

Linköping Studies in Science and Technology

Thesis No. 1353

# **A Model and Implementation of a Security Plug-in for the Software Life Cycle**

by

**Shanai Ardi**



Submitted to Linköping Institute of Technology at Linköping University in partial  
fulfilment of the requirements for the degree of Licentiate of Engineering

Department of Computer and Information Science  
Linköpings universitet  
SE-581 83 Linköping, Sweden

Linköping 2008



# **A Model and Implementation of a Security Plug-in for the Software Life Cycle**

by

Shanai Ardi

March 2008

ISBN 978-91-7393-956-0

Linköping Studies in Science and Technology

Thesis No. 1353

ISSN 0280-7971

LiU-Tek-Lic-2008:11

## **ABSTRACT**

Currently, security is frequently considered late in software life cycle. It is often bolted on late in development, or even during deployment or maintenance, through activities such as add-on security software and penetration-and-patch maintenance. Even if software developers aim to incorporate security into their products from the beginning of the software life cycle, they face an exhaustive amount of ad hoc unstructured information without any practical guidance on how and why this information should be used and what the costs and benefits of using it are. This is due to a lack of structured methods.

In this thesis we present a model for secure software development and implementation of a security plug-in that deploys this model in software life cycle. The model is a structured unified process, named S<sup>3</sup>P (Sustainable Software Security Process) and is designed to be easily adaptable to any software development process. S<sup>3</sup>P provides the formalism required to identify the causes of vulnerabilities and the mitigation techniques that address these causes to prevent vulnerabilities. We present a prototype of the security plug-in implemented for the OpenUP/Basic development process in Eclipse Process Framework. We also present the results of the evaluation of this plug-in. The work in this thesis is a first step towards a general framework for introducing security into the software life cycle and to support software process improvements to prevent recurrence of software vulnerabilities.

*This work has been supported by Vinnova (Swedish Agency for Innovation Systems) and CUGS (Swedish National Graduate School in Computer Science).*



## Abstract

---

Currently, security is frequently considered late in software life cycle. It is often bolted on late in development, or even during deployment or maintenance, through activities such as add-on security software and penetration-and-patch maintenance. Even if software developers aim to incorporate security into their products from the beginning of the software life cycle, they face an exhaustive amount of ad hoc unstructured information without any practical guidance on how and why this information should be used and what the costs and benefits of using it are. This is due to a lack of structured methods.

In this thesis we present a model for secure software development and implementation of a security plug-in that deploys this model in software life cycle. The model is a structured unified process, named S<sup>3</sup>P (Sustainable Software Security Process) and is designed to be easily adaptable to any software development process. S<sup>3</sup>P provides the formalism required to identify the causes of vulnerabilities and the mitigation techniques that address these causes to prevent vulnerabilities. We present a prototype of the security plug-in implemented for the OpenUP/Basic development process in Eclipse Process Framework. We also present the results of the evaluation of this plug-in. The work in this thesis is a first step towards a general framework for introducing security into the software life cycle and to support software process improvements to prevent recurrence of software vulnerabilities.



## Acknowledgements

---

I would like to express my gratitude to my supervisor Professor Nahid Shahmehri for introducing me to the incredible world of research in security. She has always been there to encourage me and is a great teacher for me. Without her continuously keeping me on track this thesis would not have been possible.

Several other people have contributed to this thesis. I especially thank: my colleagues David Byers and Dr. Claudiu Duma, for our interesting discussions in the initial stage of the project which helped me formulate new ideas; Professor Kristian Sandahl for providing his valuable comments on the draft of this thesis and Brittany Shahmehri for proof-reading the thesis.

I also would like to thank our project partners *Sectra Communications AB*, *Combitech AB* and *Ericsson AB*. Special thanks go to Sectra Communications for supporting our empirical tests, especially Dr. Michael Bertilsson and Robert Lidquist for providing valuable comments on the thesis.

I would like to thank my colleagues at ADIT (Division for Database and Information Techniques) for their friendship and support.

I wish to express my gratefulness to my beloved ones for their love and support. I thank my mom and dad for always believing in me and encouraging me to study and learn and I would like to thank my sister, my endless source of joy and inspiration for always supporting me. Last but not least I would like to thank my husband Behzad who has always been my champion, for his understanding and support.

Finally, I acknowledge the financial support by *Vinnova* (Swedish Agency for Innovation Systems) and *CUGS* (Swedish National Graduate School in Computer Science).





## List of the Publications

---

1. S. Ardi, D. Byers, and N. Shahmehri, "Towards a structured unified process for software security", *Proceedings of the ICSE 2006 workshop on Software Engineering for Secure Systems (SESS06)*, Shanghai, China, May 2006.
2. D. Byers, S. Ardi, N. Shahmehri, and C. Duma, "Modeling software vulnerabilities with vulnerability cause graphs", *Proceedings of the International Conference on Software Maintenance (ICSM06)*, Philadelphia, USA, September 2006.
3. S. Ardi, D. Byers, P. H. Meland, I. A. Tøndel, and N. Shahmehri, "How can the developer benefit from security modeling?", *Proceedings of the ARES 2007 International Workshop on Secure Software Engineering (SecSE07)*, Vienna, Austria, April 2007.
4. S. Ardi, Nahid Shahmehri, "Integrating a security plug-in with the OpenUP/Basic development process", *Proceedings of the International Conference on Availability, Reliability and Security (ARES08)*, Barcelona, Spain, March 2008.

List of the other publications related to this research topic:

1. D. Byers, N. Shahmehri, "Design of a process for software security", *Proceedings of the International Conference on Availability, Reliability and Security (ARES07)*, Vienna, Austria, April 2007.
2. D. Byers, N. Shahmehri, "A cause-based approach to preventing software vulnerabilities", *Proceedings of the International Conference on Availability, Reliability and Security (ARES08)*, Barcelona, Spain, March 2008.



# Contents

---

<b>CHAPTER 1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	Introduction .....	1
1.2	Motivation .....	2
1.3	Problem formulation.....	3
1.4	Research methodology .....	4
1.5	Contribution .....	5
1.6	Thesis outline .....	7
<b>CHAPTER 2</b>	<b>SUSTAINABLE SOFTWARE SECURITY PROCESS.....</b>	<b>9</b>
2.1	The structure of S <sup>3</sup> P.....	9
2.2	Model vulnerability .....	10
2.2.1	<i>Vulnerability cause graphs.....</i>	<i>10</i>
2.2.2	<i>Prevention semantics.....</i>	<i>12</i>
2.2.3	<i>Vulnerability cause graph construction.....</i>	<i>13</i>
2.2.4	<i>Graph validation and optimizations.....</i>	<i>14</i>
2.2.5	<i>Case study, VCG.....</i>	<i>15</i>
2.2.6	<i>Initial analysis.....</i>	<i>16</i>
2.2.7	<i>Vulnerability cause graph (CVE-2005-2558).....</i>	<i>17</i>
2.2.8	<i>Discussion .....</i>	<i>21</i>
2.2.9	<i>Empirical study.....</i>	<i>21</i>
2.3	Identify cause mitigations.....	22
2.3.1	<i>Security activities .....</i>	<i>26</i>
2.3.2	<i>Activity constraints.....</i>	<i>27</i>
2.3.3	<i>The semantic value of SAGs .....</i>	<i>27</i>
2.3.4	<i>Security activity graph construction.....</i>	<i>28</i>
2.3.5	<i>Case study, SAG.....</i>	<i>30</i>
2.4	Vulnerability analysis database .....	33

2.5	Define process components .....	34
2.6	Tool support.....	35
<b>CHAPTER 3</b>	<b>SECURITY PLUG-IN FOR OPENUP/BASIC .....</b>	<b>37</b>
3.1	The security plug-in.....	37
3.1.1	<i>Identifying security problems .....</i>	<i>38</i>
3.1.2	<i>Interactions between S<sup>3</sup>P and the development process .....</i>	<i>38</i>
3.1.3	<i>Staffing.....</i>	<i>40</i>
3.2	Security plug-in for OpenUP/Basic .....	41
3.2.1	<i>Security domain .....</i>	<i>44</i>
3.2.2	<i>Security discipline .....</i>	<i>45</i>
3.2.3	<i>Case study, the security plug-in.....</i>	<i>46</i>
3.3	Evaluation of the security plug-in.....	48
3.3.1	<i>Goal-Question-Metric .....</i>	<i>49</i>
3.3.2	<i>Questionnaire .....</i>	<i>51</i>
3.4	Discussion .....	53
<b>CHAPTER 4</b>	<b>RELATED WORK.....</b>	<b>55</b>
4.1	Software process improvement.....	55
4.1.1	<i>SPI management methods.....</i>	<i>56</i>
4.1.2	<i>Software process best practices.....</i>	<i>57</i>
4.2	Experience-based approaches.....	57
4.2.1	<i>Security best practices.....</i>	<i>57</i>
4.2.2	<i>CLASP .....</i>	<i>60</i>
4.2.3	<i>Security for agile development processes.....</i>	<i>61</i>
4.3	Analysis of vulnerabilities.....	65
4.3.1	<i>Root cause analysis .....</i>	<i>65</i>
4.3.2	<i>Vulnerability repositories .....</i>	<i>66</i>
4.3.3	<i>Vulnerability classifications .....</i>	<i>67</i>
4.3.4	<i>Threat modeling.....</i>	<i>67</i>
4.3.5	<i>Attack trees .....</i>	<i>68</i>
<b>CHAPTER 5</b>	<b>CONCLUSIONS AND FUTURE WORK .....</b>	<b>69</b>
5.1	Conclusions .....	69
5.2	Future work .....	71
5.2.1	<i>Improvements to S<sup>3</sup>P.....</i>	<i>71</i>
5.2.2	<i>S<sup>3</sup>P in commercial settings .....</i>	<i>71</i>
5.2.3	<i>SPI aiming at security.....</i>	<i>72</i>
5.2.4	<i>Taxonomy of causes and activities .....</i>	<i>73</i>
5.2.5	<i>Tool support.....</i>	<i>74</i>
<b>REFERENCES</b>	<b>.....</b>	<b>75</b>
<b>APPENDIX A</b>	<b>THE OPENUP/BASIC DEVELOPMENT PROCESS.....</b>	<b>81</b>
<b>APPENDIX B</b>	<b>SUMMARY OF THE EVALUATION .....</b>	<b>93</b>

**APPENDIX C ACRONYMS.....97**

**LIST OF TABLES .....98**

**LIST OF FIGURES.....99**



# Chapter 1

## Introduction

---

### **1.1 Introduction**

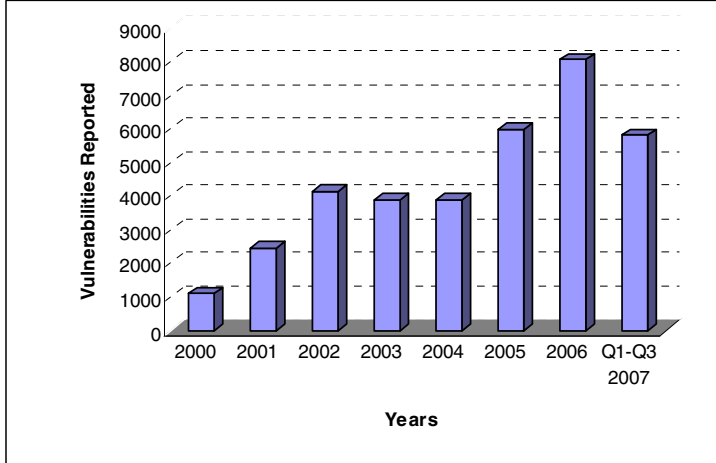
Serious security problems involving software and applications are frequently reported; they are rapidly becoming one of the most pressing issues in software engineering. These security problems are at the center of most costly software failures in recent years. For example the National Institute of Standards and Technology (NIST) report that software that is faulty in security costs the US economy \$59.5 billion annually in breakdowns and repairs [36].

According to statistics published by CERT, Coordination Center at Carnegie Mellon University (CERT/CC), the number of security vulnerabilities reported in the first three quarters of 2007 is almost as many as the number of reported vulnerabilities for whole year 2005 (see Figure 1-1) [14].

The level of risk society faces from intentional failures in software systems has increased in an almost uncontrolled fashion because [2]:

- Software is controlling, protecting, and affecting more and more critical information and systems, and this has caused a significant increase of the potential consequences of failures.
- As software becomes more complex, it tends to contain more flaws, and as it becomes more networked and converges towards a small set of open standards, its exposure to potential adversaries increases.

- Software-intensive systems are increasingly becoming viable financial and political targets for well-funded and well-motivated attackers, thus increasing the overall threat to these systems.



*Figure 1-1: Software security vulnerabilities reported to CERT/CC.*

## 1.2 Motivation

The term “software life cycle”, according to Pfleeger [42], “describes the life of a product from its conception to its implementation, delivery, use and maintenance”. Security problems can be addressed in many ways during software life cycle by considering security during software development, deployment and maintenance. Today, security is often an afterthought when developing software, rarely included in the early phases of software development. Most current solutions treat software security as an after-the-fact consideration, and focus on detecting and fixing problems after software deployment. Although solutions like firewalling, penetration testing and patch management reduce the risk by reducing the probability of security incidents or threats, exposure is more readily controlled if the focus becomes preventing problems in the first place [2]. Also, it is less expensive to fix software flaws earlier in the software development process [50]. According to IBM Systems Science Institute, fixing software defects in the testing and maintenance phases of software development increases the cost by factors of 15 and 60, respectively, compared to the cost of fixing them during design phase [50].



Efforts are being made to reduce security vulnerabilities in software and general best practices, disciplines and guidelines for software development have been published to improve the software security. Despite this, statistics on security incidents show that the industry has a long way to go (e.g. according to a preliminary analysis by CERT, over 90% of software security vulnerabilities are caused by known software defect types [43]). This is a sign that there are obvious gaps in the software development process with regard to security.

In addition to efforts to produce secure software, introducing security solutions to protect already developed software products is an important issue, particularly during maintenance, as new vulnerabilities in deployed software are discovered. Processes to build security in software life cycle can improve the software quality as well, and can result in software released with fewer defects.

Therefore in this thesis our interest lies in processes for developing secure software. We aim to introduce a structured way to create security plug-ins to the software development processes that reduce exposure by reducing the number of vulnerabilities in software.

### **1.3 Problem formulation**

There are various ways to address security problems of a software product: intrusion prevention mechanisms such as access control can be used to prevent vulnerabilities from having consequences; hardware-based solutions can be introduced to detect and prevent attacks to software systems. Standard approaches such as penetration testing and patch management, using security software, or deploying solutions like input filtering also offer security. One problem of these solutions is:

Problem 1: These mechanisms aim at software security after software is already built and are based on finding and fixing known security problems after they have been exploited in field systems [35].

New programming paradigms, methodologies, and development environments are introduced to improve the security of software during its development. Some current approaches are process specific solutions to integrate security features to a specific software development process. Other approaches are mostly ad hoc application of best practices or “secure programming techniques”.

Problem 2: In the case of process specific solutions, although security features can improve security of resulted products:

The problem of integrating these features in variants of the same process or in case of process changes remains unsolved.

Best practices and techniques are experience-based and they do indeed help prevent flaws, but:

It is difficult to say with any certainty what vulnerabilities are prevented and to what extent, or to say whether there are alternative ways of achieving the same effect [3].

More importantly for both of process specific solutions and best practices:

Problem 3: The ad hoc nature of these approaches makes evolving the process to meet new threats, or adapting it to specific situations something of a hit-and-miss affair [3].

According to the statistics published by CERT, most software vulnerabilities arise from common causes. In fact, the top ten causes account for about 75% of all vulnerabilities and many of these vulnerabilities result from defective specification, design, and implementation [43]. Because of increased public interest in computer and Internet security, this kind of security failure data is published increasingly in books, newsgroups and advisories.

Problem 4: Although security failure data and lessons learned from them can improve the security and survivability of the software systems, and can prevent the recurrence of vulnerabilities, software engineers do not use this kind of data [37].

Based on these problems we introduce our model and a security plug-in that target security issues in software development processes and proposes solutions to address mentioned problems.

## **1.4 Research methodology**

Research and development covers three activities [16]: “basic research, applied research and experimental development...*Basic research* is experimental or theoretical work undertaken primarily to acquire new knowledge of the underlying foundation of phenomena and observable facts, without any particular application or use in view. *Applied research* is also original investigation undertaken in order to acquire new knowledge. It is, however, directed primarily towards a specific practical aim or objective. *Experimental development* is systematic work, drawing on existing knowledge gained from research and/or practical experience, which is

directed to producing new materials, products or devices, to installing new processes, systems and services, or to improving substantially those already produced or installed”.

Based on this definition, the nature of our research can be categorized as *applied research*. We have focused on acquiring new knowledge and applying it to meet a specific need. From the initial phases of our research we have had collaboration with industrial partners to identify the requirements of our research based on commercial settings, and also to test our proposed solution in industrial settings. This categorizes our research as applied research in an industrial setting.

According to Adrion [1], research methodologies in software engineering are divided into the *scientific method*, the *engineering method*, the *empirical method*, and the *analytical method*. The scientific method deals with observing the real world, proposing a method or theory of some real world phenomena, measuring and analyzing, validating hypotheses of the method or theory, and if possible repeating this process. The engineering method deals with observing existing solutions, proposing better solutions, building or developing, measuring and analyzing, and then repeating until no further improvements are possible. The empirical method is based on proposing a model, developing statistical or other methods, applying to case studies, measuring and analyzing, validating the model, and repeating. The analytical method deals with proposing a formal theory or set of axioms, developing a theory, deriving results, if possible comparing with empirical observations and then refining the theory if necessary.

The research methodology we used is a combination of the engineering method (evolutionary paradigm) and the empirical method (revolutionary paradigm) [1]. We have used surveys about vulnerabilities, their properties and remedies; we have studied existing solutions presented to improve the security of software products with a goal of developing new solutions. We have proposed a new model for improving software security; we have applied our method to case studies; and we have developed prototypes based on our method.

## **1.5 Contribution**

The overall contributions of this thesis include introducing the Sustainable Software Security Process (S<sup>3</sup>P), for introducing security into the software life cycle and a security plug-in that shows how S<sup>3</sup>P can be deployed in a real development process.

S<sup>3</sup>P consists of three main steps and starts with modeling vulnerabilities based on a thorough analysis of software vulnerabilities to identify the causes that leads to them. The results of this analysis are used in the second step to identify mitigation techniques that eliminate causes of vulnerabilities as early as possible in the software life cycle. The mitigation techniques are then used in the third step to define development process components in the form of activities to be applied by development team members.

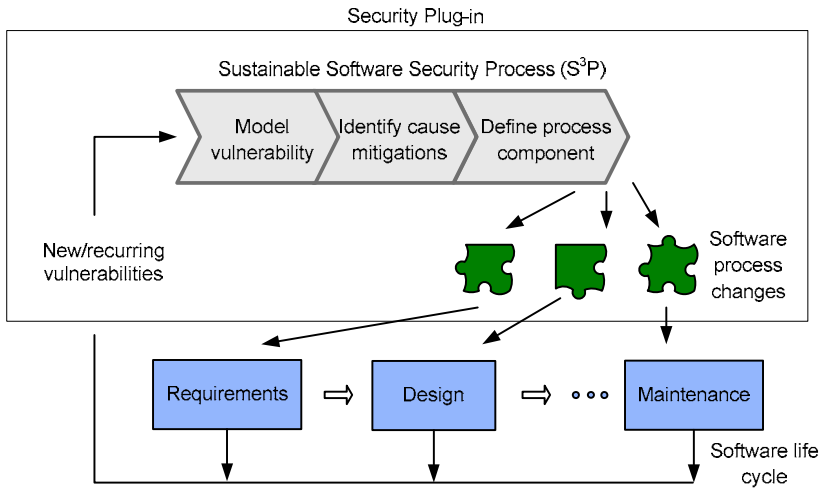
S<sup>3</sup>P is aimed to be process agnostic in order to reduce the effort involved in deploying it in a wide range of situations. We are particularly concerned that S<sup>3</sup>P will be applicable in agile processes such as extreme programming [8] or feature-driven development [41] as well as in more conventional development processes, [27], [31]. The security plug-in is composed of the process components resulting from S<sup>3</sup>P, and it provides support required for introducing these components into a particular software development process. S<sup>3</sup>P is also a software process improvement (SPI) process, which aims at improving the development process to prevent recurrence of security problems and vulnerabilities, and which allows process components to evolve as new threats and vulnerabilities are discovered.

There is already a considerable body of security know-how in literature, and individual organizations have developed specific activities and know-how related to their specific products and processes. This kind of information is integrated into S<sup>3</sup>P and the security plug-in as activities in the process components.

Figure 1-2 shows an overview of S<sup>3</sup>P and the security plug-in and their relation to the software life cycle.

Our detailed contribution is as follows:

- We introduce S<sup>3</sup>P for systematic and continuous improvement of security throughout software life cycle [3]. We also present our graphical notation, which is used in the structure of S<sup>3</sup>P.
- We present a security plug-in based on S<sup>3</sup>P, for the OpenUP/Basic development process version 0.9 [39]. This security plug-in is the first step towards developing a framework for adapting S<sup>3</sup>P to arbitrary software development processes and we present lessons learned when developing this plug-in.
- We also present the results of an evaluation of applicability of the security plug-in, which we have performed in collaboration with one of our industrial partners.



*Figure 1-2: Security plug-in in the context of software life cycle.*

## 1.6 Thesis outline

The outline of this thesis is as follows:

- Chapter 1 presents the introduction, the motivation, the problem formulation, our research methodology and the contribution of the thesis.
- Chapter 2 provides an overview of S³P and details about each step in this process.
- Chapter 3 presents the important issues related to the development of a security plug-in based on S³P and the security plug-in for the OpenUP/Basic development process.
- Chapter 4 presents the related work.
- Chapter 5 concludes the thesis and presents our future work.



## Chapter 2

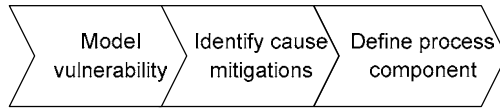
# Sustainable Software Security Process

---

### **2.1 *The structure of S<sup>3</sup>P***

S<sup>3</sup>P runs in parallel to a software development process and produces required components to improve the development process. According to definition by Komi-Sirviö [28] “Software process improvement denotes activities aiming at improving the software development process and is used for reaching a desired improvement goal”. Based on this definition, S<sup>3</sup>P is a process improvement process and the goal of the improvements is to introduce security to the software development process and to produce secure software. The workflow of S<sup>3</sup>P is shown in Figure 2-1.

“Model vulnerability” (the first step in S<sup>3</sup>P) aims at the analysis of vulnerabilities and their causes, similar to the root cause analysis. The results of this step are represented in a graph called Vulnerability Cause Graph (VCG). VCGs provide the basis for better understanding of vulnerabilities and their relationships, and identifying the activities to prevent them. Each cause in the VCG is then individually analyzed in the second step of S<sup>3</sup>P to determine how it can be mitigated. The result of this analysis and the structure of the VCG are then used to create a second graph structure called Security Activity Graph (SAG). SAGs allow us to reason about tradeoffs between different activities. Then activities are selected from SAGs and are introduced to the software development process in the form of configurable process components.



**Figure 2-1: The workflow of  $S^3P$ .**

## 2.2 Model vulnerability

When security problems or potential vulnerabilities are discovered in software during development or after deployment, they must be addressed as part of the software life cycle. Vulnerability modeling is a creative process supported by a systematic approach that provides an in-depth understanding of why and how the security problems and vulnerabilities are introduced into the software. The in-depth understanding helps to address these problems, prevent their reoccurrence, and eventually prevent the occurrence of similar problems. The process we describe for vulnerability modeling is based on our experience of analyzing a number of known software vulnerabilities [3], [10].

Vulnerability modeling starts with an initial analysis of the vulnerability in question. The vulnerability is analyzed to develop an understanding of the conditions that might lead to it. This analysis is typically performed using code review, static analysis tools, and visualization tools, execution traces, live debugging and if possible developing a working exploit. This initial analysis is considered complete when we know what conditions, inputs, and environmental issues would expose the vulnerability [10]. The results of the initial analysis are used to create the vulnerability cause graph.

### 2.2.1 Vulnerability cause graphs

A VCG is a directed acyclic graph that contains four kinds of nodes: *simple nodes*, *compound nodes*, *conjunction nodes* and an *exit node*. Simple nodes represent causes - conditions and events during software development process - that independent of any other cause or condition might result in the vulnerability. “Use of unsafe API”, “conditional range check” and “data file can contain executable code” are examples of simple causes. Compound nodes represent combinations of causes and refer to other VCGs. Compound nodes are introduced into the VCG to model complex analysis elements and facilitate analysis reuse, maintenance, and readability. Conjunction nodes represent the conjunction of two or more other nodes. Arbitrary combinations of simple and compound nodes are permitted in the conjunction nodes. The exit node is the root node that is the only node in the VCG without any successors. The exit node represents the vulnerability



being modeled by the VCG, and when a VCG is created for a compound node, the exit node represents the compound cause that VCG models. The edges in the VCG represent the relationship between causes and between causes and the vulnerability. The UML model of the VCG is shown in Figure 2-2.

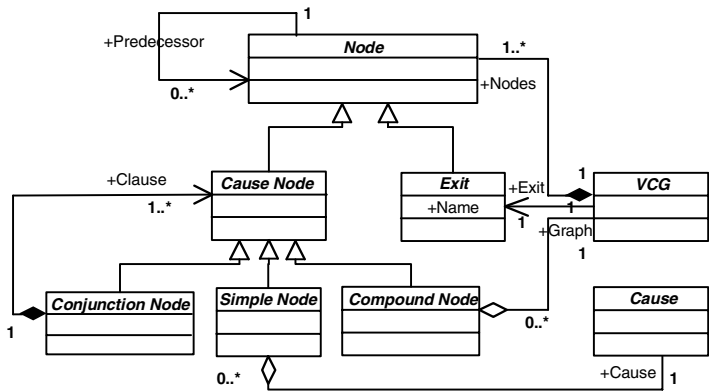


Figure 2-2: Simplified UML model of VCG.

The visual representation of VCGs is designed so that they can be easy for humans to understand. VCGs must also be well-defined so they can be used for automatic computation. Figure 2-3 contains the visual elements of VCG.

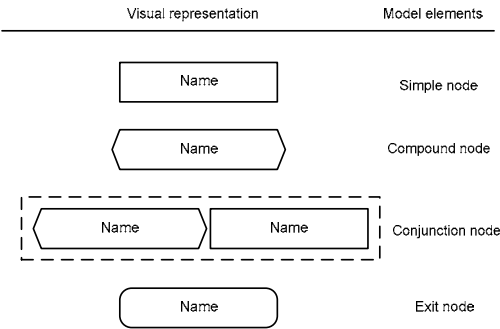
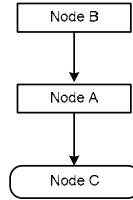


Figure 2-3: Visual representation of VCG.

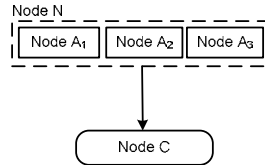
The predecessor-successor relationship in the VCG shows that if node *B* is a predecessor of node *A*, then if *B* holds (i.e. is not mitigated during development), then *A* is a concern. This implies that if the cause node *B* represents is mitigated, node *A* will not be a concern any more. Figure 2-4

shows an example sequence in the VCG. *Node A* is a direct cause and if *Node B* holds, then *Node A* and consequently *Node C* will be a concern.



**Figure 2-4: A sequence in VCG.**

In the case of conjunctions, if  $N$  is a predecessor of  $C$ , and  $N$  is a conjunction consisting of  $A_1...A_n$ , then  $C$  is a concern only if all of  $A_1...A_n$  hold (expressing conjunctions  $C = A_1 \wedge A_2 \wedge ... \wedge A_n$ ). In the example VCG in Figure 2-5, the conjunction node shows that the connection between  $A_1$ ,  $A_2$  and  $A_3$  is not causal and these three nodes jointly cause *Node C* to be a concern.



**Figure 2-5: A conjunction in VCG.**

### 2.2.2 Prevention semantics

The ultimate goal of vulnerability modeling is to determine how the vulnerability can be prevented and the semantics of VCGs are expressed in such terms [10]. Based on these semantics:

A cause that is of concern during software development is *mitigated* if actions are taken that result in the condition the cause represents being false.

A node representing a cause that is not a concern during development is considered *blocked*.

A node in a VCG is said to be blocked if it is mitigated, or all its immediate predecessors in the VCG are blocked.

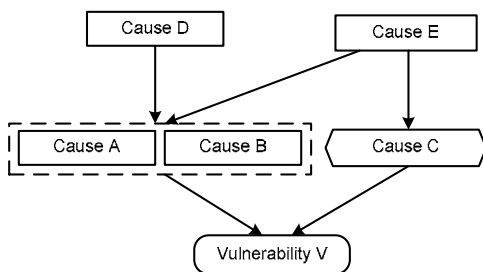
A simple node is mitigated if the cause it represents is mitigated.

A conjunction node is mitigated if any of its clauses are blocked.

A compound node is mitigated if the exit node of the VCG it is associated with is blocked.

By definition, the exit node of a VCG can never be mitigated but can be blocked as it always represents the consequences of other conditions. If the exit node of the VCG is blocked, then the vulnerability the VCG models is prevented.

In the example graph shown in Figure 2-6, *E* causes *C* and the conjunction of *A* and *B* to be a concern and vulnerability *V* is a concern because of both *C* and the conjunction of *A* and *B*. In order to prevent vulnerability *V*, *C* and the conjunction node should be mitigated. The conjunction node can be mitigated if one of *A* or *B* is mitigated. *C* will be mitigated if the exit node of its associated VCG is blocked. Another alternative is to mitigate *D* and *E*, so none of *A*, *B*, and *C* and consequently vulnerability *V* will be a concern anymore.



*Figure 2-6: A simple vulnerability cause graph.*

### 2.2.3 Vulnerability cause graph construction

The process of VCG construction starts with creating a base VCG consisting of an exit node only. Then the immediate causes of the exit node are identified and entered into the VCG as the predecessors of the exit node. Then each of the new entered nodes is further analyzed to identify their direct predecessors. The predecessors of a node represent the conditions that independent of any other condition, might cause the condition that the node represents to be a concern. For example if a node represents “range check is missing when copying into the buffer” a candidate predecessor might be “fixed-size buffer is used”.

Finding the predecessors of each node starts with answering the question “under what circumstances is this cause a concern?”. Then three steps are performed for every node entered into the VCG:

- The validity of the node is determined.
- The node is analyzed to determine if it needs to be split or converted to a compound node.
- Candidates for predecessors are found and organized.

Simple nodes entered into the VCG should always represent simple conditions, not combinations or sequences of conditions and the conditions represented by different nodes in the graph should not overlap. This is important both for the understanding of the vulnerability and for identifying mitigation techniques. Analysis of mitigation techniques is easier for simple conditions than for the complex ones. When a complex condition is identified it should be split to several nodes and possibly converted to compound or conjunction nodes. If the node being analyzed is already present in some other VCG, then its predecessors in that VCG might be suitable as predecessors in the current VCG.

The process of finding predecessors is repeated for all of the nodes in the VCG until no more additions to the VCG can be found. All compound nodes that have been introduced to the VCG should also be analyzed completely. The process of analyzing compound nodes is identical to the process of analyzing vulnerabilities.

We stop the modeling process when we find the causes related to the actions performed during the software development process and further analysis of these causes will lead to the causes related to issues out of the scope of the development process (e.g. organizational level causes).

#### **2.2.4 Graph validation and optimizations**

After completing the VCG, a second analysis is needed to validate the resulting VCG. Then the VCG is optimized - transformations are applied to it while preserving its semantic. This is an important step for improving the clarity of the VCG and to support its reuse. For example<sup>1</sup>:

- The order of every sequence in VCG needs to be verified to ensure that it is a natural order (e.g. cause-effect or temporal order).

---

<sup>1</sup> Detailed information about graph transformation is in [10].

- If a sequence lacks natural order we recommend the conversion to a conjunction node.
- Part of the graph can be converted to a compound node to support model reuse.

### 2.2.5 Case study, VCG

We have applied vulnerability modeling to a number of well-known vulnerabilities. This has resulted in a comprehensive understanding of them and alternative techniques to eliminate them. As an example, we present the analysis of CVE-2005-2558 [59] (as designated in the Common Vulnerabilities and Exposures<sup>2</sup> list [58]). This vulnerability is a buffer overflow in MySQL 4.0 before 4.0.25, 4.1 before 4.1.13, and 5.0 before 5.0.7-beta. According to the published descriptions:

“Stack-based buffer overflow in the `init_syms` function in MySQL allows remote authenticated users who can create user-defined functions to execute arbitrary code via a long `function_name` field” [59].

“The `init_syms` function uses an unsafe string function to copy a user specified string into a stack based buffer. Due to improper sanitation this buffer is able to be overflowed, overwriting portions of the stack. This allows an attacker to write 14 bytes of arbitrary data and 8 bytes of hard coded data beyond the end of the buffer. The format of the `CREATE FUNCTION` statement is as follows:

```
CREATE FUNCTION function_name RETURNS type
SONAME "library_name".
```

User specified input to the “`function_name`” field is limited to 64 characters. If this library can be successfully loaded by the operating system, control is then passed to `init_syms()`. This will attempt to copy the user string into a buffer 50 bytes in length. Hard coded strings are then copied onto the end of this string. In some older versions of MySQL this can be used to gain complete control over the EIP or copy attacker specified data to an arbitrary location. One issue of concern is because this buffer is owned by the calling function, in an environment with a stack that grows upwards, it may be possible to overwrite

---

<sup>2</sup> CVE (Common Vulnerability and Exposure) is a list of standardized names for vulnerabilities and other information security exposures and is hosted by MITRE Corporation. For more information see [www.cve.mitre.org](http://www.cve.mitre.org).

the EIP return or other sensitive values. Exploiting this vulnerability would require the ability to create user-defined functions. This is not typically granted to untrusted users; however given this vulnerability you should understand the ramifications of granting the ability to create user-defined functions” [53].

### 2.2.6 Initial analysis

We performed a detailed analysis of the vulnerability in MySQL-4.0.24 by code inspection and analysis of known exploits in a debugger. According to our analysis:

- The size of the buffer that should contain the user-defined function name is defined by the constant variable `MAX_FIELD_NAME` which is set to 34 characters. This variable together with some other variables are defined in a file named `unireg.h` to be used in `unireg` library<sup>3</sup>. Based on the comments in the code, `MAX_FIELD_NAME` is used to define maximum length of column names in the tables.
- The function used for copying function name into the buffer is `strmov()` which is defined by the programmer and is an unsafe function without any range check when it copies data.
- In MySQL-4.0.25, the patch is released for this vulnerability and the size of the buffer is defined by the constant variable `NAME_LEN`, which is defined in `mysql_com.h` file.
- The code does not contain any comments about the files, their creation time and how they are related and it is difficult to know which of the following scenarios are valid:
  - `MAX_FIELD_NAME` and `NAME_LEN` are defined to be used for different concepts (`MAX_FIELD_NAME` for column names and `NAME_LEN` for user-defined function names) and the programmer has made wrong assumption about `MAX_FIELD_NAME`.
  - Both of these constant values are defined for user-defined function names at different times with different values.

Figure 2-7 shows the structure and relationships of the folders and files in MySQL. The folder named *sql* contains modules related to user-defined functions. The folder named *include* contains modules related to the

---

<sup>3</sup> In MySQL, `unireg` is the tty interface builder. For more information see [38].

communication between server and client. Parts of the code related to this vulnerability are:

unireg.h:

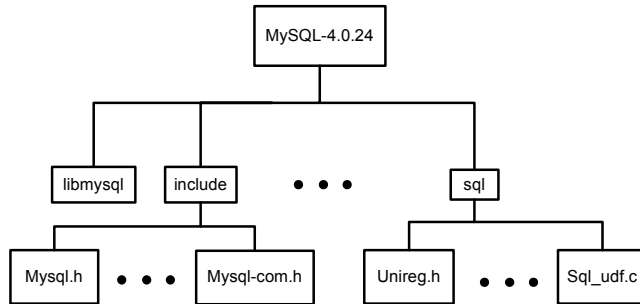
```
#define MAX_FIELD_NAME 34 /* Max column name length +2 */
```

sql\_udf.c:

```
char buf[MAX_FIELD_NAME+16], *missing;
```

mysql\_com.h:

```
#define NAME_LEN 64 /* Field/table name length */
```



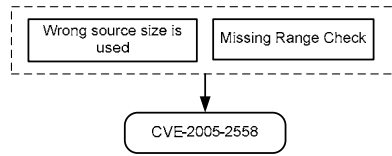
*Figure 2-7: The structure of files in MySQL.*

The constant value used in vulnerable version (MAX\_FIELD\_NAME) is from a different module (unireg module). If we assume that *include* folder contains constant values for the whole system, NAME\_LEN is the correct value to be used.

### 2.2.7 Vulnerability cause graph (CVE-2005-2558)

VCG construction starts with a single exit node labeled “CVE-2005-2558”, representing the vulnerability we are modeling:

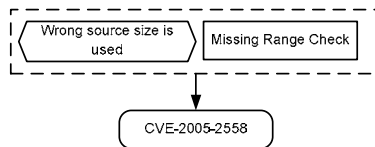
**Iteration 1:** The exit node is picked for further analysis. This node is a valid node and cannot be split or converted to a compound node. The predecessor candidates of the node are its immediate causes and according to our analysis a buffer overflow can occur because the size of the buffer that contains the user-defined function names is defined with wrong value. Besides, no range check is performed when copying data into this buffer. We enter these two causes as predecessors of the exit node. “Wrong source size is used” and “missing range check” together leads to this vulnerability and mitigating one of them can block the vulnerability. In this case these two causes should form a conjunction node (see Figure 2-8).



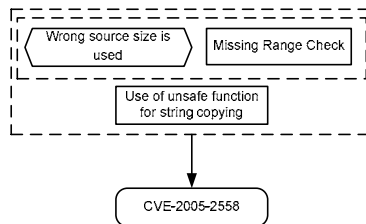
**Figure 2-8: VCG of CVE-2005-2558, Iteration 1.**

**Iteration 2:** The cause “wrong source is used” is not a simple cause and there are causes and conditions that are directly related to it. We convert this node to a compound node and will further analyze it in next iterations (see Figure 2-9).

**Iteration 3:** Based on the analysis, unsafe function *strmov* is used for string copying and because of wrong buffer size and missing range check this cause might lead to vulnerability. We enter it as a simple node to the VCG and since it causes vulnerability in conjunction with two previous nodes, we enter them as conjunction node to the VCG (see Figure 2-10).



**Figure 2-9: VCG of CVE-2005-2558, Iteration 2.**



**Figure 2-10: VCG of CVE-2005-2558, Iteration 3.**

**Iteration 4:** We analyze the resulting conjunction node and identify its direct cause to be the use of fixed-size buffers. This cause is a cause for concern because external data is copied into internal buffer and this is a cause for concern because C-like strings are used. We introduce these nodes to the VCG (see Figure 2-11).



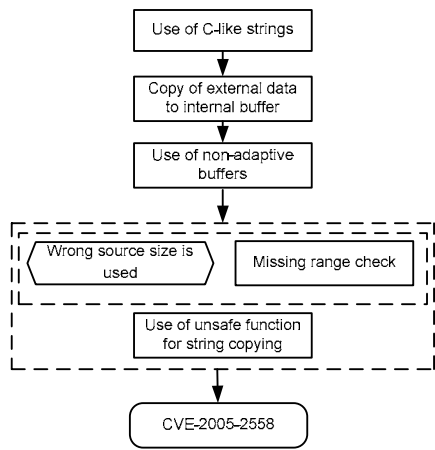


Figure 2-11: VCG of CVE-2005-2558, Iteration 4.

The compound node in the VCG is further analyzed to identify its causes. This process is similar to the process mentioned for construction of VCG with CVE-2005-2558 as exit node.

**Iteration 1:** In order to construct the VCG of this node, the exit node is created and labeled as “Wrong source size is used”. This is a valid node and it is a concern because either two different concepts (column name in tables and the name of a user defined function) have been assumed to be the same or different values have been defined for the same concept in two different modules (see Figure 2-12).

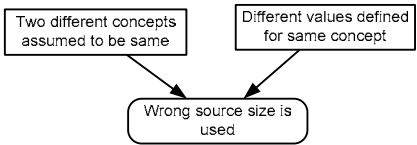
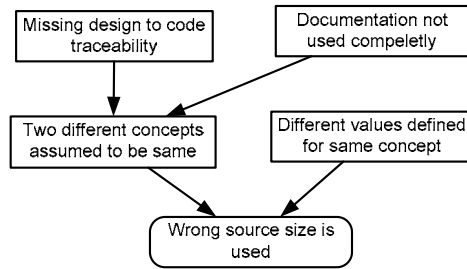


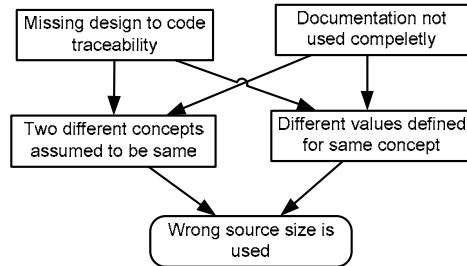
Figure 2-12: VCG of compound node, Iteration 1.

**Iteration 2:** First we analyze the node “two different concepts assumed to be same”. This is a cause for concern because of lack of traceability from design to implementation. Another cause can be not using documentation to clarify the purpose of each concept. These two causes independently can result in wrong assumption about the two concepts and respective constant values MAX\_FIELD\_NAME and NAME\_LEN (see Figure 2-13).



*Figure 2-13: VCG of compound node, Iteration 2.*

**Iteration 3:** The node “different values defined for same concept” can also be caused by “missing design to code traceability” or “documentation not used completely” (see Figure 2-14).



*Figure 2-14: VCG of compound node, Iteration 3.*

**Iteration 4:** Possible cause for not using documentation completely includes: the developer does not understand documentation and the cause for this problem can be either that the documentation quality is low (it is hard to understand), or the developer lacks skills to read documentation (e.g. language proficiency problem, etc.). We enter these causes as compound nodes because they can be further analyzed to identify why such problems are present. Further analysis of resulting nodes will lead to causes that are outside the scope of the development team (e.g. the cause for hiring a development team member that lacks skills to read the documentation) and further analysis of the node representing lack of traceability from design to implementation does not lead to any direct cause for it.

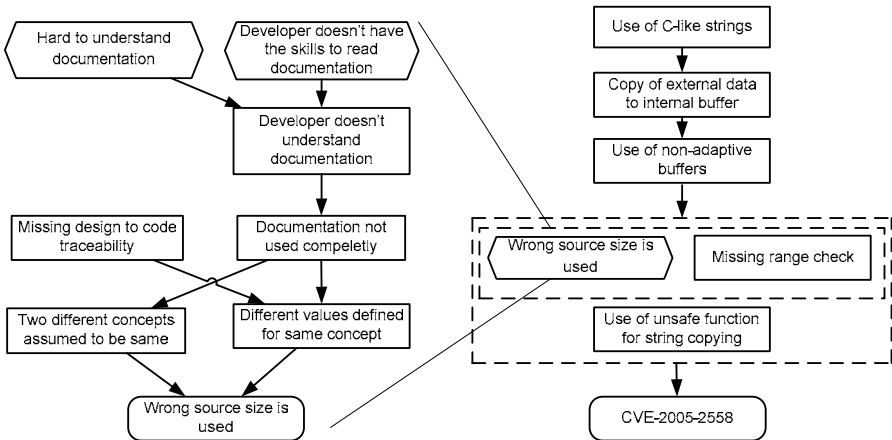
For the optimization of the resulting VCGs, we analyze the order of the sequences: for the sequence “copy of external data to internal buffer” and “use of non-adaptive buffers”, the decision on the use of non-adaptive buffers is made after the decision to copy data; for the sequence “use of C-like strings” and “copy of external data into internal buffer”, the choice of

using the C language is made early in the development. We keep the current sequences as they are. Three nodes were converted to compound nodes during the analysis and this further optimizes the VCG.

The resulting VCG for the vulnerability with one expanded compound node is shown in Figure 2-15.

### 2.2.8 Discussion

We applied our modeling method to this and several other known vulnerabilities and we found that the results gave a much more detailed understanding of the vulnerability than available published sources. For example for CVE-2005-2558 we discovered that documentation problems can cause misunderstanding and might lead to buffer overflow in stack. This detailed understanding is vital when we attempt to improve the development process and to prevent similar vulnerabilities.



*Figure 2-15: Vulnerability cause graph for CVE-2005-2558, with expanded compound node.*

### 2.2.9 Empirical study

We empirically tested the generation of vulnerability modeling in collaboration with one of our industrial partners. The goal of this experiment was to determine if our proposed vulnerability modeling methodology could be applied consistently by software developers.

Our test took the form of a small empirical experiment, involving four subjects. The subjects were experienced developers: two programmers, one tester and one security officer with development experience. The subjects were part of a development team. We were interested in having our method tested by subjects with various expertises in the development team, so subjects were chosen based on this criterion.

The test started with giving a tutorial on the vulnerability modeling and example vulnerability (vulnerability A) was analyzed and modeled to help subjects get familiar with the modeling process. Then the subjects got the description of vulnerability B and they analyzed the vulnerability and modeled it based on their own analysis. Each subject then modeled someone else's model and then the groups consisting of two subjects were formed to discuss the models of each group member and report one model for each group. We made a comparison of the models before and after validation by the groups and we also compared the models to the results of modeling performed by our research group.

The results of the experiment showed that software developers can use the method to model vulnerabilities with a small amount of training. Subjects had total agreement regarding causes in the design and coding phases and they came up with similar causes but there were some significant differences in naming the causes. The structure of the model varied before validation but after validation the structure was comparable to our modeling results. This shows that the validation phase is critical and perhaps more than one iteration should be recommended.

Our empirical test has shown that vulnerability modeling requires a mindset that is initially foreign for most developers: they tend to think in terms of fixing problems, rather than in terms of causes that lead to problems. This matches our own experience in developing our modeling process. We also realized that the names of causes should be made more consistent and the creation of taxonomy of causes should be considered as part of our future work. This empirical test validated our test method and we are planning to use a similar method to test the second step of S<sup>3</sup>P as well.

### ***2.3 Identify cause mitigations***

After modeling a vulnerability, mitigation techniques are identified to mitigate causes and block the vulnerability in question. This is the second step of S<sup>3</sup>P. The goal of this step is to determine possible software life cycle activities that would prevent vulnerability. This includes determining techniques for individual causes and composing them in a structure to show

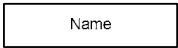
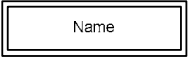
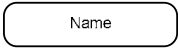
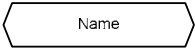
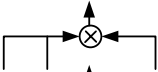
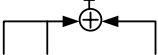
how they should be combined and performed to address the causes and to prevent the vulnerability. Most current methods for software security typically recommend certain sets of activities without support for adapting them to the conditions of the users of them. Many of these recommendations are only applicable under certain conditions (product type or organizational structures) and have little or no support for evolution [12]. Our methodology in identifying cause mitigations is developed with consideration for this shortcoming of current solutions.

The input to identifying cause mitigation step is vulnerability cause graphs. Each cause in the VCG is analyzed individually and activities that can be performed during software development to mitigate these causes are identified. The results are presented in security activity graphs. SAG is a tree consisting of nodes that are logic gates or activities. Edges represent relationships between activities and gates. SAG is a representation of a predicate logic formula with activities for terms and gates for operators. The root of the SAG is representing the semantics of the entire SAG. Figure 2-16 shows visual representation elements of SAGs and Figure 2-17 shows an example SAG. According to Figure 2-17 *cause C* can be mitigated by either performing *activity A<sub>3</sub>* or performing both of *activities A<sub>2</sub>* and *A<sub>1</sub>*.

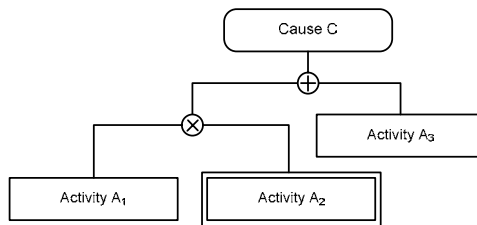
SAGs can be constructed manually or composed automatically from manually constructed SAGs. Manually constructed SAGs show how to prevent particular causes and the SAG of a vulnerability is computed automatically by combining the SAGs of each cause in the vulnerability's VCG, based on the structure of the VCG. The SAG of a vulnerability can be very large and complex but the structure of SAGs supports automatic processing.

The SAGs may contain *cause references*, a type of node that abstracts the SAG for a cause. A simplified UML model of the SAG is shown in Figure 2-18. For example, for vulnerability *V* in Figure 2-19 with two causes *A* and *B*, the complete SAG is shown in Figure 2-20. Since the two causes *A* and *B* independently can cause the vulnerability, the SAGs of these causes are combined with an *and* gate expressing that both of the causes must be mitigated (an *or* gate must be used if cause *A* and *B* have built a sequence).

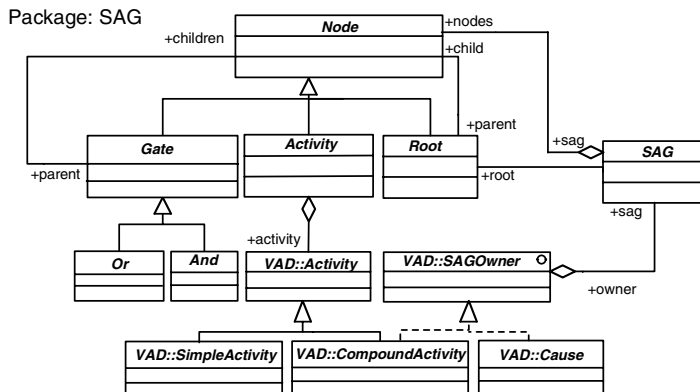
## CHAPTER 2

Visual representation	Model elements
	Simple activity
	Compound activity
	Root
	Cause references
	And gate
	Or gate

**Figure 2-16: Visual representation of SAG.**

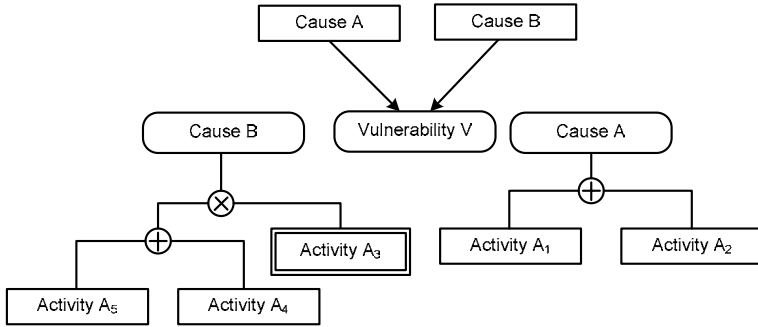


**Figure 2-17: A security activity graph for a cause.**



**Figure 2-18: UML model for Basic SAGs.**

The structure of SAG is called graph rather than tree because it can be visualized as a directed acyclic graph in which duplicated sub-trees have been merged. This can help to reduce the visual complexity of the SAG.



**Figure 2-19: Example vulnerability and SAGs of individual causes.**

The algorithm for automatic composition of SAGs is as follows<sup>4</sup>:

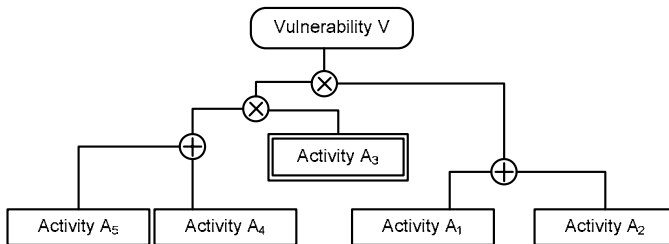
1. The SAG for a simple node is the combination of mitigation techniques that address the cause that the node represents (mitigation techniques are combined by *and* and *or* gates to show how they should be applied).
2. The SAG for a compound node is the SAG for the exit node of the VCG modeling the compound node.
3. The SAG for a conjunction node is the SAGs for the nodes in the conjunction connected by an *or* gate.
4. To construct the SAG for the exit node in a VCG:
  - a. Set the node as the start point.
  - b. Create the SAG of its direct predecessor according to the steps 1, 2, or 3 and set this SAG as the basic SAG.
  - c. Set the visited direct predecessor as the start node and repeat step b and connect the resulting SAG to the basic SAG with an *or* gate.
  - d. Repeat steps b and c until there is no predecessor for the start point.

<sup>4</sup> The algorithm in Python-esque notation can be found in [12].

- e. If there is a node that has more than one direct predecessor connect the SAGs constructed for its direct predecessors by an *and* gate.

### 2.3.1 Security activities

SAGs contain security activities and security activities represent security-related software life cycle activities. These activities are selected and performed during the software development process to prevent vulnerabilities.



**Figure 2-20: Complete SAG for vulnerability V.**

“Make all design objects identifiable” and “specify overall security policy” are examples of security activities. Every security activity contains information required to implement it and verify the implementation, and some constraints that might be faced when it is performed. Two kinds of activities are supported in our method: *simple* and *compound*. A simple activity is a natural language description of an activity and a compound activity refers to another SAG.

#### Implementation procedure

The implementation procedure of an activity is a detailed description of how the activity is implemented and if it should be expressed using a natural language (simple activity) or by a SAG (compound activity). Compound activities are mainly used to express the activities that can be performed in more than one way. Compound activities also create abstraction of complex activities and support reuse of part of complex activities in other SAGs.

#### Verification procedure

The purpose of this procedure is to verify that the implementation procedures was successful. The verification procedure can be expressed both in natural language and in form of SAG. Using SAGs for verification procedure means alternative ways can be used to verify the activity. The



verification procedure is not always required. For example, compound activities may transfer verification to constituent activities, and the constituent activities of a verification procedure expressed by a SAG typically lack a verification procedure.

### 2.3.2 Activity constraints

Activity constrains model relationships between activities called *inter-activity constraints* or between activities and the world called *external constraints*. The types of activities we have faced so far are dependency and ordering constraints (inter-activity constraints) and applicability constraints (external constraints).

#### Dependency constraints

Dependency constraints refer to the situation when for activity *A* to be successful activity *B* must be also performed. This constraint is satisfied if activity *A* is not performed or if both of activities *A* and *B* are performed. For example, the activities *A* and *B* can be connected via an *and* gate or activity *A* can be extended to contain the content of activity *B*. Sometimes dependent activities should be performed in different phases and there is a need for verification procedure to ensure that the activity is actually performed as intended.

#### Ordering constraints

If, when activity *A* and *B* are both performed, activity *A* should be performed before *B*, then there is an ordering constraint. This constraint is satisfied if none of activities *A* and *B* is performed or *A* is performed before *B*. Ordering constraints are not modeled in the SAGs, as the SAG models the information about which activities to be performed not how they should be composed to form a process component. The ordering constraints might be mentioned in the description of simple activities.

#### Applicability constraints

The applicability constraints refer to situations when a specific tool or expertise is needed to perform an activity.

### 2.3.3 The semantic value of SAGs

The semantic function of a SAG is a function of Boolean variables that return true when activities are implemented in the software life cycle to prevent the vulnerability associated with the SAG. The semantic values are computed as follows [12]:

The semantic value of the root equals the semantic value of its child.

The semantic value of an *and* gate is the conjunction of the semantic values of its predecessors.

The semantic value of an *or* gate is the disjunction of the semantic values of its predecessors.

The semantic value of an activity is a variable, which is true iff the activity is implemented.

For example the semantic value of the SAG in Figure 2-20, with  $V$  denoting vulnerability,  $B(V)$  denoting the semantic function,  $A_i$  denoting activity  $A_i$  and so on is:  $B(V) = (A_1 \vee A_2) \wedge (A_3 \wedge (A_4 \vee A_5))$ <sup>5</sup>.

### 2.3.4 Security activity graph construction

Security activity graph construction starts with the analysis of the individual causes identified in the vulnerability modeling step. The analysis process is called *cause mitigation analysis* and each cause is analyzed to determine how it can be mitigated. Determining how to mitigate a cause is a creative process, and relies on the experience of the analyst. Security know-how and best practices are used in this stage to define the security activities. The process of cause mitigation analysis consists of following steps:

1. **Determine immediate activities:** The activities that directly address the cause are identified. The following questions help us to perform this step:
  - a. What best practices are known to eliminate this cause?
  - b. What activities could eliminate this cause if performed during requirement analysis, design, implementation, and deployment?
  - c. What activities related to the organization and overall environment can eliminate this cause?

Depending on the type of cause, some of these questions will not have any answers. For example, a cause that is strongly related to the implementation may not have related activities in the requirement phase.

---

<sup>5</sup> The algorithm that computes the semantic function of a SAG can be found in Python-esque notation in [12].

2. **Determine supporting activities:** Supporting activities are those that are required for successfully implementing an activity. These activities are either dictated by the current cause or must always be performed together with the current immediate activity. The supporting activities are attached to the activities they support, via an *and* gate.
3. **Break down complex activities:** Every new activity entered in SAG is examined to determine if it should be a simple or compound activity. An activity is compound if:
  - a. There are steps of the activity that belong to different development phases.
  - b. There are steps in implementation of the activity that require different verification processes.
  - c. There are options for implementing the activity.
  - d. The implementation of the activity contains reusable parts that could be used in the implementation of other activities.
4. **Define the verification procedure:** For each new activity, an analysis is performed to determine how it can be verified that the implementation was successful.
5. **Account for detection method:** The cause is analyzed to determine how it can be detected. This cannot be used for prevention of the cause but can be used in verifying the activities and can indirectly lead to prevention of other causes. The following questions help us to perform this step:
  - a. What best practices are known to detect this cause?
  - b. What activities could detect this cause during requirement analysis, design, implementation, testing, and deployment?
  - c. How can the organization and overall environment be designed to promote detecting this cause?
6. **Cull inappropriate activities:** We need to identify the activities that have effects beyond addressing the current cause (e.g. “Use Java” for a cause like “Use of unsafe function for string copying”). We should also identify the activities that address the conditions that the current cause leads to instead of addressing the current cause. Such activities should be accounted for as part of the SAG associated with some cause in the current VCG. If no suitable

SAG is found, the current VCG should be reviewed to see if it is complete.

7. **Iterate:** All steps are iterated until no more changes are made to the SAG.

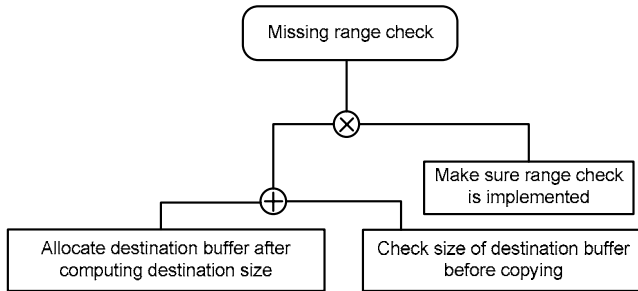
### 2.3.5 Case study, SAG

We have applied techniques for developing security activity graphs to CVE-2005-2558. Based on the VCG of this vulnerability in Figure 2-15, we have analyzed each cause and determined the mitigation techniques.

#### Missing range check

Range check is not performed when copying the name of user-defined function into the fixed-size buffer and resulting in a buffer overflow. The SAG for this cause is shown in Figure 2-21. We performed following steps to construct this SAG:

1. Range check when copying into fixed-size buffer is recommended as a best practice [22]. Immediate activities that can mitigate this cause include managing the size of the buffer either by allocating the buffer after size calculation or checking its size before copying and also ensuring that range check is actually implemented before each copy action. These activities are performed during implementation.
2. We have not identified any supporting activity.
3. The identified activities are simple.
4. Any copy action to the fixed-size buffer needs to be checked to verify that range check is implemented.



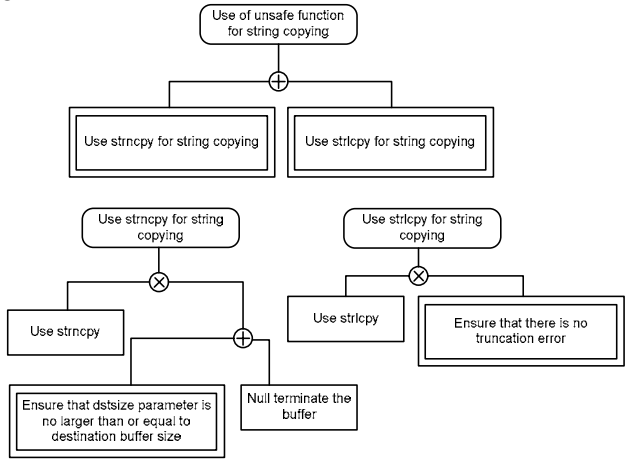
*Figure 2-21: The SAG for missing range check cause.*

#### Use of unsafe function for string copying

MySQL uses *strmov* which is very similar to the standard C function *strcpy*. We perform cause mitigation analysis and identify that:

1. This cause can be mitigated by replacing all occurrence of *strmov* with a safe function e.g. *strcpy* or *strncpy*.
2. The alternatives for replacing the unsafe function are complex activities and we introduce them as compound activities.
3. We also identify supporting activity to ensure that replacing of *strmov* is successfully implemented.
4. We also need the verification procedure to ensure that *strmov* is never re-introduced. The verification procedure is shown in Figure 2-23.

The SAG for this cause and the SAGs of its compound activities are shown in Figure 2-22.



**Figure 2-22: The SAG for use of unsafe function for string copying.**

### Use of non-adaptive buffers

The buffer used for copying the name of user-defined functions is a fixed-size buffer and does not adapt to the amount of data. Figure 2-24 shows the results of cause mitigation analysis for this cause.

### Copy of external data to internal buffer

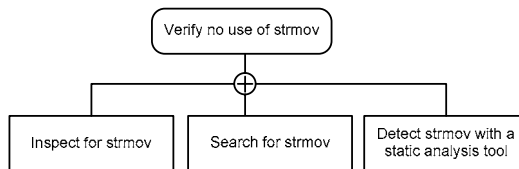
Data supplied by, or influenced by a source outside the program is copied into a buffer and this copy is performed improperly. The SAG for this cause is shown in Figure 2-25.

### Use of C-like strings

C-like strings do not contain a representation of their length and this might cause problems if they are not handled properly especially when copying. The SAG for this cause is shown in Figure 2-26.

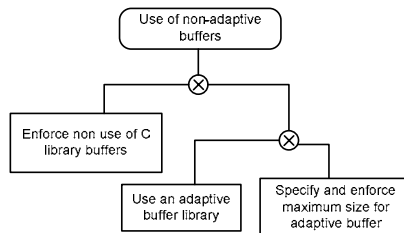
### Wrong source size is used

“Wrong source size is used” is a compound activity and its SAG is shown in Figure 2-27. Since the cause “hard to understand documentation” and “developer lacks skill to read documentation” are compound causes and need further analysis, we enter cause references instead of their SAGs. The activities to mitigate these causes are not performed during software development and are related to the qualification of the development team members. These activities will vary depending on the structures of the organizations and their policy when employing new development team members.

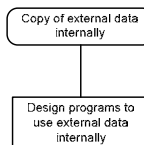


**Figure 2-23: Verification procedure for not using strmov.**

The complete SAG of CVE-2005-2558 can be composed of the SAGs that we presented here (see 2.3). The full SAG shows which activities must be combined to prevent the vulnerability. In this particular case, using a safe function or a good string library would prevent vulnerability. If these options are not accepted then the SAG shows all possible alternative activities.



**Figure 2-24: The SAG for use of non-adaptive buffers.**



**Figure 2-25: The SAG for copy of external data to internal buffers.**

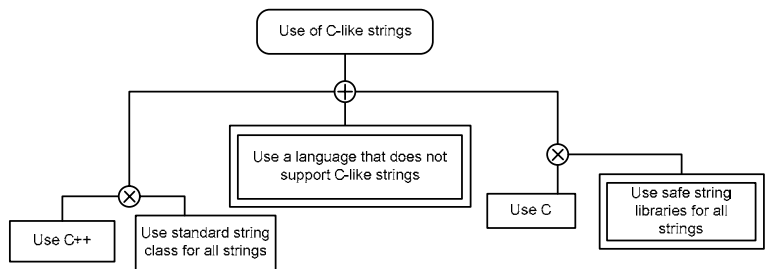


Figure 2-26: The SAG for use of C-like strings.

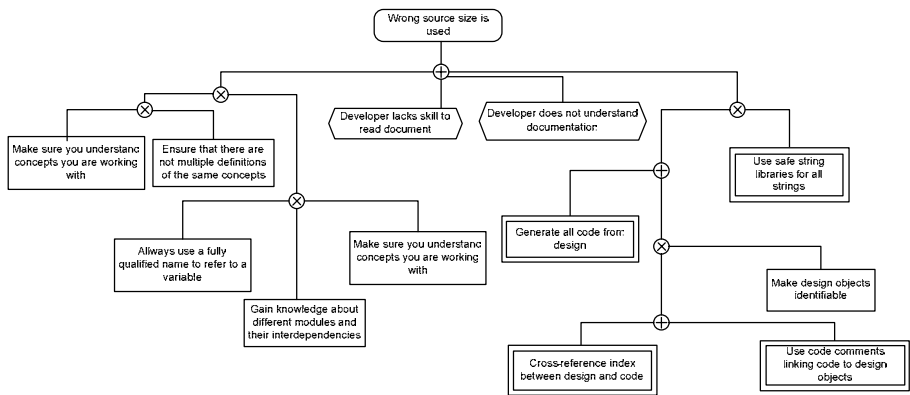
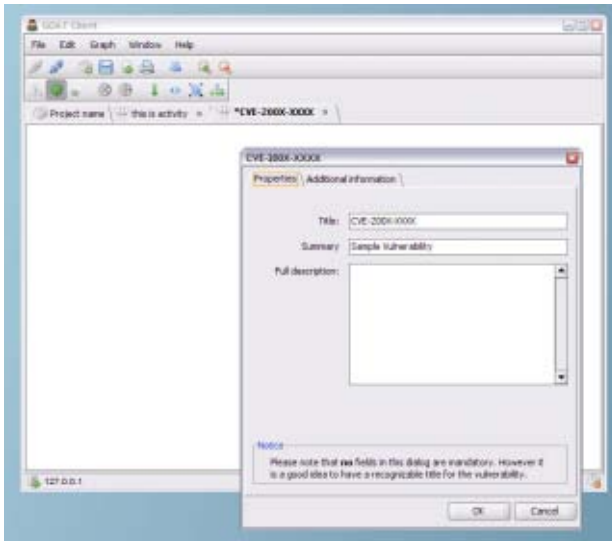


Figure 2-27: The SAG for wrong source size is used.

## 2.4 Vulnerability analysis database

All information gathered during “model vulnerability” and “identifying cause mitigations” steps is entered into a shared repository called *Vulnerability Analysis Database* (VAD). The VAD ensures that all uses of same cause, VCG, and SAG are linked; provides search mechanisms to support analysis reuse; and contains documentation needed to use our models. This database is an essential tool for effective practical application of S<sup>3</sup>P. The VAD contains five major sets of data:

- **Vulnerabilities:** Information including ID, summary, full description of the vulnerability and an in-depth analysis of the vulnerability are available for each vulnerability. The possible references to external resources are also presented in VAD. Figure 2-28 shows a screenshot of the interface that users see when entering information about a new vulnerability.



*Figure 2-28: The screen shot of the user interface of VAD.*

- Causes: Every cause has a title and ID, a brief summary and an in-depth description. There is also a code example and references to the vulnerability that cause might produce. Each cause is also linked to the SAG that presents the activities to mitigate the cause.
- Security activity: Every activity has an ID, a title, an implementation procedure, a verification procedure, and a set of constraints, also an assigned cost.
- Vulnerability cause graph: VCGs are linked to the vulnerability they present and the causes they contain.
- Security activity graphs: Every SAG has an ID. The SAG is linked to the cause it mitigates, and the activities that it contains.

## **2.5 Define process components**

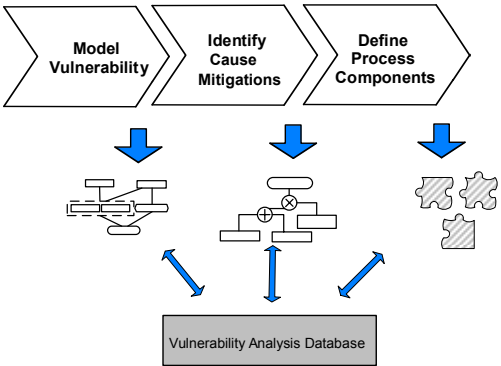
The third step in  $S^3P$  is the process of selecting activities from a set of security activity graphs. Activities are selected that are suitable to the product, development process, and organization. To accomplish this, activities are assigned *costs*, and a set of activities with *best cost* is selected. The cost of an activity depends on different factors including how it fits to the development process, staff, product, and organization. For example, the cost of training staff for performing an activity affects the cost of the



activity. An activity that the staff already know how to perform is cheaper (from a training point of view) than one that requires staff training.

Performing the selected activities during software development process satisfies the semantic function of the corresponding SAG. Finding a set of activities that satisfies the semantic function is trivial but finding a good solution is difficult. The process of assigning costs to activities and selecting a set of suitable activities is ongoing work and we are currently in the process of specifying the selection method. Further complicating matters is that sometimes the combination of certain activities is less expensive than the sum of their costs, and we also need to consider possible conflicts between activities that can affect the cost of an activity.

Note that such constraints will make the task of activity selection more difficult and we will work on this issue in our future work. These constraints are present in the SAG and are related to the development organization that performs the activity selection. Figure 2-29 shows the summarized structure and content of S<sup>3</sup>P as presented in this chapter.



*Figure 2-29: The structure of S<sup>3</sup>P.*

## 2.6 Tool support

Tool support can help the effective application of S<sup>3</sup>P. To support the initial analysis process that precedes vulnerability modeling, tools like source code checking tools are useful. It depends on the development organization how these kinds of tools are provided for the development team. Visualization and model editing tools can be used for vulnerability modeling and cause mitigation analysis.

Tool support for S<sup>3</sup>P includes tools to aid the software development process as well. The static code analysis tools help S<sup>3</sup>P during analysis and

## CHAPTER 2

also in verification procedures of security activities. These tools can evolve by taking advantage of the information provided by vulnerability modeling.

## Chapter 3

# Security Plug-in for OpenUP/Basic

---

S<sup>3</sup>P is designed to be process agnostic and is introduced into the software development process as a process plug-in. In software engineering, plug-ins are modules that add specific features to software or a service. The idea behind a process plug-in is to tailor new components into processes to better fit the needs of an enterprise. Process plug-ins support modifications to activities, roles, and other components of a process for software process improvements. Moreover, new components may be added by plug-ins to support missing features of a development process. Examples of plug-ins are a plug-in for handling capacity requirements in the OpenUP/Basic process [9] and IBM's process plug-in that extends the Rational Unified Process (RUP) to support requirements quality assurance [4].

This chapter presents a prototype of the process plug-in to introduce security into the OpenUP/basic development process. This security plug-in is our first step in deploying S<sup>3</sup>P in a real software development process. We first present key issues that must be considered when developing the security plug-in and then we present the security plug-in for OpenUP/Basic.

### **3.1 The security plug-in**

The typical scenario for using S<sup>3</sup>P in a software development process may contain the following steps:

1. Security problems and vulnerabilities are found and documented during the software development process.
2. The problems to be addressed are selected.

3. The selected problems are used to create the input document to S<sup>3</sup>P.
4. The steps of S<sup>3</sup>P are performed.
5. The process components resulting from S<sup>3</sup>P are defined based on the software development process.
6. The process components are introduced into the development process.

Considering these steps, the security plug-in must provide information about modifications to the software development process, including the interactions between S<sup>3</sup>P and the development process, and staffing issues.

### 3.1.1 Identifying security problems

Currently S<sup>3</sup>P does not mandate a particular process for identifying security problems during the software development process. However, detection methods identified when constructing SAGs are introduced in the form of process components to prevent recurrence of the vulnerabilities.

After identifying security problems, a risk analysis must be performed to determine the associated risks to each problem. This is required for prioritizing the security problems to be analyzed by S<sup>3</sup>P. The risk analysis step is not part of S<sup>3</sup>P and it is a requirement for the software development process to support this risk analysis.

### 3.1.2 Interactions between S<sup>3</sup>P and the development process

The vulnerabilities or potential vulnerabilities, and problems uncovered (and possibly fixed) during development must be used to create inputs to S<sup>3</sup>P. Since the first stage in S<sup>3</sup>P is the in-depth analysis of the vulnerability, a simple problem description is sufficient as the input. We call these reports *Security reports*. S<sup>3</sup>P will be initiated and run whenever a security report is created. It depends on the development organization how often S<sup>3</sup>P is performed (e.g. for each security report or after creating several security reports). These reports are generated in all phases of development and shouldn't be limited to test or maintenance phases.

Note that if there is no security problem identified to create the security report, a potential starting point for the use of S<sup>3</sup>P is to obtain the VAD with data on publicly known vulnerabilities, identify relevant vulnerabilities in the VAD and create process components that need to be introduced into the development process to prevent the vulnerabilities.

In addition to reporting the problems during development, security reports should be created when [11]:

1. A known vulnerability reoccurs; this indicates that the existing vulnerability or activity models are incomplete.
2. New mitigation techniques become known; in this case all existing models should be revisited to include the new mitigation technique.
3. An existing mitigation technique changes; in this case all models containing the mitigation technique need to be reviewed.
4. A new risk is identified or a known risk changes; new vulnerability models may be created to model potential vulnerabilities implied by the risk.
5. The criteria that influence process component definition changes (e.g. development process, staffing, or some other basis for cost assignment).

Figure 3-1 shows examples of descriptions that can be included in a security report. In some of the cases, there is no need to perform all steps of S<sup>3</sup>P. For example in the case of changes in the criteria for defining process components, the iteration will contain only the last step performed to re-select the activities.

Security Report,

Reported by: Designer  
Date: 2007-09-10

There is ambiguity in the behavior of `mprotect()` function. It seems that it should be used to switch the write permission on and off to a readonly segment of the shared memory. Why do we attach readonly to this segment at first place?

Reported by: Project manager  
Date: 2007-09-11

The static analysis tool AAA is available now for development team.  
The VAD needs to be updated in this regard.

**Figure 3-1: A security report.**

The results of S<sup>3</sup>P are selected activities and it depends on the structure of the development process what kind of process components should be created to introduce these activities into the development process. For example *Security checklist* can be created containing activities selected from SAGs. Activities in the security checklist must be included in the software

development process as routines the development team always follows. For example, activities represented by Figure 3-2 are a checklist of security activities that prevent security problems resulting from an inadequately low memory response in combination to unsafe use of *malloc*. Depending on the organization and the development process these tags might be different (e.g. referring to the role that performs the activity, not the phase in which activity is performed). The general view of the interaction between S<sup>3</sup>P and the development process, with security reports as input and a security checklist as output is shown in Figure 3-3. This figure shows one iteration of the interaction.

Note that training might be required to successfully perform some security activities defined in the security checklist.

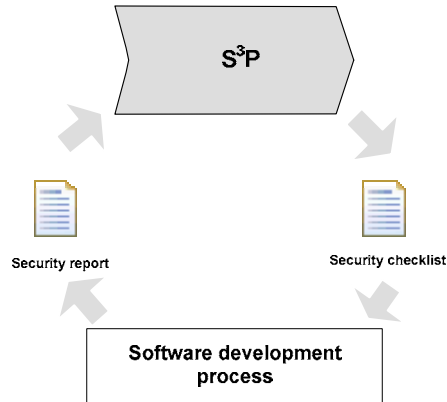
Security checklist v.x.y, updated 2007-10-10
<ul style="list-style-type: none"> <li>• D.1. Specify low memory behavior</li> <li>• D.2. Design low memory behavior</li> <li>[...]</li> </ul>
<ul style="list-style-type: none"> <li>• I.1. Avoid use of implicit malloc</li> <li>• I.2. Implement low memory response mechanism</li> <li>• I.3. Avoid use of libc memory management</li> <li>[...]</li> </ul>

*Figure 3-2: Example page of a security checklist.*

### 3.1.3 Staffing

All members of the development team can create security reports. Then at least two individuals are required to perform the steps of S<sup>3</sup>P (two or more individual are required to perform model validation). These two individuals are trained in vulnerability modeling and cause mitigation identification [11]. Based on our experience from our empirical test of vulnerability modeling (see 2.2.9), the more experience individuals have, the more varied the causes they identify. We recommend that one of the individuals in the team performing S<sup>3</sup>P steps is a senior developer with experience from all phases of software development.

Since process components are used to improve the development process, the team responsible for S<sup>3</sup>P must be supported by an individual or team with the mandate to alter the development process. This individual or team is a good candidate for ownership of the overall process [11]. We call this role *Security Auditor* (or security auditor team).



**Figure 3-3:** Example of the interaction between  $S^3P$  and a software development process.

### 3.2 Security plug-in for OpenUP/Basic

OpenUP/Basic is an open source software development process that takes an agile approach to software development. OpenUP/Basic targets small teams of 3-6 people with 3-6 months of development effort and preserves essential characteristics of the Rational Unified Process (RUP), e.g. iterative development and risk management. We have worked with version 0.9 of OpenUP/Basic. OpenUP/Basic provides an approach to assigning tasks and responsibilities within a development organization. These tasks and responsibilities based on OpenUP/Basic are described in two dimensions: the time dimension that shows the dynamic aspect of the process, and the content dimension that shows the static aspect of the process.

The static aspect describes what OpenUP/Basic contains in terms of modeling elements: artifacts, tasks, roles and workflows. Roles refer to responsibilities (stakeholder, analyst, architect, developer, tester and project manager). Tasks refer to activities to be performed by roles and workflows are sequences of tasks. Roles perform tasks that consume and produce artifacts and workflows define which tasks should be performed and when. For example, the task *Create test cases* develops test data for the requirements to be tested, and is performed by *tester*. The input to this task is the *use case* artifact that captures the sequence of actions and the behavior of a system from the perspective of the end user. The output of this task is a *test case* artifact, which is the specification of a set of test inputs and expected outputs. Tasks in OpenUP/Basic are organized into six *disciplines* (requirements; configuration and change management; project management;

analysis and design; implementation and test). Five domains organize the artifacts of OpenUP/Basic (architecture, development, project management, requirements, and test).

The dynamic aspect describes how OpenUP/Basic is performed over time, in terms of cycles, phases, iterations and milestones. Phases are: inception, elaboration, construction, and transition. Each of these phases consists of a number of activities. The activities show the tasks and their sequences. More information about the structure of OpenUP/Basic can be found in the Appendix A.

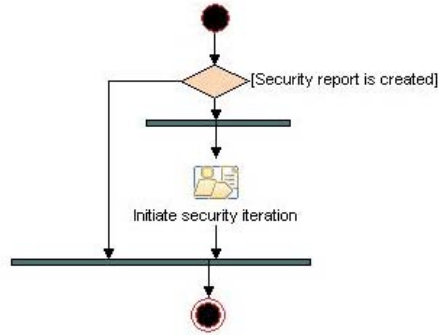
OpenUP/Basic has not been developed for secure development, although its aim is to deploy software development best practices such as iterative and incremental development. Tests focus on attributes of stated requirements for the developed product including integrity (resistance to failure), ability to be installed and executed on different platforms, and ability to handle many requests simultaneously. To achieve these kinds of goals different type of tests are developed such as a function test, security test, integrity test and stress test. The security test focuses on ensuring the data related to the product are accessible only to those actors for which they are intended, and relies on the expertise of the developers in designing such tests.

Testing cannot prove the absence of vulnerabilities in software because testers can apply only limited numbers of tests [60]. In order to build security into a system, security testing needs to be coupled with other security activities performed throughout the software development life cycle to effectively validate design assumptions, and to discover vulnerabilities and implementation issues that may lead to vulnerabilities.

We developed the security plug-in as an extension to OpenUP/Basic to run S<sup>3</sup>P and to use its results during software development. This plug-in introduces modifications to the OpenUP/Basic dynamic and static aspects: *Security Domain* and *Security Discipline* extend the static aspect of OpenUP/Basic and introduce modifications to roles, tasks and artifacts; the *Security Iteration* extends the dynamic aspect, and shows how the new roles, tasks, and artifacts are used to support security. According to our scenario in 3.1, when security problems or potential vulnerabilities are discovered, they need to be reported in security reports. Analyst, architect and developer are responsible for reporting problems such as ambiguities in the specification of requirements, conflicts between requirements from a security point of view, problems encountered during architecture design and the results of developer tests. The security iteration consists of the tasks and artifacts required for performing the three steps of S<sup>3</sup>P and security reports are used for initiating



a security iteration and as inputs to this iteration. Figure 3-4 shows the overview of workflow for initiating the security iteration.



*Figure 3-4: The overview of workflow for security iteration.*

We have designed the security plug-in as a method plug-in<sup>6</sup> in the Eclipse Process Framework (EPF) [20]. In this framework the elements like artifacts, tasks, roles, and capability patterns related to the plug-in are created and introduced to development processes. “Method content variability” is used in EPF when introducing new elements in a plug-in. According to the method content variability, the new element can contribute to, extend, or replace an existing element:

- **Contributes:** A new element contributes to an existing (base) element and the resulting element is the combination of the element.
- **Extends:** A new element inherits attribute values of an existing (base) element and also can have its own additional attribute values.
- **Replace:** A new element replaces an existing (base) element.

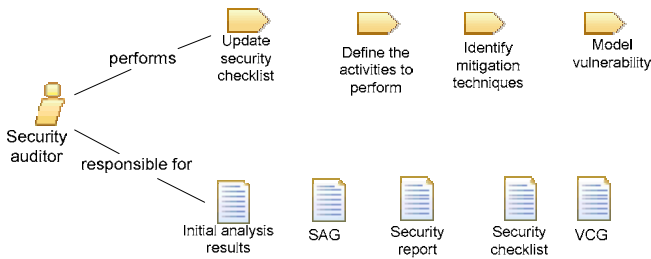
In addition to these content variability alternatives, “Not applicable” can be selected when a new element is created without connection to any other element. For example, the task “Report the issues discovered in developer test” in the security plug-in contributes to the “Implement developer test” task of OpenUP/Basic.

The *Security Team* is responsible for performing the security iteration. We introduce two roles in the security team and these roles extend two existing roles in OpenUP/Basic:

<sup>6</sup> In Eclipse Process Framework the content of a plug-in is organized in method plug-ins [20].

- **Security auditor:** The owner of the security iteration and the primary performer of S<sup>3</sup>P tasks. This role extends the project manager role.
- **Developer:** The person in this role is a developer and a member of the security team and, together with the security auditor, performs security tasks. This role contributes to the developer role of OpenUP/Basic.

Figure 3-5 and Figure 3-6 show security auditor and developer roles, the related artifacts and related tasks. The symbols used for tasks (pentagon) and artifacts (rectangles) in these figures are the same as symbols used in EPF.



*Figure 3-5: The security auditor.*

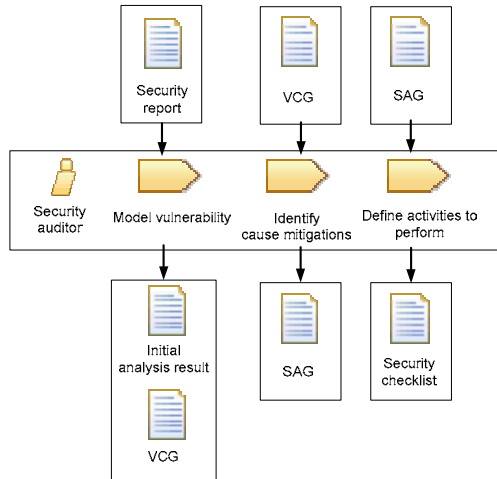


*Figure 3-6: The security developer.*

### 3.2.1 Security domain

Security reports, the results of performing initial analysis in vulnerability modeling, vulnerability cause graphs, and security activity graphs are the artifacts used for initiating and performing the security iteration. Checklists

are used in OpenUP/Basic to identify a series of items that need to be completed or verified and the results of the security iteration are introduced into OpenUP/Basic in security checklist artifact (see 3.1.2). The workflow in Figure 3-7 shows how these artifacts are used.



*Figure 3-7: The workflow and artifacts in the security plug-in.*

### 3.2.2 Security discipline

This discipline contains tasks for initiating and performing the security iteration and the task that supports the use of the results. In order to initiate a security iteration the security reports must be created and the tasks to create these reports must be defined in the plug-in. The tasks performed in a security iteration are defined based on the three steps of S<sup>3</sup>P and :

- **Report security issues in requirements, report inconsistencies in design, and report issues discovered in developer test** are tasks related to reporting potential vulnerabilities and problems and creating security reports.
- **Report new risk, new mitigation and change in criteria:** This task is for creating the security report whenever there is a need for revisiting the models (see 3.1.2).
- **Model vulnerability:** This task is the first step, in which S<sup>3</sup>P identifies the causes that might lead to vulnerability. The result is in the form of vulnerability cause graphs.

- **Identify mitigation techniques:** This task is the second step of S<sup>3</sup>P, identifies the cause mitigations. The output is in the form of security activity graphs.
- **Define the activities to perform:** This task is the last step of S<sup>3</sup>P, to select activities to be performed during development, the output is the list of activities that should be used to create security checklist.
- **Create-update security checklist:** After selecting activities, those activities that should be introduced as process improvements are used to update the security checklists.

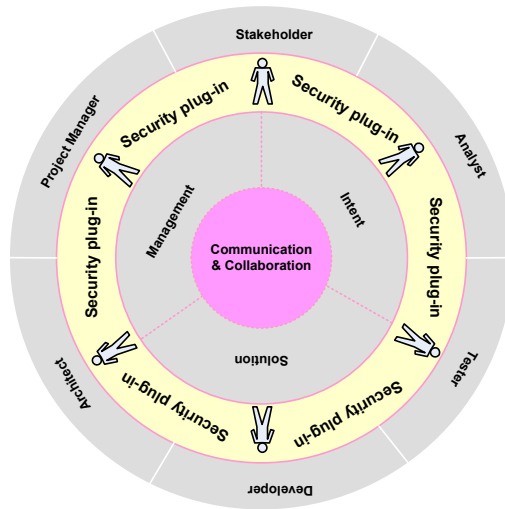
The security plug-in extensions to OpenUP/Basic are presented in Table 3-1 and Figure 3-8 shows the organization of contents for *Secure OpenUP/Basic* (OpenUP/Basic after introducing the security plug-in).

Modified roles	New tasks (Security Discipline)	New artifacts (Security Domain)
<ul style="list-style-type: none"> <li>- Security auditor: Project manager with extended responsibilities</li> <li>- Security developer: Developer to assist the security auditor</li> <li>- Analyst, architect and developer: to report security issues</li> </ul>	<ul style="list-style-type: none"> <li>- Report security issues in requirements, design and developer test</li> <li>- Model vulnerability</li> <li>- identify cause mitigations</li> <li>- Define activities to perform</li> <li>- Update security checklist</li> <li>- Report new risk, new mitigation techniques</li> </ul>	<ul style="list-style-type: none"> <li>- Initial analysis results</li> <li>- VCG</li> <li>- SAG</li> <li>- Security checklist</li> </ul>

*Table 3-1: Extensions to OpenUP/Basic.*

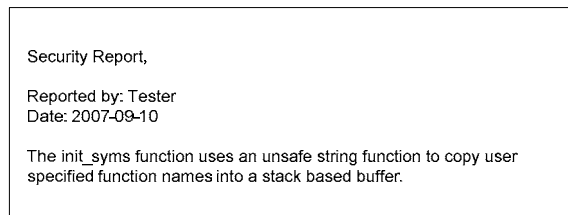
### 3.2.3 Case study, the security plug-in

We assume that CVE-2005-2558 has been identified during the test of the MySQL 4.0.24 (e.g. one of the testers realized that *strmov* behaves similarly to *strcpy*, which is an unsafe function) and is reported by the security report



**Figure 3-8: The organization of content in Secure OpenUP/BASIC.**

in Figure 3-9. This security report is input to security iteration. The security team performs “model vulnerability” task and the results are the initial analysis and VCGs (see 2.2.6, 2.2.7). Then VCGs are used to perform a “identify cause mitigations” task and the SAGs presented in section 2.3.5 are created. The complete SAG (see Figure 3-10) is used to select activities in “define process components” task<sup>7</sup>.

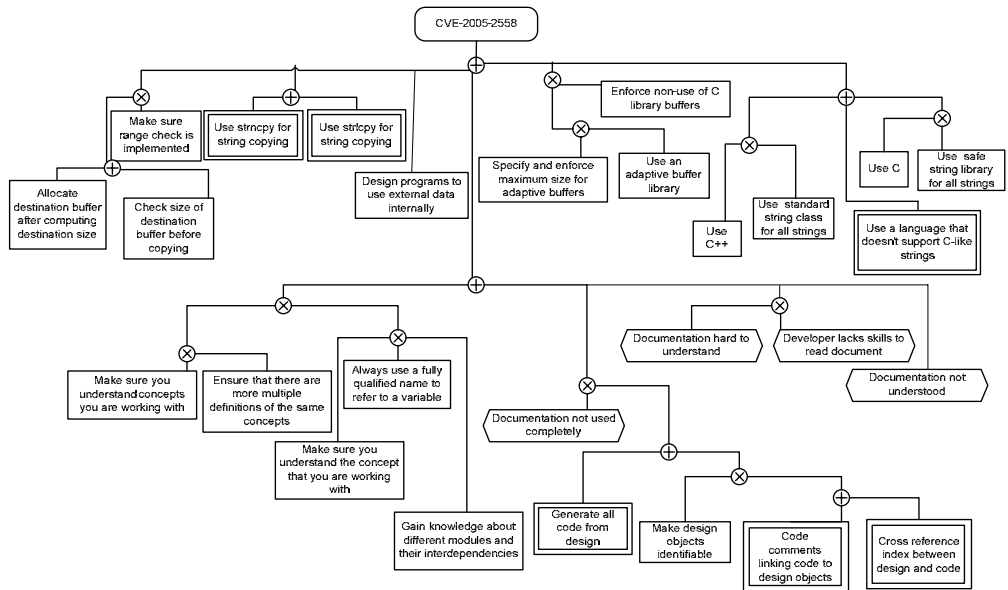


**Figure 3-9: Security report for CVE-2005-2558.**

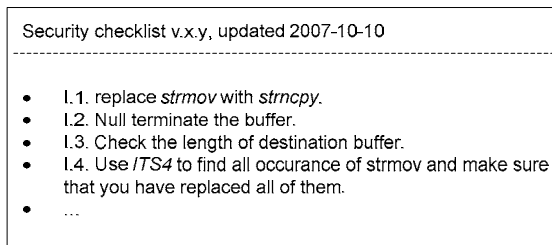
The SAG shows that the use of safe functions for copying data would be sufficient to prevent the vulnerability. This is one of the alternatives and other options might be selected depending on the organization’s preference. The knowledge about the vulnerability and the results of each of these tasks

<sup>7</sup> This SAG is simplified: sub-trees are eliminated and cause references are shown for four of the causes.

are entered into VAD, and the security checklist is updated to include the selected activities (see Figure 3-11). The activities in the security checklist must be performed during implementation (by developer role in OpenUP/Basic).



**Figure 3-10: Complete SAG for CVE-2005-2558.**



**Figure 3-11: Security checklist to prevent CVE-2005-2558.**

### 3.3 Evaluation of the security plug-in

We evaluated the application and adoption of the security plug-in in collaboration with one of our industrial partners. The goal of this evaluation

was to gather knowledge about the advantages and disadvantages of deploying the plug-in, its adoptability, and an estimation of time and effort required for its deployment.

We have used a questionnaire to ask experienced development team members about how the security plug-in would fit into a development process and how much value it adds to the development process after incorporation. We used “Goal-Question-Metric” [6] to design the questionnaire and to conduct the evaluation.

### 3.3.1 Goal-Question-Metric

The Goal-Question-Metric (GQM) paradigm is a mechanism for defining and interpreting software measurements [6]. It combines models of an object of study e.g. process, product, and one or more focuses, e.g. models aimed at viewing the object of study and studies the particular characteristics of the object. This study can be done for any purpose e.g. characterization, evaluation, prediction, motivation, improvement, etc.

Applying GQM involves:

1. Developing a set of goals,
2. Generating questions that define those goals as completely as possible in a quantifiable way,
3. Specifying measures that needed to be collected to answer those questions,
4. Developing a mechanisms for data collection, and
5. Collecting, validating and analyzing the data in real-time to provide feedback for corrective actions.

Goals may be defined for any object, for a variety of reasons, with respect to various models of quality, from various points of view. The goal is defined by filling a set of values for the various parameters in the template [6]. The template parameters are: purpose (what object and why), perspective (what aspect and who), and the environmental characteristics (where).

We are interested in the underlined template parameters for our evaluation:

***Purpose:***

Analyze some

(object: processes, products, other experience models)

for the purpose of

(why: characterization, evaluation, prediction, motivation, improvement)

***Perspective:***

with respect to

(focus: cost, correctness, defect removal, changes, process context, adherent, resulted product, ...)

from the point of view of

(who: user, customer, manager, developer, corporation, ...)

***Environment:***

in the following context

(problem factors, people factors, resource factors, process factors,...)

In our case, we analyze Secure Open/UP/Basic for the purpose of evaluation, with respect to cost, process context, adherent and resulting product, from the point of view of the manager and development team. The model of the object is “the software development process” and the model of perspective of interest is “producing secure software”.

Different sets of guidelines exist for different objects of study to apply QQM. Based on these guidelines for each process under study, there are three major areas that need to be addressed: definition of the process, definition of the quality of perspectives of interest, and feedback from using this process relative to the quality perspective of interest. Definition of the process includes questions related to:

- Process conformance (an assessment of how well the process is performed), and
- Domain conformance (an analysis of the process performer’s knowledge concerning the object).

Quality of perspective of interest is related to the validity of the model for the particular environment, and the validity of the data collected. Feedback includes questions related to improving the process and things learned with regard to process.

In QQM paradigm, metrics for process measurements are collected from the activities used in the process. Examples include cost and quality. Cost includes the measures of any resource used e.g. staff months, computer time, calendar time, etc. Quality measures include examples like reliability, ease of change, etc.



Based on these guidelines and information about GQM, we developed a questionnaire to evaluate our plug-in. There were three participants: one project manager and two development team members received a tutorial to get started with OpenUP/Basic and the security plug-in. The assumption is that the participants are acquainted with OpenUP/Basic and the security plug-in when answering the questionnaire.

### 3.3.2 Questionnaire

The questionnaire consists of three sections: process conformance questions, domain conformance questions and feedback. Process conformance questions target the characterization of S<sup>3</sup>P and the security plug-in. Domain conformance questions aim at the analysis of the evaluation participant's knowledge concerning software development.

#### Process conformance questions:

1. Do you think that applying the security plug-in (as an example plug-in based on S<sup>3</sup>P) is important?
  - a. Not very important but it may be applied to some products.
  - b. Important and should be applied to most products.
  - c. Extremely important, part of the essence of the development process (all products).
  - d. Critical, without the security plug-in the product will be insecure.
2. The security plug-in shows the details about how it can be adapted to OpenUP/Basic. Based on these details, what is your estimate of the cost of deploying the security plug-in if you were using OpenUP/Basic in your organization?

#### Training

- a. A large amount of time needs to be spent on training courses.
- b. A small amount of time needs to be spent in courses.
- c. Just using plug-in for self-training is enough.
- d. I can't answer.

#### Staff-Months

- a. All team members need to be heavily involved with the tasks of the security plug-in.
- b. Only the security team needs to be heavily involved with the tasks of the security plug-in.

- c. The level of involvement of the development team is not important compared to the results gained by using the security plug-in.
- d. Not possible to estimate before deployment.

Project Schedule

- a. It is possible to adjust the time spent on each security iteration, based on project constraints.
  - b. It will not affect the schedule.
  - c. Whatever it takes to perform the security iteration, the results are so important that the cost will be acceptable.
  - d. It is not possible to estimate how much performing the security iteration will affect project schedule before using it. (Give comments if you choose this answer.)
3. How long does a normal iteration take in your organization and what is your estimation about the length of the security iteration?
4. How often do you think the security iteration will have to be run? (Answer by considering the experience of your development team in reporting potential vulnerabilities or problems.)
5. Estimate the staff-time required for each of the following tasks:
- Creating security report
  - Creating and updating security checklists
  - Vulnerability modeling
  - Identifying mitigation techniques
  - Using security checklists
6. Do you think that there may be principles or issues related to the security plug-in that will not be completely accepted by development team to perform?
7. Do you think that the development team in your organization can create security reports based on the currently available problem reports (before deployment of the security plug-in in your organization)?
8. How closely you think your team will follow the principles and practices related to the security plug-in? Answer on a scale of 1-5, (5 means completely following the principles and 1 not following at all).
9. Do you see any significant problems that may hinder the deployment of the security plug-in?

10. Do you think that your development team supports the use of security checklist?

**Domain conformance questions:**

1. What kind of products does your organization develop?
2. What is your team size?
3. What is your role in the organization?
4. How familiar are you with the OpenUP/Basic (or RUP) development process model and concepts? Answer on a scale of 1-5, (5 means completely confident and 1 no familiarity).
5. How much do you think S<sup>3</sup>P will help you address security requirements? Answer on a scale of 1-5, (5 means completely addressing, and 1 not at all addressing). Please comment on your answer.
6. How much do you think the use of S<sup>3</sup>P may influence customer satisfaction? Answer on a scale of 1-5, (5 means influence a lot, and 1 no influence at all). Please comment on your answer.
7. How much are security handbooks used by your development team?
  - a. Not at all.
  - b. Used for some products.
  - c. Used for most products.
  - d. Essential to the development process

**Feedback:**

1. What are your suggestions to refine S<sup>3</sup>P?
2. Do you think the interface for the plug-in was usable for you to get required information about S<sup>3</sup>P?
3. Do you think that OpenUP/Basic helped you to get the idea of the deployment of S<sup>3</sup>P in an organization?
4. What parts of S<sup>3</sup>P you think need to be automated?
5. How much time did it take for you to answer to this questionnaire?

### **3.4 Discussion**

The detailed answers to the questionnaire are summarized in Appendix B and we present an overview of the evaluation result here. The subjects in the evaluation were three development team members: a system and software

engineer, a developer, and a project manager. The subjects are familiar with the principles of the OpenUP/Basic process and used the published web pages for the security plug-in to get familiar with it. The organization develops embedded systems, PC applications and communication products.

Results of the evaluation show that the subjects see the deployment of the security plug-in as a way to improve the security of the software products and two subjects even see it as an important method to be used in development of most of the software products. According to the subjects the security plug-in can be applied by the development team after a reasonable amount of training and the staff-months required for applying the plug-in cannot be estimated before deployment; however, only the security team will be heavily involved with the security iteration. Subjects state that when deploying the security plug-in it is possible to adjust the time spent on security iterations, based on the constraints of every project. Also deploying the plug-in would improve customer satisfaction.

Checklists are already used in the organization for some of the development activities but in order to ensure the proper use of security checklists a process is needed to manage and update the checklists. The frequency and length of the security iteration and each of its tasks depend on how many vulnerabilities are addressed in each security iteration and how often in the project's lifetime this iteration is run. Subjects believe that security reports can be created based on the current available problem reports in their organization and development team members would more and less follow the principles of the security plug-in.

According to the evaluation, the provided EPF-based interface has the quality required to assist the development team.

There are also several questions asked during this evaluation that need to be answered in future work:

- How can the security plug-in be integrated with security solutions already in use in the organization (e.g. Common Criteria)?
- Is there a way to export security reports to Common Criteria for tracking?
- Why isn't the security plug-in used for all defects?
- How can S<sup>3</sup>P and the security plug-in be used to capture security requirements?
- How can we be sure that security checklist is followed?

## Chapter 4

### Related Work

---

#### **4.1 Software process improvement**

According to the definition of ISO 15504 standards [25] a *software process* is “the process or the set of processes used by an organization or project to plan, manage, execute, monitor, and control its software related activities”. ISO 15504 further defines *process improvements* as an “actions taken to change an organization’s processes, so that they meet the organization’s business needs and achieve its business goals more effectively”. Based on this definition S<sup>3</sup>P is a software process improvement process for organizations where development of secure software is one of their business goals.

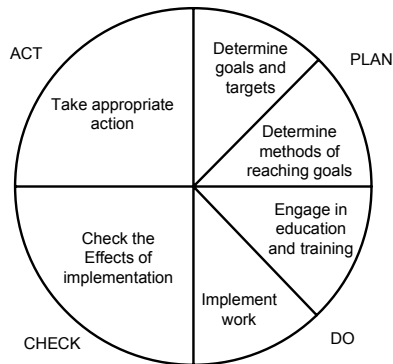
Various SPI methods have been suggested in the literature since the late 1980’s, to support the software development and software process improvements [7], [24]. These methods are classified to SPI management methods, software process best practices, measurement methods, product quality and knowledge management methods [28]. The purpose of management methods is to manage improvement initiatives. Software process best practices aim to improve software development processes using software development best practices. Measurement methods are used to determine the status of processes and products before and after applying improvement activities. Knowledge management methods aim to use the knowledge management principles to effectively share knowledge of software development. S<sup>3</sup>P shares the ideas of the management methods and software development best practices.

#### 4.1.1 SPI management methods

The examples of SPI management methods are Total Quality Control Model (TQC) [24], Quality Improvement Paradigm (QIP) [7], and The IDEAL Model [28]. TQC and QIP use models similar to *Deming's cycle* [18] and the IDEAL model is developed to support SW-CMM<sup>8</sup> and aims at improving the management in strategic and tactical levels.

S<sup>3</sup>P is similar to TQC in the way that they both start by analyzing the cause of the problems and visualizing the results of analysis in a graphical representation (VCG in S<sup>3</sup>P and Ishikawa diagrams in TQC).

TQC is based on the Control Circle (refined version of Deming cycle). Figure 4-1 shows the structure of Control Circle. The TQC model starts by defining the improvement goals and then in the PLAN phase, the Cause and Effect Fishbone diagrams (known as Ishikawa diagrams) are used for collecting cause factors which may have an influence on implementing the quality characteristics. These diagrams are created in an analysis session and based on the opinion of people familiar with the development process in question. The results of these analysis sessions are used to plan actions to improve the process.



**Figure 4-1: Control Circle [28].**

TQC does not support any specific process to identify and visualize the actions required for improving the process and it has no support for introducing these actions into the development process. The management improvement methods aim to improve the management of the process in

<sup>8</sup> The capability maturity model for software from Software Engineering Institute. For more information see [www.sei.cmu.edu/cmm..](http://www.sei.cmu.edu/cmm..)

general and do not focus on a specific property of the resulting product e.g. security.

#### **4.1.2 Software process best practices**

Methods in this class are classified into two categories: assessment-based approaches and software process standards. Assessment-based approaches focus on assessing existing development process and compare the process with a specific reference process [28]. Examples of these methods are SEI Capability Maturity Model and ISO 15504 (SPICE) [19]. Software process standards take the software best practices and create standardized definitions for them, and then these standards are mandatory or recommended for process improvements. One example of these methods is ISO 9000-3 [26].

S<sup>3</sup>P uses these software process best practices and standards of software development in the form of activities in SAGs. The knowledge available in the VCGs and SAGs also help to find out about the benefits of applying a certain best practice and the consequences of not using it.

### **4.2 Experience-based approaches**

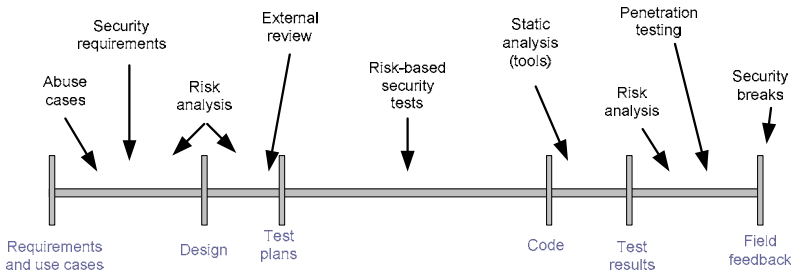
Currently, most approaches to software security are based on experience. These approaches are either ad hoc application of best practices and techniques or processes specific solutions. Examples of ad hoc solutions are security best practices suggested by Howard [23], software security approach by McGraw [35] and Microsoft's trustworthy computing security development life cycle [32]. Examples of process specific solutions are CLASP [48], and security for agile development processes [21], [49].

#### **4.2.1 Security best practices**

Howard defines the basic best practices that must be performed by a security team in the software development process. These practices are: security-focused development process goals, creating a central security team, education and raising awareness, understanding the adversary, using secure design practices, building secure code, security-focused events, performing security review and establishing a response process [23].

McGraw focuses on applying software security practices to various security artifacts [35]. For example, building abuse cases can help to identify the security requirements. Risk analysis should be applied during design to identify possible attacks on the system. External review is necessary and static analysis tools must be used to find implementation flaws. Risk-based

security testing is necessary and penetration testing is useful. Operations people should carefully monitor the system during use for security breaks. In this approach the artifacts are laid out according to traditional waterfall model, but it is possible to cycle through the steps more than once if the organization follows an iterative approach. Figure 4-2 shows how the security best practices are applied.



**Figure 4-2: Software security best practices applied to software artifacts [35].**

Although evidence clearly shows that these methods do prevent vulnerabilities, they typically have several drawbacks. One of the drawbacks is a lack of specificity. For example, Howard’s approach makes high-level recommendations without any practical guidance on how they are implemented. McGraw recommends a security review without specifying how this review should be performed and what the results of the review might be. In both of these approaches it is not clear which security problems are addressed by applying the practices and to what extent; if there are any alternatives to the practices; what the concrete benefits of the practices are and what the consequences of not applying one or more of them might be.

Many other proposals for security best practices have the same deficiencies. In our approach, the alternatives, benefits, and consequences of activities are presented by SAGs, VCGs, and the relationships between them. We also offer a flexible framework for selecting activities based on the needs of the organization.

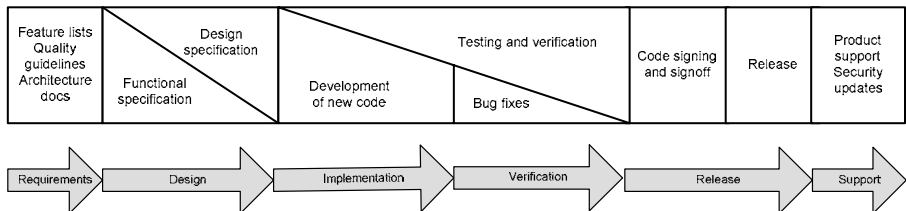
The trustworthy computing security development life cycle is a process adopted by Microsoft for “development of software that needs to withstand malicious attack” [32]. The process is formed of a series of security activities. These activities are:

- The development of threat models during software design
- The use of static analysis code-scanning tools



- The conduct of code reviews and security testing during a focused *security push*
- The final security review by a team independent from the development group

The generally accepted development process in Microsoft is shown in Figure 4-3. The process has five milestones and although the figure shows a waterfall model, the process is spiral. Requirements and design are often revisited during implementation in response to changes.



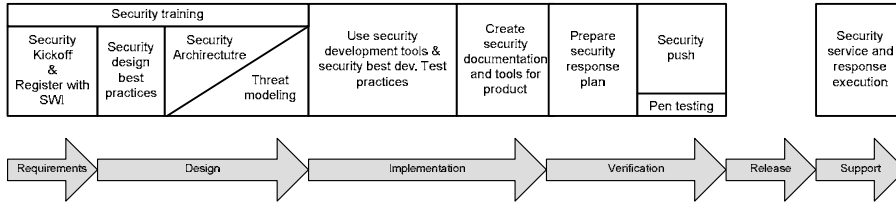
**Figure 4-3: The standard Microsoft development process [32].**

Security principles listed by Microsoft are:

- Secure by design: The software should be designed and implement so it can resist attacks.
- Secure by default: Software's default state should promote security to minimize the harm that attacks may cause.
- Secure in deployment: Tools and guidance should accompany software to help users to use it securely.
- Communications: Developers should be prepared for the discovery of the vulnerabilities and should communicate openly with the users.

Integrating security measures that support these principles results in the Secure Development Life cycle (SDL) process shown in Figure 4-4. In this process, during the requirements phase, the product team and central security team collaborate in planning the process and a security advisor advises the product team on security milestones. In the design phase the guidelines for security architecture and design are used, threat modeling (see 4.3.4) is conducted and documented. Then coding and testing standards are applied and static analysis tools are used in implementation. In the verification phase a security push that includes security code reviews is applied and security-focused testing is performed. In release phase, the software is subject to a

final security review and then is ready to be delivered. According to SDL, after deployment, the product team must be prepared to respond to newly discovered vulnerabilities in the software product.



**Figure 4-4: SDL improvements to the Microsoft development processes [32]<sup>9</sup>.**

SDL, like McGraw's approach [35], recommends some best practices for each phase. It concentrates on threats and attacks as well, and performs threat modeling and defines the elements of the software attack surface. This process follows the high-level guidelines of Howard's approach [23]. The experimental results at Microsoft show that the number of security bulletins released when SDL is used is significantly less than the number of bulletins when using the baseline process.

SDL is presented based on the baseline process of Microsoft and is a specific solution for a specific process, process model and organization. Integrating the security measures of SDL in other organizations requires the complete replacement of the development process. This is not always an accepted solution by organizations. SDL recommends security support for the maintenance phase but it is not clear how this support is performed and how new vulnerabilities are discovered and mitigated.

## 4.2.2 CLASP

CLASP [48] is Comprehensive, Lightweight Application Security Process which is designed as a security plug-in for RUP. CLASP is an activity centric approach containing twenty-four activities to be performed by software development team members. Figure 4-5 and Figure 4-6 show an example activity and example vulnerability respectively.

<sup>9</sup> SWI in the requirement phase activities refers to Secure Windows Initiative team in Microsoft.

<b>Specify operational environment</b>	
Purpose:	- Document assumptions and requirements about the operating environment so that the impact on security can be assessed.
Owner:	Requirement specifier
Key contributor:	Architect
Applicability:	All projects
Relative impact:	Medium
Risks in omission:	- Risks specific to the deployment environment may be over-looked in design. - May not properly communicate to users the design decisions with security impact.
Activity frequency:	Generally, one per iteration.
Approximate man Hours:	- 20 man hours in the first iteration. - <4 hours per iteration in maintenance.

**Figure 4-5: An example activity in CLASP.**

CLASP has a database that contains vulnerabilities categorized in sets of problem types: range and type errors, environmental problems, synchronization and timing errors, protocol errors, general logic errors and malware. The example vulnerability in Figure 4-6 is a range and type error.

While CLASP is a comprehensive and practical approach, it can only be used with RUP, while S<sup>3</sup>P supports any process and adapts to each user's needs. Security activities are limited to twenty four activities and possible alternatives or additional activities are not supported in this approach. The database of vulnerabilities in CLASP provides information about classes of vulnerabilities and general error types; in contrast we perform cause analysis in S<sup>3</sup>P to identify the causes of individual problems as well as common causes of classes of vulnerabilities.

### **4.2.3 Security for agile development processes**

Agile software development processes such as extreme programming [8] and feature-driven development (FDD) [41] are of increasing interest in software development, especially for web applications [21]. Security for agile processes has been discussed in a few articles mostly focusing on adding security features in specific agile development processes. Ge et al. [21] presents an agile development method for secure web applications. In this approach, FDD is the baseline development process and security activities like risk analysis are integrated into this process. The goal of the

authors is to design a development process that decreases the life cycle time, accepts frequent change of requirements, and, through risk analysis tasks and subtasks integrates security design throughout the development. The overall process then contains following tasks:

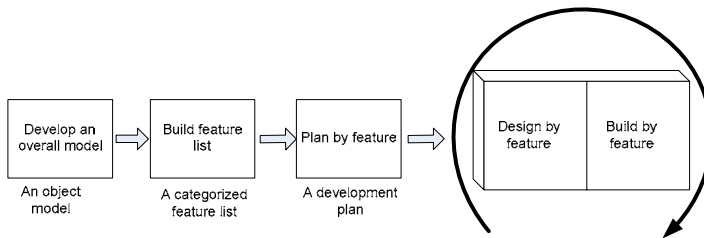
<b>Buffer underwrite</b>	
<b>Overview</b>	
A buffer underwrite condition occurs when a buffer is indexed with a negative number, or pointer arithmetic with a negative value results in a position before the beginning of the valid memory location.	
<b>Consequences</b>	
Availability: Buffer underwrites will very likely result in the corruption of relevant memory, and perhaps instructions, leading to a crash.	
Access Control (memory and instruction processing): If the memory corrupted memory can be effectively controlled, it may be possible to execute arbitrary code. If the memory corrupted is data rather than instructions, the system will continue to function with improper changes, ones made in violation of a policy, whether explicit or implicit.	
Other: When the consequence is arbitrary code execution, this can often be used to subvert any other security service.	
<b>Exposure period</b>	
Requirements specification: The choice could be made to use a language that is not susceptible to these issues.	
Implementation: Many logic errors can lead to this condition. It can be exacerbated by lack of or misuse of mitigating technologies.	
<b>Platform</b>	
Languages: C, C++, Assembly	
Operating Platforms: All	
<b>Required resources</b>	
Any	
<b>Severity</b>	
High	
<b>Likelihood of exploit</b>	
Medium	
<b>Avoidance and mitigation</b>	
Requirements specification: The choice could be made to use a language that is not susceptible to these issues.	
Implementation: Sanity checks should be performed on all calculated values used as index or for pointer arithmetic.	
<b>Examples</b>	
The following is an example of code that may result in a buffer underwrite, should find() returns a negative value to indicate that ch is not found in srcBuf:	
<pre>int main() {     ...     strcpy(destBuf, &amp;srcBuf[find(srcBuf, ch)], 1024);     ... }</pre>	
If the index to srcBuf is somehow under user control, this is an arbitrary write-what-where condition.	
<b>Related problems</b>	
Buffer Overflow (and related issues)	
Integer Overflow	
Signed-to-unsigned Conversion Error	
Unchecked Array Indexing	

**Figure 4-6: One example from the root-cause database in CLASP.**

- Requirement analysis: The aim is to determine the needs and expectations of relevant stakeholders.
- Security policy decision: A set of rules, principles and practices that determines how security is implemented, defined and followed during the development.

- Use case analysis: The purpose is to list a categorized list of features offered by the system, as well as a development plan.
- Content design: This is the major design activity, which is to design the structure and functionality of the system based on the features.
- Security risk analysis: the risk analysis is performed based on a risk model and the features in the system.
- Implementation: Finally the model of the features is implemented.

Figure 4-7 and Figure 4-8 show FDD and the secure web application development process based on FDD, respectively.

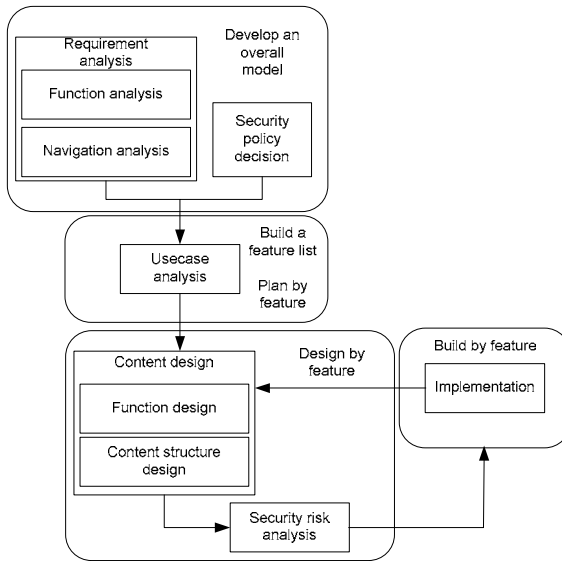


**Figure 4-7: Description of FDD process.**

This approach focuses on risk analysis as the main security feature in software development. Performing risk analysis is an important way to identify security flaws in design but developers need to know how to manage each particular risk and what the most cost-effective practice might be to address a certain risk. The development team should be aware of the problems and vulnerabilities that might be introduced to software and there is also a need for verifying that applying a particular set of practices has helped to control a particular risk.

One of the most important drawbacks that all the aforementioned approaches share is that they provide little or no support for evolution. For example, the evolution of CLASP is accomplished by waiting for the next revision of the manual. By continuously revising the vulnerabilities, their VCGs and SAGs, S<sup>3</sup>P and the security plug-in based on S<sup>3</sup>P evolves to meet new threats.

All the methods discussed here tend to be centered on best practices in one way or another. Our approach is not different in that respect: many security activities in our approach are best practices, and a thorough knowledge of best practices is essential to successfully identify cause mitigations.



**Figure 4-8: Secure web application development process.**

According to Turner [56] the software best practices have little implementation in software projects because the answers to the following questions are unclear:

1. How much will the practice really cost?
2. What is the actual benefit of applying the best practice?
3. What environment is assumed?
4. Will the management buy-in the practice?
5. Can we really implement it?
6. What is the pedigree?
7. Does the practice provide immediate benefits?

Our approach eliminates most of these obstacles in implementation of best practices, by clarifying the implementation procedure, the benefits of each practice, the cost of implementation, and the consequences of the activity.

### **4.3 Analysis of vulnerabilities**

Our overall approach to software security is related to root cause analysis (RCA) [44] and defect causal analysis (DCA) [13]. The vulnerability modeling method presented in section 2.2 can be seen as a method for root cause analysis of security-related software failures.

#### **4.3.1 Root cause analysis**

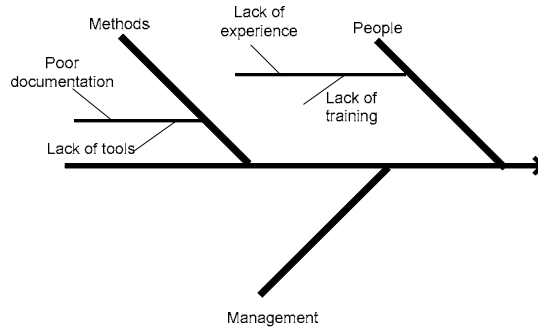
RCA is a process for identifying the root causes of problems or events. Root cause in this process refers to the fundamental cause, basis, or source from which the event or problem derives. The primary goal is to identify what happened, how it happened, why it happened, and the actions for preventing recurrence. The possible fields of application of RCA include project management, quality control, health and safety, business process improvement, change management, etc. In software engineering RCA can be used for defect analysis. The basic approach for using root cause analysis is:

- Define the problem
- Gather information and evidence
- Identify all issues and events that contributed to the problem
- Determine the root cause
- Identify recommendations for eliminating the recurrence of problems
- Implement the identified solutions

There are also several methodologies based on this basic approach such as the “5-why” analysis, change analysis, and Pareto analysis [44]. Defect causal analysis is a kind of root cause analysis based on Ishikawa diagrams and is derived from the quality management process (see 4.1.1). Cause-effect diagrams are used to visualize the causes of a problem in DCA. Figure 4-9 shows an example Ishikawa diagram with categories of plan, people, and policy.

Our method meets many of the requirements of a root cause analysis method. There are, however, some differences. We are concerned not only with what *did* cause the vulnerability, but with what *might have* caused the vulnerability. One of the reasons for this is that in many situations, there will not be sufficient evidence available to determine the actual causes.

Furthermore, RCA and DCA are typically most concerned with root causes, while we are equally concerned with contributing causes (e.g. through implementation-related, rather than design or requirements-related, activities).



*Figure 4-9: An example fishbone (Ishikawa) diagram.*

S<sup>3</sup>P supports a high degree of formalism in the representation of the analysis. This formalism is important for automation purposes when creating VCGs and SAGs. Although some RCA methods use a formal representation, it is not a general requirement, and we found Ishikawa diagrams very limited for supporting the required formalism.

#### 4.3.2 Vulnerability repositories

Information about known vulnerabilities can already be found in several publicly available repositories or databases. The examples of these repositories are the web-based vulnerability database from Security Focus [47] and the related mailing list *Bugtraq*, and the Open Source Vulnerability database [40]. These databases provide descriptions and catalogs of publicly available vulnerabilities and information about possible exploits, the affected software products and released patches. They do not provide information about how vulnerabilities are caused or how they can be prevented. Vulnerability modeling in S<sup>3</sup>P is a complement to these efforts in the way that the in-depth analysis in vulnerability modeling provides understanding of the vulnerability and its causes. This information is used to prevent the vulnerability and to identify common causes of vulnerabilities to prevent similar problems.



### 4.3.3 Vulnerability classifications

There are several approaches to classifying software vulnerabilities. For example, Aslam et al. [5] define a classification of security faults in the UNIX operating system with two classes: coding faults, and emergent faults (improper installation of software). Another example is Krusl's [29] approach that classifies vulnerabilities based on the assumptions that programmers make regarding the environment in which the software will be used. The classes are design flaws, environmental flaws, coding flaws and configuration flaws.

Vulnerability modeling could benefit from these classification efforts, as vulnerabilities in the same class are likely to have similar causes. Classification efforts might also benefit from vulnerability analysis, as vulnerabilities with similar causes are likely to be related in some way. The analysis of one form of buffer overflow vulnerability and example IP fragmentation vulnerability are published by Krsul et al. [30]. The analysis attempts to identify the characteristics of the vulnerabilities and the kinds of policy violations by their exploitation. This kind of information can be used during the initial step of vulnerability modeling, as it provides a thorough understanding of the vulnerability in question.

### 4.3.4 Threat modeling

Threat modeling [52], [57] is mainly performed during the design phase of development and the goal is to understand what kind of threats the software faces and how adversaries might try to attack the system. Threat modeling consists of following activities [57]:

1. Scoping the process: threat modeling of an entire product is too complex and realistic threat modeling can be performed on the components of the system. In order to threat model the individual components, it is possible to enumerate all dependencies and entry points and list manageable number of components or it is possible to perform a high-level analysis of each feature of the system and then combine the logically related features to get an overview of the threats of whole system.
2. Gathering background information to understand the software system: this information is entered into the *threat-model document* and this document assists the entire process.
3. Identifying the entry points to software like interfaces to other software, hardware and users: entry points represent interfaces

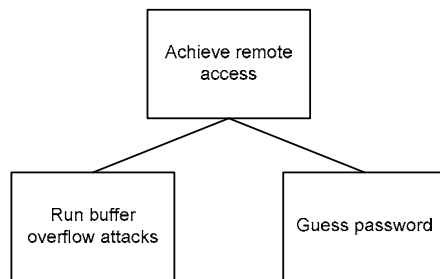
with other software, hardware, and users. Any part of the system that sends or receives data to or from an external entity is an entry point.

4. Obtaining threats: after gathering all information about the system a brainstorming session is arranged to enumerate all potential threats.
5. Resolving threats: when potential threats are listed, a meeting with the key stakeholders is scheduled and everyone in the meeting agrees on the resolution of threats.
6. Following up: follow up is needed to track and verify the assumptions made in previous steps and to consider changes.

Threat modeling is performed as part of Microsoft's SDL (see 4.2.1).

#### 4.3.5 Attack trees

Attack trees [45] model attacks against a system in a tree structure. The root node in the tree represents the goal (a successful attack) and different ways of achieving the goal is represented by leaf nodes. Attack trees can be used in both requirements and design phases. A simple example attack tree is shown in Figure 4-10.



*Figure 4-10: An example attack tree.*

Threat modeling, attack trees, and vulnerability modeling aim at improving the understanding of security issues so they can be dealt with effectively. Vulnerability modeling models causes where the direct or indirect actors are software developers. Threat modeling and attack trees deal with the attacker's behavior. All of these approaches aim at developing an understanding of security problems and identifying countermeasures.

## Chapter 5

### Conclusions and future work

---

#### **5.1 Conclusions**

Security is rapidly becoming one of the most important issues in software engineering. Software security aims to reduce exposure to security incidents by reducing the number of vulnerabilities in software. In this thesis we have identified four problems related to the current status of software security (see 1.3) and we have presented our approach, which aims at addressing these problems:

1. Today, security is often an afterthought when developing software and most of the currently available mechanisms aim at software security after software is already built. We believe that security needs to be built into the software from the early phases of software development, and security-related activities need to be performed throughout the software life cycle. In order to accomplish this we present our approach that starts from the beginning of software life cycle and introduce security in a structured way. The security plug-in based on our approach systematically introduces security activities to the software development process.
2. Current approaches to software security are either process-specific solutions or they are mostly ad hoc applications of security best practices and it is difficult to say what vulnerabilities are prevented by applying a

certain set of these practices. S<sup>3</sup>P is a process-agnostic solution and can be introduced to both requirements-driven and agile processes by a security plug-in. VCGs and SAGs in S<sup>3</sup>P provides information about the costs and benefits of applying security best practices.

3. The problem with both process specific techniques and best practices is that none of these approaches evolve the development process to meet new threats. One of the properties of our security plug-in is that it supports the idea of sustainable security. It is designed to help its users continuously take care of new threats and vulnerabilities as they evolve.
4. We focus on analysis of vulnerabilities based on published knowledge about possible attacks and exploits against them to gain a detailed understanding of the causes of vulnerabilities. This use of available published data about vulnerabilities and attacks helps to identify common causes of vulnerabilities and to prevent a wide range of vulnerabilities and classes of vulnerabilities.

In addition, VCGs and SAGs can be used in other applications, such as process analysis and estimating risk of vulnerabilities. It can also be used to assess existing development processes.

In this thesis we presented a prototype of our security plug-in, which shows how S<sup>3</sup>P can be introduced to the OpenUP/Basic development process. In this prototype we focused on using the results of S<sup>3</sup>P in a development team. We did not discuss how these results might be used at an organizational level; however our case study (see 2.2.5 and 2.3.5) showed that S<sup>3</sup>P helps to identify the causes outside the development process (e.g. employing underqualified developers).

The process of developing the security plug-in for OpenUP/Basic helped us to identify issues that must be considered when deploying S<sup>3</sup>P. For example, we identified that defining interactions between S<sup>3</sup>P and the development process is an important step when deploying S<sup>3</sup>P, and roles in the plug-in need to be connected to the roles in the development process. We also found that we might need tasks that are not performed in S<sup>3</sup>P but they must be included in the development process to support S<sup>3</sup>P (e.g. identifying and reporting potential problems discovered during software development, and support for risk analysis to decide on which vulnerabilities to prevent).

The process of developing the security plug-in shows that the S<sup>3</sup>P part of the security plug-in is process agnostic and it is the structure of the plug-in that follows the structure of the development process. In the case of

OpenUP/Basic, the security plug-in is designed based on both static and dynamic aspects of OpenUP/Basic. We evaluated this plug-in; the evaluation results helped us to identify the strengths and weaknesses of our plug-in and possible directions for our future work.

## **5.2 Future work**

We have identified several directions for our future work:

### **5.2.1 Improvements to S<sup>3</sup>P**

We are currently working on attaching costs to activities in SAGs to quantify tradeoffs between cost and security, and optimize the selection of activities given a desired cost or level of mitigation. When costs are attached to security activities they can be selected manually or automatically. Manual selection requires no tool and there is no need for optimization, since the user can choose the set of activities that fits their situation. Note that complete SAGs tend to be large graphs and there is a need to support automation in activity selections. This is a part of our future work to define how the optimal solution can be defined and what the optimal solution is to satisfy the semantic function of SAGs.

Currently, we assume that mitigation techniques (sometimes several in conjunction) are 100% effective at eliminating vulnerability causes. We are unable to model mitigation techniques that merely reduce the risk of a vulnerability. One of our priorities for the future is extending VCGs and SAGs so risk-reducing mitigation techniques can be accurately modeled and combined. However, cost modeling and modeling risk-reducing mitigation techniques should clarify preferences between actions in a more flexible way.

The current versions of VCGs and SAGs do not express the costs of applying security activities, the order in which they are applied, or the constraints. To some extent, order is implied by the actions, but we expect that explicit ordering will be needed at some point. Similarly, it is likely that we will want to express preference or priority between actions.

### **5.2.2 S<sup>3</sup>P in commercial settings**

We have begun validating key properties of S<sup>3</sup>P and we are planning to incorporate the entire process into one or more of our partners' development processes, hoping to show that it leads to a reduction in vulnerabilities. Our results to date indicate that the vulnerability modeling step, which is the basis for the entire process, works as intended. We have also empirically

evaluated the security plug-in. The next step will be testing other steps of S<sup>3</sup>P and then a full-scale test thereafter.

The security plug-in for OpenUP/Basic is our first step in deploying S<sup>3</sup>P in a real development process. Although the process of developing this prototype helped us to identify the key aspects of deploying S<sup>3</sup>P, in order to support the deployment of S<sup>3</sup>P in various development processes (both agile and requirement-driven), a framework is needed that specifies how a security plug-in must be developed in a structured way. We use the term framework as a defined support structure that specifies which steps need to be performed and what elements need to be defined in order to create a security plug-in based on S<sup>3</sup>P. Research must also be conducted to define how the plug-in is developed with respect to different process models. The evaluation of the security plug-in shows that it is necessary to consider the integration with the solutions that organizations use to deal with security problems. For example, one of our industrial partners uses Common Criteria in the requirements phase and as future work we need to clarify how the artifacts of the security plug-in and Common Criteria can collaborate.

Adopting S<sup>3</sup>P in an organization requires management support and buy-in. In our prototype we have focused on integrating the security plug-in into a development process and we have not considered the issues related to integrating the resulting development process (e.g. Secure OpenUP/Basic) into an organization. One of the directions for future work is to identify the factors that motivate the deployment of security solutions in general and S<sup>3</sup>P as a concrete example in various development organizations. Research is needed to determine how adopting security solutions might affect management issues (e.g. recruitment policy and financial issues), and how security budgets and ROI (Return On Investment) on security can be managed.

### 5.2.3 SPI aiming at security

S<sup>3</sup>P is not only a security solution to be included in a software development process but also a software process improvement process. The current version of S<sup>3</sup>P uses software engineering standards and practices as activities that mitigate software vulnerabilities. As a direction for future work, we may look at the possibility of using S<sup>3</sup>P as an assessment-based SPI approach to define how S<sup>3</sup>P can be used as a method for evaluating and improving process properties from a security point of view. In order to be used as a SPI, S<sup>3</sup>P needs extensions to be able to answer two main questions: “what to improve” and “how to improve”. In order to answer these questions we need to define security properties of a software development process, metrics required for assessment and evaluation of a process from a security point of view, methods to measure the security properties and interpret the

measurements, and methods to identify the improvements to development processes. More research is needed to explore how to make use of improvement results.

One further research area in this regard arises from differences in the nature of agile and requirements-driven development processes. SPI strategies for security must provide support for both of these software development methods. We believe that the research area of SPI for improving development processes from a security point of view is in its infancy and requires more attention and investment.

#### **5.2.4 Taxonomy of causes and activities**

In order to verify that VCGs and SAGs are sufficiently expressive for vulnerability modeling and visualizing cause mitigations, we need to completely characterize the set of vulnerabilities, their causes, and associated mitigation techniques. We currently have partial characterizations of a number of problems related to buffer overflows.

Our empirical work has shown that when modeling vulnerabilities and identifying cause mitigations, the process of identifying cause candidates and defining mitigation activities depend on the creativity of S<sup>3</sup>P team members. We have not formalized the process of labeling the nodes in VCGs and SAGs yet. Such formalism supports the use of information in VCGs and SAGs for automation purposes.

Taxonomies of causes and mitigation activities can facilitate both of these processes and increase the performance of S<sup>3</sup>P team (e.g. in terms of creating complete models in a short time). Since vulnerabilities in the same classes tend to have common causes, such taxonomies can be developed gradually by modeling examples of the classes of vulnerabilities and as an essential part of the VAD. We have already started to create a taxonomy for buffer overflow vulnerability class, in which various vulnerabilities and memory structures are involved. Buffer overflows together with format string vulnerabilities, covert storage channel vulnerabilities and injection problems are range and type errors [48]. The next step can be extending the taxonomy to these classes of vulnerabilities as well as environmental problems, serialization and timing errors and protocol errors. The taxonomy of causes can show the distribution of causes in various phases of the software development process and this information can be used if statistics are needed when improving a software development process.

Additionally, knowledge gathered in the taxonomy of causes and the relationship between causes and vulnerabilities may provide the required background to see if we can identify combinations of the causes that has not been identified yet and might lead to potential vulnerabilities. This might

lead us to determine if vulnerabilities can be predicted and removed before they are introduced into software products.

### 5.2.5 Tool support

S<sup>3</sup>P can be applied without the use of any tools but tool support is helpful for effective application of the process. Tools for finding potential vulnerabilities, visualization and modeling tools, tools to estimate costs of activities, and tools to visualize and make activity selections are needed. In addition to the aforementioned tools, in order to ensure the application of selected security activities during software development, there is a need for mechanisms such as workflow engines that can assist the use of e.g. security checklists to ensure that activities are really performed during software development.

Currently S<sup>3</sup>P does not mandate any process for initial analysis during vulnerability modeling and our examples are known vulnerabilities that are already reported in literature. One of the future work directions is to use available knowledge e.g. attack patterns; identify possible vulnerability patterns in source code; and develop tools based on these patterns to assist in the initial analysis step. The solution can be extended to other software artifacts such as requirements specification and design specification.

Several types of tools are currently available to help to improve security in software development (e.g. verification and validation of formal specifications, design tools, and static analysis tools). These tools need process support to be maximally effective in software development. We believe modeling vulnerabilities, making these models openly available, and making development tools and methods take advantage of them will contribute to the creation of more secure software. Future work directions related to this issue might be identifying formal security models to express security information e.g. vulnerabilities, causes of vulnerabilities, attacks, and their mitigation techniques; providing openly available repositories of security information; and integrating access to this information into development tools and methods. This direction is the subject for a European project (FP7) called SHIELDS<sup>10</sup> and our research group is one of the actors involved with this project.

---

<sup>10</sup> For more information see <http://er-projects.gf.liu.se/~shields>.



## References

---

- [1] W. R. Adrion, “Research methodologies in software engineering”, *Workshop on Future Directions in Software Engineering*, Software Engineering Notes, Summary of the Dagstuhl, vol. 18, no. 1, 1993.
- [2] S. Ardi, D. Byers, P. H. Meland, I. A. Tøndel, N. Shahmehri, “How can the developer benefit from security modeling?”, *Proceedings of the ARES 2007 International Workshop on Secure Software Engineering (SecSE07)*, Vienna, Austria, April 2007.
- [3] S. Ardi, D. Byers, N. Shahmehri, “Towards a structured unified process for software security”, *Proceedings of the ICSE 2006 workshop on Software Engineering for Secure Systems (SESS06)*, Shanghai, China, 2006.
- [4] A. Altmannsberger, F. Buhler, M. Rowe, “A rational unified process (RUP) plug-in to support requirements quality assurance”, *Midwest Instructions and Computing Symposium 2006*, [http://www.micsymposium.org/mics\\_2006/papers/AltmannsbergerBuhlerRowe.pdf](http://www.micsymposium.org/mics_2006/papers/AltmannsbergerBuhlerRowe.pdf), accessed August 2007.
- [5] T. Aslam, I. Krsul, E. Spafford, “Use of a taxonomy of security faults”, *Proceedings of the 19<sup>th</sup> NIST-NCSC National Information System Security Conference*, 1996.
- [6] V. R. Basili, “Software modeling and measurement: the Goal/Question/Metric paradigm”, *University of Maryland, CS-TR-2956, UMIACS-TR-92-96*, September 1992.
- [7] V. R. Basili, G. Caldiera, H. D. Rombach, “Experience factory”, *Encyclopedia of Software Engineering*, vol. 1, ed. JohnWiley & Sons, 1994.

## REFERENCES

- [8] K. Beck, "Embracing change with extreme programming", *IEEE Computer*, vol. 32, no. 10, pp. 70-77, 1999.
- [9] A. Borg, M. Patel, K. Sandahl, "Extending the OpenUP/Basic requirements discipline to specify capacity requirements", *Proceedings of the 15<sup>th</sup> International Requirements Engineering Conference (RE 2007)*, Delhi, India, 2007.
- [10] D. Byers, S. Ardi, N. Shahmehri, C. Duma, "Modeling software vulnerabilities with vulnerability cause graphs", *Proceedings of the International Conference on Software Maintenance (ICSM06)*, Philadelphia, USA, September 2006.
- [11] D. Byers, N. Shahmehri, "Design of a process for software security", *Proceedings of the International Conference on Availability, Reliability and Security (ARES07)*, Vienna, Austria, April 2007.
- [12] D. Byers, N. Shahmehri, "A cause-based approach to preventing software vulnerabilities", *Proceedings of the International Conference on Availability, Reliability and Security (ARES08)*, Barcelona, Spain, March 2008.
- [13] D. N. Card, "Learning from our mistakes with defect causal analysis", *IEEE Software*, vol. 15, no. 1, 1998.
- [14] CERT Coordination Center, CERT/CC statistics 2000-3Q 2007.
- [15] Common Criteria Portal, <http://www.commoncriteriaportal.org>, accessed Sept. 2007.
- [16] C. L. S. Coryn, "The fundamental characteristics of research", *Journal of MultiDisciplinary Evaluation*, no. 5, ISSN 1556-8180, September 2006.
- [17] N. Davis, W. Humphrey, S. T. Redwine, G. Zibulski, G. McGraw, "Processes for producing secure software", *Security & Privacy Magazine, IEEE*, vol. 2, no. 3, pp. 18-25, May-June 2004.
- [18] W. E. Deming, "Out of the crises: quality, productivity and competitive position", *MIT Center for Advanced Engineering Study*, Cambridge, MA, 1986.
- [19] J. N. Drouin, "The SPICE project", *Elements of Software Process Assessment and Improvement*, ed. K. El Emam, N. Madhavji, IEEE Computer Society, Los Alamitos, CA, 1999.

## REFERENCES

- [20] Eclipse Process Framework Project (EPF), <http://www.eclipse.org/epf>, accessed June 2007.
- [21] X. Ge, R. F. Paige, F. A. C. Polack, "Agile development of secure Web applications", *Proceedings of the International Conference on Web Engineering*, CA, USA, 2006.
- [22] M. Howard, D. LeBlanc, "*Writing secure code*", Microsoft Press, Second Edition, 2003.
- [23] M. Howard, "Building more secure software" *Security & Privacy Magazine, IEEE*, vol. 2, no. 6, pp. 63-65, November-December 2004.
- [24] K. Ishikawa, "*What is total quality control? The Japanese way*", Prentice Hall, Englewood Cliffs, NJ, 1985.
- [25] ISO/IEC 15504-9, "Information technology-software process assessment-part 9: Vocabulary", *Technical Report*, 1998.
- [26] ISO/IEC 9000-3, "Guidelines for the application of ISO 9001:1994 to the development, supply, installation, and maintenance of computer software", *Internal Organization for Standardization*, 1997.
- [27] I. Jacobson, G. Booch, J. Rumbaugh, "*Unified software development process*", Addison-Wesley, 1999.
- [28] S. Komi-Sirviö, "Development and evaluation of software process improvements methods", *VTT Publications 535*, 2004.
- [29] I. Krsul, "Software vulnerability analysis", *PhD thesis, Purdue University*, 1998.
- [30] I. Krsul, E. Spafford, M. Tripunitra, "An analysis of some software vulnerabilities", *Proceedings of the 21<sup>st</sup> NIST-NCSC National Information Systems Symposium*, pp. 111-125, 1998.
- [31] R. C. Linger, "Cleanroom process model", *IEEE Software*, vol. 11, no. 2, pp. 50-56, March 1994.
- [32] S. B. Lipner, "The trustworthy computing security development life cycle", *Proceedings of the 20<sup>th</sup> Annual Computer Security Application Conference*, pp. 2-13, December 2004.
- [33] G. McGraw, "Building secure software: better than protecting bad software", *IEEE Software*, vol. 19, no. 6, pp. 57-58, November-December 2002.

## REFERENCES

- [34] G. McGraw, "From the ground up: The DIMACS software security workshop", *IEEE Security & Privacy*, vol. 1, no. 2, pp. 59-66, March-April 2003.
- [35] G. McGraw, "Software security", *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80-83, March-April 2004.
- [36] N. R. Mead, "Security requirements engineering", *Build Security In 2006-08-10*, <http://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/requirements/243.html>, accessed August 2007.
- [37] A. P. Moore, R.J. Ellison, R. C. Linger, "Attack modeling for information security and survivability", *Dependable Systems and Networks Conference*, Gothenburg, Sweden, 2001.
- [38] MySQL reference Manual, [http://www.yesky.com/imagesnew/software/mysql/manual\\_Unireg.html](http://www.yesky.com/imagesnew/software/mysql/manual_Unireg.html), accessed September 2007.
- [39] OpenUP Component, [http://www.eclipse.org/epf/openup\\_component/open\\_up\\_index.php](http://www.eclipse.org/epf/openup_component/open_up_index.php).
- [40] OSVDB. The open source vulnerability database. <http://osvdb.org>, accessed September 2007.
- [41] S. R. Palmer, J. M. Felsing, "*A practical guide to feature-driven-development*", Prentice Hall, 2002.
- [42] S. L. Pfleeger, "*Software Engineering: theory and practice*", Prentice Hall, Second Edition, 2001.
- [43] S. T. Redwine, N. Davis, "Task force for improving security across the development life cycle", *Task Force Report, Appendix B: processes to produce secure software*, 2004.
- [44] Root cause analysis, *Office of Financial Management*, <http://www.ofm.wa.gov/rmd/erm/root.asp>, accessed September 2007.
- [45] B. Schneier, "Attack trees: modeling security threats", *Dr. Dobb's Journal*, December 1999.
- [46] R. Scumacher, R. Ackermann, R. Steinmetz, "Towards security at all stages of a system's life cycle", *Proceedings of the International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2000.

## REFERENCES

- [47] SecurityFocus. Security focus vulnerability database. <http://www.securityfocus.com/vulnerabilities>.
- [48] Secure Software, Inc., "The CLASP application security process", <http://www.securesoftware.com>, accessed June 2007.
- [49] M. Siponen, R. Baskerville, T. Kuivalainen, "Integrating security into agile development methods", *Proceedings of the 38<sup>th</sup> Hawaii International Conference on System Sciences*, 2005.
- [50] K. Soo Hoo, A. W. Sundbury, A. R. Jaquith, "Tangible ROI through secure software engineering", *Secure Business Quarterly*, vol. 1, no. 2, Fourth Quarter 2001.
- [51] E. Spafford, "The Internet worm program: An analysis", *Computer Communication Review*, vol. 19, no. 1, 1989.
- [52] F. Swiderski, W. Snyder, "Threat modeling", Microsoft Professional, 2004.
- [53] Team SHATTER Security Alert, Application Security Inc. <http://www.appsecinc.com/resources/alerts/mysql/2005-002.html>, accessed September 2007.
- [54] The common vulnerabilities and exposures list, <http://cve.mitre.org>.
- [55] H. H. Thompson, J. A. Whittaker, "Rethinking software security", *Dr. Dobbs's Journal*, January 2004.
- [56] R. Turner, "Seven pitfalls to avoid on the hunt for best practices", *IEEE Software*, vol. 20, no. 1, 2003.
- [57] P. Torr, "Demystifying the threat-modeling process", *IEEE Security & Privacy*, vol. 3, no. 5, pp. 66-70, 2005.
- [58] US-CERT/NIST. National vulnerability database, <http://nvd.nist.gov>.
- [59] US-CERT/NIST, Vulnerability summary CVE-2005-2558, National Vulnerability Database, <http://nvd.nist.gov/nvd.cfm?cvename=CVE-2005-2558>.
- [60] A. J. A. Wang, "Information security models and metrics", *Proceedings of 43<sup>rd</sup> ACM Southeast Conference*, 2005.



## Appendix A

# The OpenUP/Basic Development Process

---

### OpenUP/Basic

OpenUP/Basic is an open source development process, developed as part of the Eclipse Process Framework (EPF) [20]. OpenUP/Basic is a subset of OpenUP that takes an agile approach to software development. The essential characteristics of the Rational Unified Process (RUP) are preserved in OpenUP, including iterative development, use cases, and risk management. OpenUP/Basic targets small teams of 3-6 people and 3-6 months of development effort. The version of OpenUP/Basic we use in this thesis is version 0.9 [39] (OpenUP/Basic version 1.0 is in its review stage)<sup>11</sup>.

OpenUP/Basic contains the fundamental contents of RUP providing a simplified set of work products, roles, tasks, and guidance. The focus is more on the collaboration and stakeholder benefits than on unnecessary deliverables and formality. Four core principles characterize OpenUP/Basic [39]:

- Collaborate to align interests and share understanding
- Balance competing priorities to maximize stakeholder value
- Focus on articulating the architecture
- Evolve continuously to obtain feedback and improve

---

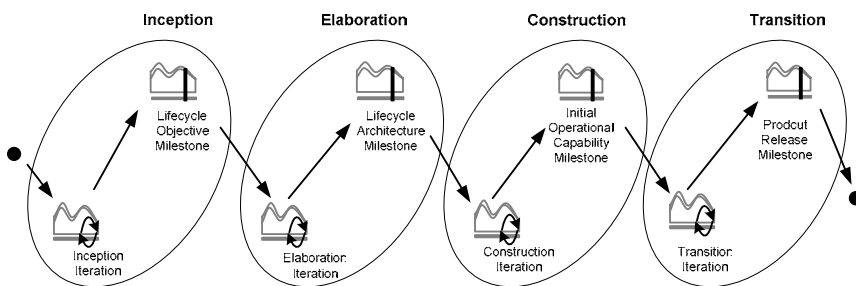
<sup>11</sup> The review version is available in <http://www.epfwiki.net/wikis/openupreview/>

Based on RUP, the OpenUP basic process has two aspects: dynamic and static. The dynamic aspect is over time and expresses the process in terms of cycles, phases, iterations and milestones. The static aspect describes the process in terms of activities, artifacts, tasks, roles and workflows.

Based on the dynamic organization of OpenUP/Basic, the development process is broken into development cycles and each cycle has four phases:

- **Inception phase:** This phase contains activities to initiate the project, manage the iterations, manage requirements, and determine architectural feasibility.
- **Elaboration phase:** The main objectives for elaboration are related to better understanding requirements, creating and establishing a baseline for the architecture for the system, and mitigating top-priority risks.
- **Construction phase:** The architecture should be stable when this phase starts and functionality is continuously implemented and tested.
- **Transition phase:** The main objectives are to fine-tune functionality, performance, and overall quality of the beta product from the end of construction phase.

Every cycle works on a new generation of product and each phase in the cycle has a well-defined milestone (see Figure A-1) [39].



*Figure A-1: Phases in OpenUP/Basic.*

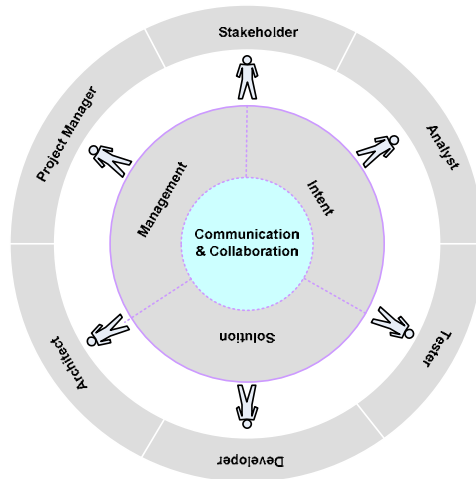
Based on the core principles, the content of OpenUP/basic is organized into four major areas also known as sub-processes: collaboration and communication, intent, solution, and management. The collaboration and



communication sub-process is the foundation for OpenUP/Basic. The management sub-process deals with the management of the projects, including project planning, iteration planning, and iteration assessment. The intent sub process deals with how to transfer the intent of stakeholders to the rest of the development team, to ensure that the intent of the stakeholder will be reflected by validated builds. The solution sub-process describes all aspects of creating the architecture, designing, implementing, and testing the application. Figure A-2 shows the organization of contents in OpenUP/Basic. Each sub-process contains four kinds of *modeling elements* which are:

- **Roles** refer to responsibilities. These are stakeholder, analyst, architect, developer, tester and project manager (see Figure A-2).
- **Tasks** refer to activities to be performed by roles (organized by disciplines).
- **Work products** contain all artifacts as inputs and outputs of tasks (organized by domains).
- **Workflows** are sequence of tasks (defined in capability patterns).

Roles perform tasks that consume and produce artifacts. OpenUP/Basic describes the minimal set of roles, tasks, and artifacts involved in software development.



**Figure A-2: Organization of content in OpenUP/Basic.**

The distribution of the modeling elements in each of the sub-processes is shown in Table A-1. OpenUP/Basic supports the use of *Guidance* which is a general term for supplemental information that can be added to most elements. For example, the modeling elements are linked to *concepts* that are a kind of guidance to outline key ideas or basic principles behind the elements; *checklists* are another type of guidance to be used in reviews such as walkthroughs or inspections. An example checklist in OpenUP/Basic is *Qualities of good requirements*, which provide guidance on assessing the quality of requirements.

The process description of OpenUP/Basic is published as a set of web pages, linked based on different perspectives including phases and disciplines [39]. An example page that describes the artifact *Architectural Proof-of-Concept* is shown in Figure A-3. This artifact is a work product kind of concept, and its relationships with other elements are described in this page. As it is shown in Figure A-4, there is a guideline that describes how to select and validate the architectural proof-of-concept.

The collaboration and communication sub-process is not included in Table A-1. This sub-process involves all roles, and instead of containing tasks and artifacts, provides *a set of practices* that motivate effective collaboration. These practices help development team members to jointly define the intent of the stakeholder, jointly develop the solution and jointly manage the project. The provided practices are applied to all of sub-processes. The practices are:

**Maintain a common understanding:** Project participants should share information and use work products that helps to align understanding between the stakeholder and developers. These work products could be Vision, Work item list and Requirements.

**Foster a high-trust environment:** Planning activities in detail, and supervising and auditing them can create a high-trust environment. Project participants should feel safe communicating their ideas and taking initiatives.

**Share responsibility:** While assigning primary responsibilities of work products to individuals, sharing responsibilities of work products supports team members in asking for help when needed.

**Learn continuously:** Continuous development of technical and interpersonal skills is very important for members of a software development team. This will help with personal development as software development field is evolving.

**Organize around the architecture:** As the size of projects grows the communication between team members become complex. Organizing around the architecture will provide team members a common vocabulary and shared mental model of the system.

Sub-processes Elements	Intent	Management	Solution
<b>Disciplines</b>	Requirements Configuration and change management	Project management	Analysis and design Implementation Test
<b>Work products (Domain)</b>	Requirements - Actors -Glossary - Supporting requirements - Use cases - Usa case model - Vision	Project management - Iteration plan - Project plan - Risk list - Status assessment - Work item list	Architecture - Architectural proof of concept - Architecture Development - Build - Design - Developer test - Implementation Test - Test case - Test log - Test script
<b>Capability patterns</b>	Manage requirements Ongoing tasks	Initiate project Plan and manage Iteration	

*Table A-1: The distribution of modeling elements in sub-processes.*

## Disciplines

Tasks to be performed during the OpenUP/Basic development process are organized into six disciplines: requirement, analysis and design, implementation, test, project management, and configuration and change management.

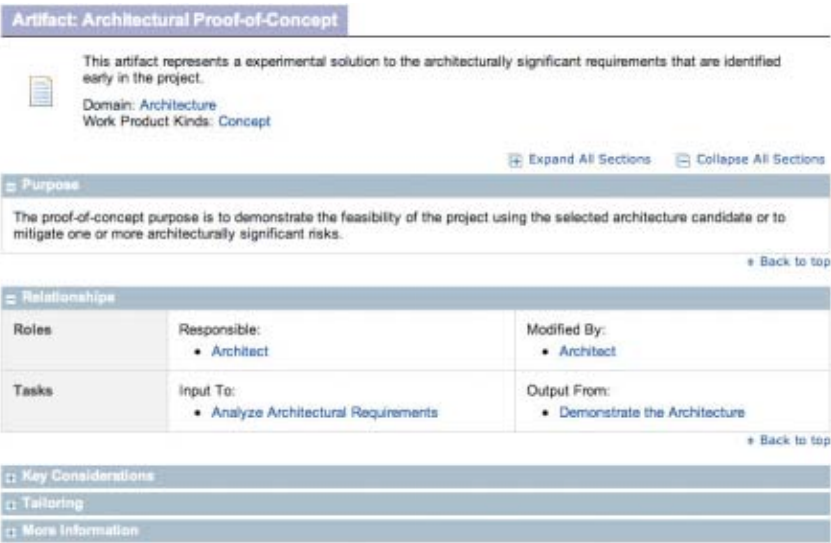
**Requirement discipline:** The tasks specified for the Requirements discipline of OpenUP/Basic are:

- **Define vision:** The analyst role negotiates with the stakeholder to gain agreement on the problem to solve, capture a common vocabulary and identify the constraints on the system.
- **Define requirements:** The purpose of this task is to describe one or more requirements in sufficient detail to validate understanding of the requirements. The responsible role is analyst.

- **Find and outline requirements:** This task is performed to identify and capture domain terms and to communicate the requirements to the development team.

**Analysis and design discipline:** The specified tasks in this discipline are:

- **Analyze architectural requirements:** The architect identifies architectural goals and architectural constraints and captures architectural decisions.
- **Demonstrate the architecture:** The architect illustrates at least one architecture that supports the requirements of the system. This illustration reduces the risk of reworking the software architecture.

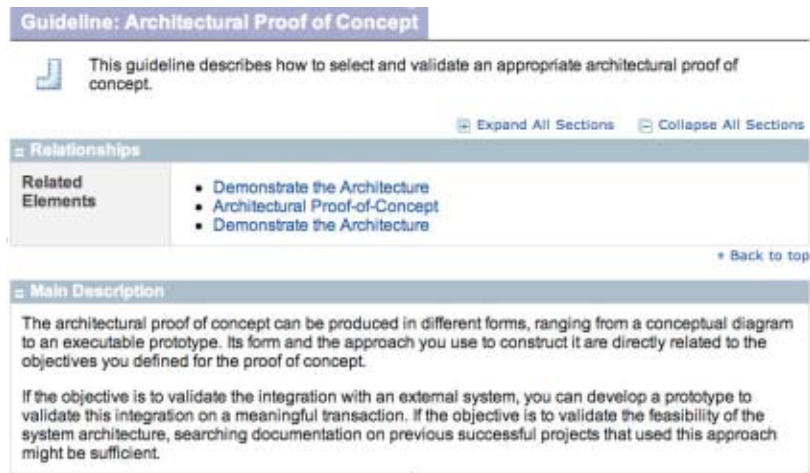


*Figure A-3: Snapshot of an artifact.*

- **Design the solution:** This task is about designing part of the system not the whole and the developer describes the elements of the system so that they support the required behavior.
- **Develop the architecture:** The architect provides a skeletal design to enable more comprehensive design activities to be performed coherently by the team.

**Implementation discipline:** This discipline aims to incrementally build the system and to verify that technical units used to build the system work as specified. The discipline contains three tasks:

- **Implement developer tests:** The developer refines the scope and identifies the test. This test is based on the expected behavior of code units.
- **Implement solutions:** The developer determines the strategy and transforms design into implementation; writes source code and evaluates the implementation.
- **Run developer tests:** To verify that the implementation works as specified, the developer runs tests and evaluates execution of tests.



*Figure A-4: Snapshot of a guideline.*

**Test discipline:** The purpose of this discipline is to find and document defects and to validate the requirements, design and implementation. The tasks in this discipline are:

- **Create test cases:** The tester examines the requirements to be tested and identifies test data.
- **Implement tests:** The tester selects appropriate implementation techniques and implements tests.
- **Run tests:** The tester runs and evaluates test results.

**The project management discipline:** The goal is to keep the team focused on continually delivering a tested software product for stakeholder evaluation.

- **Plan project:** The project manager describes a roadmap that provides direction to the team and continually adapts it based on feedback and changes in the environment.
- **Plan iteration:** The project manager defines the iteration objectives, produces a detailed plan and defines the evaluation criteria.
- **Assess results:** The project manager gathers stakeholder feedback and assesses the results.
- **Manage iteration:** The project manager identifies and manages risks and handles exceptions and problems.

**Configuration and change management discipline:** This discipline explains how to control changes to artifacts, ensuring a synchronized evolution of the set of work products composing a software system. The only task in this discipline is:

- **Request change:** Any role in the development team can gather request information and update the work item list to document the information that is gathered in the previous steps.

## Domain

The artifacts (work products) that are used by or that resulted from tasks are organized into domains. These domains are:

**Architecture domain:** Two artifacts are defined in this domain.

- **Architectural proof-of-concept** demonstrates the feasibility of the project and is used to explore the understanding of problem domain.
- **Architecture** is used to concentrate on the structure of the design, essential elements and key scenarios.

**Development domain:** The artifacts in this domain are:

- **Build** is the operational version of the system that demonstrates the capabilities provided by the final product.
- **Design** describes the system in terms of components to serve the understanding of the functionality of the system.
- **Developer test** validates the performance of the individual software components.
- **Implementation** contains software code files including source, binary or executable code, data files and documentation files.

**Project management domain:** The artifacts related to this domain are as follows:

- **Iteration plan:** The iteration plan describes the objectives, work assignments and the evaluation criteria of the iteration.
- **Project plan:** This artifact gathers all information required to manage the project.
- **Risk list:** This is a sorted list of known and open risks to the project.
- **Status Assessment:** This artifact captures the results that show if the project is on track and if there are opportunities for improvements.
- **Work item list:** This artifact contains a list of all scheduled work to be done as well as the proposed work that may affect the product.

**Requirements domain:** This domain contains six artifacts:

- **Actor:** This artifact represents a set of rules interacting with the system. The analyst uses this artifact to define the system boundaries and to identify external interfaces. The developers use it to capture characteristics of human actors when creating user interfaces.
- **Glossary:** This artifact defines important terms used by the project. The goal is to provide a common vocabulary for all stakeholders.
- **Supporting requirements:** This artifact captures system-wide requirements that are not captured in scenarios and use cases.
- **Use case:** This artifact shows the sequence of actions a system performs and is used to capture the behavior of the system from the end user's point of view.
- **Use case model:** This artifact is used to present an overview of the system as a base for agreement on the functionality of the system between the customer and the project team.
- **Vision:** This is a high-level conceptual view of the system.

**Test domain:** The artifacts related to this domain are:

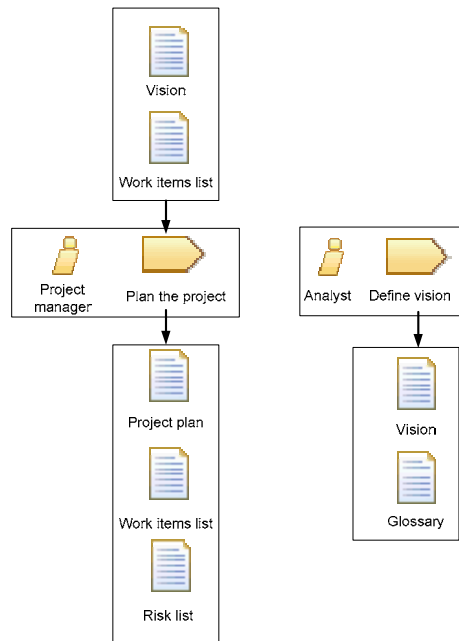
- **Test case** is used to specify the set of test inputs, execution conditions and expected results.
- **Test log** collects raw outputs during each unique execution of a test to provide verification that a set of tests was executed and to provide information related to the success of those tests.

- **Test script** contains step by step instructions that realize a test. This can be documented textual instructions that are executed manually.

## Capability Patterns

Capability patterns describe the workflow of each sub-process by defining the activities to be performed in the form of tasks and work products. The capability patterns of each sub-process are shown in Table A-1. For example Figure A-5 shows structure of the work breakdown for “Initiate project” capability pattern. The capability patterns in OpenUP/Basic are:

**Manage requirements:** This capability pattern belongs to intent sub-process and describes the tasks performed to gather, specify, analyze and validate a subset of system’s requirements prior to implementation and verification.



**Figure A-5: The work breakdown structure for “Initiate project”.**

**Ongoing tasks:** This is another capability pattern that belongs to the intent sub-process and includes a single task, *Request Change*. This task may occur anytime during the life cycle in response to an observed defect, a desired enhancement, or a change request.



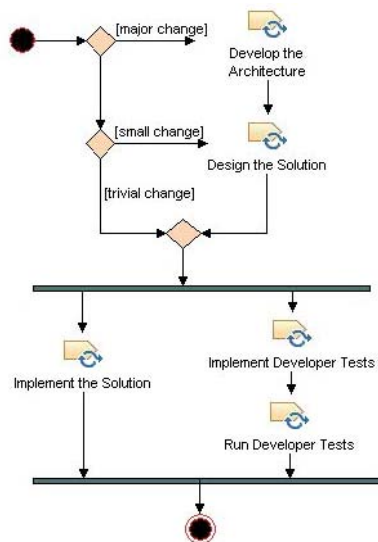
**Initiate project:** This capability pattern belongs to the management sub-process. This capability pattern describes the activities that take place at the beginning of the first iteration, when the project starts to establish the vision for the project.

**Plan and manage iteration:** This capability pattern describes how to initiate an iteration and assign work to team members, and belongs to the management sub-process.

**Define the architecture:** This capability pattern explains the activities to complete the architecture for iteration and belongs to the solution sub-process.

**Determine architecture feasibility:** This capability pattern in the solution sub-process confirms that the project is feasible by constructing an architectural proof-of-concept.

**Develop solution:** The project manager uses this capability pattern to perform goal-based planning and management. During each iteration this capability pattern occurs multiple times (one instance for each requirement planned for the iteration). Figure A-6 shows how this capability pattern is used.



*Figure A-6: The overview of workflow for developing solution.*

**Test:** This capability pattern describes the activities to create and run tests to validate the system according to the intent.



## Appendix B

### Summary of the Evaluation

---

#### **Answers to the process conformance section**

##### Importance:

- Different products have different security requirements; the security plug-in is one way of achieving security.
- Important for most products (for all defects)
- Extremely important for most products

##### Training:

- Small amount of time (training course)
- Small amount of time (training course)
- Self training (plug-in published webpage)

##### Staff-Months:

- Don't know yet
- Don't know yet (based on the acceptance factors, 4 man-week to setup everything to start)
- Only the security team is involved.

##### Schedule:

- Adjustable to projects
- Adjustable to projects
- Adjustable to projects

##### Frequency and length:

- One security iteration e.g. every second month
- For one vulnerability the time spent is one day
- One iteration for 4-10 defects, at least once per development iteration

Detailed staff-time:

- Don't know yet
- Don't know yet
- Depends on the vulnerability, e.g. one vulnerability:
  - Security report: one hour
  - Security checklist creation: two hours
  - Vulnerability modeling: one hours
  - Identifying techniques and using checklist: ?

Reason for not accepting:

- Formalism in modeling (VCG and SAG)
- New process is difficult to be accepted (solved if used for all defects)
- No problem for accepting

Following principles: (1-5)

- 3
- 3
- Depends on the acceptance (usually quick fixes are appreciated rather than processes)

Problem in deployment:

- Acceptance, root causes are not usual in real life and the security process should be integrated with CC
- How to find vulnerabilities in first place? Who decides which is bug and not vulnerability? What is the relation to security requirements and assets to be protected?
- No problem

Your team supports use of checklists:

- Yes
- Yes
- Yes and no, first day it is used and then forgotten. (Parsers may help)

Addressing security requirements: (1-5)

- 3 (CC is good for handling and tracking security requirements, why use the security plug-in)
- 3 (CC is good and prevents building vulnerabilities into software. S<sup>3</sup>P can be used after development)
- 2 (Security requirements have not been mentioned in the description of S<sup>3</sup>P)

### **Answers to domain conformance section**

Products of organization:

- Embedded systems
- PC applications
- Communication products

Role of subject in development team:

- System and software engineer, responsible for software competence
- Project manager
- Developer

Team size:

- 4
- 3-10
- 5-10

Familiarity with OpenUP: (1-5)

- 2
- 2
- Just theory

Influence on the customer satisfaction: (1-5)

- 3
- 3 (Customers are already concerned about well-defined security processes)
- A security flaw destroys our business completely

Use of security handbooks:

- For most products
- For some products (CC)
- Essential to development process (architecture, pre-design and design, not on a daily basis but essential to understand)

## **Answers to feedback section**

Refinement:

- The security plug-in can be used for handling all kind of defects in controlled way.
- Make it possible to export security reports to CC for tracking.
- It could be used in maintenance phase (there often is a lack of process in that phase)

The quality of proposed Interface:

- Don't know
- Yes
- A bit (templates and examples are needed for artifacts)

OpenUP helped:

- Yes
- Yes
- Hard to say, I was already familiar with S<sup>3</sup>P.

Automation:

- Tool for modeling VCG and SAG
- Checklists should be performed by parsers if possible
- Don't know

Time spent to answer the questionnaire:

- 4 hours
- 3 hours
- 3 hours

## Appendix C

### Acronyms

---

API	Application Program(ming) Interface
CERT	Computer Emergency Response Team
CERT/CC	CERT Coordination Center
CLASP	Comprehensive Lightweight Application Security Process
CVE	Common Vulnerabilities and Exposures
DCA	Defect Causal Analysis
EPF	Eclipse Process Framework
FDD	Feature Driven Development
GQM	Goal-Question-Metric
ISO	International Standards Organization
NIST	National Institute of Standards and Technology
QIP	Quality Improvement Paradigm
RCA	Root Cause Analysis
RUP	Rational Unified Process
ROI	Return On Investment
S <sup>3</sup> P	Sustainable Software Security Process
SAG	Security Activity Graph
SDL	Secure Development Life cycle
SEI	Software Engineering Institute
SPI	Software Process Improvement
SPICE	Software Process Improvement and Capability dEtermination
SW-CMM	Capability Maturity Model for Software
TQC	Total Quality Control
UML	Unified Modelling Language
VAD	Vulnerability Analysis Database
VCG	Vulnerability Cause Graph

# List of Tables

---

Table 3-1: Extensions to OpenUP/Basic.....	46
Table A-1: The distribution of modeling elements in sub-processes. ....	85



## List of Figures

---

Figure 1-1: Software security vulnerabilities reported to CERT/CC. ....	2
Figure 1-2: Security plug-in in the context of software life cycle.....	7
Figure 2-1: The workflow of S <sup>3</sup> P.....	10
Figure 2-2: Simplified UML model of VCG. ....	11
Figure 2-3: Visual representation of VCG. ....	11
Figure 2-4: A sequence in VCG.....	12
Figure 2-5: A conjunction in VCG. ....	12
Figure 2-6: A simple vulnerability cause graph. ....	13
Figure 2-7: The structure of files in MySQL. ....	17
Figure 2-8: VCG of CVE-2005-2558, Iteration 1. ....	18
Figure 2-9: VCG of CVE-2005-2558, Iteration 2. ....	18
Figure 2-10: VCG of CVE-2005-2558, Iteration 3. ....	18
Figure 2-11: VCG of CVE-2005-2558, Iteration 4. ....	19
Figure 2-12: VCG of compound node, Iteration 1. ....	19
Figure 2-13: VCG of compound node, Iteration 2. ....	20
Figure 2-14: VCG of compound node, Iteration 3. ....	20
Figure 2-15: Vulnerability cause graph for CVE-2005-2558, with expanded compound node. ....	21
Figure 2-16: Visual representation of SAG. ....	24
Figure 2-17: A security activity graph for a cause. ....	24

## LIST OF FIGURES

Figure 2-18: UML model for Basic SAGs. ....	24
Figure 2-19: Example vulnerability and SAGs of individual causes. ....	25
Figure 2-20: Complete SAG for vulnerability V.....	26
Figure 2-21: The SAG for missing range check cause.....	30
Figure 2-22: The SAG for use of unsafe function for string copying. ....	31
Figure 2-23: Verification procedure for not using strmov. ....	32
Figure 2-24: The SAG for use of non-adaptive buffers. ....	32
Figure 2-25: The SAG for copy of external data to internal buffers. ....	32
Figure 2-26: The SAG for use of C-like strings.....	33
Figure 2-27: The SAG for wrong source size is used. ....	33
Figure 2-28: The screen shot of the user interface of VAD. ....	34
Figure 2-29: The structure of S <sup>3</sup> P.....	35
Figure 3-1: A security report.....	39
Figure 3-2: Example page of a security checklist. ....	40
Figure 3-3: Example of the interaction between S <sup>3</sup> P and a software development process. ....	41
Figure 3-4: The overview of workflow for security iteration.....	43
Figure 3-5: The security auditor.....	44
Figure 3-6: The security developer. ....	44
Figure 3-7: The workflow and artifacts in the security plug-in.....	45
Figure 3-8: The organization of content in Secure OpenUP/Basic. ....	47
Figure 3-9: Security report for CVE-2005-2558.....	47
Figure 3-10: Complete SAG for CVE-2005-2558. ....	48
Figure 3-11: Security checklist to prevent CVE-2005-2558.....	48
Figure 4-1: Control Circle [28]. ....	56
Figure 4-2: Software security best practices applied to software artifacts [35].....	58
Figure 4-3: The standard Microsoft development process [32]. ....	59
Figure 4-4: SDL improvements to the Microsoft development processes [32].....	60
Figure 4-5: An example activity in CLASP. ....	61
Figure 4-6: One example from the root-cause database in CLASP. ....	62

Figure 4-7: Description of FDD process..... 63

Figure 4-8: Secure web application development process. .... 64

Figure 4-9: An example fishbone (Ishikawa) diagram..... 66

Figure 4-10: An example attack tree..... 68

Figure A-1: Phases in OpenUP/Basic. .... 82

Figure A-2: Organization of content in OpenUP/Basic. .... 83

Figure A-3: Snapshot of an artifact..... 86

Figure A-4: Snapshot of a guideline. .... 87

Figure A-5: The work breakdown structure for “initiate project”..... 90

Figure A-6: The overview of workflow for developing solution..... 91





**LINKÖPINGS UNIVERSITET**

**Avdelning, institution**  
Division, department

Institutionen för datavetenskap

Department of Computer  
and Information Science

**Datum**  
Date

2008-03-18

**Språk**

Language

☐

Svenska/Swedish

☒

Engelska/English

☐

\_\_\_\_\_

**Rapporttyp**

Report category

☒

Licentiatavhandling

☐

Examensarbete

☐

C-uppsats

☐

D-uppsats

☐

Övrig rapport

**URL för elektronisk version**

**ISBN**

978-91-7393-956-0

**ISRN**

LiU-Tek-Lic-2008:11

**Serietitel och serienummer**

Title of series, numbering

**ISSN**

0280-7971

Linköping Studies in Science and Technology

Thesis No. 1353

**Titel**

Title

A Model and Implementation of a Security Plug-in for the Software Life Cycle

**Författare**

Author

Shanai Ardi

**Sammanfattning**

Abstract

Currently, security is frequently considered late in software life cycle. It is often bolted on late in development, or even during deployment or maintenance, through activities such as add-on security software and penetration-and-patch maintenance. Even if software developers aim to incorporate security into their products from the beginning of the software life cycle, they face an exhaustive amount of ad hoc unstructured information without any practical guidance on how and why this information should be used and what the costs and benefits of using it are. This is due to a lack of structured methods.

In this thesis we present a model for secure software development and implementation of a security plug-in that deploys this model in software life cycle. The model is a structured unified process, named S<sup>3</sup>P (Sustainable Software Security Process) and is designed to be easily adaptable to any software development process. S<sup>3</sup>P provides the formalism required to identify the causes of vulnerabilities and the mitigation techniques that address these causes to prevent vulnerabilities. We present a prototype of the security plug-in implemented for the OpenUP/Basic development process in Eclipse Process Framework. We also present the results of the evaluation of this plug-in. The work in this thesis is a first step towards a general framework for introducing security into the software life cycle and to support software process improvements to prevent recurrence of software vulnerabilities.

**Nyckelord**

Keywords

Software security, Vulnerability modelling, Plug-in, Software development process, Software life cycle.



**Linköping Studies in Science and Technology**  
**Faculty of Arts and Sciences - Licentiate Theses**

- No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
- No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
- No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.
- No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
- No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.
- No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
- No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
- No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.
- No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.
- No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
- No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
- No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.
- No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
- No 113 **Ralph Rönquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
- No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
- No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.
- No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
- No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.
- No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
- No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
- No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
- No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.
- No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.
- No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.
- No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.
- No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
- No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.
- No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
- No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.
- No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
- No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
- No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.
- No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
- No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
- No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
- No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.
- No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.
- No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
- No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.
- No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
- No 318 **Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
- No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
- No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.
- No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.
- No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
- No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.
- No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.
- No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
- No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.
- No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.
- No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
- No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
- No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
- No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
- No 398 **Johan Boye:** Dependency-based Groudnness Analysis of Functional Logic Programs, 1993.

- No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
- No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
- No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
- No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
- No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
- No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
- No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
- FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
- FHS 4/94 **Karin Pettersson:** Informationssystemstrukturer, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
- No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
- No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
- No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
- No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
- No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
- No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
- FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
- No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
- No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
- No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
- No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
- No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
- No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
- No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
- No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
- FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
- No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
- No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
- No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
- No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
- No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
- No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
- FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
- FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
- No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
- No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
- No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
- No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
- No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
- No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
- No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
- FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
- No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
- No 550 **Kristina Larsen:** Föresättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.
- No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
- No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
- No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
- No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
- No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
- No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
- No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
- No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
- No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
- No 591 **Niclas Wahlöf:** A Default Extension to Description Logics and its Applications, 1996.
- No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
- No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.



- No 598 **Rego Granlund:** C<sup>3</sup>Fire - A Microworld Supporting Emergency Management Training, 1997.
- No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
- No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
- No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
- FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
- FiF-a 6 **Tommy Wedlund:** Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
- No 615 **Silvia Coradeschi:** A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
- No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
- No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
- No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
- No 629 **Gunilla Iwefors:** Krigsspel och Informationsteknik inför en oförutsägbart framtid, 1997.
- No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
- No 639 **Jukka Mäki-Turja:** Smalltalk - a suitable Real-Time Language, 1997.
- No 640 **Juha Takkinen:** CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
- No 643 **Man Lin:** Formal Analysis of Reactive Rule-based Programs, 1997.
- No 653 **Mats Gustafsson:** Bringing Role-Based Access Control to Distributed Systems, 1997.
- FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
- No 674 **Marcus Bjärelund:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
- No 676 **Jan Håkegård:** Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
- No 668 **Per-Ove Zetterlund:** Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
- No 675 **Jimmy Tjäder:** Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
- FiF-a 14 **Ulf Melin:** Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
- No 695 **Tim Heyer:** COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
- No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
- FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
- No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
- No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
- No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
- No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
- No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
- No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
- No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
- No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
- FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
- FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
- No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
- No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
- FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
- No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
- No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
- No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
- No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
- No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
- No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
- No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
- No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
- No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
- FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
- No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
- No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
- No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
- No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
- No 800 **Anders Subotic:** Software Quality Inspection, 1999.
- No 807 **Svein Bergum:** Managerial communication in telework, 2000.

- No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
- FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
- No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
- No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
- No 823 **Lars Hult:** Publika Gränssytor - ett designexempel, 2000.
- No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
- FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
- No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
- No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
- FiF-a 37 **Ewa Braf:** Organisationens kunskapsverksamheter - en kritisk studie av "knowledge management", 2000.
- FiF-a 40 **Henrik Lindberg:** Webbaseade affärsprocesser - Möjligheter och begränsningar, 2000.
- FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
- No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
- No 863 **Dan Lawesson:** Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
- No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
- No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
- No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
- FiF-a 47 **Per-Arne Segerkvist:** Webbaseade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
- No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
- No 906 **Lin Han:** Secure and Scalable E-Service Software Delivery, 2001.
- No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
- No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
- FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
- FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
- No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
- No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
- No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
- No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
- No 938 **Bourhane Kadmiry:** Fuzzy Control of Unmanned Helicopter, 2002.
- No 942 **Patrik Haslum:** Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
- No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
- FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
- No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
- No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
- No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
- FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
- No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
- No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
- No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
- No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
- No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.
- No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002
- No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
- No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
- No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
- FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
- No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
- No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
- No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
- No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
- No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
- No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
- No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Trigged Distributed Embedded Systems, 2003.
- FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.

- No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.  
 No 1034 **Arja Vainio-Larsson:** Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.  
 No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.  
 FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.  
 No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.  
 No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.  
 No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.  
 FiF-a 71 **Emma Eliason:** Effekttanalys av IT-systems handlingsutrymme, 2003.  
 No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.  
 No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.  
 FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.  
 No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.  
 No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.  
 FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.  
 No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.  
 No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.  
 No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.  
 No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.  
 No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.  
 FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.  
 No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.  
 No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.  
 No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.  
 No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.  
 No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.  
 No 1149 **Vaida Jakoniene:** A Study in Integrating Multiple Biological Data Sources, 2005.  
 No 1156 **Abdil Rashid Mohamed:** High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.  
 No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.  
 No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.  
 FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.  
 No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.  
 No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.  
 No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.  
 FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.  
 No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.  
 FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.  
 No 1172 **Petter Ahlström:** Affärsstrategier för seniorbostadsmarknaden, 2005.  
 No 1183 **Mathias Cöster:** Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.  
 No 1184 **Åsa Horzella:** Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.  
 No 1185 **Maria Kollberg:** Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.  
 No 1190 **David Dinka:** Role and Identity - Experience of technology in professional settings, 2005.  
 No 1191 **Andreas Hansson:** Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.  
 No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.  
 No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.  
 No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.  
 No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.  
 No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.  
 No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.  
 No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.  
 No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.  
 No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.  
 No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.  
 No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.  
 No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.  
 No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.  
 No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation-What are the Barriers and Enablers, 2006.  
 FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.  
 No 1272 **Raquel Flodström:** A Framework for the Strategic Management of Information Technology, 2006.

No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.

No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.

FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.

No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006

No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.

No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.

No 1303 **Daniel Andreasson:** Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.

No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.

No 1306 **Gustaf Svedjemo:** Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.

No 1307 **Gianpaolo Conte:** Navigation Functionalities for an Autonomous UAV Helicopter, 2007.

No 1309 **Ola Leifler:** User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.

No 1312 **Henrik Svensson:** Embodied simulation as off-line representation, 2007.

No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.

No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.

No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.

No 1323 **Magnus Lundqvist:** Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.

No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.

No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.

No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.

No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.

No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.

No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.

No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.

No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.