more difficult because the development sites are scattered over distinct places throughout the world. Sophisticated tools facilitate and foster the sharing of information.

4 Conclusion

We listed a catalog of the most important issues regarding software modeling. We complemented the catalog with issues that we collected during industrial software projects. Moreover, we strove to present these issues in relation to each other. Additionally, we sought to investigate their relationship with the respective situation. Thus, we stated five patterns of software modeling which exist in today's industrial software development. In addition, we stated the typical context of each pattern. We examined the sequential development from one pattern to the next. It is important to understand the dynamics of this development: long periods of steady development are disrupted by crises which are a watershed in the application of a new modeling pattern. In order to make our developmental model as generic as possible, we omitted unnecessary details and abstracted it to only five basic patterns. Certainly in a real software project, the development will never exactly follow our model. Nevertheless, some issues are still symptomatic. Knowledge of these symptomatic issues helps to characterize the actual problems and design new solutions with the contextual setting in mind.

Acknowledgment

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure systems GmbH for support.

[4] www.atlassian.com/JIRA
[5] http://www.collab.net/products/teamforge

References

1. Greiner, L.: Evolution and revolution as organizations grow. 1972. Harvard Business Review (January 1997)
2. Alexander, C., Ishikawa, S., Jacobsen, M., Fiksdahl-King, I., Angel, S.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (August 1977)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Pearson Education (October 1994)
4. Watzlawick, P., Weakl, J.H., Weakland, J.H., Fisch, R.: Change; Principles of Problem Formation and Problem Resolution. W. W. Norton (1973)
5. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley (2008)
6. Messerschmitt, D.G., Szyperski, C.: Software Ecosystem. MIT Press (2002)
7. Manikas, K., Hansen, K.M.: Software ecosystems – A systematic literature review. Journal of Systems and Software **86**(5) (May 2013) 1294–1306
8. McGregor, J.D.: A method for analyzing software product line ecosystems. In: Proceedings of the Fourth European Conference on Software Architecture. (2010) 73–80
9. Messnarz, R., Ross, H.L., Habel, S., König, F., Koundoussi, A., Unterrreitmayer, J., Ekert, D.: Integrated Automotive SPICE and safety assessments. Software Process: Improvement and Practice **14**(5) (September 2009) 279–288
10. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI Guidlines for Process Integration and Product Improvement. Addison-Wesley Longman Publishing Co., Inc. (2003)
11. : Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final (July 2009)
12. Standardization, I.I.O.f.: ISO 26262 Road vehicles Functional Safety Part 1-10. (2011)

Where does all this waste come from?

Wolfgang Raschke[1], Massimiliano Zilli[1], Johannes Loinig[2], Christian Steger[1] and Christian Kreiner[1]

[1] Institute for Technical Informatics, Graz University of Technology, Austria {wolfgang.raschke, massimiliano.zilli, steger, christian.kreiner}@tugraz.at
[2] NXP Semiconductors Austria GmbH, Gratkorn, Austria johannes.loinig@nxp.com

Abstract

Agile development processes are more flexible than conventional ones. They emphasize iterative development and learning over feedback loops. Nevertheless, we experienced some pitfalls in the application of agile processes in dependable software systems. We present here the experiences we gathered in the construction of high-quality industrial software. Moreover, we will digest our experiences into a conceptual model of waste creation. This model will be refined to a case study where we take appropriate measurements in order to provide empirical evidence for it. Finally, we discuss the implications of the developed model, which helps to estimate the trade-off between agile and traditional software processes.

Keywords

Software Process Improvement, Innovation, Agile Processes

1 Introduction

Today's dependable systems face the challenge of being built as safe, secure and reliable systems. The construction of such systems utilizes quality assurance methodologies such as code reviews, testing and static code analysis. Industry-relevant frameworks for process-based quality assurance exist for process maturity (SPICE [1]) and security (Common Criteria [2]). A higher capability level in such a framework implies more processes accomplished at a higher level of maturity. Generally, a higher level of assurance comes with a higher development effort.

We investigated the development process of a highly secure software system. This project is developed with agile methodologies: Test-Driven Development [3], Continuous Integration [4] and Scrum [5]. These agile processes have considerable benefits in the industrial software project. Nevertheless, we observed some inconsistencies with regards to waste. Waste is an agile terminology, which denotes unnecessary work, which does not add to customer value [6]. We believe that it can be traced back to the agile way of working in the context of dependable systems.

In order to build a suitable model for software development in dependable systems, we discuss a pattern of evolution, which has been observed in diverse industries. First, we apply this pattern to a general model for dependable systems. Second, we refine the dependable systems pattern in order to

© 2014. Reprinted, with permission, from Proceedings of 21st EuroSPI Conference (EUROSPI'14).

create a case study, which will give us the opportunity to provide evidence.

We will discuss the implications which our model has on various levels of granularity and explain them regarding the agile development model and the V-model. Moreover, we show how our model can be used as framework of considering software development processes on a high level of abstraction. Finally, we conclude our findings and provide guidelines for scoping agile versus traditional processes.

2 Experience From an Industrial Project

We gained experience from an industrial software project during the development of a highly secure system. It is imperative that evidence be provided for the purposes of security via certification. The so-called *Common Criteria* [2] is a documentation-based approach. To guarantee high levels of assurance, this evidence is the result of carefully designed and accomplished processes. A great amount of effort has been put into composing the appropriate documentation for these processes.

We investigated the development process through interviews with software architects and developers. As a reference model, we utilized the V-model. Within the reengineered V-model, we identified two agile iterative cycles. Our observations are outlined in Figure 1.

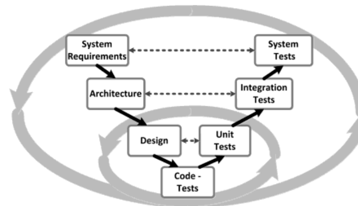


Figure 1: Iterative cycles in architecture and on component level.

The inner cycle is an iterative loop that represents the agile development process that mainly focuses on the design and implementation of user stories. Since we are applying Test-Driven Development (TDD), writing tests belongs to the code-and-test phase. Conversely, the execution of the tests is part of the unit test phase. The design comprises of the specification of the component and its interface at component level. It is allowed to change interfaces and connections between components in this inner cycle. It pushes the outer architectural cycle. This is usually regarded as a bottom-up design methodology.

The outer cycle represents the development of system requirements, architecture, integration tests and system verification (certification) in the V-model. As mentioned above, the outer cycle is driven by the inner cycle, which means that the architecture is first established incrementally at code and design level and then explicitly defined in the architecture phase. The system requirements do not evolve directly out of the inner circle but the linkages between them and the architecture have to be permanently maintained.

We found that there is a friction between both cycles, namely in the form of synchronizing implementation level artifacts (code and design) with architectural artifacts. Such a friction occurs in several manifestations which we describe in the following.

First, changing interfaces affects testing. Thus, unit tests and integration tests have to be adopted according to the new interfaces. We will show this effect later in more detail.

Second, changing interfaces have an impact on security evaluation because interface descriptions are part of the respective documentation.

Third, some interfaces have to be known to link the components together because different programming languages are utilized. Hence, interfaces have to be reworked in the product integration, as well.

Finally, we conclude that changing an interface causes rework in at least three obvious cases. The real cost cannot be measured reliably.

3 Model of Change Impact

In order to investigate potential causes for waste, we introduce a conceptual model of an innovation lifecycle. We apply the pattern described in [7] because it is simple and allows mapping to diverse circumstances.

3.1 Product Innovation vs. Process Innovation

Abernathy [7] describes a lifecycle, where at the beginning of a new development, product innovation is accomplished at a high pace and process innovation on a relatively low one. Gradually, product innovation decreases its rate whereas process innovation increases. There is a lag between product innovation and process innovation. It is important to understand the meaning of *product* and *process* innovation, which we will clarify in the following.

3.2 Product Innovation

Product innovation is the way a product is altered. Generally, product innovation is the change of *what* we build. Henderson [8] defines a model of innovation, where the rate of changing links between components is an important indicator. The change of links can be seen as the change of interfaces. In the following, we will use the term architectural evolution to label the change of interfaces

3.3 Process Innovation

Process innovation defines *how* we build products. This how usually includes all processes that embrace quality engineering, such as reviews, tests and static code analysis. It is likely that a change in an interface causes rework in the quality processes. A common practice is to increase the rate of process improvement when the interfaces are more or less stable [7].

3.4 Mapping to Software Terminology

In order to facilitate the construction of a hypothesis, which can be evaluated empirically in a software project, we map the previously outlined terminology to software engineering methodologies.

Product innovation is the sum of all activities that immediately affect a product. In a typical software project, these activities are: architecture, design and coding.

We assume architectural evolution as a quantifiable indicator for product innovation. We define architectural evolution here as the rate of interface changes between at least two components. For example, in the architecture phase of the V-model, the rate of interface changes is high. In design and coding, this rate is, in comparison, low.

Process Innovation is the aggregate of all activities that indirectly affect a product. In a software project these activities are usually: unit testing, integration testing, system testing, code review and static code analysis. In order to facilitate the interpretation we assume that process innovation is equal to all quality activities in a software project. The measurable quantity is the amount of quality activities at a certain point in time.

3.5 Definition of Waste

Ikonen [6] defines waste as "basically everything that does not add to the customer value of a product". This viewpoint regards everything except coding as waste, because it does not directly add value to the customer. This expectation is too narrow in our opinion, because testing and quality assurance add value indirectly. So, we regard waste as activities that neither *directly* nor *indirectly* add value.

3.6 Relation between Waste, Product and Process Artifacts

Our observations in the industrial case study suggest that waste is proportional to the evolution of the architecture in terms of changed interfaces *dA*. The definitions of the terms are given in Table 1. The following formula states the generic model of waste creation:

$$W = \sum_i c_i \cdot dA = Q \cdot dA$$

Definition	Interpretation	Short
Architectural evolution	Change of component interfaces	dA
Waste	Effort with no additional value	W
Quality processes effort	Total effort for *all* quality activities	Q
Constant	Coupling factor *no. i*	c_i

Table 1: Terminology of the generic model of waste creation.

4 Evaluation With Empirical Data

The empirical evaluation has been conducted in cooperation with the company *NXP Semiconductors*. We evaluated a software project which is dedicated to the implementation of a highly-secure embedded system. About 100 developers participate at the project. The project includes several development teams, a testing team and a dedicated security team. The number of tests is typical for a high-quality embedded system of medium size: approximately 80 modules are tested by several thousand tests (unit tests, integration tests and acceptance tests).

In order to apply the previously mentioned model to real data, we refine it in the following. We measured the number of failing test cases W. In addition, we recorded the number of changed interfaces *dA* on a daily basis. We provide the explanation of the terms in Table 2. Finally, we can construct a hypothesis, which can be evaluated with empirical data:

$$W_t = c_t \cdot dA$$

Definition	Interpretation	Short
Architectural evolution	Change of component interfaces	dA
Waste	Number of failing test cases	W
Constant	Coupling factor	c_t

Table 2: Terminology of the refined model which is evaluated empirically.

Change of component interfaces: this is the accumulated number of changes in interfaces per day. This number has been measured on a daily basis, seven days a week. This information was retrieved from the source code repository.

Number of failing test cases: the number of failing test cases has been measured on a daily basis, seven days a week. The tests have been automatically executed each day on a continuous integration server. All test results have been logged.

We apply pre-processing to the data, which describes the changing of interfaces for the following reasons: First, there are remarkably few changing of interfaces on Saturdays and Sundays. Hence, there is a fundamental oscillation on a weekly basis. Second, the time span between altering an interface to the effect on the number of failing tests is not constant. This variation of time from cause to effect exists, because each and every interface is not immediately integrated into the tested product.

In order to mitigate the mentioned effects, we apply a staged pre-processing to the recorded data, which is described in the following:

Step 0: Raw Data

For the raw data, the Cross-Correlation Coefficient (*CCC*) equals 0.16.

Step 1: Constant Moving Average Filter

In order to smooth the vector *dA*, we apply a simple moving average with a length of *N=7*:

$$dA_{avg}(n) = \frac{1}{N} \sum_{j=0}^{N-1} dA(n+j)$$

The *CCC* equals 0.51 and thus is remarkably higher, as before. We explain this by the filtering out of peaks that occur on weekends.

Step 2: Vector Norm

Both vectors *W* and *dA* have considerably different scales. In order to apply vector normalization, we divide both vectors by their length and obtain the respective unit vectors.

$$dA_{norm} = \frac{dA_{avg}}{\|dA_{avg}\|} \quad , \quad W_{norm} = \frac{W_t}{\|W_t\|}$$

The normalization does not affect the Cross-Correlation Coefficient. The normalized signals are shown in Figure 2.

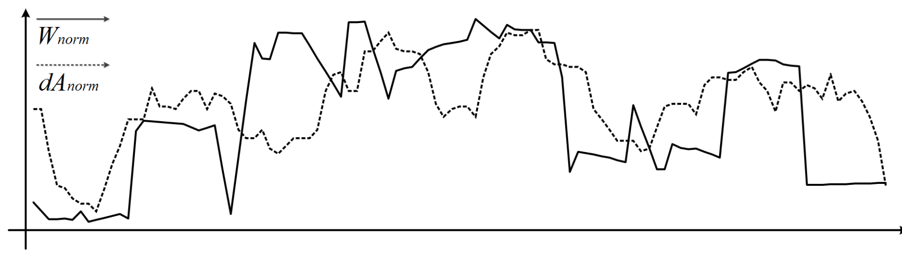


Figure 2: *Wnorm* and *dAnorm* after the Normalization.

Step 3: Dynamic Time Warping

In order to cope with the time variation, we apply *dynamic time warping* [9, 10], a non-linear signal processing algorithm. Dynamic time warping compares two signals and locally stretches and compresses the time axis in order to find an optimal alignment between both. A good alignment is characterized by a high similarity of both data series. The warping path *p* is a timely mapping between *dAavg* and *Wnorm* which is computed by the dynamic time warping algorithm:

$$p = dtw(dA_{avg}, W_{norm})$$

The time mapping p allows the reconstruction of the signal *dArec*:

$$dA_{rec}(t) = dA_{norm}(p(t))$$

The *CCC* equals 0.87 and shows that there is a high correlation between the pre-processed data. The reconstructed signal *dArec* and the reference signal *Wnorm* now show a remarkable visual similarity, as can be seen in Figure 3. Thus, we can assume a proportional relation between both signals.

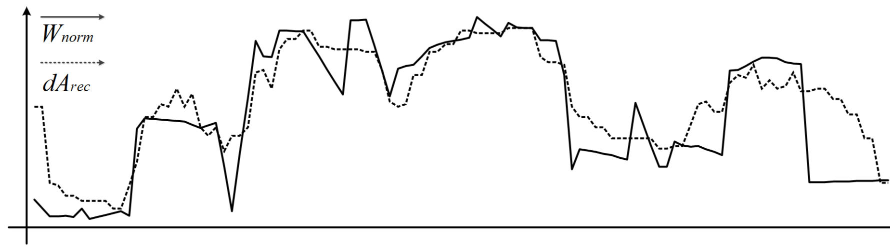


Figure 3: After data processing step 3, the signals *Wnorm* and *dArec* are obviously similar.

5 Implications

In the presented study, there is correlation between the evolution of the architecture and failing test cases. This undermines the general hypothesis that the evolution of the architecture causes rework in quality processes. The implications can be applied to the V-model and agile development processes.

5.1 Implications for the V-Model

The V-model reflects Abernathy's pattern of innovation [7] where in a cycle of experimentation, learning and refinement, an architecture is defined. In this architecture phase, there are a high amount of changing interfaces (see dotted line in Figure 4 a) but quality processes are not unfolded to full maturity (see continuous line in Figure 4 a). So, the product of both curves is in the middle range (see Figure 4 b).

Certainly, there occurs a point in time, where architectural activities slow down. In the V-model this is usually the phase of implementation, which is the pivotal element of the V and thus has connections to architectural activities *and* to quality processes. In this phase, architecture *and* quality processes are assumed to operate at a medium velocity. The product of both is high in this phase. In the third phase, the interfaces are mostly stable and quality assurance is at its highest level of the whole lifecycle. The product of both curves is in the middle range.

For a better illustration of the curves in Figure 4, we apply sample values in Table 3. The sum of a column represents the architectural evolution (*dA*), effort for quality processes (*Q*) and the resulting total waste (*W*) of the model.

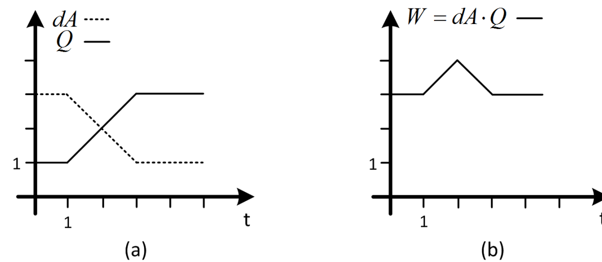


Figure 4: Coherence between waste *W*, changing interfaces *dA* and total effort for quality processes *Q* in the V-model.

5.2 Implications for the Agile Model

In agile processes, the sequence of process steps is abandoned. Rather, the activities are accomplished concurrently in iterative cycles. In such an iterative cycle, each activity (architecting, coding, testing and verification) has to be performed. We assume for architectural evolution (see dotted line in Figure 5 a) and for quality activities (see continuous line in Figure 5 a) a constant and medium velocity. As can be seen in Figure 5 b, the product of both is constantly very high. Therefore, the interface changes create huge effort because all quality activities in the current iteration are immediately affected.

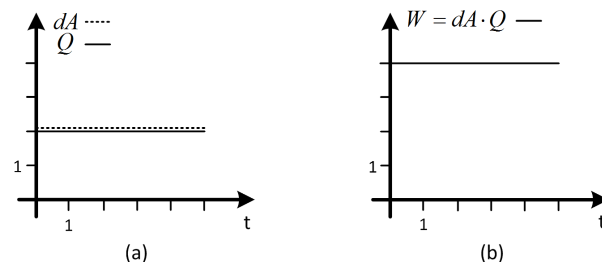


Figure 5: Coherence between waste *W*, changing interfaces *dA* and total effort for quality processes *Q* in the V-model.

For a better illustration of the curves in Figure 5, we apply sample values in Table 3. The sum of a column represents the agile evolution (*dA*), the agile effort for quality processes (*Q*) and the resulting total waste (*W*) of the model.

6 Conclusion

In Table 3, the effort for architecture and processes is equal (Sum = 10). This allows a comparison of the resulting waste: for agile working practices, the model predicts a higher resulting waste compared to the V-model. Of course, the presented model of waste creation is an abstraction. In reality there is some amount of approximation in the application of such a model. Nevertheless, the theoretical model is suitable for deliberation. Such deliberation suggests the following conclusion: agile methods are more appropriate for software projects with low demand for quality and possibly only acceptance tests. Traditional processes, like the V-model process is more appropriate in software projects with a high demand for quality and respective activities.

Time	V-model *dA*	V-model *Q*	V-model *W*	Agile *dA*	Agile *Q*	Agile *W*
0	3	1	3	2	2	4
1	3	1	3	2	2	4
2	2	2	4	2	2	4
3	1	3	3	2	2	4
4	1	3	3	2	2	4
Sum	10	10	16	10	10	20

Table 3: Sample data for the model of waste creation for the V-model and agile processes.

7 Acknowledgment

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure::systems GmbH for support.

8 Literature

1. Messnarz R., et al. Integrated Automotive SPICE and safety assessments. Software Process: Im-provement and Practice, 14(5):279–288, 2009.
2. Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final, 2009
3. Janzen D., Saiedian H. Test-driven development: Concepts, taxonomy, and future direction. Computer Science and Software Engineering, 2005.
4. Duvall P., Matyas S., Glover A. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007.
5. Cohn M. Succeeding with Agile. Pearson Education, 2009.
6. Ikonen M., Kettunen P., Oza N. Exploring the sources of waste in Kanban software development pro-jects. Conference on Software Engineering and Advanced Applications, SEAA 2010.
7. Abernathy W. J., Utterback J. M. Patterns of Industrial Innovation. Technology review, 64: 254-28, 1978.
8. Henderson R., Clark K. Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. Administrative science quarterly, 9-30, 1990.
9. Giorgino T. Computing and visualizing dynamic time warping alignments in R: the dtw package. Journal of statistical Software, 31(7):1-24 , 2009.
10. Müller M. Dynamic Time Warping. Information retrieval for music and motion, 69–84, Springer, 2007

9 Author CVs

Wolfgang Raschke

He completed his studies in telematics in 2012. Since February 2012 he works as a PhD student at the Graz University of Technology. His current research interests are: software product lines for secure systems, innovation and evolution of software and software models.

Massimiliano Zilli

He received his Master's degree in Electronic Engineering at the University of Udine in 2007. From 2008 to 2012 he worked as a software developer for embedded systems. Since February 2012 he works as a PhD-Student at the Graz University of Technology. His research focuses on optimization techniques for embedded systems.

Johannes Loinig

received the Dr.techn. degree (PhD) in electrical engineering with focus on Secure Embedded Systems from Graz University of Technology in 2012. Currently he works as software and security architect in several projects at NXP Semiconductors Austria and is in the lead of several research projects with focus on HW/SW codesign and information security.

Christian Steger

received the Dipl.-Ing. degree (M.Sc.) 1990 and the Dr.techn. degree (PhD) in electrical engineering from Graz, University of Technology, Austria, in 1995, respectively. He is key researcher at the Virtual Vehicle Competence Center (VIF, COMET K2) in Graz, Austria. From 1989 to 1991 he was software trainer and consultant at SPC Computer Training GmbH, Vienna. Since 1992, he has been Assistant Professor at the Institute for Technical Informatics, Graz University of Technology. He heads the HW/SW codesign group at the Institude for Technical Informatics. His research interests include embedded systems, HW/SW codesign, HW/SW coverification, SOC, power awareness, smart cards UHF RFID systems, multi-DSPs.

Christian Kreiner

graduated and received a PhD degree in Electrical Engineering from Graz University of Technology in 1991 and 1999, respectively. From 1999 to 2007 he served as the head of the R&D department at Salomon Automation, Austria, focusing on software architecture, technologies, and processes for logistics software systems. He was in charge to establish a company-wide software product line development process and headed the product development team. During that time, he lead and coordinated a long-term research programme together with the Institute for Technical Informatics of Graz University of Technology. There, he currently leads the Industrial Informatics and Model-based Architectures group. His research interests include systems and and software engineering, software technology, and process improvement.

International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015 37

Balancing Product and Process Assurance for Evolving Security Systems

Wolfgang Raschke, Institute for Technical Informatics, Graz University of Technology, Graz, Austria

Massimiliano Zilli, Institute for Technical Informatics, Graz University of Technology, Graz, Austria

Philip Baumgartner, NXP Semiconductors Austria GmbH, Gratkorn, Austria

Johannes Loinig, NXP Semiconductors Austria GmbH, Gratkorn, Austria

Christian Steger, Institute for Technical Informatics, Graz University of Technology, Graz, Austria

Christian Kreiner, Institute for Technical Informatics, Graz University of Technology, Graz, Austria

ABSTRACT

At present, security-related engineering usually requires a big up-front design (BUFD) regarding security requirements and security design. In addition to the BUFD, at the end of the development, a security evaluation process can take up to several months. In today's volatile markets customers want to be able to influence the software design during the development process. Agile processes have proven to support these demands. Nevertheless, there is a clash between traditional security design and evaluation processes. In this paper, the authors propose an agile security evaluation method for the Common Criteria standard. This method is complemented by an implementation of a change detection analysis for model-based security requirements. This system facilitates the agile security evaluation process to a high degree. However, the application of the proposed evaluation method is limited by several constraints. The authors discuss these constraints and show how traditional certification schemes could be extended to better support modern industrial software development processes.

Keywords: *Agile Development, Common Criteria, Model-Based Software Development, Model Evolution, Security, Traceability*

DOI: 10.4018/ijsse.2015010103

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

© 2015 IGI-Global. Reprinted, with permission, from Proceedings of International Journal of Secure Software Engineering (IJSSE).

38 International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015

1. INTRODUCTION

Traditional security engineering requires a big up-front design (BUFD) which includes the following engineering tasks: threat analysis, security requirements elicitation, security design and (security) architecture. Reviews ensure the consistency of these artifacts. In highly secure systems, the consistency of the security requirements and the security architecture is formally verified. Altogether, this constitutes a huge effort, before the implementation is even started upon. After the security design, the system is implemented according to requirements and design. Thereafter, evidence of security has to be provided in the form of documentation. This documentation is then delivered to the evaluation body. The evaluation may take several months. If the feedback from the evaluator is negative, the system has to be re-designed. If this is the case, then a large amount of time and effort would be required. Moreover, the pressure to deliver the product to market is high. Summarizing, the challenges involved in security engineering are: early validation and time-to-market. The software industry came up with the trend of agile development practices (Cockburn, 2006), such as XP, Scrum, and Test-driven development. Agile methods focus on customer interaction and incremental software development. Feedback loops, such as customer on-site, pair programming and refactoring tend to minimize errors. Generally, agile processes react flexibly to changing customer requirements. In contrast, traditional processes tend to fulfill contractually specified requirements and are inflexible, by nature. Agile processes have demonstrated a high success rate in several software projects (Cohn, 2010). Unfortunately, agile methods have not been well studied for high-level security engineering and evaluation. Case studies and success stories are lacking, especially for the *Common Criteria* standard (Common Criteria, 2012). Despite the fact that traditional security engineering seems to counter agile practices (Beznosov, 2004), we think that a synthesis of both is not contradictory, if well considered.

In the background section we provide material, we think helps to understand the following sections. As we do not claim a contribution in this section, we want to keep this section as simple as possible.

Our contribution is the analysis of two evaluation approaches, which are possibly suitable for an agile security evaluation. Both evaluation approaches are compared and one of them is then proposed for an agile security evaluation. In an agile security evaluation, all security properties of the system are kept in a model, which is capable of creating documentation for an evaluation round. In the agile evaluation approach only the increment of the model has to be reviewed. This has several benefits: the entire system does not have to be reviewed in each iteration. The evaluator starts with a small system and gains experience as the system matures. Early feedback enables an early, and thus more inexpensive, correction of the system. With elaborated algorithms we are able to create a difference model for all changes during an increment. Additionally, we demonstrate the feasibility of automation by customizing existing open source software.

The presented agile security evaluation approach is applicable under certain constraints. We discuss those constraints, which are fulfilled in a large portion of today's industrial software projects. However, the existing certification scheme could be enhanced in order to better support agile development processes. As a contribution, we propose to view security assurance under two perspectives: structural product assurance and behavioral process assurance. Both of the mentioned assurance paradigms are well established in other domains, such as automotive safety engineering (Habli, 2006; Habli, 2007; Hawkins, 2010). These paradigms span a two dimensional assurance space. The concept of such an assurance space could provide a more flexible certification scheme that could support traditional and agile development processes.

Despite the fact that there are few Common Criteria certifications (139 in 2014)[1] we consider improving the certification process important. First of all, many companies do not have the

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015 39

choice to certify their products for another standard (e.g. ISO 27001). This is especially true for the smart card industry, which certified 57 products in 2014 up to now[1] which is a portion of 41 per cent. In the past the smart card industry focused on mass production of standardized product and thus the number of certifications was relatively low. However, there is currently a shift in this industry: an increasing number of customers demand a tailored functionality and also a tailored security of those products. Currently, the expensive and inflexible certification hinders the individualization of products.

In the discussion with practitioners, we found that evaluation processes for Common Criteria, such as delta and compositional certification are hardly relevant in practice because they are considered as too complex. In order to start supporting security evaluations with the model-based approach, we strived to start with the least complex challenge. In practice, this challenge is the following: the iterative certification is already accepted by certification bodies and industry but this approach is informal at the moment and thus less structured.

For exchanging increments of documents, currently text based diff-tools[2] are used. However, this approach is work-intensive and error-prone in complex software projects. We strive to improve the situation by taking advanced model-evolution techniques into account. Our work proposes to change the certification standard. Thus, we do not directly address the certification body here. We address the organizations which are responsible for issuing the standard. We show here the concerns and possible improvements from the viewpoint of smart card development. As mentioned above, in 2014, 42 per cent of certified products are smart cards and smart card operating systems[1]. Regarding this high number, considering an adaptation would make sense in order to better support the majority of Common Criteria certifications.

An improvement of the certification could not only support the smart card industry but also other industries. As a result, we expect a fast growing number of Common Criteria certifications due to the individualization of products.

2. BACKGROUND

2.1. Requirements for the Security-Relevant Agile Development Process

We developed this method mainly to gain improvements regarding time-to-market and early validation. In addition, we strive to improve the semi-automation of the document creation with the model-based approach.

2.1.1. Time-to-Market

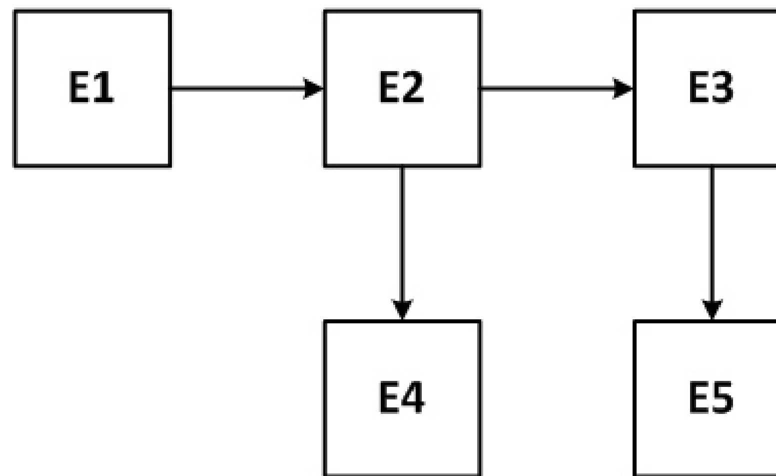
Time-to-market is one of the key success factors in the high-security business. In a traditional certification, the evaluation is accomplished after the implementation is finished. The time-to-market can significantly be reduced, if the evaluation starts in parallel with the development. Some documents can only be created after the development is finished completely. However, the time span from the finished product to the certificate can be reduced.

2.1.2. Early Validation

In the traditional certification approach, the evaluation starts late, when the product is already finished. If there is a negative evaluation result, the refactoring is expensive and seriously delays the completion of the product. In order to avoid these issues, several companies employ consultants to support them in creating and managing the certification process. These consultants are in contact with the evaluation body during the development phases. With our proposed method, we want to raise the level of maturity, so that the certification processes are *repeatable*. *Repeatable* means that a certification process that has been established with the help of consultants, can be repeated for new project without external support.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 1. Sample dependency graph. E4 is impacted directly by E2 and intermediately by E1 (via E2).



2.2. Change Detection Analysis

A Change Detection Analysis (CDA) computes all changes between two versions. The CDA is more commonly known under the name diff analysis: diff[2] is a software that compares two versions of a text and marks the differences between them. It is a standard tool for software versioning and configuration management. The CDA for models is similar but does not make a line-based comparison of the text. Such a line-based comparison between two models is not sufficient because it does not take into account the structure of a model. In contrast, the CDA recognizes the type of the changed artifact. For example, the CDA recognizes, if an interface has changed in the source code. In this case, more related artifacts need to be reviewed: the design, the tests and the like. If an interface is not affected by a change, much fewer artifacts need to be taken into account. Textual diff tools do not have these capabilities. Basically, the CDA detects three different kinds of changes (Altmanninger, 2009):

1. **Add:** Add a model element. A model element is a node in a model. It may have parent and child nodes.
2. **Delete:** Delete an element.
3. **Update:** Change of an element property. A property (attribute) belongs to an element.

2.3. Traceability Impact Analysis

The Traceability Impact Analysis (TIA) is an extension of the CDA. It takes all changed elements and finds all elements which are possibly affected by them. For this purpose, the TIA has to follow all dependencies (traces) of the changed elements. Thus, an element can be directly affected (by a changed element) or indirectly via several intermediate dependencies.

2.3.1. Dependency Matrix:

In order to perform a TIA, the dependency matrix has to be constructed. In Figure 2 the matrix D represents the dependency graph shown in Figure 1. In the matrix, each line lists the depending elements of a certain element. So, line 1 states that element E2 depends on element E1.

2.3.2. Reachability Matrix

The reachability matrix (see Figure 3) can be computed from the dependency matrix. It states all direct and indirect dependencies between two

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 2. Dependency matrix

$$D = \begin{array}{c} \text{Element} \\ E1 \\ E2 \\ E3 \\ E4 \\ E5 \end{array} \begin{array}{c} \begin{array}{ccccc} E1 & E2 & E3 & E4 & E5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

elements. The reachability matrix can be used to calculate all impacts of a changed element.

For example, in Figure 4, the reachability matrix is multiplied with a vector that indicates that element E2 has changed (the 1 in the second row). The outcome of the multiplication is the vector IS that lists all impacted elements. So, if element E2 changes, element E3, E4 and E5 are impacted.

2.4. Delta Evaluation

The *Common Criteria information statement on Reuse of Evaluation Results and Evidence* (Common Criteria, 2002) requires the following evidence for a reuse of evaluation results:

1. Product and supporting documentation
2. New Security Target
3. Original Security Target
4. Original Evaluation Technical Report

Figure 3..Reachability matrix

$$R = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

42 International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015

Figure 4. Calculation of impacted elements

$$IS = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

5. Original Common Criteria Certificate
6. Original Evaluation Work Packages

The document states: "The evaluation facility would be required to perform a delta analysis between the new security target and the original security target to determine the impact of changes on the analysis and evidence from the original evaluations" (p. 2) and: "However, the evaluation facility conducting the current evaluation should not have to repeat analysis previously conducted where requirements have not changed nor been impacted by changes in other requirements" (p. 2).

Summarizing, this approach states the necessary documentation and the need for a Change Impact Analysis. A more detailed process for this evaluation is not given and has to be defined together with the evaluation facility.

2.5. Security Model

The security model describes the properties and relations of the developer evidence. Basically, the developer evidence contains the Security Target (ST), the design documentation and the implementation representation. Such a model for a Common Criteria evaluation is very complex due to the very hierarchical composition of the Common Criteria artifacts. Therefore, we only describe the basic properties of such a model which are necessary for understanding the proposed evaluation approach (see Figure 5). The Security Target is the security claim and describes the operating context and how the TOE establishes its security. The ST is published and thus does not contain a detailed description of the design and implementation. The model of the ST contains the following main parts:

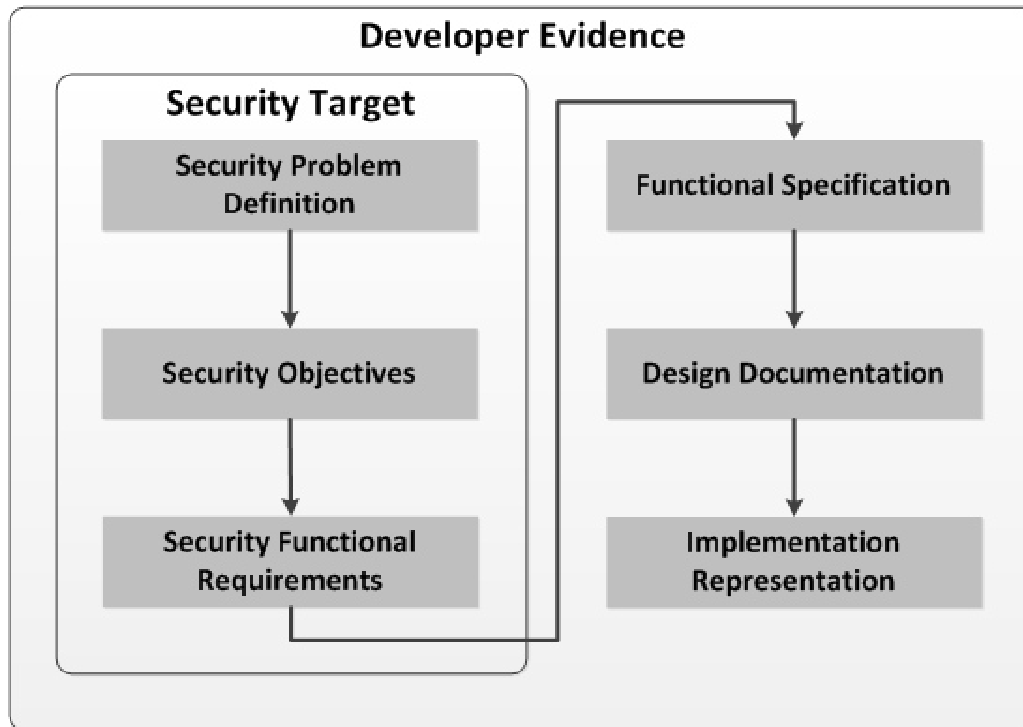
1. *The Security Problem Definition* describes the environment of the TOE: threats, assumptions about its operation and its security relevant assets.
2. *The Security Objectives* are high-level security goals which are then refined by the Security Functional Requirements.
3. *The Security Functional Requirements* are a set of well understood requirements in a security domain and have to be fulfilled by the TOE.

The remaining developer evidence basically contains the functional specification, design documentation and the implementation representation. The evaluation has to prove that the TOE fulfills the security claim of the ST. For this reason, mainly the following developer evidence is necessary:

1. *The Functional Specification* describes the interfaces through which security functionality can be invoked. In addition to the interface description, it has to provide a specification on how the security functional requirements are fulfilled.
2. *The Design Documentation* contains the functional specification and the design specification.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 5. Basic parts of the security model. The arrows indicate a fulfillment operation. For example, the functional specification must fulfil the security functional requirements.



3. *The Implementation Representation* is the source code of the implementation.

3. IMPACT ANALYSIS FOR SECURITY EVALUATION

In this section we will explain and compare two possible approaches incorporating an impact analysis for a security evaluation. The first approach is more formal and suits a delta certification approach. The second one is more informal and more appropriate for an agile security evaluation. The main difference between a delta and an agile evaluation is the frequency of the evaluation: the delta evaluation is based on a previous evaluation result which has been ascertained some time ago. Basically two product versions are compared. In the agile approach the evaluator receives updated documentation in each agile iteration (which lasts several weeks). Nevertheless, the certificate is only ascertained once, at the end of the product development lifecycle. The following approaches are based on the change impact analysis described by Bohner (2002).

3.1. Security Evaluation with Traceability Impact Analysis

In the following, we will describe the security evaluation with TIA with an example. The resulting sets are listed in Table 1. The graph in Figure 1 shows the structure of the two versions N and N+1.

The structure of the model does not alter during this increment. For the example, we assume, that the content of the node E1 has changed. Figure 6 indicates that the CDA detects all model differences between both versions. The output is the Starting Impact Set (SIS). In this case the SIS is the changed element E1:

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

44 International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015

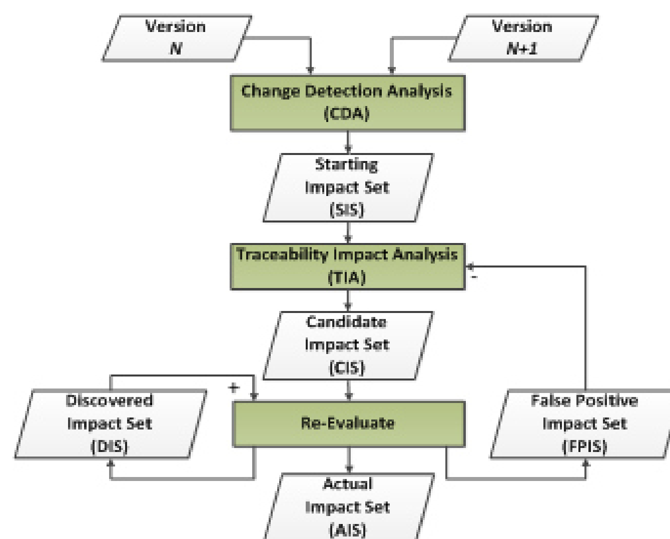
Table 1. Sample sets for the process with traceability impact analysis. The corresponding example is described in section 3.1.

	Set	Output Of
Starting Impact Set	{E1}	CDA
Candidate Impact Set	{E1,E2,E3,E4,E5}	TIA
Discovered Impact Set	{}	re-evaluation
False Positive Impact Set	{E3,E4,E5}	re-evaluation
Actual Impact Set	{E1,E2}	re-evaluation

SIS = {E1}. The TIA then resolves all traces and dependencies of the SIS and computes the so-called Candidate Impact Set (CIS). The CIS is the set of all possibly impacted artifacts. In this case, the CIS is: CIS = {E1,E2,E3,E4,E5}. Nevertheless, the CIS has to be manually re-evaluated:

1. If the candidate has a possible impact, the corresponding evaluation artifacts need to be updated. The set of all re-evaluated artifacts form the Actual Impact Set (AIS). The Actual Impact Set in this example is: AIS ={E1,E2}.
2. During the re-evaluation, new dependencies may be found: the Discovered Impact Set. Also the newly discovered impacts (DIS) have to be re-evaluated, again. In this example, the DIS is empty.
3. If the candidate has no impact, it is part of the False Positive Impact Set (FPIS). In this case, a further security evaluation of the corresponding artifacts can be omitted. The FPIS in this example is: FPIS = {E3,E4,E5}. Basically, the TIA predicts the AIS. This prediction is of course partly wrong, because new impacts are discovered (DIS) and some predicted impacts actually have no impact (FPIS). In the below

Figure 6. Re-evaluation process with Traceability Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts.



Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 2. Sample sets for the process with experiential impact analysis. The corresponding example is described in section 3.2.

	Set	Output Of
Candidate Impact Set	{E1}	CDA
Discovered Impact Set	{E2}	re-evaluation
False Positive Impact Set	{}	re-evaluation
Actual Impact Set	{E1,E2}	re-evaluation

equation, the relationships between the mentioned sets are stated:

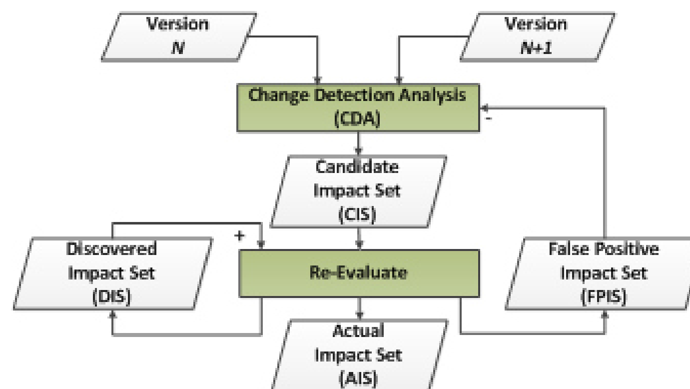
$$AIS = CIS + DIS - FPIS$$

3.2. Security Evaluation with Experiential Impact Analysis

We will describe the security evaluation with EIA with an example. The resulting sets are listed in Table 2. The graph in Figure 1 shows the structure of the two versions N and N+1. As shown in Figure 7, the process with Experiential Impact Analysis (EIA) takes as input two versions (N, N+1) of a model.

The CDA detects all model differences between both versions. In the example, the structure of the model between both versions remained unchanged. Only the content of node E1 has changed. The resulting Candidate Impact Set (CIS) is then: CIS = {E1}. As can be seen, the TIA is omitted: the re-evaluation is accomplished directly after the CDA. This leads to two consequences: first, the number of false positives is much lower (compare Table 1 and Table 2), because the CIS is smaller without a TIA, which also takes into account intermediate dependencies. Second, the DIS is larger because more impacts need to be detected during the re-evaluation (compare Table 1 and Table 2). In this example, the manually detected impacts are: DIS = {E1,E2}. This evaluation relies on Experiential Impact Analysis (EIA) which has been described by Kilpinen (2007). The EIA is based on expert design knowledge and review techniques: code inspections and walkthroughs. In any case, these techniques have to be performed during a security evaluation (Common Criteria, 2012).

Figure 7. Re-evaluation process with Experiential Impact Analysis. The Actual Impact Set is finally the re-evaluated set of software artifacts.



Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Table 3. Comparison of the processes with traceability impact analysis and experiential impact analysis

	TIA Process	EIA Process
DIS	SMALL	LARGE
FPIS	LARGE	SMALL
Presumed Experience	LOW	HIGH
Iteration Cycle	Months/Years	Weeks

It can be argued that the effort for EIA is expensive because it has to be conducted manually. Thus, a major issue in this process is keeping the cost of detecting the DIS low. This cost can be kept low, if the evaluators are involved early and regularly. Thus, they can build up knowledge of the possible impact between software artifacts. This ensures that the EIA can be accomplished efficiently if each evaluator works on the same but narrow set of software modules.

3.3. Comparing the Cost of the Approaches

It is an important observation here that the DIS and the FPIS are both discovered manually, whereas the CIS can be detected automatically with tools. The below equation indicates that the total effort can be estimated by summing up the effort for manually discovering new impacts finding false positives:

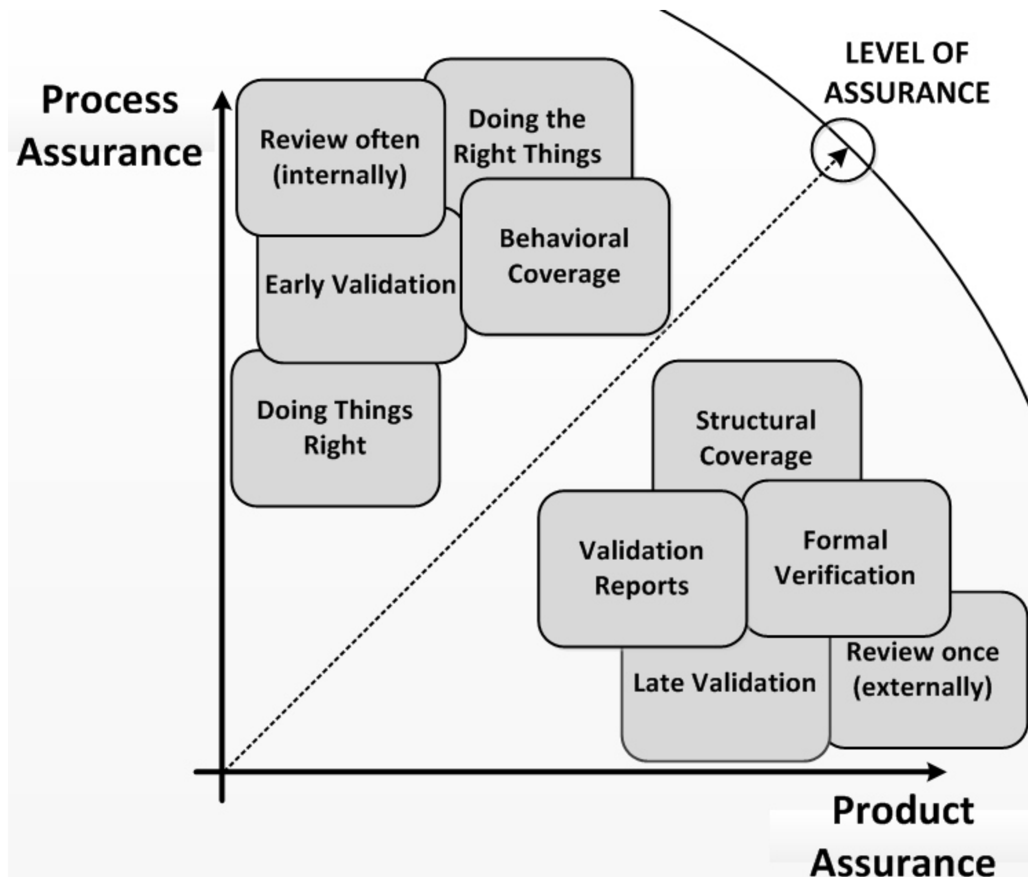
$$Effort_{total} = Effort(DIS) + Effort(FPIS)$$

This equation is a rough estimate of the total effort. However, it indicates, that the number of DIS and FPIS are a major factor for selecting an appropriate evaluation approach. Nevertheless, there are more parameters which influence such a selection. The most important factors are listed in Table 3. The evaluation with Traceability Impact Analysis (TIA) calculates a large Candidate Impact Set because the number of traces and dependencies is usually high (see Table 1). Thus, the number of direct plus indirect possible impacts soars. On the one hand, a large CIS reduces the likelihood of manually detected impacts (DIS) because all direct and indirect traces have still been resolved by the TIA (see Table 1). On the other hand, a large CIS will contain many false positives (see Table 1 and also Bohner, 2007) which need to be detected manually. For human beings it is easier to falsify a visible impact than to detect an invisible impact, if no experience exists. When there are long time intervals between two iterations, it can usually be presumed that there is no experience of the software under evaluation. Thus, an approach which detects many candidate impacts is more appropriate in this case.

The evaluation with Traceability Impact Analysis (TIA) calculates a large Candidate Impact Set because the number of traces and dependencies is usually high (see Table 1). Thus, the number of direct plus indirect possible impacts soars (see Table 1). On the one hand, a large CIS reduces the likelihood of manually detected impacts (DIS) because all direct and indirect traces have still been resolved by the TIA (see Table 1). On the other hand, a large CIS will contain many false positives (see Table 1 and also Bohner, 2002) which need to be detected manually. For human beings it is easier to falsify a visible impact than to detect an invisible impact, if no experience exists. For long time intervals between two iterations, usually no experience of the software under evaluation can be presumed. Thus, an approach which detects many candidate impacts is more appropriate in this case. The process with EIA detects more impacts via the re-evaluation (DIS, see Table 2). Fewer false positives are detected,

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 8. Assurance tactics are located in different places in the assurance space. It is indicated that the level of assurance is an aggregate of both dimensions.



because the number of candidate impacts is smaller without a TIA (see Table 2). This approach fits better for an iterative evaluation with short intervals between the re-evaluation activities. Its strength is to build up knowledge of the discovered impact set which makes the re-evaluation efficient. Although there are several iterations of re-evaluation, only one certificate is issued, in the end. In the following we will show a toolchain that can assist a process with EIA.

3.4. Mapping the Security Evaluation Approaches to Assurance Paradigms

In this section we describe two orthogonal assurance paradigms. As can be seen in Figure 8 they span an assurance space whose dimensions are: structural product assurance and behavioral process assurance. Both assurance paradigms are well established in other domains and evaluation standards, such as ISO 26262:

A safety argument that argues safety through direct appeal to features of the implemented item (e.g. the behavior of a timing watchdog) is often termed a product argument. A safety argument that argues safety through appeal

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

48 International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015

Table 4. Matched pairs of security evaluation tactics. A matched pair contains an assurance tactic from the process and also the product dimension.

Matched Pair	Process Assurance	Product Assurance
Review	Review Often (internally)	Review Once (externally)
Moment of Validation	Early Validation	Late Validation
Validation Relevance	Doing the Right Things	Validation Reports
Validation Depth	Doing Things Right	Formal Verification
Coverage	Behavioral Coverage	Structural Coverage

to features of the development and assessment process (e.g. the design notation adopted) is often termed a process argument. (ISO, 2011)

Standards in other domains are only relevant for this article to show that the idea of process and product assurance is established and not completely new. Therefore, we dared to cite here a standard which is not related to the rest of this work. For traditional product assurance only one iteration is necessary. Contrarily, process assurance can only be obtained with recurring activities. In the following, we will elaborate more on the two assurance paradigms.

1. **Product Assurance:** The product is evaluated against the claimed properties. Traditionally, the input of such an evaluation is a final product, which is not altered any more. The evaluation usually comprises verification and validation on all levels: requirements, architecture, design, source code and tests. This approach takes into account the static structure of a product in order to gain assurance. Examples of standards which focus on product assurance are the *Common Criteria* (Common Criteria, 2012) standard. The *IEC 61508* (Bell, 2006) and *ISO 26262* (ISO, 2011) standards incorporate functional safety requirements and thus ascertain assurance partly product-based.
2. **Process Assurance:** In addition to the product itself, a sound and appropriate development process is also assurance for the evaluated system properties. Thusly, continuous review, security design and testing are arguments for assurance. These activities are especially of value, if they are accomplished continuously. Process assurance takes into account the activities and the involved roles and their competencies. An example for process-based software assurance is the *DO-178-B* (Do, 1992) standard. The standards *IEC 61508* (Bell, 2006) and *ISO 26262* (ISO, 2011) demand process assurance in addition to the functional safety requirements. Several frameworks for describing job roles and relevant competencies have been described (Hilburn, 2013; Kreiner, 2013; Reiner, 2014). Process assurance is a behavioral strategy to ascertain quality.

Figure 8 shows the position of some assurance tactics in the assurance space. It can be seen that most of these tactics have corresponding counterparts in the other dimension. Figure 8 indicates that the level of assurance is an aggregate of process assurance and product assurance. In Table 4 we match these counterparts of both dimensions to corresponding pairs. In the following, we will dive into the details of those matched pairs.

- **Review:** An external review is traditionally accomplished only once at the end of product development because of the high cost of evaluation. In contrast, internal reviews are conducted often and can contribute to the product quality from the beginning. The involved roles, their competencies, the

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015 49

rigor and the frequency of review activities can be used as arguments for assurance.

- **Moment of Validation:** Early validation starts early in the product development and helps to mitigate risks regarding defects in architecture and design. The early validation comes along with additional effort for creating respective documentation. This additional effort may thwart an agile way of working to a certain degree. Late validation is accomplished when the product development is finished to show that it complies with certain standards. Late feedback from external evaluation can only be integrated with a high amount of effort.
- **Validation Relevance:** What is actually required for a certain level of assurance: tests, review documents or design documents? For product assurance the relevant documentation needs to be delivered. For example the *Common Criteria* (Common Criteria, 2012) standard does not require documentation for ATE_TDS (design description of the TOE) to achieve EAL 1. In order to get a certificate for a higher assurance level (EAL2 – EAL7) a documentation for ATE_TDS is imperative. For process assurance, it is important to accomplish the relevant tasks to achieve a relevant level of assurance. For example, the *Common Criteria* (Common Criteria, 2012) standard requires for EAL2 to EAL 7 evidence for ATE_FUN (functional testing). In this case, providing documents does not suffice. Relevant test have to be performed according to a test plan.
- **Validation Depth:** This matched pair indicates the depth and rigor of the assurance argumentation. Regarding process assurance this depth can be achieved by involving the appropriate roles. In addition, the validation depth can be increased by keeping to validation guidelines such as moderation of reviews and checking against lists. Validation depth in product assurance can be increased by formal verification in addition to informal verification.
- **Coverage:** Behavioral coverage denotes the frequency of assurance activities. In contrast, the coverage is structural in product assurance and takes into account requirements coverage, test coverage and the like.

In Table 5 we show how the matched pairs relate to the previously described security evaluation approaches (EIA and TIA). For each matched pair we show if the evaluation approach relates to process assurance or product assurance. In Table 6 we sum up the occurrences of process assurance and product assurance for the EIA and TIA evaluation approaches (see Table 5). It can be seen that the TIA perfectly matches the product assurance dimension. The iterative EIA evaluation approach refers to process assurance and also to product assurance. It can be seen that for the iterative approach (EIA) another dimension of assurance can be taken into consideration. Ideally, an additional dimension can provide more flexibility for certification. We discuss this issue in the next section in more detail.

3.5. Trade-Off in the Security Evaluation

In this section, we motivate the demand to make the security evaluation process of Common Criteria more flexible. First, we list some of the current problems of a Common Criteria evaluation when agile development processes are applied. Then, we propose a more flexible security certification approach, which would facilitate tremendously the development of secure systems in an agile way. The problems are located in the following area of tension:

1. The Architecture must be nearly completed before the Common Criteria Evaluation is started because otherwise the evaluators cannot compile evaluation reports. This seems to be a contradiction to the previously described agile security evaluation. However, the agile security evaluation is feasible if an implicit architecture still

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

50 International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015

Table 5. For each matched pair we describe how it relates to the EIA and the TIA evaluation approaches. A matched pair can relate to process assurance, product assurance or both of them.

Matched Pair	EIA	TIA
Review:	Often but Informal	Once and Formal
Process Assurance	X	
Product Assurance		X
Moment of Validation:	Early	Late
Process Assurance	X	
Product Assurance		X
Validation Relevance:	Validation Reports	Validation Reports
Process Assurance		
Product Assurance	X	X
Validation Depth:	Doing things right	Formal Verification
Process Assurance	X	
Product Assurance		X
Coverage:	Behavioral and Structural Coverage	Structural Coverage
Process Assurance	X	
Product Assurance	X	X

exists. Frequently, such implicit architectures exist in the form of a domain model. Such a model is usually well understood, if it focuses on a narrow domain. In such a narrow domain, the products resemble each other to a high degree and thus so does the architecture. Nevertheless, if a new product is developed in a completely new domain, the agile security evaluation may find its limits.

2. Security is often a system-wide property that cannot be isolated to specific components or parts of the system. Security is also a non-composable property (Mantel, 2002; Santen, 2002) in the following sense: if two or more components are secure that does not necessarily mean that the aggregation of those components is secure. Therefore, for any kind of security evaluation, all components and their connections (architecture) must be in place. However, in practice often a base software system exists and a custom-specific product release evolves out of this base system. In this case the agile security evaluation can provide a system-wide model at any iteration.
3. The agile way of working states that communication shall be over documentation. Frequent interaction with the customer and quickly building and adapting a running system is imperative. Regarding the quick and flexible development, development

Table 6. Affinity between evaluation approaches (EIA and TIA) and assurance dimensions

Affinity	EIA	TIA
Process Assurance	XXX	
Product Assurance	XX	XXXXX

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015 51

Table 7. Mapping between assurance families, assurance tactics, process assurance and product assurance

	ATE_FUN	ATE_COV	ATE_DPT
Assurance Tactic	*doing things right*	*structural coverage*	*structural coverage*
Process Assurance	X		
Product Assurance		X	X

and documentation in parallel seems to be counterproductive.

This area of tension was also discussed previously in (Raschke, 2014a) where the authors empirically evaluated the impact of agile development practices on effort for quality assurance. Their cost model basically shows that evolving software architecture causes continuous rework of the documentation for quality assurance, if it is done in parallel.

However, the additional effort for documentation in parallel does not necessarily increase time-to-market. This additional effort can be compensated with additional resources, such as a dedicated security team. Anyway, much of the documentation effort can be automated: the model can be synchronized with the source code, the tests, and other relevant artifacts. A considerable degree of the documentation can be generated from the model.

Summarizing, it can be seen that there are problems of agile security evaluation in general, but in many practical cases it is still applicable. In the case of an evolving architecture in a new domain the agile security evaluation has some limitations. In this case, we propose a solution in the following with the assurance class ATE (tests) as example. In order to simplify the example, we omit the assurance family ATE_IND (independent testing). First, we will map the assurance families of the assurance class ATE to assurance tactics. The assurance tactic helps to identify, whether an assurance family contributes to process or product assurance. Then, we will build up a goal model which represents the currently applied and inflexible way of certification. Finally, we will build a goal model, which represents the proposed and more flexible way of certification.

The assurance component ATE_FUN is described (Common Criteria, 2012) as follows: "Functional testing performed by the developer provides assurance that the tests in the test documentation are performed and documented correctly" (Part 3, p. 161). As we can see the phrase "performed [...] correctly" relates to the assurance tactic *doing things right*. As can be seen in Table7, this assurance tactic provides process assurance.

The "ATE_COV addresses the rigour with which the functional specification is tested" (Common Criteria, 2012, Part 3, p. 153). The applied assurance tactic here is *structural coverage* which contributes to product assurance.

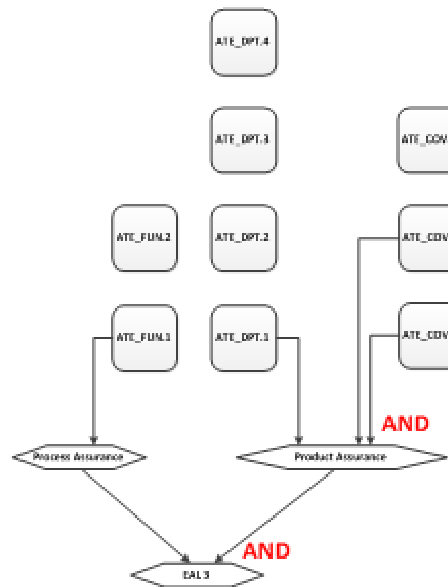
The ATE_DPT assurance family is described as follows: "The components in this family deal with the level of detail to which the TSF is tested by the developer. Testing of the TSF is based upon increasing depth of information derived from additional design representations and descriptions (TOE design, implementation representation, and security architecture description)" (Common Criteria, 2012, Part 3, p. 157).

Also this assurance family is an analysis of structural coverage: it analyses the testing depth with additional representations and descriptions. Table 7 summarizes the assurance families, the applied assurance tactics and whether the assurance family provides process or product assurance.

The hard goal model in Figure 9 describes the currently applied aggregation of assurance

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 9. The current Common Criteria evaluation does not leave much room for flexibility. The hard goals are only fulfilled, if all (AND) inputs are true.



components. In order to achieve EAL 3 for the ATE class (without the omitted ATE_IND family), the following assurance components need to be fulfilled: ATE_FUN.1, ATE_DPT.1, ATE_COV.1 and ATE_COV.2 (Common Criteria, 2012, part 3, p. 31). All these required components are connected to the hard goals *process assurance* or *product assurance*. Assurance components which are not required for EAL 3, are not connected to any hard goal.

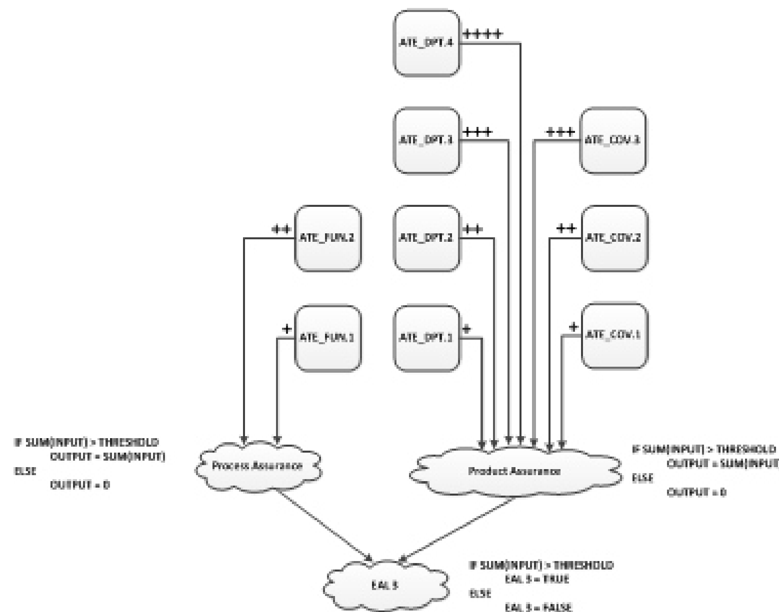
The soft goal model shown in Figure 10 indicates how the aggregation of assurance components can be made more flexible. Each assurance component contributes to a soft goal, which is indicated by the plus signs. If the sum of all inputs is above a defined threshold, the output is either the sum of all inputs (*process assurance* or *product assurance*) or TRUE (*EAL 3*). If the sum of all inputs is below this threshold, the output is either zero of FALSE. In the proposed way, a higher process assurance can compensate for a lower product assurance and *vice versa*.

3.6. How Model-Merge and Model-Diff Support Process Assurance

In Figure 11, we provide a use case which demonstrates, how model-merge and model-diff support process assurance. This use case shall provide an impression of how these algorithms can leverage argumentation for process assurance. In Figure 11 the workspace represents the local development space of the software developer. The repository denotes the common basis for the whole software project. The developer is allowed to change an interface locally in the workspace. The changed interface does not affect the common basis until it is checked into the repository, which shall contain code of high quality. If the developer commits the changed interface to the repository, she triggers the merge and review sub-process. In this sub-process, the security architect obtains a list of all interface changes. The architect reviews the changes and accepts or rejects them. A review protocol is added to the repository. The described sub-process can be refined in order to improve the associated assurance argument: the review could be done by more than one security architect. In

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 10. A more flexible evaluation of assurance levels can be modelled with soft goals. The plus signs next to the assurance components indicate the contribution to a soft goal. The output of the soft goals depends on the sum of its inputs. If this sum is below a specified threshold, the output is zero or FALSE. It the sum is above this threshold, the output is either the sum of all inputs or TRUE.



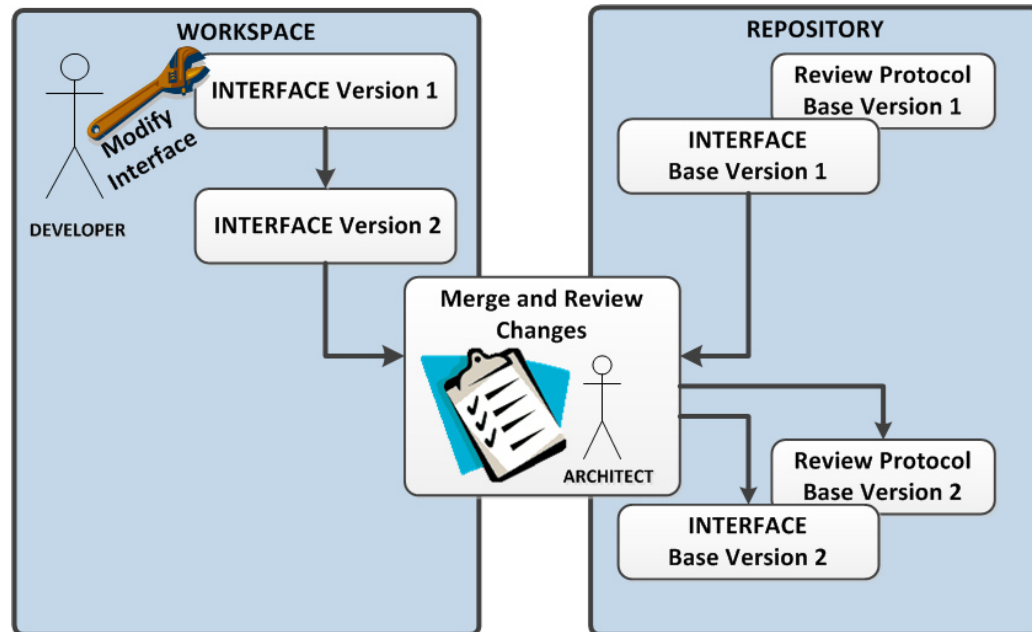
addition, a dedicated moderator and a formal review procedure can improve the assurance argument. The review could also follow a formal procedure. All this information is then stored in the review protocol. The accepted changes are then merged automatically to the base version of the interface. This approach supports agile development methods, because the architecture can be developed in a bottom-up process: the developer simply changes the interface and subsequent review activities are triggered by this change. Contrarily, in a traditional waterfall development process the architecture would be planned before the implementation is started upon. If tools and methods which support architecture evolution are missing, late changes are expensive.

4. TOOL SUPPORT FOR AUTOMATION OF THE CHANGE DETECTION ANALYSIS

In order to support an agile evaluation process with tooling, we implemented a Change Detection Analysis. The CDA for the security model takes as input two versions and delivers a list of all differences. For example, the CDA lists all changed source code and test entities. This has the benefit that the unchanged parts do not have to be taken into account for re-evaluation. Basically, such a CDA for models is not new. Our contribution here is to show how such a tool can be built with existing open source software. Moreover, we faced some issues because we work with large models. These issues suggest a careful design of the model structure: each node within a model shall be tagged with a unique identifier.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 11.This use case illustrates how model merge and model diff contribute to an assurance argument. This argument is supported by the appropriate role (architect) who performs a process (review).



4.1. Basic Requirements for the Tooling

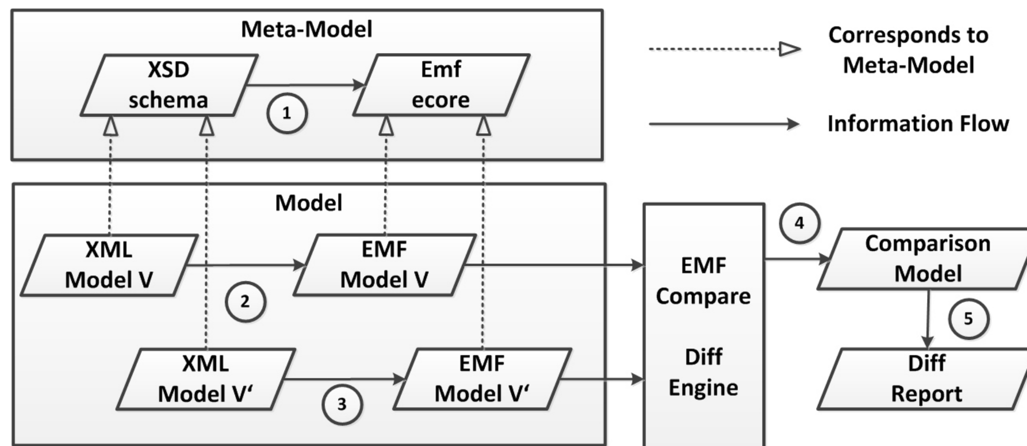
First of all, the diff engine shall work as a standalone and independently of other proprietary tools. Unfortunately, many modeling tools do not support sophisticated model-diff engines, at present. Such a diff engine shall accept a standard format as input. The produced diff report shall be in a format which can be easily read by humans and software. Second, we must handle large models with several thousands of artifacts. Comparing them with an inefficient algorithm is not feasible. Thus, we strive to utilize a generic diff engine which can be extended by customizable algorithms. Third, the diff engine shall be able to deal with different meta-models without much refactoring of the algorithms.

4.2. Basic Tool and Format Overview

The technical process for the CDA is shown in Figure 12. First (1), the XSD schema[3] is imported to the EMF[4] framework and is then an EMF ecore meta-model. Such a meta-model describes the structure and properties of an EMF model. Then (2 and 3) the XML models are imported to the EMF framework and are then represented as EMF models. The EMF compare engine takes the two models as input and computes (4) a comparison model. The comparison model can be used to generate a customized diff report. This diff report is then stored in a file and contains all add, delete and update operations and the unique identifiers of the changed elements. The diff report can be accessed from other tools and GUI's.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Figure 12. Process and tooling for a change detection analysis



4.3. EMF Compare Customization

We identified *EMF compare*[5] as a highly customizable and extensible tool. It is open source and can be embedded in java applications. EMF compare can be used without Eclipse in a standalone fashion. For this purpose only the appropriate EMF compare and EMF ecore libraries need to be included to a java application. Actually, we experienced some runtime issues because we were differencing large models with many differences. This is due to the standard match algorithm of EMF compare. A match engine is part of the difference engine and determines which elements of the old and new model correspond to each other. Basically, for this purpose, a similarity metric is computed for each pair of elements. The pair with the highest similarity metric is then a matched pair. Apparently, this algorithm performs poorly for models with a high number of elements. Although some optimizations of the pairwise comparison have been implemented (Brun, 2008; Wenzel, 2014) they perform well under the assumption that not many differences exist between two versions of a model. This assumption is not true in our case because we compare large models at a time interval of several weeks (an agile iteration). Thus, we decided to implement a match algorithm based on a comparison of unique identifiers which is much faster because no pairwise comparison is needed. EMF compare provides the possibility to utilize such a matching algorithm.

5. RELATED WORK

The Assert4SOA[6] project is concerned with the issue of certifying software which is composed of several services. It is difficult to ascertain trust in heterogeneous software in a software ecosystem. The traditional certifying approaches do not take into account such heterogeneous systems. This project deals mainly with the issue of composing evidence for security for such systems. An approach for Common Criteria Certification of such systems is taken into account.

The EURO-MILS[7] project strives to exploit virtualization techniques in order to enable a separation between different levels of security for networked embedded systems. If an appropriate separation between highly secure software and low security software is possible, only the highly secure software needs to be certified which makes the approach more cost effective. In addition to the virtualization, the communication between the separated software entities also needs to be taken into account. The SecureChange[8] project deals with supporting security evaluation during the evolution of the

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

software system at all levels of the software development process. A key aim of this project is to focus on the delta between software releases in order to concentrate only on the software artifacts that have changed. Tools and processes have been developed in order to meet the mentioned objectives.

Hayley (Haley, 2008) describes a framework for security requirements engineering. This framework consists of a meta-model which mainly contains goals, requirements and traces to architecture. Additionally, the framework describes a process for the elicitation of security requirements in several iterations. Additionally, he provides a notation of security requirement satisfaction arguments. The framework is applied in an industrial case study. An iterative approach for security requirements engineering is described in this paper. However, the authors do not investigate the issues that emerge in large-scale and complex software systems. The authors mention the application of a graphical model for the security assurance arguments. In a complex industrial software project, many engineers would need to work on the same model, concurrently. This would imply that changing dependencies and resulting change impacts happen more frequently than in pre-defined iteration cycles. In this paper, we address these problems with model-evolution approaches. In addition, we show that those model-evolution techniques can be exploited to construct a process argument for security assurance.

Jürjens (2011) extends the UMLSec (UML) profile with annotations, so that model evolutions can be registered in the model. The original UMLSec extension can verify the model for specific security properties. The UMLSec can use change annotated models and compute a difference model from it. The verification is then applied to the difference model. It has been shown that such an incremental verification of security properties is much more efficient (in terms of calculation time) than a full model verification.

The combination of product and process argument for a safety assurance argument (safety case) has been described by (Habli, 2006; Habli, 2007). The authors state that even the best product argument lacks assurance when a process argument is omitted. They show how both arguments can be combined to achieve a better argumentation with applying the so-called *Goal Structuring Notation*. Hawkins (2010) describes a structured approach for providing safety evidence. He emphasizes the need to determine the capabilities of the resources which are involved in the processes. Only appropriate processes which involve resources with appropriate capabilities form a sound assurance argument.

Frameworks for assessing software process improvement have been well established in the past (Team, 2002; Messnarz, 2009). It is interesting to see that recently there is an emerging trend for the certification of competencies in addition to processes. This trend stems from the need to apply process improvement for software assurance purposes. The *Software Engineering Institute*[9], which issued the CMMI standard (Team, 2002), recently released a *Software Assurance Competency Model* (Hilburn, 2013). This competency model strives to support companies in building software of high quality and security. The *European Qualification Framework*[10] (Reiner, 2014) was established to certify training courses, the corresponding job roles and examinations. The AQUA[11] project (Kreiner, 2013) is an initiative of car manufacturers and universities to establish a set of training courses and job roles which are ECQA certified. The rationale behind the AQUA project is to provide an assessment framework for the competencies of developers involved in a safety assurance process (ISO, 2011; Messnarz, 2009).

6. CONCLUSION

Today's software systems are developed with changing requirements regarding functionality and security. Moreover, fast delivery is an important issue which is impacted by the duration of development and security evaluation. We deduced a conceptual background for an agile security evaluation which allows

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

the management of changing requirements and fast time-to-market constraints. Moreover, this evaluation approach provides early feedback regarding the security concept of a product and thus avoids late and costly refactoring. We described a process which utilizes Change Detection Analysis and Experiential Change Impact analysis to improve such an iterative approach. In addition, we described an implementation of a change detection analysis for model-based security requirements and design.

Regarding the Common Criteria evaluation, we found some mismatches with agile development paradigms. In order to enable a better support for agile software development we propose to extend the product-focused evaluation philosophy of Common Criteria by a process assurance dimension. This second dimension would allow the trade-off of assurance components in a more modular way in order to tailor the security evaluation to the applied development processes.

We discussed that there are matched pairs of assurance tactics. Future work could focus on finding such matched pairs. It would be important to evaluate their correspondence. Further, it would be necessary to find rules of aggregation for assurance components. Such rules of aggregation need to be evaluated with the following respect: do different clusters of assurance components provide the same assurance of aggregation? If yes, the aggregation rules and matched pairs are appropriate. If no, they have to be adjusted to deliver the same level of assurance for different compositions.

ACKNOWLEDGEMENT

Project partners are NXP Semiconductors Austria GmbH and TU Graz. The project is funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FIT-IT contract FFG 832171. The authors would like to thank pure::systems GmbH for support.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

REFERENCES

Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., & Wimmer, M. (2009, October). Why model versioning research is needed!? an experience report. *In Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS* (Vol. 9).

Bell, R. (2006, April). Introduction to IEC 61508. *In Proceedings of the 10th Australian workshop on Safety critical systems and software*-Volume 55 (pp. 3-12). Australian Computer Society, Inc.

Beznosov, K., & Kruchten, P. (2004, September). Towards agile security assurance. *In Proceedings of the 2004 workshop on New security paradigms* (pp. 47-54). ACM.

Bohner, S. A. (2002, December). Extending software change impact analysis into cots components. *In Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE* (pp. 175-182). IEEE.

Brun, C., & Pierantonio, A. (2008). Model differences in the eclipse modeling framework. UPGRADE. *The European Journal for the Informatics Professional*, *9*(2), 29–34.

Cockburn, A. (2006). *Agile software development: the cooperative game*. Pearson Education.

Cohn, M. (2010). *Succeeding with agile: software development using Scrum*. Pearson Education.

Common Criteria. (2002). *Reuse of evaluation results and evidence*.

Common Criteria. (2012). *Common criteria for information technology security evaluation* (Part 1-3, Version 3.1., Revision 4).

Do, R. T. C. A. (1992). *178B: Software considerations in airborne systems and equipment certification*.

Habli, I., & Kelly, T. (2006). Process and product certification arguments: Getting the balance right. *ACM SIGBED Review*, *3*(4), 1–8. doi:10.1145/1183088.1183090

Habli, I., & Kelly, T. (2007). Achieving integrated process and product safety arguments. In *The Safety of Systems* (pp. 55–68). Springer London. doi:10.1007/978-1-84628-806-7_4

Haley, C. B., Laney, R., Moffett, J. D., & Nuseibeh, B. (2008). Security requirements engineering: A framework for representation and analysis. Software Engineering. *IEEE Transactions on*, *34*(1), 133–153.

Hawkins, R., & Kelly, T. (2010, October). A structured approach to selecting and justifying software safety evidence. *In System Safety 2010, 5th IET International Conference on* (pp. 1-6). IET.

Hilburn, T. B., Ardis, M., Johnson, G., Kornecki, A. J., & Mead, N. (2013). *Software assurance competency model.*

ISO, C. (2011). *26262, road vehicles–Functional safety*. ISO Standard.

Jürjens, J., Marchal, L., Ochoa, M., & Schmidt, H. (2011). Incremental security verification for evolving UMLsec models. In *Modelling Foundations and Applications* (pp. 52–68). Springer Berlin Heidelberg. doi:10.1007/978-3-642-21470-7_5

Kilpinen, M. S., Eckert, C. M., & Clarkson, P. J. (2007, March). The emergence of change at the interface of system and embedded software design. *In Conference on Systems Engineering Research*, Hoboeken, NJ.

Kreiner, C., Messnarz, R., Riel, A., Ekert, D., Langgner, M., Theisens, D., & Reiner, M. (2013). Automotive Knowledge Alliance AQUA–Integrating Automotive SPICE, Six Sigma, and Functional Safety. In *Systems, Software and Services Process Improvement* (pp. 333–344). Springer Berlin Heidelberg. doi:10.1007/978-3-642-39179-8_30

Mantel, H. (2002). On the composition of secure systems. *In Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on* (pp. 88-101). IEEE.

Messnarz, R., Ross, H. L., Habel, S., König, F., Koundoussi, A., Unterrreitmayer, J., & Ekert, D. (2009). Integrated Automotive SPICE and safety assessments. *Software Process Improvement and Practice*, *14*(5), 279–288. doi:10.1002/spip.429

Raschke, W., Zilli, M., Baumgartner, P., Loinig, J., Steger, C., & Kreiner, C. (2014b). Supporting evolving security models for an agile security evaluation. In *Evolving Security and Privacy Requirements Engineering (ESPRE), 2014 IEEE 1st Workshop on* (pp. 31-36). IEEE. doi:10.1109/ESPRE.2014.6890525

Raschke, W., Zilli, M., Loinig, J., Weiss, R., Steger, C., & Kreiner, C. (2014a). Where does all this waste come from. *In Industrial Proceedings of the 18th EuroSPI Conference* (pp. 3.1-3.10). DELTA, Denmark.

Reiner, M., Sauberer, G., & Messnarz, R. (2014). European Certification and Qualification Association – Developments in Europe and World Wide. *In Industrial Proceedings of the 18th EuroSPI Conference* (pp. 3.1-3.10). DELTA, Denmark.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

International Journal of Secure Software Engineering, 6(1), 37-60, January-March 2015 59

Santen, T., Heisel, M., & Pfitzmann, A. (2002). Confidentiality-preserving refinement is compositional—sometimes. In *Computer Security—ESORICS 2002* (pp. 194–211). Springer Berlin Heidelberg. doi:10.1007/3-540-45853-0_12

Team, C. P. (2002). *Capability maturity model® integration (CMMI SM),* version 1.1. Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA, Tech. Rep. SEI-2002-TR-012.

Wenzel, S. (2014). Unique identification of elements in evolving software models. *Software & Systems Modeling*, *13*(2), 679–711. doi:10.1007/s10270-012-0311-7

Wiegers, K. E. (2002). *Peer reviews in software: A practical guide*. Boston: Addison-Wesley.

ENDNOTES

1. http://www.commoncriteriaportal.org/products/, visited in November 2014
2. http://www.gnu.org/software/diffutils/
3. http://www.w3.org/XML/Schema.html
4. http://www.eclipse.org/modeling/emf/
5. http://www.eclipse.org/emf/compare/
6. http://www.assert4soa.eu/
7. http://www.euromils.eu/
8. http://www.securechange.eu/
9. http://www.sei.cmu.edu
10. http://www.ecqa.org
11. http://automotive-knowledge-alliance.eu

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Wolfgang Raschke completed his studies in telematics in 2012. Since February 2012 he works as a PhD student at the Graz University of Technology. His current research interests are: software product lines for secure systems, innovation and evolution of software and software models.

Massimiliano Zilli received his Master's degree in Electronic Engineering at the University of Udine in 2007. From 2008 to 2012 he worked as a software developer for embedded systems. Since February 2012 he works as a PhD-Student at the Graz University of Technology. His research focuses on optimization techniques for embedded systems.

Philip Baumgartner completed his studies in telematics in 2014. Currently, he works as security engineer at NXP Semiconductors Austria. His current research interests are: software product lines, requirements engineering and secure systems.

Johannes Loinig received the Dr.techn. degree (PhD) in electrical engineering with focus on Secure Embedded Systems from Graz University of Technology in 2012. Currently he works as software and security architect in several projects at NXP Semiconductors Austria and is in the lead of several research projects with focus on HW/SW codesign and information security.

Christian Steger received the Dipl.-Ing. degree (M.Sc.) 1990 and the Dr.techn. degree (PhD) in electrical engineering from Graz, University of Technology, Austria, in 1995, respectively. He is key researcher at the Virtual Vehicle Competence Center (VIF, COMET K2) in Graz, Austria. From 1989 to 1991 he was software trainer and consultant at SPC Computer Training GmbH, Vienna. Since 1992, he has been Assistant Professor at the Institute for Technical Informatics, Graz University of Technology. He heads the HW/SW codesign group at the Institute for Technical Informatics. His research interests include embedded systems, HW/SW codesign, HW/SW coverification, SOC, power awareness, smart cards UHF RFID systems, multi-DSPs.

Christian Kreinergraduated and received a PhD degree in Electrical Engineering from Graz University of Technology in 1991 and 1999, respectively. From 1999 to 2007 he served as the head of the R&D department at Salomon Automation, Austria, focusing on software architecture, technologies, and processes for logistics software systems. He was in charge to establish a company-wide software product line development process and headed the product development team. During that time, he lead and coordinated a long-term research programme together with the Institute for Technical Informatics of Graz University of Technology. There, he currently leads the Industrial Informatics and Model-based Architectures group. His research interests include systems and and software engineering, software technology, and process improvement.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Bibliography

[1] J. A. Kupsch and B. P. Miller, "Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed?," *Continuous Software Assurance Marketplace*, vol. 22, 2014.

[2] Ericsson, *Ericsson Mobility Report*. November 2013.

[3] C. Christensen and M. Raynor, *The innovator's solution: Creating and sustaining successful growth*. Harvard Business Review Press, 2013.

[4] *Virtual Machine Specification*. Java Card Platform, Version 3.0.1, Classic Edition, 2009. Sun Microsystems.

[5] C. Steger and J. Loinig, "DAVID - Design-Flow für Java Betriebssysteme auf Low-End Smart Cards," tech. rep., February 2012. Internal.

[6] M. Zilli, W. Raschke, J. Loinig, R. Weiss, and C. Steger, "On the dictionary compression for Java card environment," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, pp. 68–76, ACM, 2013.

[7] M. Zilli, W. Raschke, R. Weiss, C. Steger, and J. Loinig, "A light-weight compression method for Java card technology," *SIGBED Review*, vol. 11, no. 4, pp. 13–18, 2014.

[8] M. Zilli, W. Raschke, R. Weiss, J. Loinig, and C. Steger, "Instruction Folding Compression for Java Card Runtime Environment," in *Proceedings of the 17th Euromicro Conference on Digital System Design (DSD'14)*, pp. 228–235, IEEE, 2014.

[9] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, vol. 24, no. 12, pp. 21–29, 1999.

[10] S. Mauw and M. Oostdijk, "Foundations of attack trees," in *Information Security and Cryptology (ICISC'05)*, pp. 186–198, Springer, 2006.

[11] T. Tidwell, R. Larson, K. Fitch, and J. Hale, "Modeling internet attacks," in *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, vol. 59, 2001.

[12] J. McDermott and C. Fox, "Using abuse case models for security requirements analysis," in *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC'99)*, pp. 55–64, IEEE, 1999.

[13] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements Engineering*, vol. 10, no. 1, pp. 34–44, 2005.

[14] J. McDermott, "Abuse-case-based assurance arguments," in *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC'01)*, pp. 366–374, IEEE, 2001.

[15] P. Hope, G. McGraw, and A. I. Antón, "Misuse and abuse cases: Getting past the positive," *Security & Privacy, IEEE*, vol. 2, no. 3, pp. 90–92, 2004.

[16] H. Mouratidis and P. Giorgini, "Secure tropos: a security-oriented extension of the tropos methodology," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 02, pp. 285–309, 2007.

[17] P. Giorgini, H. Mouratidis, and N. Zannone, "Modelling security and trust with secure tropos," *Integrating Security and Software Engineering: Advances and Future Vision*, pp. 160–189, 2006.

[18] F. Massacci, J. Mylopoulos, and N. Zannone, "Computer-aided support for secure tropos," *Automated Software Engineering*, vol. 14, no. 3, pp. 341–364, 2007.

[19] J. Jürjens, "UMLsec: Extending uml for secure systems development," in *UML 2002—The Unified Modeling Language*, pp. 412–425, Springer, 2002.

[20] J. Jürjens, "Towards development of secure systems using UMLsec," in *Fundamental Approaches to Software Engineering*, pp. 187–200, Springer, 2001.

[21] B. Best, J. Jurjens, and B. Nuseibeh, "Model-based security engineering of distributed information systems using UMLsec," in *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pp. 581–590, IEEE, 2007.

[22] I. Côté, M. Heisel, H. Schmidt, and D. Hatebur, "UML4PF-a tool for problem-oriented requirements analysis.," in *Proceedings of the 19th IEEE International Requirements Engineering Conference (RE'11)*, pp. 349–350, 2011.

[23] K. Beckers, S. Faßbender, D. Hatebur, M. Heisel, and I. Côté, "Common Criteria compliant software development (CC-CASD)," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1298–1304, ACM, 2013.

[24] F. Moyano, C. Fernández-Gago, K. Beckers, and M. Heisel, "Enhancing Problem Frames with Trust and Reputation for Analyzing Smart Grid Security Requirements," in *Smart Grid Security*, pp. 166–180, Springer, 2014.

[25] C. B. Haley, J. D. Moffett, R. Laney, and B. Nuseibeh, "A framework for security requirements engineering," in *Proceedings of the 2006 International Workshop on Software Engineering for Secure Systems*, pp. 35–42, ACM, 2006.

[26] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[27] A. Van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *Proceedings of the 26th International Conference on Software Engineering*, pp. 148–157, IEEE Computer Society, 2004.

[28] J. Jürjens, L. Marchal, M. Ochoa, and H. Schmidt, "Incremental security verification for evolving UMLsec models," in *Modelling Foundations and Applications*, pp. 52–68, Springer, 2011.

[29] I. Habli and T. Kelly, "Process and product certification arguments: getting the balance right," *ACM SIGBED Review*, vol. 3, no. 4, pp. 1–8, 2006.

[30] *Common Criteria. Common Criteria for Information Technology Security Evaluation - Part 1-3. Version 3.1 Revision 3 Final (July 2009).*

[31] *ISO, C. (2013). 27001, Information technology –Security techniques – Information security management systems – Requirements. ISO Standard.*

[32] *ISO, C. (2011). 26262, Road vehicles–Functional safety. ISO Standard.*

[33] R. Bell, "Introduction to IEC 61508," in *Proceedings of the 10th Australian workshop on Safety Critical Systems and Software. Volume 55*, pp. 3–12, Australian Computer Society, Inc., 2006.

[34] L. A. Johnson, "DO-178B, Software considerations in airborne systems and equipment certification," *Crosstalk, October*, 1998.

[35] R. Messnarz, H.-L. Ross, S. Habel, F. König, A. Koundoussi, J. Unterrreitmayer, and D. Ekert, "Integrated Automotive SPICE and Safety Assessments," *Software Process: Improvement and Practice*, vol. 14, no. 5, pp. 279–288, 2009.

[36] C. P. Team, "Capability Maturity Model ® Integration (CMMI SM), Version 1.1," tech. rep., 2002.

[37] T. B. Hilburn, M. Ardis, G. Johnson, A. J. Kornecki, and N. Mead, "Software Assurance Competency Model," tech. rep., 2013.

[38] M. Reiner, G. Sauberer, and I. Richard Messnarz, "European Certification and Qualification Association-Developments in Europe and World Wide," tech. rep., 2014.

[39] I. Habli and T. Kelly, "Achieving integrated process and product safety arguments," in *The Safety of Systems*, pp. 55–68, Springer, 2007.

[40] S. E. Toulmin, *The uses of argument.* Cambridge University Press, 2003.

[41] R. Hawkins and T. Kelly, "A structured approach to selecting and justifying software safety evidence," in *Proceedings of the 5th IET International Conference on System Safety*, pp. 1–6, IET, 2010.

[42] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Patterns for Hardware-Independent Development for Embedded Systems," in *Proceedings of the 20th European Conference on Pattern Languages of Programs*, ACM, 2015. in print.

[43] V. Handziski, J. Polastre, J. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible hardware abstraction for wireless sensor networks," in *Proceedings of the Second European Workshop onWireless Sensor Networks*, pp. 145–157, IEEE.

[44] J. Grenning, "Applying test driven development to embedded software," *Instrumentation & Measurement Magazine, IEEE*, vol. 10, no. 6, pp. 20–25, 2007.

[45] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[46] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Communications of the ACM*, vol. 42, no. 1, pp. 94–97.

[47] W. Raschke, M. Zilli, P. Baumgartner, J. Loinig, C. Steger, and C. Kreiner, "Balancing Product and Process Assurance for Evolving Security Systems," *International Journal of Secure Software Engineering (IJSSE)*, vol. 6, no. 1, pp. 47–75, 2015.

[48] A. D. Sinnhofer, W. Raschke, C. Steger, and C. Kreiner, "Evaluation paradigm selection according to Common Criteria for an incremental product development," in *International Workshop on MILS: Architecture and Assurance for Secure Systems*, 2015. http://mils-workshop.euromils.eu/ [Online; accessed 24-March-2015].

[49] W. Raschke, M. Zilli, P. Baumgartner, J. Loinig, C. Steger, and C. Kreiner, "Supporting evolving security models for an agile security evaluation," in *Proceedigns of the 1st Workshop on Evolving Security and Privacy Requirements Engineering (ESPRE'14)*, pp. 31–36, IEEE, 2014.

[50] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Proceedings of the 5th IEEE International Symposium on Requirements Engineering.*, pp. 249–262, IEEE, 2001.

[51] W. Raschke, M. Zilli, , J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Where does all this waste come from?," in *Industrial Proceedings of the 21st EuroSPI Conference*, pp. 3.1–3.10, DELTA, Denmark, 2014.

[52] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Patterns of Software Modeling," in *On the Move to Meaningful Internet Systems: OTM 2014 Workshops*, pp. 428–437, Springer, 2014.

[53] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Test-Driven Migration Towards a Hardware-Abstracted Platform," in *Proceedings of the 2014 International Conference on Pervasive and Embedded Computing and Communicaton Systems, PECCS 2015*, 2015.

[54] M. Lackner, R. Berlach, W. Raschke, R. Weiss, and C. Steger, "A defensive virtual machine layer to counteract fault attacks on java cards," in *Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems*, pp. 82–97, Springer, 2013.

[55] W. Raschke, M. Zilli, J. Loinig, R. Weiss, C. Steger, and C. Kreiner, "Embedding research in the industrial field: a case of a transition to a software product line," in *Proceedings of the 2014 International Workshop on Long-Term Industrial Collaboration on Software Engineering*, pp. 3–8, ACM, 2014.

[56] *CESeCore TOE Design (ADV TDS) - Version 1.1.2* . https://www.cesecore.eu/ [Online; accessed 24-March-2015].

[57] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pp. 317–322, IEEE, 2009.

[58] T. Gilb, *Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage.* Butterworth-Heinemann, 2005.

[59] T. Kelly and R. Weaver, "The goal structuring notation–a safety argument notation," in *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, Citeseer, 2004.