EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Extending the agile development process to develop acceptably secure software

*Citation for published version (APA):*
Ben Othmane, L., Angin, P., Weffers, H. T. G., & Bhargava, B. (2013). *Extending the agile development process to develop acceptably secure software*. (Computer science reports; Vol. 1306). Technische Universiteit Eindhoven.

*Document status and date:*
Published: 01/01/2013

*Document Version:*
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

*Please check the document version of this publication:*

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

*General rights*
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

*Take down policy*
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

# Extending the Agile Development Process to Develop Acceptably Secure Software

Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava

13/06

ISSN 0926-4515

All rights reserved
editors: prof.dr. P.M.E. De Bra
prof.dr.ir. J.J. van Wijk

Reports are available at:
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Author&level=1 and
http://library.tue.nl/catalog/TUEPublication.csp?Language=dut&Type=ComputerScienceReports&Sort=Year&Level=1

Computer Science Reports 13-06
Eindhoven, July 2013

# Extending the Agile Development Process to Develop Acceptably Secure Software

Lotfi ben Othmane[1], Pelin Angin[2], Harold Weffers[1], and Bharat Bhargava[2]

[1] Laboratory for Quality Software, Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands
{l.ben.othmane,h.t.g.weffers}@tue.nl
[2] CERIAS and Computer Sciences, Purdue University, USA
{pangin,bbshail}@purdue.edu

**Abstract.** The agile software development approach makes developing secure software challenging. Existing approaches for extending the agile development process, which enables incremental and iterative software development, fall short of providing a method for efficiently ensuring the security of the software increments produced at the end of each iteration. This paper (a) proposes a method for security reassurance of software increments and demonstrates it through a simple case study, (b) integrates security engineering activities into the agile software development process and uses the security reassurance method to ensure producing acceptably secure–for the business owner–software increments at the end of each iteration, and (c) discusses the compliance of the proposed method with the agile values and its ability to produce secure software increments.

**Keywords:** Agile software development, secure software, security assurance cases.

## 1 Introduction

Developing *secure software* that continue to function correctly under malicious (intended) attacks [1] requires integrating security engineering activities and verification and validation gates into the development process. The *security engineering activities* capture and refine protection requirements and ensure their integration into the software through purposeful security design [2]. The verification and validation gates ensure traceability [3] of analysis, design, coding, and testing artifacts; which helps addressing the weakest link (i.e., least protected point) problem [4] by ensuring completeness of the protection mechanisms.

The sequential software development approach suits the integration of the security engineering activities, commonly used in sequence, and the use of verification and validation gates between the development stages: analysis, design, coding, and testing. For example, Microsoft had a sequential software development method that integrates security activities [5] [6].

In contrast, the iterative and incremental nature [7] of the Agile Software Development (ASD) [8] approach enables developing software in regular intervals, i.e., iterations, producing the software in increments. The iterative and incremental nature of the ASD approach limits its ability to accommodate the security engineering activities and the use of verification and validation gates. The development process it employs does not fit the sequential use of the security engineering activities and the set of verification and validation gates.

There are several challenges that limit the use of ASD for developing secure software [9,10,11,12]. The proposed solutions for extending the ASD process to produce secure software, e.g., [13,14,15] fall short of ensuring the security of the increments produced in each iteration. It is also not possible to implement all security requirements of software in the first iteration of the software and perform all security verification and validation tasks, (e.g., OWASP verification requirements [16]) during each iteration of the software.[3]

This paper proposes extending the agile development process to support producing secure software increments. It addresses two questions:

1. How to efficiently ensure the security of software increments produced by development iterations?
2. How to extend the agile development process to support security engineering activities and produce acceptably secure software in each iteration?

We answer the first question through proposing a method for security reassurance of software increments. The common approach for security assurance is to use checklists of security verifications (e.g., [16]) that require the verification of all the security requirements for each software increment. We address the issue through using security assurance cases [17]. The tree-like structure of assurance cases supports efficient security reassurance of the software increments.

We answer the second question through extending the agile development process with security engineering activities and use of the proposed security reassurance method while preserving the ASD values.

In the following we provide an overview of the agile software development approach, secure software development, and security assurance cases (Section 2), discuss related work (Section 3 ), describe our method for security reassurance of software increments (Section 4), propose a process for developing acceptably secure software using the ASD approach (Section 5), discuss the proposed method (Section 6) and conclude the paper (Section 7).

## 2 Background

This subsection gives an overview of the ASD approach, secure software development and security assurance cases.

[3] The problem, in our opinion, is similar to passing an elephant through the eye of a needle.

### 2.1 Agile software development approach

Seventeen software developers met on February 2001 in the Wasatch Mountains of Utah, USA, aiming to share their perceptions of software development [8]. They agreed on four values that the software methods they created share: individuals and interactions, working software, customer collaboration, and responding to change. They crafted a manifesto that includes these values and named the approach they created the Agile Software Development approach.

The ASD approach is implemented by several methods including: Scrum [18], Extreme Programming (XP) [19], Agile Modeling (AM) [20], and Feature-Driven Development (FDD) [21]. It enables producing potentially shippable working software at regular intervals [20](i.e., iterations), which enables providing customers with high value features (customer-valued product functionalities) in a short time. It accommodates several classes of software, such as Web applications [22]. It applies a greedy-like approach with incomplete information for selecting functionalities to develop.[4] The approach relies on the use of patterns, principles, and best practices for developing "good" software.

The agile approach has several advantages. First, it reduces the chance of project failure because it enables early detection of gaps between business expectations and developer understanding. Second, it enables discovery of customer needs rather than customer wishes since customers can see demos of the product while being developed and adapt the requirements based on their needs. Third, it enables early discovery of technical barriers since the developers experiment their ideas and use the results to adapt the system architecture and work plan.

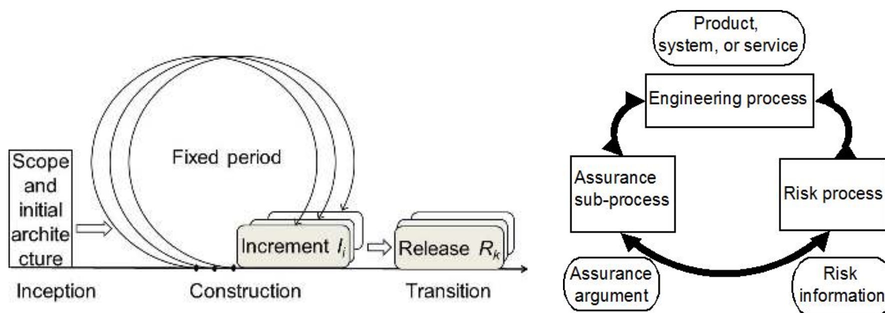


**Fig. 1.** The agile software development process.

**Fig. 2.** CMM security engineering process.

The ASD methods share a similar process, shown in Figure 1. The process supports developing software in an iterative and incremental fashion. It has 3

[4] The greedy approach (the term "greedy algorithm" is used in the literature) makes the choice that looks best at the moment; that is, makes the locally optimal choice with the hope to have an optimal solution [23].

phases (cf. [24]): inception, construction, and transition. The *inception phase* is for defining the scope of the project and model of the initial architecture. The *construction phase* is for developing the software in a set of iterations. For each iteration, the business owner and development team determine the goal of the iteration and select a set of user stories to achieve the goal. Then, they elicit the requirements for the stories, update the design to address the requirements, and code a software increment that addresses the requirements and is potentially shippable. They demonstrate the user stories and review the team efficiency at the end of the iteration. The *transition phase* is for integration testing and for hardening the increment to make it ready as a release[5] for use in a production environment.

### 2.2 Secure software development

Secure software are developed using processes that integrate security activities for capturing and refining protection requirements and for ensuring their integration into the software through purposeful security design [2]. A known reference model of engineering secure software is the System Security Engineering-Capability Maturity Model (SSE-CMM) [25]. Figure 2 shows the Capability Maturity Model (CMM) security engineering process (SSE-CMM). The process has three sub processes: risk process, engineering process, and assurance process. The risk process enables identifying threats to and vulnerabilities of a given system along with their associated impacts and likelihood of occurrence [26]; that is, their risks. The security engineering process supports determining and implementing solutions to the threats. The security assurance process ensures that the security features (high-level security requirements that express protection capabilities of the software to mitigate the threats [27]), practices, procedures, and architecture of software accurately mediate and enforce the security policy [2]. *Security policies* state the required protection of the information objects [28]; they are rules for sharing, accessing, and using information, hardware, and software.

For instance, Figure 3 demonstrates that it is not possible to claim that a software product mitigates a security risk unless all its components, together, are verified to mitigate the risk.

### 2.3 Security assurance cases

Security assurance enables developing coherent objective arguments–which could be reviewed–to support claiming that a software product mitigates its security risks. A *security assurance case* [17], a semi-formal approach for security assurance, is a collection of security-related claims, arguments, and evidences where a *claim*, i.e., a security goal, is a high-level security requirement, an *argument* is a justification that a set of (objective) evidences justify that the related claim is satisfied, and an *evidence* is a result of a verification through, for example,

[5] A software could have several releases.

**Microsoft Bob: a Design Flaw**

Microsoft Bob would pipe up when the program determined that the user was stuck doing something. Bob's most insecure function occurred when a user attempted three times (unsuccessfully) to type in his or her password. Bob would pop up and proclaim: "I see you have forgotten your password, please enter a new password." Then the user was allowed to change the password even though the user apparently had no idea of the old one.

Microsoft Bob, Hacker's friend.

**Fig. 3.** Example of (a possibly apocryphal story) security flaw [1].

security testing, source code security review, mathematical proofs, checking use of secure coding standards, qualification of the developers in terms of training on developing secure software (cf.[29]), etc.

Ensuring that software mitigates a security risk requires:

- ensuring that collected evidences indeed support the related claim. For example, a source code static analysis of a Web application cannot justify that the communication between a client and a Web server is secure–it only supports the claim.
- having evidences that sufficiently justify the claims.[6] For example, a verification of compliance with standards for writing secure code [31] helps avoiding source code vulnerabilities, such as buffer overflow [32], but does not justify a claim that the code is free from source code vulnerabilities.
- evaluating and addressing conflicts and dependencies between both the claims and the evidences.

A security assurance case has a tree structure, where the root is the main claim, intermediate nodes are either sub-claims or arguments, and the leaves are the evidences. A common way to represent assurance cases is to use the Goals-Structuring Notation (GSN) [33]. *Goals-Structuring Notation*[7] is a graphical argumentation notation that represents each element of the assurance case and the relationships between these elements. Figure 4 shows an example of security assurance cases that uses GSN.

The main steps of creating a security assurance case (cf. [35,36]), in sequence, are:

1. **Identify the claims**–decompose the claim "the software is secure" into sub-claims such that satisfying the sub-claims induce satisfying the claim. The sub-claims (which in turn become claims) could be iteratively subdivided, until getting verifiable sub-claims.

[6] Note that security is a system property [30].

[7] GSNs could be traced back to McDermid's work [34], but without the graphical notations.

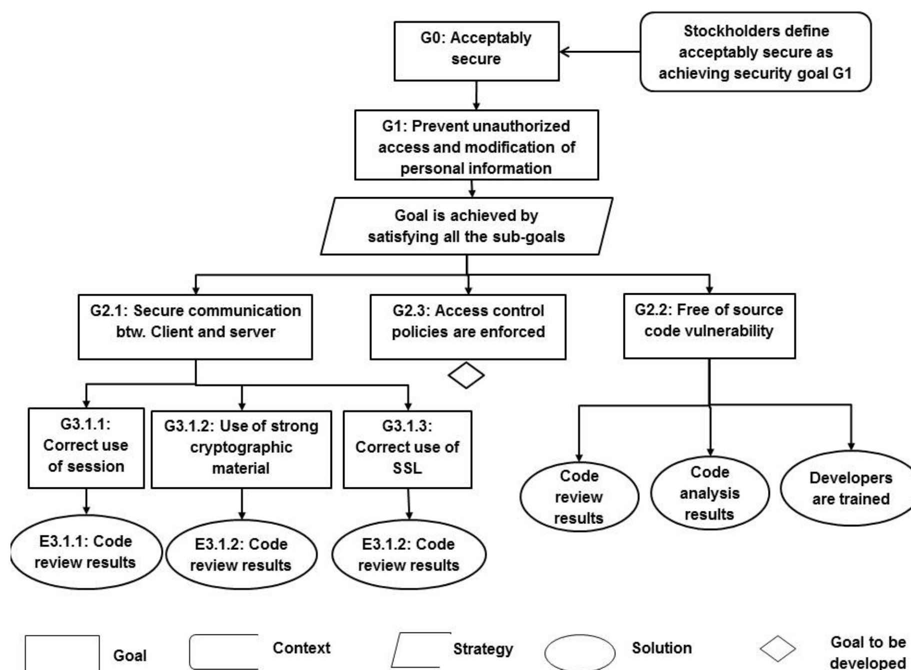


**Fig. 4.** Example of security assurance case using GSN.

2. **Establish the context**–specify additional information for claims, such as definitions, reference to documents, explanations, and assumptions.
3. **Identify the strategies**–provide information on how a claim is decomposed into sub-claims. The strategy could be explicitly described in the assurance case or be implicit if no strategy is specified.
4. **Identify evidences**–collect the result of using the security evaluation techniques, such as security testing and security review of source code to evaluate the security countermeasures [27] used to eliminate or reduce the risk of the threats to the software and achieve the related security goals.
5. **Specify the arguments**–show implicitly or explicitly that an evidence supports a claim. For example, the results of a security analysis tool may report that the software has a set of code security vulnerabilities (e.g., buffer overflow). The argument describes that the "errors" are false positives and the claim is satisfied.

There are two classes of security claims for software: business security claims and technical security claims. *Business security claims* are security features that implement protection mechanisms and are perceived by the customers, e.g., protection against unauthorized access and use of sensitive data, secure communication between a client and a server and enabling activities audit. *Technical security* claims are coding and configuration best practices that aim to prevent

bypassing the security features. Examples are validation of all input data, use of strong cryptography, handling all errors and exceptions, secure use of security configuration, and checking existence of malicious code.

The two main advantages of security assurance cases over checklists are (a) richness of argumentation and (b) completeness of decomposition. Security assurance cases provide richness of argumentation because they record the evidences, arguments, assumptions, and contexts that justify why the evaluator[8] believes that a claim is satisfied. For instance, it records all results generated by a code security analysis tool, lists false positives and the arguments for ignoring them. In contrast, checklists require to assert for each claim[9] whether it is satisfied, but does not justify how the evidence supports the related claim.

## 3 Related work

This section describes three approaches for extending the agile development process or methods to enable developing secure software.

**OWASP approach.** The Open Web Application Security Project (OWASP) [13] proposes developing evil user stories (hacker abilities to compromise the system), and expressing them in a conversational style. An example for an evil story is: As a hacker, I can send bad data in URLs, so I access data for which I'm not authorized. The stories address authentication, session management, access control, input validation, output encoding/escaping, cryptography, error handling and logging, data protection, communication security, and HTTP security features.

The developers mitigate all applicable evil stories in every iteration and perform a security-focused code review at the completion of each iteration. Successful completion of code review includes initial code review, addressing potential issues found in the initial code review, and passing a re-review.

**Microsoft approach.** Sullivan [14,37,38] proposed integrating security-related activities into the development life-cycle considering their required frequency of completion. He divides the activities into three groups based on their required completion frequency. The groups are: one-time activities, cycle security-related activities, and bucket security-related activities. The one-time activities group includes activities that are performed only once in each project, during the preparation phase. The cycle security-related activities group includes activities that are performed in each iteration of the project. The bucket security-related activities group includes one verification task, one design review task, and one response planning task. Verification tasks include, for example, attack surface analysis; design review tasks include, for example, review of code that uses cryptographic operations; and response planning tasks include, for example, update of security response contacts.

[8] In general, an evaluator is a specialized professional who evaluates whether the software complies with the requirements and how it does so.

[9] Checklists use the term security requirement to mean a claim.

**Risk-driven secure software development.** Vähä-Sipilä [15,39] proposes a risk-based approach for developing secure software. The method is based on managing the security risks and implementing security solutions.

Managing security risks is reducing the risks of security threats, through implementing security controls, to a level acceptable by the business owner. The security threats include the threats related to the functionalities and the architecture of the software (e.g., as a user, I do not want my data to be used by anybody, except for processing my transaction), threats related to specific user stories, and threats related to code vulnerability and data validation. The risks of threats are reduced using security mechanisms, which are associated with implementation and operation costs.

Implementing security solutions is transforming the security user stories into functional user stories that implement security countermeasures to prevent, protect or detect threats. It also includes use of secure coding standards and security assessment activities, such as testing the existence of vulnerabilities.

## 4 Security reassurance of software increments

Evolving software, through adding user stories, requires reassessing the security assurance of the software. The reason is: the changes to the software components could invalidate the evidences and claims of the security assurance case. For example, evidences collected using a code security analysis tool become invalid if new code is added to the software. Also, the claim "access control to files is enforced" becomes invalid if the files become available in several locations.

The safe approach for assessing the security assurance of a new increment of software is to discard the security assurance case of the previous increment and perform a new security assessment for the new increment. However, performing a new security assessment of software for each increment is not practical because it is time consuming and has high cost. For instance, it is not possible to perform the 121 verifications required by OWASP [16] for a Web application that has a new increment every week.

We describe in the following our method for developing security reassurance cases of software increments. First, we analyze the security verification requirements of OWASP [16] and identify a set of common properties. Then, we analyze the relationship between security assurance case elements and software components. Next, we describe our method of security reassurance of software increments. We also demonstrate the algorithm through a case study.

### 4.1 Analysis of security verification areas of OWASP

The Open Web Application Security Project developed a standard for evaluating the security of Web applications [16]. We classified the 121 security verification requirements of the standard based on classes of security assessment techniques, locality (the assessment concerns specific components or all components of the software), and automation (the assessment is automatic or manual). Table 1

**Table 1.** Analysis of the security verification areas of OWASP [16]

| Security verification areas | Assessment technique class | Locality (Local/ Global) | Automation (Automatic/ Manual) |
|---|---|---|---|
| Security architecture | RA | 4/2 | 1/5 |
| Authentication | RA/TFU | 12/3 | 3/12 |
| Session management | RA/TFU | 5/8 | 4/9 |
| Access control | RA/TFU | 13/2 | 6/9 |
| Input validation | RA/TFU | 8/1 | 4/5 |
| Output encoding/ escaping | RA | 9/1 | 1/9 |
| Cryptography | RA | 9/1 | 0/10 |
| Error handling and logging | RA/TFU | 9/3 | 2/10 |
| Data protection | RA | 6/0 | 0/6 |
| Communication security | RA/TFU | 7/2 | 1/8 |
| HTTP security | RA | 6/1 | 3/4 |
| Security configuration | RA | 0/4 | 0/4 |
| Malicious code search | RA | 2/0 | 0/2 |
| Internal security | RA | 0/3 | 0/3 |
| | | 90/31 | 25/96 |

Note: We use the following abbreviation: RA for review and analysis and TFU for testing of security features and their use.

shows the statistics that we obtained. Based on the analysis, we observe the following:

- Assessment automation: We use assessment techniques to collect evidences. These techniques are either manual (i.e., performed by humans) or automatic (i.e., using tools and scripts). We observe that most of the security verification requirements, 79%, require manual assessment.
- Evidence locality: Some evidences apply to only parts of the software (e.g., a component, a class) and others apply to the whole software. For instance, verifying an authentication mechanism requires verifying only the web pages and business logic that implement the security feature; there is no need to assess all the software. The analysis shows that most of the verification requirements, 74%, are local.

We also observe that claims have dependencies, e.g., an access control mechanism depends on the authentication mechanism. The main relationships among 2 claims $A$ and $B$ are:

| | | |
|---|---|---|
| $A$ depends on $B$ | $\Longrightarrow$ | $B$ is a condition for $A$ |
| $A$ supports $B$ | $\Longrightarrow$ | $A$ is a condition for $B$ |
| $A$ implements $B$ | $\Longrightarrow$ | $A$ is a specialization of $B$ |
| $A$ abstracts $B$ | $\Longrightarrow$ | $A$ is a generalization of $B$ |
| $A$ conflicts $B$ | $\Longrightarrow$ | satisfying $A$ prevents satisfying $B$ |

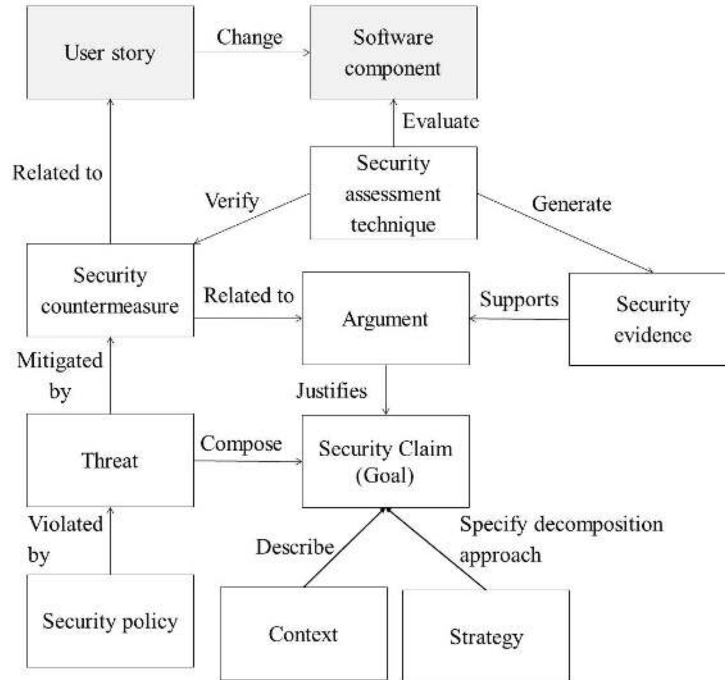


**Fig. 5.** Conceptual model of the relationship between security assurance case concepts and software development artifacts. (White rectangles model security assurance case concepts, blue rectangles model software development artifacts and arrows model relationships between two concepts.)

When analyzing the security verification requirements, we observed that checklists do not help to easily identify all the divisions of claims. For instance the requirement V7.5: "verify that cryptographic module failures are logged" does not require that the logs are protected from unauthorized access, which may be considered as a security flaw. In contrast, security assurance cases have a tree-like structure, which simplifies identifying (known) sub-claims of a claim; that is, they help ensure completeness of decomposition of claims.

### 4.2 Relationship between security assurance case elements and software components

Figure 5 shows the conceptual model of security assurance cases for agile software development. In this model, the project owner specifies the user stories and the security policies. A *user story* describes a functionality valuable to the user of the software [40]. A *security policy* states the required protection of the information objects [28].

A security claim, i.e., a goal, specifies a capability of the system to protect, prevent, detect, or deter a set of threats. For example, the security claim "prevent

unauthorized users from accessing and using the application" specifies that the software prevents the threat "unauthorized access and use of the application." A claim could be decomposed into sub-claims using a decomposition strategy and described using a context annotation.

Threats are mitigated by security countermeasures. A *countermeasure* is an action, device, procedure, or technique that counters a threat by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken [27]. Countermeasures are related to user stories, e.g., a countermeasure is implemented by a set of user stories. They are also related to arguments, e.g., secure coding could contribute to the argument of the claim "minimum source code vulnerabilities."

Implementing a user story may require adding new components,[10] removing components, and/or updating existing components–i.e., changing their structure or behavior. These operations could invalidate the elements of the security assurance case of the software, e.g., a claim becomes false. The invalidation of claims and evidences depends on the application because changing a component may or may not invalidate related evidences. *Invalidate* means "does not support" for evidences and "is not true" for claims. The invalidation types are:

I1. Changes to a context could invalidate related claims. We formulate this invalidation using the function:
$I_{cl}(C_x) = \{C_l^i\}$.

I2. Changes to a component could invalidate related evidences. We formulate this invalidation using the function:
$I_{ev}(C_o) = \{E_v^i\}$.

I3. Invalidation of evidences invalidates related claims. We formulate this invalidation using the function:
$I_{cl}(E_v) = \{C_l^i\}$.

I4. Claims could have relationships, such as dependency and conflict. Changes to a claim could invalidate a related claim. We formulate this relationship using the function:
$T_{cl}(C_l) = \{C_l^a\}$.

We use symbols $C_x$ for context, $C_o$ for component, $E_v$ for evidence, $C_l$ for claim, $C_l^i$ for invalidated claim, $E_v^i$ for invalidated evidence, and $C_l^a$ for potentially affected claim. We also use $I_{cl}(C_x)$ to denote a function that provides the claims associated with the context provided as a parameter, $I_{ev}(C_o)$ to denote a function that provides the evidences associated with the component provided as a parameter, $I_{cl}(E_v)$ to denote a function that provides the claims associated with the evidence provided as a parameter, and $T_{cl}$ to denote a function that provides the claims related to the claim provided as a parameter.

Note also that invalidation of claims makes evidences associated to the claims useless. Also, invalidation of claims and evidences makes associated arguments

[10] A *software component* is a unit of composition (and is also subject to composition) with specified interfaces and explicit context dependencies and can be deployed independently [41].

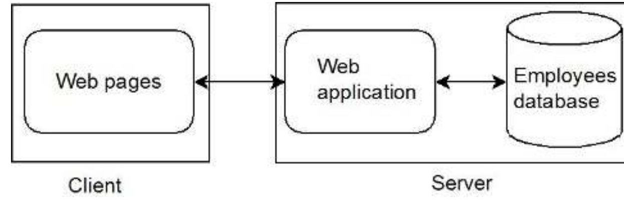


**Fig. 6.** Architecture of the employees Web portal application.

**Table 2.** List of user stories.

| Id | User story |
| --- | --- |
| U1 | As an HR officer, I want to create/view/update employee records. |
| U2 | As an employee, I want to view and update my personal information. |
| U3 | As an HR officer, I want to view and update employees' holidays. |

useless. (We do not discuss invalidation of arguments because the process is straightforward.)

### 4.3 Methodology of security reassurance of software increments

The method of security reassurance of software increments needs to maintain the security assurance cases. For instance, each assurance case shares a set of artifacts with the security assurance case of the previous iteration. The security assurance case of a new increment requires only new evidences for invalidated claims or new claims.

An efficient method for security reassurance of software increments minimizes the reassurance task, which requires minimizing the number and size of claims to evaluate. We exploit the decomposition of software into a set of components and we maintain the security assurance case based on changes to the components.

A software increment could affect the security assurance case in several ways:

- **Case 1: New claim.** The set of claims that need evaluation due to adding a new claim includes the new claim, sub-claims, and parent claims–i.e., related claims.
- **Case 2: Context change.** The set of claims that need evaluation due to context change of a claim includes the claim and related claims.
- **Case 3: Component update.** Component update could invalidate evidences and claims associated with the updated component and *potentially* affects claims and evidences associated with components related to the modified component.[11]

[11] We do not have rules to identify claims and evidences associated to the related components that become invalid. It is for the security assessor to identify such cases.

**Table 3.** Relationships between the security claims.

| Id | Security claim | Stakeholder | Dependencies |
|---|---|---|---|
| C01 | Prevent anonymous access to the software | HR officer | Supports C02 |
| C02 | Prevent unauthorized access and modification of personal information | HR officer | Depends on C01 |
| C03 | Inputs are validated | Technical | |
| C04 | Reduced source code vulnerabilities | Technical | |

– **Case 4: New component.** The set of claims that need evaluation due to adding a new component includes claims related to the new component and claims that could be affected by updating other components connected to the new component–e.g., they have a dependency with the new component.

Note that in the worst case, an iteration invalidates all claims of the security assurance case, which requires reevaluation of these claims and in the best case it preserves all the claims of the previous iteration and evaluates the claims associated with the new components integrated to the increment.

Note also that Vivas et al. [17] proposed a method for security assurance-driven software development that focuses on relating the assurance cases to the software through the development life-cycle (analysis, design, development, and test). Our method relates the security assurance case to the software as it evolves in increments.

### 4.4 Case study: Web portal for employees

In this section, we illustrate the use of our method for security reassurance of software increments to assure the security of a simple employees Web portal.

**Table 4.** Relationships between presentation and business layer components.

| Presentation Layer | Action | Business layer class | Method |
|---|---|---|---|
| Login | Input | Authentication | Authenticate |
| CreateEmployee | Input | Employee | setEmployee |
| ViewEmployee | View | Employee, Holidays | getEmployee, getHolidays |
| UpdateEmployee | View, Input | Employee | getEmployee, setEmployee |
| ManageHoliday | View, Input | Employee, Holidays | getEmployee, getHolidays, SetHoliday |
| ManageResour-ces | Input, Search, View | Resource | setResource, findResource, getResource |
| AssignAccess-Control | Input, View | AccessControl | setAccessControl, getAccess-Control |

**Table 5.** Impact of user stories on the components of the architecture.

| User story Id | Affected components |
|---|---|
| U1 | CreateEmployee (A), ViewEmployee (A), UpdateEmployee (A), Employee (A) |
| U2 | |
| U3 | ViewEmployee (U), UpdateEmployee (U), Login (U), ManageHoliday (A), Holidays (A) |
| U4 | Login (A), Authenticate (A) |
| U5 | Login(U), Resource (A), ManageResources (A), AccessControl (A), AssignAccessControl (A) |

We use the code “A” for adding a new component and “U” for updating an existing component.

**Description of the system** The employees Web portal is a Web application for viewing and updating employees’ personal information and holidays. The security policies set by the business owner are: (1) each employee can view and update his/her own personal information, and (2) HR officers can view and update all employee records. The application is to be developed using Java and run on an Apache Web server [42].

At the project inception the business owner and developers sketch the architecture of the intended software as depicted by Figure 6 and identify the initial user stories as provided in Table 2. They also identify the main security claims, which are provided in Table 3. Table 4 shows the relationship between the components of the presentation and business layers.

**Building the security assurance case** The security assurance case of the application evolves as the developers implement the user stories and produce software increments. We summarize the impacts of the user stories on the components of the software in Table 5. The table includes, besides the user stories of Table 2, a security user story for authenticating users (U4) and another for controlling use of the application (U5).

In the following, we show in detail the evolution of the security assurance case of the Employees Portal application through three development iterations.

**Iteration 1.** This iteration involves the development of the user story $U1$. The top level security claim “the system is acceptably secure” is decomposed into three sub-claims: $C01$, $C03$ and $C04$. The components of the increment are the `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` Web forms and the `Employee` class. Figure 7 shows the security assurance case of this iteration.

**Table 6.** List of evidences.

| Id | Description |
|---|---|
| E01.1 | Results of deployment architecture review |
| E01.2 | Successful tests with the authentication mechanism |
| E03.1.1 | Successful tests for invalid input to CreateEmployee |
| E03.1.2 | Code review results for CreateEmployee |
| E03.2.1 | Successful tests for invalid input to ViewEmployee |
| E03.2.2 | Code review results for ViewEmployee |
| E03.3.1 | Successful tests for invalid input to UpdateEmployee |
| E03.3.2 | Code review results for UpdateEmployee |
| E03.4.1 | Successful tests for invalid input to Login |
| E03.4.2 | Code review results for Login |
| E04.1.1 | Code security analysis results for Employee |
| E04.2.1 | Code security analysis results for CreateEmployee |
| E04.3.1 | Code security analysis results for ViewEmployee |
| E04.4.1 | Code security analysis results for UpdateEmployee |
| E04.5.1 | Code security analysis results for Login |
| E04.6.1 | Code security analysis results for Authentication |

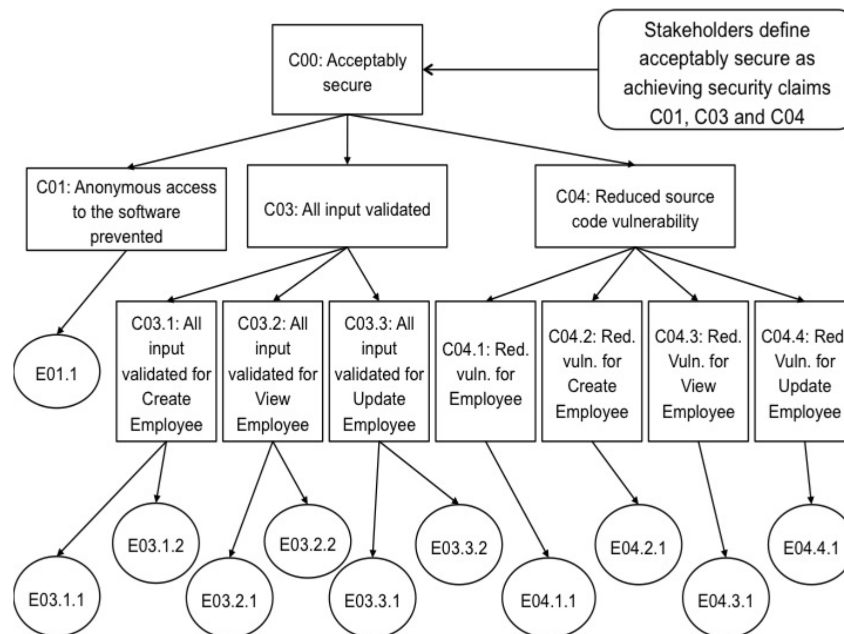


**Fig. 7.** Security assurance case for Iteration 1. (Table 6 describes the evidence codes.)

Claim $C01$ is satisfied by the evidence collected from a review of the software deployment architecture, showing that the application is hosted on a desktop accessed only by HR officers.

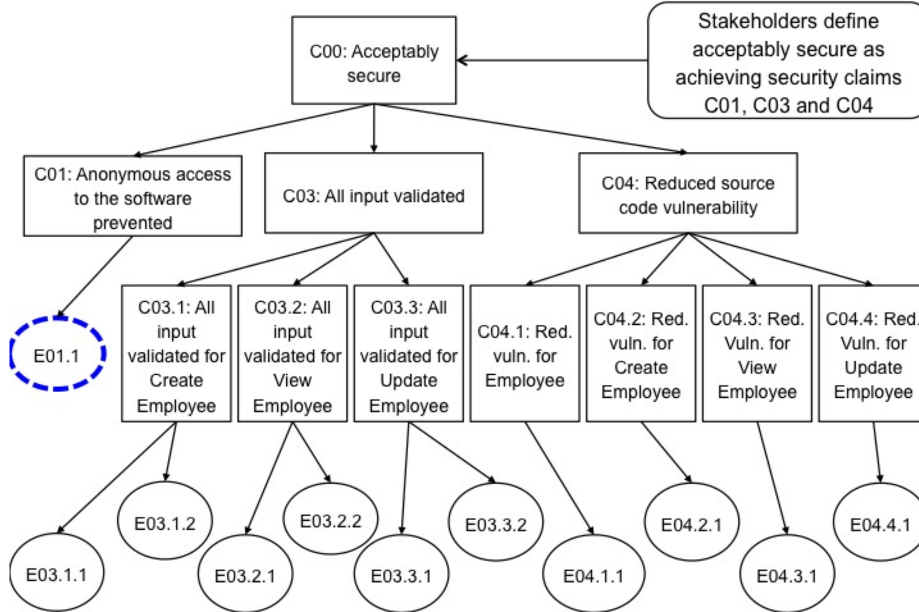


**Fig. 8.** Invalidated evidences upon the transition from Iteration 1 to Iteration 2. (Table 6 describes the evidence codes.)

Claim $C03$ is satisfied because all input for the `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` Web forms are validated, hence the sub-claims: $C03.1$, $C03.2$, and $C03.3$ are satisfied. This argument is supported by two forms of evidences: results of code review for the Web forms: `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee`; and security testing of the three Web forms with invalid input handled successfully by the application.

Claim $C04$ is satisfied because its sub-claims $C04.1$, $C04.2$, $C04.3$ and $C04.4$, which state that source code vulnerabilities were reduced as much as possible for the `Employee`, `CreateEmployee`, `ViewEmployee`, and `UpdateEmployee` components respectively, are satisfied. The claim is supported by the results from the code security analysis tool run on the software components.

**Iteration 2.** This iteration involves implementing user story $U2$. The resulting increment could be used by HR officers to create, view, and update employee records and by employees to view and update their records. Implementing user story $U2$ requires hosting the application on a server accessible by HR officers and employees. The top level security claim is composed of the three sub-claims $C01$, $C03$, and $C04$ as before. The change of the deployment architecture invalidates the argument of evidence $E01.1$. An alternative approach to satisfy claim $C01$ (anonymous access to the software is prevented) is to develop an authentication mechanism, which we formulate as security user story $U4$.

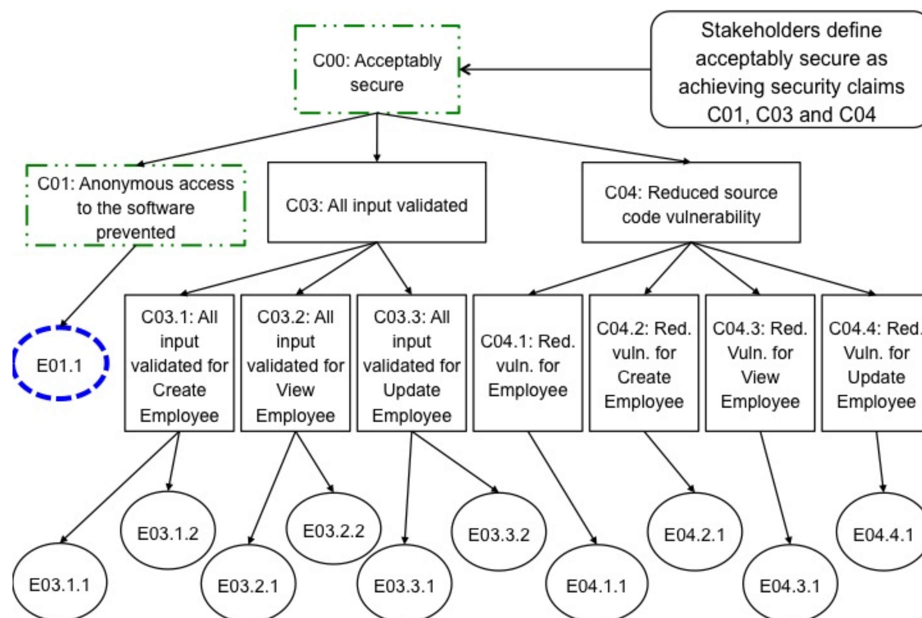


**Fig. 9.** Invalidated claims upon the transition from Iteration 1 to Iteration 2. (Table 6 describes the evidence codes.)

This iteration results in adding the `Login` and `Authentication` components to the increment and updating the configuration files of the application to require the user to be authenticated.

Figure 8 shows the invalidated evidences upon the transition from Iteration 1 to Iteration 2; the blue dashed circle is an invalidated evidence as a result of this software increment. Two security claims of Iteration 1 are invalidated as a result of invalidating the evidence. Figure 9 shows the changes to the claims using green dashed rectangles. (We observe, from this exercise, that claims become invalid as a result of invalidating supporting evidences or sub-claims.)

Figure 10 shows the updated security assurance case for Iteration 2. It includes new evidences, shown in dashed circles, and the new sub-claims shown in dashed rectangles. Claim $C01$ is now supported by the evidence from successful tests with the authentication mechanism implemented in addition to a review of the deployment architecture. A sub-claim is added to claim $C03$ to ensure the validation of input to the Login Web form. Sub-claims are also added to claim $C04$ to ensure reduced source code vulnerability for the newly added Login and Authentication components.

Note that the security claims for this iteration enforce user authentication to gain access to the system, but do not ensure proper access control for the system resources, i.e. do not distinguish between HR officers and regular employees for authorizing specific actions.

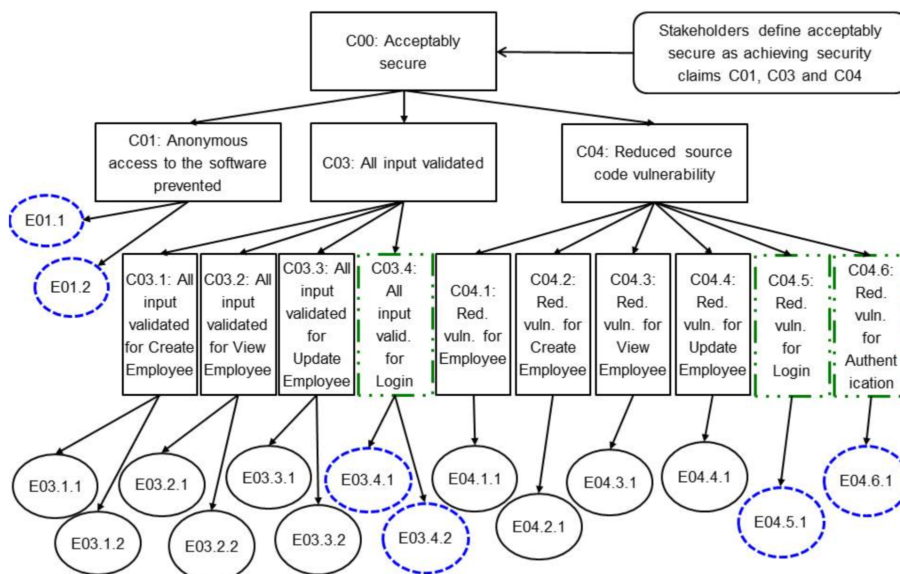


**Fig. 10.** Security assurance case for Iteration 2. (Table 6 describes the evidence codes.)

**Iteration 3.** This iteration involves implementing user story $U3$. The resulting increment could be used by HR officers to create/view/update employee records and manage holidays, and by employees to view/update their records. In this iteration, the top level security claim is composed of the three sub-claims: $C02$, $C03$, and $C04$. (Claim $C01$ becomes a sub-claim of claim $C02$.) Claim $C02$ (Prevent unauthorized access and modification of personal information) is satisfied by implementing an access control mechanism, which we formulate as security user story $U5$.

This iteration results in adding the `ManageHoliday`, `Holidays`, `ManageResources`, `Resource`, `AssignAccessControl`, and `AccessControl` components to the increment and updating the `CreateEmployee`, `ViewEmployee`, `UpdateEmployee` and `Login` Web forms to enforce the access control policies. Figure 11 shows the invalidated evidences as a result of this increment in blue dashed circles, and Figure 12 shows the invalidated claims in green dashed rectangles.

The security assurance case for this iteration extends that of the previous iteration by adding sub-claims under $C03$ for the `ManageHoliday`, `ManageResources` and `AssignAccessControl` components; gathering new evidences for the claims $C03.1$, $C03.2$, $C03.3$, $C03.4$; adding sub-claims under $C04$ for the `ManageHoliday`, `Holidays`, `ManageResources`, `Resource`, `AssignAccessControl`, and `AccessControl` components; and gathering new evidences for the claims $C04.2$, $C04.3$, $C04.4$, $C04.5$. The newly added claim $C02$ is supported by the already existing claim $C01$ and evidences gathered from successful test scenarios for the access control mechanism.

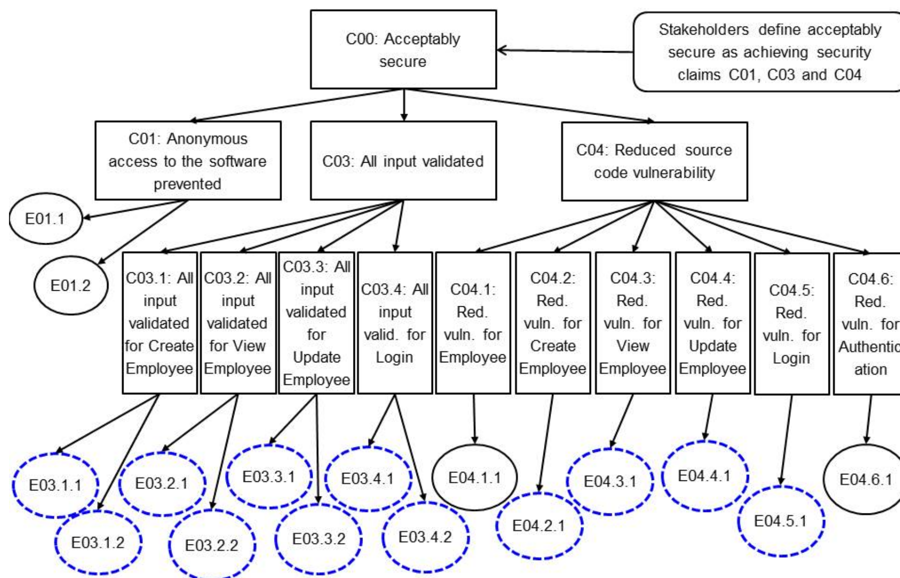


**Fig. 11.** Invalidated evidences upon the transition from Iteration 2 to Iteration 3. (Table 6 describes the evidence codes.)

Note that although Figure 12 shows that most of the claims (10 out of 14 claims) require a new assessment, the evaluation task is limited because it concerns only the changes related to the iteration. We believe that a more granular management of security reassurance may help to localize better the effects of the code modification and minimize the assessment effort.

## 5 Extending the agile software development process with security engineering activities

The approach that we use to extend the agile development process is based on: (1) redesigning the use of the security engineering activities (risk assessment, engineering, and security assurance) to accommodate the iterative nature of the agile software development process (see Subsection 2.1); (2) using our method of security reassurance to reassure the security of software increments; and (3) using a risk-based approach for selecting security threats that the software should mitigate in each increment.

We extend the agile development process by integrating the security subprocesses: risk assessment, engineering, and security assurance (see Subsection 2.2) to the agile development process, and ensuring production of acceptably secure software at each development iteration. Recall that *acceptably secure* software is a software increment that demonstrates a set of security goals selected by the business owner for the iteration.

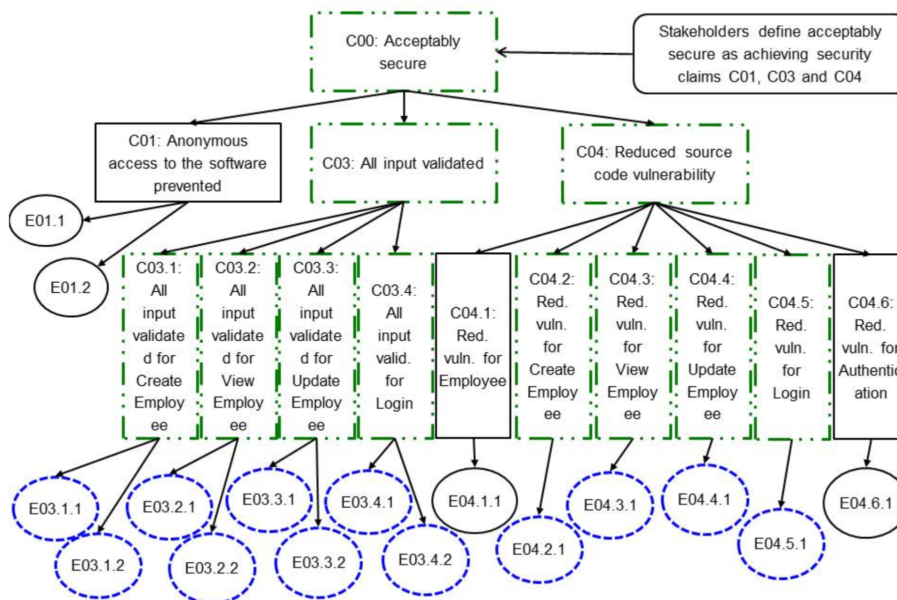


**Fig. 12.** Invalidated claims upon the transition from Iteration 2 to Iteration 3. (Table 6 describes the evidence codes.)

Figure 13 depicts the new process, called *agile secure software development process*. The description of the phases of the process follows.

### 5.1 Extending the inception phase.

We extend the inception phase with three activities: threat modeling, risk estimation, and security goals identification, which are performed after scoping the project, sketching the architecture, and identifying the main user stories. The focus in this phase is on assessing the security risks to the software and not on finding solutions that we may never use [43].

The threat modeling should be performed in a workshop with participation of the developers, project owner, and a security expert. The team identifies the threats relevant to the software, e.g., using misuse cases [44], and possible violations of key security protection mechanisms, e.g., authorization and data validation. Next, the developers provide their perception of the likelihood of occurrence of the threats (the “chance” that an attacker exploits a weakness or vulnerability and attacks the software) and the business owner provides his/her perception of the severity of the impact of the threats–level of the damage of a threat when successfully triggered. The likelihood and severity estimates are combined for each threat to obtain its risk level [45].

The threats to the software are classified into security goals. The grouping minimizes changes due to addressing other known security threats. The security

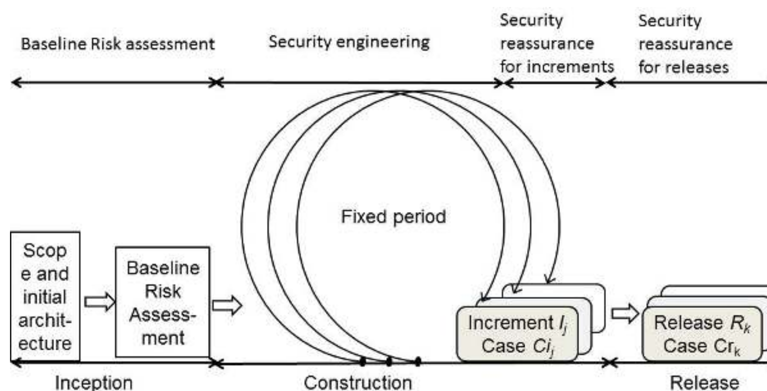


**Fig. 13.** Process for agile development of secure software.

goals take the form of claims for the assurance cases and are added as security user stories to the project backlog.

### 5.2 Extending the construction phase

We extend the construction phase by adding a set of activities to enable producing "acceptably secure" software increments. At the beginning of each iteration, the business owner defines "acceptably secure" software for the iteration; that is, the security claims that the increment should satisfy. (The business owner takes responsibility for choosing the threats to mitigate and accepting the impacts of the remaining threats in each iteration or release.)

There are two types of user stories that the developers and business owner could select from: functional user stories and security user stories. We discuss in the following how to implement both.

**Developing functional user stories.** In addition to the regular software activities required in the development of a user story (see Subsection 2.1), the development team performs a risk assessment for the selected user stories and develops security user stories to mitigate the threats they identify. The team members need also to apply secure coding and data validation techniques [46] to avoid source code vulnerabilities. For example, they need to carefully use memory allocation and exception handling.

**Developing security user stories.** The steps are:

1. Elicit the security requirements–use a threat-based security requirements eliciting process, such as Sindre and Opdahl method [47], to derive the security requirements related to the chosen security goal.
2. Develop security test scenarios–develop attack scenarios to test if the software satisfies its security requirements. For example, they design experiments for bypassing access data policy through performing a SQL injection attack.

3. Design a security feature for the user story–design a security feature that implements the security requirements and that could be integrated to the software architecture. The design transforms the security requirements expressed as constraints to a software behavior.
4. Develop the security feature–split the security feature into a set of user stories and implement them.
5. Test the security feature–implement and execute the security test scenarios to evaluate the compliance of the increment with the security requirements. The team concludes that the software implements the security user story if the results of the tests are positive.

**Security assurance.** We review, at the end of each iteration, the effects of changing the context of "acceptably secure" (from the previous iteration) on the security assurance case of the software. Then, we apply the security reassurance of increments that we described in Subsection 4.3.

We note that we record the evidences collected from the test scenarios of a security feature in the assurance case only when the security feature is fully developed; the assurance case notation does not provide a simple way to record partial satisfaction of a claim without ambiguity–in our opinion.

### 5.3 Extending the transition phase

We extend the transition phase with external review of the assurance case and perform the automated security tests and analysis on all the software. The extra tests aim to identify inconsistencies and increase the confidence in the security of the software. The team could also perform security assurance tasks that require extensive time and cost, only once, at this phase.

## 6 Discussion

This section discusses how the proposed method preserves agile development values and produces secure software. It also lists the limitations of the method.

### 6.1 Preserving agile values

This subsection discusses how the proposed method preserves the agile values [8].

**Individuals and interactions.** The proposed method favors individuals and interactions over processes and tools because several of the security engineering activities are performed in collaboration between the developers and the business owner. For instance, the threat modeling and risk estimation activities are performed in workshops that include the developers and the business owner.

**Working software.** The proposed method favors implementing security mechanisms and uses a security assurance approach (i.e., security assurance cases) that does not require extensive documentation, but rather connects the claims

to evidences that justify them. We consider the evidences as light documents that record the results of the security assessment activities.

**Customer collaboration.** The proposed method considers the customer perspective of secure software instead of the attacker perspective. For instance, the business owner, who represents the customers, collaborates with the developers in identifying the security threats and estimating the risks to the software. He/She is also responsible for selecting the priority of achieving the security claims.

**Responding to change.** The proposed method accommodates changes through (a) identifying emerging threats related to user stories selected for implementation in a new increment and (b) reassuring the security of the new increment.

### 6.2 Producing secure software

The proposed method produces secure software from a risk management perspective. For instance, it integrates the security engineering sub-processes: risk assessment, engineering, and security assurance. The sub-processes ensure identification of the threats to the software, engineering of security mechanisms for the main threats as perceived by the business owner, and ensuring the implemented security mechanisms mitigate the threats. The use of security assurance cases builds confidence into the security of the software; it provides the arguments justifying the evidences supporting the claims.

### 6.3 Limitations of the method

The proposed method has three main limitations. Their descriptions follow.

**Limited scalability.** The maintenance of the security assurance cases relies on tracing the impacts of developing new user stories on the components of the software, the evidences, arguments, and claims. The difficulty of tracing the impacts grows very fast as the number of components of the software, the number of claims, evidences, and arguments grow, which limits the use of the method.

We propose to address the issue through grouping the user stories and grouping the components of the software. This requires associating the claims, evidences, and arguments to groups of user stories and groups of components. The loss of granularity in specifying the relationships between the assurance artifacts and the software artifacts limits the efficiency of the method because it increases the number of claims, arguments, and evidences that could be affected by a change to the components.[12]

**Extra cost.** The periodic security reassurance of software[13] and the need to reassess a set of claims increases the cost of security assurance. The cost is far

[12] A change to a component, member of a group, requires assessing the claims and evidences associated to all the components of the group instead of only the claims and evidences associated to the component itself.

[13] We perform a security reassurance at the end of each development iteration.

lower than reevaluating all the claims at the end of each iteration as it would be required if we used check lists of security requirements.

The rework cost can be justified for the case of software that evolve based on the needs of the customers–mainly produced by commercial companies. However, it may not be justified if the security goals are all known in advance and the software can be only released for use if all the goals are achieved. This occurs, for example, for the case of safety-critical systems.

**Requirement of high independence between the software components.** The method is based on the assumption of *high* independence between the components of the software. The assumption limits the security reassurance of a new increment to the reevaluation of a set of claims associated with the components that have changed and the few claims that are associated with all the components of the software.

The assumption is valid for software that have components-based [48] architecture or Service Oriented Architecture (SOA) [49] because they reduce the number of connections between the components and are highly testable[14] [50].

This assumption is strong for the general case. For instance, Ren et al. [51] analyzed the impact of code changes on the tests of the functionalities of software package, Daikon, which evolved over the period of a year. They found that, on average, 52% of the tests are affected by the changes made in a week. The statistics show the limitation of the efficiency of our approach for software that have high dependency between the components. This requires more research to analyze the impacts of software changes on the different kinds of security claims and the efforts required to reevaluate invalidated claims.

## 7 Conclusion

This paper concludes that the agile software development approach does not prevent ensuring the security of software increments produced at the end of each development iteration. It proposes a method for security assurance of software increments and integrates security engineering activities into the agile software development process. The method enables ensuring the delivery of secure software at the end of each iteration.

The method could be used by companies to iteratively and incrementally improve the security of the software they produce by delivering "acceptably secure" software that partially mitigates their associated threats–while ensuring the validity of the security claims.

The main advantages of the approach are: (1) helping reduce the cost of reassurance of software security, and (2) helping reduce the cost of mitigating threats. The reason for the first advantage is: the development team could reuse parts of the security assurance cases in the assessment of the new increments. The reason for the second advantage is: the approach ensures that security goals achieved in the early iterations of the development are preserved in the subsequent software increments.

[14] easier to write tests that ensure the required properties

The main limitations of the current method are: (1) it applies to modular software, where security claims are associated with specific components; (2) it is not scalable enough; and (3) it does not consider security design principles– such as the "fail safe" principal. Also, the current work *simulates* the method on (fabricated) case studies. We plan in the future work to practice with the method on real projects and address the three mentioned limitations.

## References


1. McGraw, G.: Software Security: Building Security In. Addison-Wesley Software Security Series. Addison-Wesley (2006)
2. CNSS Glossary Working Group: National information assurance (IA) glossary. `http://jitc.fhu.disa.mil/pki/documents/committee_on_national_security_systems_instructions_4009_june_2006.pdf` (June 2006) CNSS Instruction No. 4009.
3. Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grnbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., Mder, P.: Traceability Fundamentals. In: Software and systems traceability. Springer-Verlag London Limited (2012) 3–22
4. Grossklags, J., Johnson, B.: Uncertainty in the weakest-link security game. In: Proc. of the First International Conference on Game Theory for Networks. GameNets'09, Istanbul, Turkey (May 2009) 673–682
5. Lipner, S., Howard, M.: The trustworthy computing security development lifecycle. In: Proc. of the 20th Annual Computer Security Applications Conference. ACSAC '04, Tucson, AZ (March 2005) 2–13 Revised version.
6. Howard, M., Lipner, S.: The security development lifecycle: SDL, a process for developing demonstrably more secure software. Microsoft Press Series. Microsoft Press (2006)
7. Larman, C., Basili, V.R.: Iterative and incremental development: A brief history. Computer **36**(6) (June 2003) 47–56
8. Agile Alliance: Agile alliance. `http://www.agilealliance.org/` (Sep. 2012)
9. Beznosov, K., Kruchten, P.: Towards agile security assurance. In: Proc. of the 2004 Workshop on New Security Paradigms. NSPW '04, Nova Scotia, Canada (Sep. 2004) 47–54
10. Wayrynen, J., Boden, M., Bostrom, G.: Security engineering and extreme programming: an impossible marriage? In: Lecture Notes in Computer Science. Volume 3134., Calgary, Canada, Springer (Aug, 2004) 117–128 4th Conference on Extreme Programming and Agile Methods.
11. Goertzel, K.M., Winograd, T.: Enhancing the development life cycle to produce secure software. Technical Report DAN 358844, Defense Technical Information Center (DTIC) (2008)
12. Goertzel, K.M., Winograd, T., McKinley, H.L., Holley, P., Hamilton, B.A.: Security in the software lifecycle. `http://www.cert.org/books/secureswe/SecuritySL.pdf` (August 2006) Draft version 1.2.
13. OWASP: Agile software development: Don't forget evil user stories. `https://www.owasp.org/index.php/Agile_Software_Development:_Don%27t_Forget_EVIL_User_Stories` (August 2011)
14. Sullivan, B.: Agile security; or, how to defend applications with five-day-long release cycles. Black Hat, DC, (Jan. 2010)

15. Vähä-Sipilä, A.: Product security risk management in agile product management. https://www.owasp.org/images/c/c6/OWASP_AppSec_Research_2010_Agile_Prod_Sec_Mgmt_by_Vaha-Sipila.pdf (June 2010) accessed in May 2013.
16. Boberski, M., Williams, J., Wichers, D.: Owasp application security verification standard 2009. https://www.owasp.org/images/4/4e/OWASP_ASVS_2009_Web_App_Std_Release.pdf (June 2009)
17. Vivas, J., Agudo, I., Lpez, J.: A methodology for security assurance-driven system development. Requirements Engineering **16**(1) (2011) 55–73
18. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)
19. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. 1 st edition edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2006)
20. Ambler, S.W.: The agile scaling model (ASM): adapting agile methods for complex environments. ftp://ftp.software.ibm.com/common/ssi/sa/wh/n/raw14204usen/RAW14204USEN.PDF (December 2009)
21. Palmer, S., Felsing, J.: A practical guide to feature-driven development. 1 edition edn. The Coad series. Prentice Hall PTR (Feb. 2002)
22. Jazayeri, M.: Some trends in web application development. In: Proc. Future of Software Engineering. FOSE '07, Washington, DC, USA (May 2007) 199–213
23. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Greedy Algorithms. In: Introduction to algorithms. 2nd edn. MIT press (2001) 414–450
24. Ambler, S.W.: The agile system development lifecycle (SDLC). http://www.ambysoft.com/essays/agileLifecycle.html (November 2006) accessed in May 2013.
25. International Organization for Standardization and International Electrotechnical Commission: Information technology – systems security engineering – capability maturity model (SSE-CMM) (2008)
26. Meier, J., Mackman, A., Wastell, B., Bansode, P., Taylor, J., Araujo, R.: Security engineering explained. http://www.microsoft.com/en-us/download/details.aspx?id=20528 (October 2005) accessed in May 2013.
27. Shirey, R.: Internet security glossary, version 2. http://www.ietf.org/rfc/rfc4949.txt (aug 2007) RFC 4949 (Informational).
28. Kissel, R.: Glossary of key information security terms. http://csrc.nist.gov/publications/nistir/ir7298-rev1/nistir-7298-revision1.pdf (February 2011)
29. Jackson, D., Cooper, D.: Where do software security assurance tools add value? In: Workshop on Software Security Assurance Tools, Techniques, and Metrics. SSATTM05, Long Beach, CA, National Institute of Standards and Technology (NIST) (November 2005) 14–21 NIST Special Publication 500-265.
30. Bellovin, S.M.: Security as a systems property. Security & Privacy, IEEE **7**(5) (sept.-oct. 2009) 88
31. Software Engineering Institute - Carnegie Mellon University: Cert secure coding standards. https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards accessed on January 2013.
32. Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Proc. DARPA Information Survivability Conference and Exposition. Volume 2 of DISCEX '00., Hilton Head, SC (Jan. 2000) 119 –129
33. Kelly, T., Weaver, R.: The goal structuring notation–a safety argument notation. In: Proc. Dependable Systems and Networks–Workshop on Assurance Cases, Florence, Italy (July 2004)

34. McDermid, J.A.: Support for safety cases and safety arguments using SAM. Reliability Engineering & System Safety **43**(2) (1994) 111 – 127 Special issue on software safety.
35. Goodenough, J., Lipson, H., Weinstock, C.: Arguing security - creating security assurance cases. `https://buildsecurityin.us-cert.gov/bsi/articles/knowledge/assurance/643-BSI.html` (June 2012) accessed in May 2013.
36. Bloomfield, R.E., Guerra, S., Masera, M., Miller, A., Saydjari, O.S.: Assurance cases for security. In: Workshop on Assurance Cases for Security, Arlington, VA (June 2005)
37. Microsoft: Security development lifecycle for agile development. `http://www.blackhat.com/presentations/bh-dc-10/Sullivan_Bryan/BlackHat-DC-2010-Sullivan-SDL-Agile-wp.pdf` (June 2009) accessed in May 2013.
38. Microsoft: Agile development using microsoft security development lifecycle. `http://www.microsoft.com/security/sdl/discover/sdlagile.aspx` (October 2012) accessed in May 2013.
39. Vähä-Sipilä, A.: Software security in agile product management. `http://www.fokkusu.fi/agile-security/Software%20security%20in%20agile%20product%20management.pdf` (2011) accessed in May 2013.
40. Cohn, M.: User Stories Applied: For Agile Software Development. pearson Education, Inc., Boston, MA (2004)
41. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
42. The Apache Software Foundation: Apache http server project. `http://httpd.apache.org/` accessed in May 2013.
43. Meier, J., Mackman, A., Wastell, B.: Walkthrough: Creating a threat model for a web application. Technical report, Microsoft Corporation (May 2005)
44. Alexander, I.: Misuse cases: Use cases with hostile intent. IEEE Software **20**(1) (2003) 58–66
45. Stoneburner, G., Goguen, A., Feringa, A.: Risk management guide for information technology systems – recommendations of the national institute of standards and technology. `http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf` (2002) Special Publication 800-30, accessed in May 2013.
46. Software Assurance Forum: Secure coding. `ftp://ftp.sei.cmu.edu/pub/pruggiero/bsi-swa/1/SecureCoding_PocketGuide_v2%200_05182012_PostOnline.pdf` (May 2012) accessed in May 2013.
47. Sindre, G., Opdahl, A.L.: Eliciting security requirements with misuse cases. Requirement Engineering **10**(1) (January 2005) 34–44
48. Garlan, D., Shaw, M.: An introduction to software architecture. In: Advances in Software Engineering and Knowledge Engineering. Volume 2. World Scientific Publishing Company, New Jersey, NJ (1993) 1–39
49. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
50. Chivers, H., Paige, R.F., Ge, X.: Agile security using an incremental security architecture. In: Proc. of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering. XP'05, Sheffield, UK (June 2005) 57–65
51. Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O.: Chianti: a tool for change impact analysis of java programs. In: Proc. of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '04, Vancouver, BC, Canada (October 2004) 432–448

**Science Reports** **Department of Mathematics and Computer Science**
**Technische Universiteit Eindhoven**

If you want to receive reports, send an email to: wsinsan@tue.nl (we cannot guarantee the availability of the requested reports).

## *In this series appeared (from 2009):*

| | | |
|---|---|---|
| 09/01 | Wil M.P. van der Aalst, Kees M. van Hee, Peter Massuthe, Natalia Sidorova and Jan Martijn van der Werf | Compositional Service Trees |
| 09/02 | P.J.l. Cuijpers, F.A.J. Koenders, M.G.P. Pustjens, B.A.G. Senders, P.J.A. van Tilburg, P. Verduin | Queue merge: a Binary Operator for Modeling Queueing Behavior |
| 09/03 | Maarten G. Meulen, Frank P.M. Stappers and Tim A.C. Willemse | Breadth-Bounded Model Checking |
| 09/04 | Muhammad Atif and MohammadReza Mousavi | Formal Specification and Analysis of Accelerated Heartbeat Protocols |
| 09/05 | Michael Franssen | Placeholder Calculus for First-Order logic |
| 09/06 | Daniel Trivellato, Fred Spiessens, Nicola Zannone and Sandro Etalle | POLIPO: Policies & OntoLogies for the Interoperability, Portability, and autOnomy |
| 09/07 | Marco Zapletal, Wil M.P. van der Aalst, Nick Russell, Philipp Liegl and Hannes Werthner | Pattern-based Analysis of Windows Workflow |
| 09/08 | Mike Holenderski, Reinder J. Bril and Johan J. Lukkien | Swift mode changes in memory constrained real-time systems |
| 09/09 | Dragan Bošnački, Aad Mathijssen and Yaroslav S. Usenko | Behavioural analysis of an I²C Linux Driver |
| 09/10 | Ugur Keskin | In-Vehicle Communication Networks: A Literature Survey |
| 09/11 | Bas Ploeger | Analysis of ACS using mCRL2 |
| 09/12 | Wolfgang Boehmer, Christoph Brandt and Jan Friso Groote | Evaluation of a Business Continuity Plan using Process Algebra and Modal Logic |
| 09/13 | Luca Aceto, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers | A Rule Format for Unit Elements |
| 09/14 | Maja Pešić, Dragan Bošnački and Wil M.P. van der Aalst | Enacting Declarative Languages using LTL: Avoiding Errors and Improving Performance |
| 09/15 | MohammadReza Mousavi and Emil Sekerinski, Editors | Proceedings of Formal Methods 2009 Doctoral Symposium |
| 09/16 | Muhammad Atif | Formal Analysis of Consensus Protocols in Asynchronous Distributed Systems |
| 09/17 | Jeroen Keiren and Tim A.C. Willemse | Bisimulation Minimisations for Boolean Equation Systems |
| 09/18 | Kees van Hee, Jan Hidders, Geert-Jan Houben, Jan Paredaens, Philippe Thiran | On-the-fly Auditing of Business Processes |
| 10/01 | Ammar Osaiweran, Marcel Boosten, MohammadReza Mousavi | Analytical Software Design: Introduction and Industrial Experience Report |
| 10/02 | F.E.J. Kruseman Aretz | Design and correctness proof of an emulation of the floating-point operations of the Electrologica X8. A case study |

| | | |
|---|---|---|
| 10/03 | Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, MohammadReza Mousavi and Michel A. Reniers | On Rule Formats for Zero and Unit Elements |
| 10/04 | Hamid Reza Asaadi, Ramtin Khosravi, MohammadReza Mousavi, Neda Noroozi | Towards Model-Based Testing of Electronic Funds Transfer Systems |
| 10/05 | Reinder J. Bril, Uğur Keskin, Moris Behnam, Thomas Nolte | Schedulability analysis of synchronization protocols based on overrun without payback for hierarchical scheduling frameworks revisited |
| 10/06 | Zvezdan Protić | Locally unique labeling of model elements for state-based model differences |
| 10/07 | C.G.U. Okwudire and R.J. Bril | Converting existing analysis to the EDP resource model |
| 10/08 | Muhammed Atif, Sjoerd Cranen, MohammadReza Mousavi | Reconstruction and verification of group membership protocols |
| 10/09 | Sjoerd Cranen, Jan Friso Groote, Michel Reniers | A linear translation from LTL to the first-order modal μ-calculus |
| 10/10 | Mike Holenderski, Wim Cools Reinder J. Bril, Johan J. Lukkien | Extending an Open-source Real-time Operating System with Hierarchical Scheduling |
| 10/11 | Eric van Wyk and Steffen Zschaler | 1st Doctoral Symposium of the International Conference on Software Language Engineering (SLE) |
| 10/12 | Pre-Proceedings | 3rd International Software Language Engineering Conference |
| 10/13 | Faisal Kamiran, Toon Calders and Mykola Pechenizkiy | Discrimination Aware Decision Tree Learning |
| 10/14 | J.F. Groote, T.W.D.M. Kouters and A.A.H. Osaiweran | Specification Guidelines to avoid the State Space Explosion Problem |
| 10/15 | Daniel Trivellato, Nicola Zannone and Sandro Etalle | GEM: a Distributed Goal Evaluation Algorithm for Trust Management |
| 10/16 | L. Aceto, M. Cimini, A.Ingolfsdottir, M.R. Mousavi and M. A. Reniers | Rule Formats for Distributivity |
| 10/17 | L. Aceto, A. Birgisson, A. Ingolfsdottir, and M.R. Mousavi | Decompositional Reasoning about the History of Parallel Processes |
| 10/18 | P.D. Mosses, M.R. Mousavi and M.A. Reniers | Robustness os Behavioral Equivalence on Open Terms |
| 10/19 | Harsh Beohar and Pieter Cuijpers | Desynchronisability of (partial) closed loop systems |
| 11/01 | Kees M. van Hee, Natalia Sidorova and Jan Martijn van der Werf | Refinement of Synchronizable Places with Multi-workflow Nets - Weak termination preserved! |
| 11/02 | M.F. van Amstel, M.G.J. van den Brand and L.J.P. Engelen | Using a DSL and Fine-grained Model Transformations to Explore the boundaries of Model Verification |
| 11/03 | H.R. Mahrooghi and M.R. Mousavi | Reconciling Operational and Epistemic Approaches to the Formal Analysis of Crypto-Based Security Protocols |
| 11/04 | J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius | Benefits of Applying Formal Methods to Industrial Control Software |
| 11/05 | Jan Friso Groote and Jan Lanik | Semantics, bisimulation and congruence results for a general stochastic process operator |
| 11/06 | P.J.L. Cuijpers | Moore-Smith theory for Uniform Spaces through Asymptotic Equivalence |
| 11/07 | F.P.M. Stappers, M.A. Reniers and S. Weber | Transforming SOS Specifications to Linear Processes |
| 11/08 | Debjyoti Bera, Kees M. van Hee, Michiel van Osch and Jan Martijn van der Werf | A Component Framework where Port Compatibility Implies Weak Termination |
| 11/09 | Tseesuren Batsuuri, Reinder J. Bril and Johan Lukkien | Model, analysis, and improvements for inter-vehicle communication using one-hop periodic broadcasting based on the 802.11p protocol |

| | | |
|---|---|---|
| 11/10 | Neda Noroozi, Ramtin Khosravi, MohammadReza Mousavi and Tim A.C. Willemse | Synchronizing Asynchronous Conformance Testing |
| 11/11 | Jeroen J.A. Keiren and Michel A. Reniers | Type checking mCRL2 |
| 11/12 | Muhammad Atif, MohammadReza Mousavi and Ammar Osaiweran | Formal Verification of Unreliable Failure Detectors in Partially Synchronous Systems |
| 11/13 | J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius | Experience report on developing the Front-end Client unit under the control of formal methods |
| 11/14 | J.F. Groote, A.A.H. Osaiweran and J.H. Wesselius | Ananlyzing a Controller of a Power Distribution Unit Using Formal Methods |
| 11/15 | John Businge, Alexander Serebrenik and Mark van den Brand | Eclipse API Usage: The Good and The Bad |
| 11/16 | J.F. Groote, A.A.H. Osaiweran, M.T.W. Schuts and J.H. Wesselius | Investigating the Effects of Designing Control Software using Push and Poll Strategies |
| 11/17 | M.F. van Amstel, A. Serebrenik And M.G.J. van den Brand | Visualizing Traceability in Model Transformation Compositions |
| 11/18 | F.P.M. Stappers, M.A. Reniers, J.F. Groote and S. Weber | Dogfooding the Structural Operational Semantics of mCRL2 |
| 12/01 | S. Cranen | Model checking the FlexRay startup phase |
| 12/02 | U. Khadim and P.J.L. Cuijpers | Appendix C / G of the paper: Repairing Time-Determinism in the Process Algebra for Hybrid Systems ACP |
| 12/03 | M.M.H.P. van den Heuvel, P.J.L. Cuijpers, J.J. Lukkien and N.W. Fisher | Revised budget allocations for fixed-priority-scheduled periodic resources |
| 12/04 | Ammar Osaiweran, Tom Fransen, Jan Friso Groote and Bart van Rijnsoever | Experience Report on Designing and Developing Control Components using Formal Methods |
| 12/05 | Sjoerd Cranen, Jeroen J.A. Keiren and Tim A.C. Willemse | A cure for stuttering parity games |
| 12/06 | A.P. van der Meer | CIF MSOS type system |
| 12/07 | Dirk Fahland and Robert Prüfer | Data and Abstraction for Scenario-Based Modeling with Petri Nets |
| 12/08 | Luc Engelen and Anton Wijs | Checking Property Preservation of Refining Transformations for Model-Driven Development |
| 12/09 | M.M.H.P. van den Heuvel, M. Behnam, R.J. Bril, J.J. Lukkien and T. Nolte | Opaque analysis for resource-sharing components in hierarchical real-time systems - extended version – |
| 12/10 | Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien | Efficient reprogramming of sensor networks using incremental updates and data compression |
| 12/11 | John Businge, Alexander Serebrenik and Mark van den Brand | Survival of Eclipse Third-party Plug-ins |
| 12/12 | Jeroen J.A. Keiren and Martijn D. Klabbers | Modelling and verifying IEEE Std 11073-20601 session setup using mCRL2 |
| 12/13 | Ammar Osaiweran, Jan Friso Groote, Mathijs Schuts, Jozef Hooman and Bart van Rijnsoever | Evaluating the Effect of Formal Techniques in Industry |
| 12/14 | Ammar Osaiweran, Mathijs Schuts, and Jozef Hooman | Incorporating Formal Techniques into Industrial Practice |
| 13/01 | S. Cranen, M.W. Gazda, J.W. Wesselink and T.A.C. Willemse | Abstraction in Parameterised Boolean Equation Systems |
| 13/02 | Neda Noroozi, Mohammad Reza Mousavi and Tim A.C. Willemse | Decomposability in Formal Conformance Testing |

| | | |
|---|---|---|
| 13/03 | D. Bera, K.M. van Hee and N. Sidorova | Discrete Timed Petri nets |
| 13/04 | A. Kota Gopalakrishna, T. Ozcelebi, A. Liotta and J.J. Lukkien | Relevance as a Metric for Evaluating Machine Learning Algorithms |
| 13/05 | T. Ozcelebi, A. Weffers-Albu and J.J. Lukkien | Proceedings of the 2012 Workshop on Ambient Intelligence Infrastructures (WAmIi) |
| 13/06 | Lotfi ben Othmane, Pelin Angin, Harold Weffers and Bharat Bhargava | Extending the Agile Development Process to Develop Acceptably Secure Software |