Chapter 9
Maintaining Security in Software Evolution

Jan Jürjens, Kurt Schneider, Jens Bürger, Fabien Patrick Viertel, Daniel Strüber, Michael Goedicke, Ralf Reussner, Robert Heinrich, Emre Taşpolatoğlu, Marco Konersmann, Alexander Fay, Winfried Lamersdorf, Jan Ladiges, and Christopher Haubeck

J. Jürjens (✉) · J. Bürger · D. Strüber
Institute for Computer Science, University of Koblenz-Landau, Koblenz, Germany
e-mail: juerjens@uni-koblenz.de; buerger@uni-koblenz.de; strueber@uni-koblenz.de

K. Schneider · F. P. Viertel
Institute of Software Engineering, Leibniz Universität Hannover, Hannover, Germany
e-mail: kurt.schneider@inf.uni-hannover.de; fabien.viertel@inf.uni-hannover.de

M. Goedicke
paluno – The Ruhr Institute for Software Technology, Specification of Software Systems, Universität Duisburg-Essen, Essen, Germany
e-mail: michael.goedicke@s3.uni-due.de

R. Reussner · R. Heinrich
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: reussner@kit.edu; robert.heinrich@kit.edu

E. Taşpolatoğlu
Department of Software Engineering, FZI Forschungszentrum Informatik, Karlsruhe, Germany
e-mail: taspolat@fzi.de

M. Konersmann
Institute for Software Technology, Research Group Software Engineering, Universität Koblenz-Landau, Koblenz, Germany
e-mail: konersmann@uni-koblenz.de

A. Fay · J. Ladiges
Institute of Automation Technology, Helmut Schmidt University, Hamburg, Germany
e-mail: alexander.fay@hsu-hh.de; jan.ladiges@hsu-hh.de

W. Lamersdorf · C. Haubeck
MIN-Faculty, Department of Informatics, Distributed Systems, Universität Hamburg, Hamburg, Germany
e-mail: lamersd@informatik.uni-hamburg.de; haubeck@informatik.uni-hamburg.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_9

The engineering of security-critical software systems faces special challenges regarding evolution. Even if a substantial effort went into ensuring security during the system's initial development, it is uncertain if the system remains secure when changes to the software, the execution platform, or the system environment occur. Relevant changes that might endanger security include new or evolving system requirements, changing laws, or updated knowledge regarding attacks and mitigations. Failure to keep up with such changes can lead to substantial breaches and losses, highlighting the need to actively maintain an established level of security [And08].

For preserving security in long-living systems, ongoing and systematic support for the evolution of knowledge and software is required. Reflecting the guiding theme *Methods and processes for evolution* of the priority program, there is a need for techniques, tools, and processes to support the evolution of systems in order to ensure *lifelong* compliance with security requirements. These techniques, tools, and processes need to address two main challenges, as outlined in the chapter "Challenges" of this book:

1. *How can security knowledge, available via diverse non-formal sources, be incorporated and utilised for long-living system design?* Establishing security depends on given *security knowledge*, which may only be available in a non- or semi-formal textual form. Whenever the security knowledge changes, earlier assumptions about the security of the system may no longer hold true; the system needs to be re-evaluated and adapted with regard to the security requirements.
2. *How can developers and security experts be supported to react to context evolution that may compromise the system's security design or compromise the system at run time?* New available security knowledge, as well as suspicious behaviour in the running system, may rely on a human developer for diagnosis and hardening. These human stakeholders can be assisted by providing them with appropriate information, for example about a relevant security pattern or behaviour violation.

To address these challenges, we present a suite of approaches that contribute to a three-layered framework (Fig. 9.1). On the bottom layer, developer, system, and environment activity is *monitored*. Usually, this activity is monitored in a non-invasive manner, for example by logging executed methods using a framework such as Kieker [VWH12] or in case of a production system by monitoring input and output signals [Hau+14a]. At this level, results from model-based security testing can be exploited (such as [JW01]). Additional aspects of human behaviour might be considered as well. On the middle layer, collected monitoring data are *analysed* in various ways: It is important to identify deviations of monitored behaviour from expected behaviour. Since there are different sources and types of expectations, the details of each analysis may differ. Assumptions associated with design patterns will be investigated in a different way than expectations held about user interaction. At this level, approaches such as model-based security analysis can

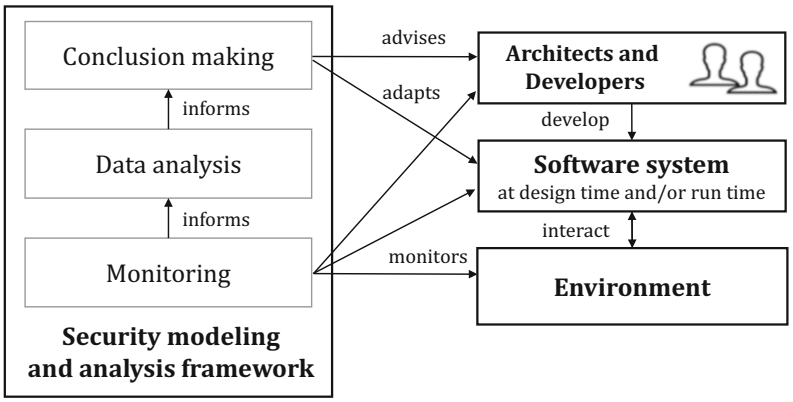

Fig. 9.1 Overview of a three-layered framework for maintaining security in software evolution at design time and run time

be used (such as [Jür01]). On the top layer, *conclusions* are derived. Warnings, hints, or technical adaptations are generated and released in order to preserve security, using approaches such as the SecReq approach to security requirement engineering [Sch+12].

In this chapter, we present a suite of five approaches that employ the above-mentioned framework. In combination, the approaches address all identified challenges for security maintenance at design time and run time. The first two approaches focus on the design time. The first approach (presented in Sect. 9.2) uses knowledge extracted from natural-language documents to identify potential steps for co-evolving the system design. The second one (Sect. 9.3) is on integrating architecture model information with program code. It creates a bidirectional mapping between model elements and code structures to automatically structure program code so that it contains model-based security properties and therefore survives code evolution. The third approach (Sect. 9.4) bridges design time and run time to support architects as the software evolves. It formally documents contextual information gathered from run time in architectural models at design time. Architects use model-based catalogues containing several security-related elements like attack types or security patterns, which are exploited as a lightweight metric for an architectural security analysis. The two remaining approaches focus on run-time security maintenance. The fourth one (Sect. 9.5) monitors run-time information in order to detect suspicious behaviour, which is reacted to automatically by adapting the system with mitigation measures. The fifth one (Sect. 9.6) focuses on interdisciplinary changes in automation software. It compares actual observed behaviour with intended behaviour expressed in signal-based models to find behaviour anomalies during run time.

Having presented these five approaches in detail, the chapter concludes with a discussion of how each approach contributes to addressing the challenges (Sect. 9.7).

9.1 Foundations

Modelling, Meta Modelling, and Model-Driven Software Development A model can be seen as an abstraction of a subject. An example for models can be mathematical formulas that describe the reality while ignoring factors that are irrelevant for the use case. Models of software are often represented as interconnected elements, for example structural models or behavioural models of the Unified Modeling Language (UML). Modelling is the activity to create models.

Meta models define a language for modelling. This means the elements where models are built from and how they can be connected. Thus, meta models define the abstract syntax of models that comply with the meta model. A model that complies with a meta model is called an *instance* of the meta model. Classically, the key concepts behind meta modelling are the relationship between a model element (often called *object* or *instance* in this context) and its meta-model element (*classifier* or *class*) and the ability to navigate from an object to its classifier. Multiple levels of instance-of relationships are possible, where the classifier of an object is itself the instance of a "higher level" classifier. Two *meta levels* mean that one level of objects and one level of classifiers exist. An arbitrary number of meta levels is possible, although typically two to four levels are used [Obj16, Section 7.3]. Instance-of relations in meta modelling build directed acyclic graphs, which build a hierarchy. In this context, we do not explore further the generalisations made by deep modelling [AG12].

Meta modelling is the activity to create meta models. This can follow a top-down or a bottom-up approach. Top-down meta modelling means to define a meta model for a subject to model and to create models afterwards. Bottom-up means to derive a meta model out of a modelled subject to classify the already modelled elements.

Model-driven software development (MDSD) [SVC06] uses models as central artefacts for software development activities. In MDSD, parts of the software are described using models that comply to domain-specific meta models [Mar10]. These domain models are refined with detailed technical models that are relevant not to the domain but to the platform that will run the software. Such models are the basis for automated code generation. The generated code has to be enriched with implementation details.

9.2 Design Time: Leveraging Knowledge from Natural Language for Design-Time System Adaptation

In this section, we present an approach to leverage knowledge from natural language for design-time system adaptation. This approach was developed in the SecVolution project within the priority program. To address the challenges overviewed in the

introduction, the key idea is to maintain a *knowledge base* that collects knowledge about security concepts and instantiations within the given software system. Using this knowledge base, we can react to vulnerabilities occurring during evolution, such as changes in requirements, knowledge, or other environmental aspects. The knowledge base contains information on how to deal with a triggering change in order to preserve security. A semi-automated mechanism uses the knowledge base to update the system models.

The SecVolution approach harnesses formal design artefacts available in the regular development process, such as UML-based system models. However, many of the monitored sources of change and evolution are informal. In particular, we need to deal with artefacts on the requirements side, which include natural-language documents like the system's requirement specification or laws. To this end, we developed socio-technical methods for supporting elicitation of relevant changes in the environment. Like in our previous works (confer [Sch06, Sch09, Pha+13, AKK14]), the relevant knowledge is captured during regular development tasks with as little extra burden for the security expert as possible. Steps for restoring security are integrated into existing tasks as well and aimed to be as unintrusive as possible. By avoiding additional tasks and assignments and by keeping extra-effort low, acceptance by developers and security experts is increased and chances rise for effectively applying the SecVolution approach.

The approach can be applied to existing (*long-living*) software systems for which this information can be provided.

9.2.1 Overview

The overall design-time approach developed in SecVolution is shown in Fig. 9.3. It uses the FLOW notation [SSK08] (summarised in Fig. 9.2) to visualise the information flow within, and to highlight important aspects of, the SecVolution approach. FLOW is used in Figs. 9.3 and 9.19. Relevant aspects of a FLOW model

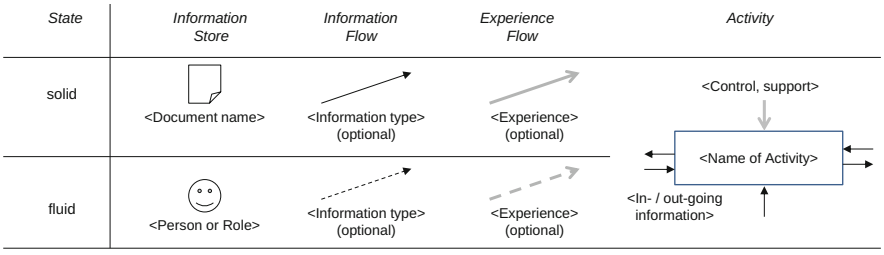

Fig. 9.2 FLOW notation symbols according to Schneider et al. [SSK08]

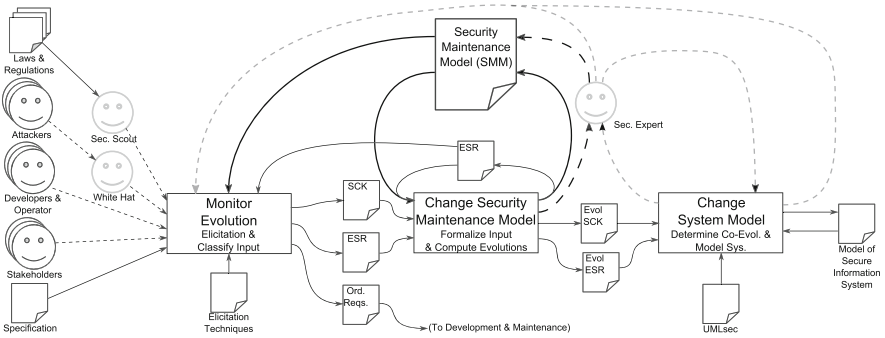

Fig. 9.3 Overview of the SecVolution design-time approach using the information flow syntax described in Fig. 9.2

include the fluid/solid state of information, the route of information, and the role of experience as a cross-cutting type of information:

- Document symbols and solid arrows represent documented project information or knowledge. That knowledge can be retrieved at any time without the need to involve the author.
- Faces and dashed arrows symbolise direct communication, for example in meetings, conversations, phone calls, or emails. This type of information representation is called *fluid* in FLOW [SSK08], as opposed to the *solid* information in documents and artefacts
- Rectangles represent activities with certain incoming and other outgoing information flows. As a black box, the internals of an activity are hidden. They may be detailed by another FLOW diagram.
- Project-specific information (in black) is attached to the left and right of an activity, whereas control and support enter from top and bottom of the activity rectangles.
- Grey colour indicates knowledge and experience. They are more generic than project-specific information and, thus, can be reused in other parts of the system.

In the following, we present the main process of SecVolution based on Fig. 9.3. The left part of Fig. 9.3 shows various sources of relevant knowledge. These are monitored for changes relevant to the approach. For deciding which information is relevant, Natural Language Processing methods are used to retrieve information from natural-language sources (Sect. 9.2.2). The monitored data is then split into three types:

- Ordinary requirements, which do not have any security relevance. These can be forwarded to the ordinary development process.
- New or evolved *Essential Security Requirements*. These are requirements that define basic security-relevant requirements for the system like "*Use secure encryption algorithms*".

- Information relevant for the security knowledge base, the *Security Context Knowledge*. This context knowledge is necessary to annotate the system model with concrete security requirements, for example a concrete encryption algorithm and appropriate key length.

The security knowledge base constituted by the *Security Context Knowledge (SCK)* requires a suitable representation, which we provide by using ontologies (Sect. 9.2.3). Security Context Knowledge and Essential Security Requirements both are managed within the *Security Maintenance Model*. Updating the Security Maintenance Model can make design decisions necessary, for example introduce a new cipher family because attacks have become known.

Security Maintenance Rules (SMR, Sect. 9.2.4) are the final part of the Security Maintenance Model. They decide if an evolution of the knowledge given by Security Context Knowledge and Essential Security Requirements makes co-evolution necessary. Figure 9.4 describes the overall idea and relationship between evolution and co-evolution. The development of a system model is accompanied by Security Context Knowledge. At its initial design, the system model was compliant with regard to the security knowledge that was current then. Over time, the Security Context Knowledge evolves. The system model now has to be co-evolved so that the then evolved system model is compliant with regard to the updated security knowledge.

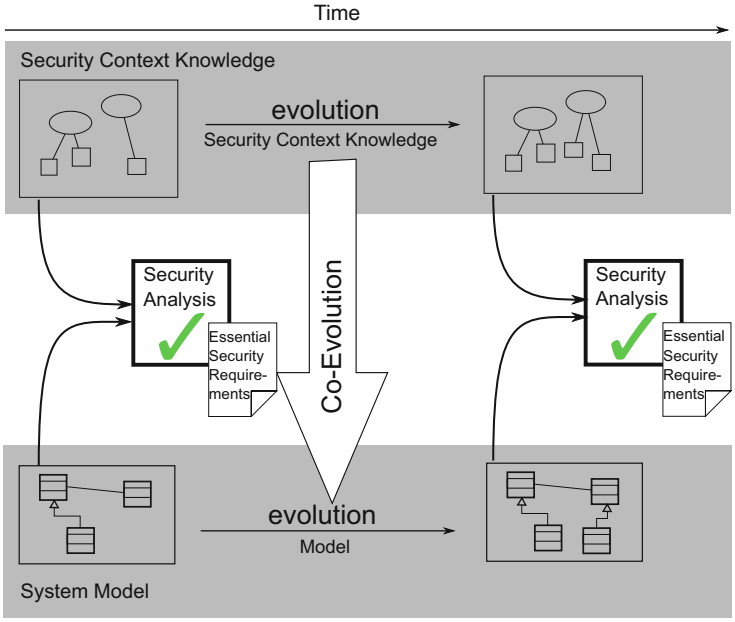

Fig. 9.4 Basic idea behind the design-time adaption, shown by the relationship of evolution to co-evolution

When required, co-evolution is carried out by using generic security knowledge and specific impact information derived from Security Maintenance Model by means of heuristic techniques. To express security requirements in the system design, the UMLsec approach [Jür05] is used. UMLsec provides a profile for UML that allows annotating UML models with security requirements, assumptions, and potential attackers. It also provides checks that determine if certain security requirements are satisfied by a given system model.

9.2.2 Capturing Security Requirements Using Natural-Language Processing

Requirements (e.g. in a specification) can be relevant for security. For example, requirements referring to *buying* something *on the Internet* may be more security-related than *buying* something in *cash in a local store*. Usually requirements are written in natural language so that everybody involved in a project is able to understand and review them. This refers to requirement engineers, developers, customers, and possibly managers. It is highly unlikely that they would learn a special language to understand requirements and check them for security relevance. The voluminous specifications of long-living software systems cannot be effectively screened manually by experts. Our approach provides an automated identification of requirements that are most likely security relevant. This identification starts from requirements in natural language, which can be ambiguous and imprecise. It makes use of the knowledge on security from several sources, including human experts and documented guidelines (Fig. 9.3). Heuristic automated support can reduce the load on human experts substantially. The final goal is to focus their attention and valuable time to the most security-relevant requirements.

To cope with problems of natural language and to semi-automate this process, we perform a linguistic approach to identify the semantic similarity of words. Two words have a high semantic similarity if they have the same meaning with simultaneous syntactical difference [Sch94a]. A numeric value identifies the level of semantic similarity of words. For the security assessment of requirements, we use a heuristic reasoning technique, which is based on Natural Language Processing. The heuristics and the calculation of the similarity value are described in more detail in Sect. 5.3. In this chapter, the security knowledge itself is seen as tacit knowledge of developers. In some cases, they also have a feeling of the security relevance of requirements, but sometimes they cannot explain the reason of the decision that a requirement is security related or not.

If the value of similarity is above a predefined threshold, the term is considered security related. In general, this method uses Security Context Knowledge of the knowledge base to determine about the security relevance of a requirement. The knowledge base is a hierarchical-structured ontology containing security-relevant

words, which is introduced in Sect. 9.2.3. Security knowledge changes over time, such that it is necessary to keep it up-to-date by domain and security experts.

The security relevance of words is domain specific. In some domains, a word can be classified as non-security related; in another domain, it is highly security dependent. For example, on the one hand if we speak about a park in general, a *bank* is a place to sit down for multiple people. In this context, it is not a security-related word. On the other hand, in the context of Common Component Modeling Example (CoCoME) as online shopping platform, the word *bank* with the same syntax has a different semantic meaning. It refers to a company where customers can store their money and transfer it to another owner. The bank details of customers are highly security dependent. To handle this type of context ambiguity, our knowledge base must be enriched by domain-specific knowledge. If a term, which is into the knowledge base and in a requirement, has a linguistic dependency to another term, the requirement engineer is questioned whether the two terms mean the same. By similar meaning of the terms, the knowledge base will be enriched by the new security-relevant term. For example, in the requirement "The user enters an identification number and a PIN", *PIN* is a security-related term. Through the linguistic dependency between *PIN* and *identification number*, the requirement engineer is questioned whether the two terms mean the same. We interleaved the semi-automated acquisition of knowledge enrichment into the security assessment of requirements. The knowledge acquisition and the heuristics are described in detail in Sect. 5.4.3. In Fig. 9.5, the enrichment of the knowledge base through a requirement engineer or a security expert, for example in the context of linguistic dependencies, is visualised. The requirement engineer has to make a decision about the security relevance of words with linguistic dependencies. The approach takes advantage of the collected knowledge described in Sect. 9.2.

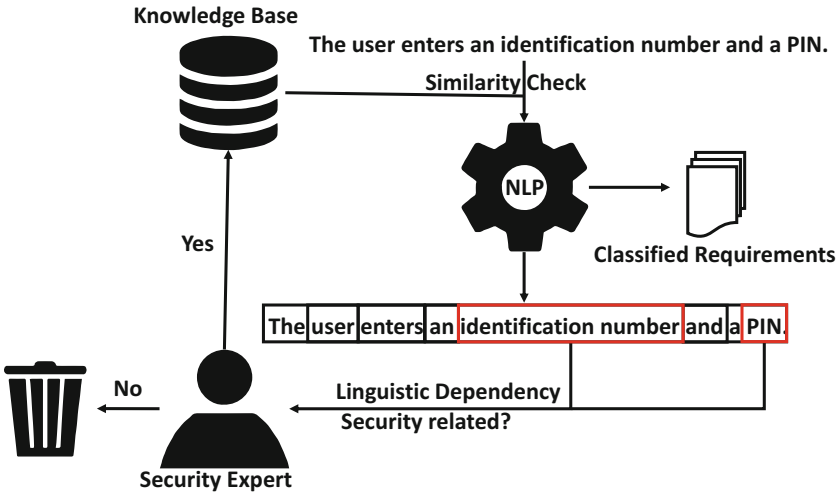

Fig. 9.5 Natural language processing approach—enrichment of security knowledge

9.2.3 Representing Security Knowledge Using Ontologies

The security context knowledge mined from the various information sources needs a suitable representation that is storable, updatable, and flexible enough to support different levels of abstraction and uncertainty. Specifically, security issues cannot be foreseen at system design time and are considered *unknown unknowns* [MH05]. Thus, a suitable knowledge representation that can be adapted to entirely new fields of knowledge is required. To this end, we use the knowledge representation concept of *ontologies* [Gru93]. An ontology contains the key concepts of a domain and the relationships between them. Our technical realisation of ontologies is based on Web Ontology Language (OWL), a standard ontology representation format [OWL09].

Software systems tend to get complex. Consequently, the knowledge necessary to ensure security during its life span grows accordingly. To support the handling of complexity and the sharing of knowledge between different projects within the same domain, the ontologies in our approach have three layers. We work with *nested ontologies* that include, for example, an upper ontology of general security concepts; a domain ontology of system-independent, domain-specific knowledge; and a system ontology of system-specific knowledge.

- We provide a generic *upper ontology* that is independent of a particular software domain or application. It represents the most general software security concepts, such as "encryption algorithm" and "attack". To identify these concepts, we performed a basic literature study [S G+14], followed by a more detailed systematic literature review (SLR, [Gär16, Bür+18]). SLR is an empirical method used to aggregate, summarise, and critically assess all available knowledge on a specific topic [KC07]. Figure 9.6 shows the resulting upper ontology, providing a taxonomy to define a system, its usage, and the surrounding security knowledge.
- *Domain ontologies* allow domain knowledge, as well as concrete security issues and measures, to be captured' for example, the encryption algorithm "DES" is subject to a specific attack called "Davies' attack". Domain ontologies (illustrated below) have to be created for each domain anew and can be shared by different systems in the same domain.
- *System ontologies* express security-relevant knowledge about a concrete system, for example that a specific banking system uses "DES" as its encryption algorithm. These system ontologies can be produced from existing artefacts, such as a UML-based system model.

Figure 9.7 illustrates an evolution step of the domain and system layers of a nested ontology. Class `data` are considered an asset of a system; thus, data are a subclass of class `Asset`. The domain level initially provides the information that data are to be further distinguished into anonymous data and personal data. Furthermore, the salary of an employee has to be considered personal data. Personal data is split into two further categories, and the individual `faith` is added. The evolution step is inspired by a refinement regarding privacy, in accordance with the German federal data protection law (Bundesdatenschutzgesetz, BDSG). Between

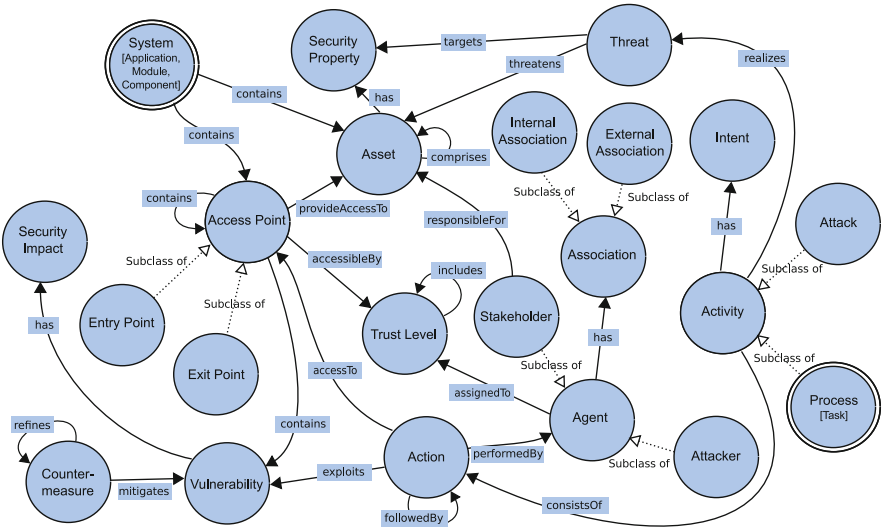

Fig. 9.6 Upper ontology for security

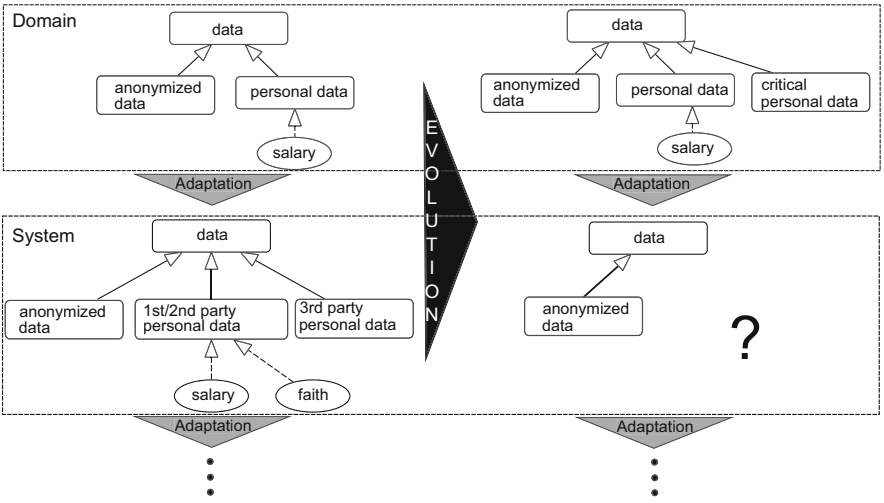

Fig. 9.7 Example of ontology layering and evolution

the 1990 and 2001 revision of the law [J B+15, Ruh+14a, Bür+18], an additional notion of *critical personal data* has been added, which leads to a change of the involved domain layer. Immediately, the question arises how we can adapt the system layer to be consistent with the domain layer again. This question leads us to model co-evolution, as discussed in the next section.

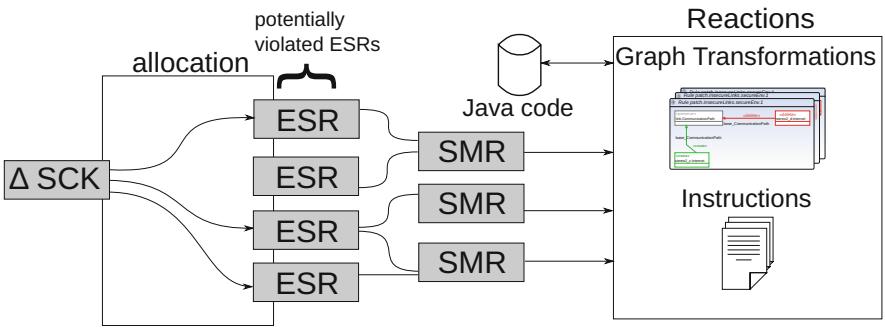

Fig. 9.8 Overview of SecVolution's co-evolution methodology

9.2.4 Rule-Based Model Co-evolution

Whenever the environment changes, the monitoring component may produce a corresponding change of the Security Context Knowledge, written as ΔSCK. Each change is analysed and, when necessary, used to produce reactions based on the process shown in Fig. 9.8.

First, in a step called *allocation*, we determine which Essential Security Requirements (ESRs) are potentially violated. This allocation is given by a mapping and can be supported, for example, by detailed information, where the ΔSCK occurs, that is relationship to specific elements of the upper ontology like *encryption*. To check if the system indeed is impacted by one or more flaws, the system model is investigated using model queries as given by the Security Maintenance Rules (SMRs). To express the model queries, we use model transformation technology, building on the model transformation language and framework Henshin [Are+10]. A system model can be interpreted as a graph. For instance, a class can be interpreted as node and the reference to another class can be interpreted as an edge. Thus, graph transformation techniques can be used to investigate and manipulate system models [JJS15]. A single graph transformation typically consists of two graphs, one called the left-hand side (LHS) and the other called the right-hand side (RHS). Whenever the LHS is matched to a given model, the matched parts are transformed according to the RHS. Elements can be added, removed, or preserved. LHS and RHS are connected through a mapping. Typically, a set of graph transformation rules is called a graph grammar. In Henshin, a number of graph transformation rules are encapsulated by a transformation model.

We implement model queries as rules in which the LHS equals the RHS. Thus, no changes to the model are made, but we can make use of the underlying matching algorithm. We use Henshin since it is built on EMF [Ecl], a standard platform to model software, and it allows us to specify change actions, as required for the next step.

To carry out the co-evolution, Security Maintenance Rules is used. To support this step appropriately, a Security Maintenance Rule consists of three parts:

- A link to ΔSCK
- A *precondition* that the model needs to fulfil for the Security Maintenance Rule to be applied
- A series of *reactions* to realise the actual co-evolution

The reactions itself can be of three different types, as shown in Fig. 9.8. The most formal way is using model transformations to directly alter the system model. Where model transformation approaches like Henshin fall short, for example complex clone operations or path-based analyses, reaction steps can be supported by Java code. Finally, showing the security expert (textual) instructions is meant as basic support for vulnerabilities, which additionally require reactions regarding a system's data or implementation (e.g. user passwords).

Using the *partial match* feature of Henshin, the application of co-evolution steps can be simplified [JJS15, J B+15]. For example, model queries or evaluation of preconditions provide links to concrete model elements of the system model for the transformation rule, that is EMF node instances. These instance links are used to populate the transformation rules, which in turn alter the system model. Using this technique and by additionally utilising the flexibility of Henshin transformation rule EMF objects, our rules actually need fewer elements and can be used in a flexible way. This helps to keep the set of transformation rules low and their understandability high. The co-evolution steps are applied semi-automatically. The security expert can be asked to make design decisions, that is choose an encryption algorithm to replace the now insecure one. Additionally, instructions can be given to the expert, for example "All users have to pick a new password".

9.2.5 Related Work

Natural Language Processing of Security Requirements Compagna et al. [Com+08] integrate legal patterns into a requirement engineering methodology for the development of security and privacy patterns using neuro-linguistic programming (NLP). The pattern design and validation process requires legal experts to describe patterns in natural language. This description is parsed by a natural language processor on the basis of a semantic template. Gegick and Williams [GW07] developed a methodology for the early identification of system vulnerabilities for existing threats. While in our approach we use suspicious sequences to encode hypotheses of possible attack patterns, they use Regular Expressions to encode attack patterns extracted from different web-based security vulnerability databases. A catalogue of such patterns is supplied to map the threat types to elements in the system model. By using a linguistic approach, the requirement engineer can concentrate on the domain-specific problem rather

than modelling it formally. Thus, natural language provides a more flexible notation, and changes can be managed more efficiently.

Haley et al. [Hal+08] present a framework not only for security requirements elicitation but also for security analysis. Their method is based on constructing a context for the regarded system. Describing this context with a problem-oriented notation makes it possible to validate the system against the security requirements. The approach is powerful but needs a lot of security expertise to build the context and understand the results of the analysis. Evolution of the context is not supported.

9.2.6 Leveraging Security Knowledge to Infer Adequate Reaction to Context Changes

Tsoumas and Gritzalis [TG06] provide a security-ontology-based framework for enterprises linking high-level policy statements and deployable security controls. Security ontology is built by extending the Distributed Management Task Force (DMTF) Common Information Model standard. In contrast to our approach, it is focused on organisational controls like how to secure server hardware, recommendations for configuration of intrusion detection systems, and so on.

Ernst et al. [EBM11] use a formal description language to relate requirements to their implementation. Changes identified in the requirement specification are then used to trigger software evolution. The approach is rather formal and aims at providing a graph-based guidance for implementation rationale. Co-evolution is not considered so far, as well as an interface to system design level.

The *Water wave phenomenon* inspired Li et al. [Li+13] to develop an impact assessment approach based on call graphs. First, they analyse the core, which consists of the direct affected software artefacts. After that, the call graph is analysed, taking the interference of different changes into account. Their approach is focused on predicting how big (in terms of number of methods to change) the impact of changing a number of methods in a given source code project will be. Contrary to this, our approach aims at analysing impact regarding security properties.

9.2.7 Summary

In this section, we presented three contributions. First of all, we introduced systematic and experience-based elicitation and management of multiple, heterogeneous knowledge sources throughout the life cycle of a long-living system. This is considered a fundamental step in the process of overcoming the multitude of information sources for the sake of leveraging it do manage long-living systems. As soon as the knowledge has been elicited and structured, it needs to be investigated to assess the effects on a system's security. Thus, as a second contribution, we

introduced a systematic analysis and optimisation of deciding how new knowledge affects the security of long-living software systems. After all, knowledge and reasoning made about the system's security is an additional challenge so that we showed, as a third contribution, how to maintain a consistent database of security requirements and security-relevant environment knowledge during evolution of a long-living information system.

The three core challenges tackled by the SecVolution design-time approach are related to the first challenge, as introduced in the chapter's preface (*How can security knowledge, available via diverse non-formal sources, be incorporated and utilised for a long-living system design?*). A process for elicitation of various knowledge sources is provided, which is able to deal with ever-changing knowledge that long-living systems are confronted with. Maintaining a database of security knowledge during evolution contributes to the second challenge (*How can developers and security experts be supported to react to context evolution, which may compromise the system's security design or compromise the system at run time?*). Developers are provided a consistent knowledge base that can be kept up to date when facing context evolution. The SecVolution design-time approach is focused on typical design-time development artefacts like UML models.

The SecVolution design-time approach has made the following contributions in detail. We developed a security assessment technique for supporting the maintenance of long-living information systems independent of the process model, domain, or technology in use [S G+14]. We created a core ontology usable for different security areas (e.g. privacy, information flow, attacker model) [Bür+18], as well as techniques for reusing and structuring the knowledge-modelling process [Ruh+14a]. We used UML profiles to define extension points in the models that are connected to the knowledge. For the case of an initially secure system, we developed a model-based security verification strategy [S W+14] that can efficiently determine whether a particular co-evolution restores security requirements that were satisfied before the evolution [Bür+18]. The strategy is supported by an extensible tool platform, CARiSMA, [Ahm+17] that reads the annotations of UML 2 models and computes a delta model containing all possible evolution paths of the given model. We presented an approach [JJS15, J B+15, Bür+18] in which changing security knowledge is analysed and possible reactions are derived. It also covers newly occurring knowledge about security issues or attacks.

9.3 Integrating Model-Based Security Constraints with Program Code

The reliability of security analyses is crucial for effective security strategies in long-living software systems. Figure 9.2 uses Unified Modeling Language (UML) profiles to define security information in models. One kind of model to which security information can be attached is architecture models, which usually

describe components and their interconnection. This information can be used for architecture-based security evaluations. These evaluations are only reliable as long as the architecture implementation is consistent with the architecture models. This consistency can be invalidated via multiple influence factors during the life cycle of a long-living software system: (1) The program code evolves, so that it is no longer consistent with the security models. (2) Security models may be based on architecture models. When the underlying architecture models evolve, the architecture-based security model is inconsistent with the actual architecture. In both cases, the analysis models must be changed accordingly and the security analysis must be repeated or adapted, or the results of the security analysis might be invalid.

In this section, we present an approach to create a continuous consistency between architecture model information, architecture-based security information, and the program code. The approach has been developed as a part of the ADVERT project within the priority program. It addresses the challenge to document security information so that it is strongly related to the program code, to support the analysis and monitoring of security aspects.

9.3.1 Codeling: Integrating Architecture Model Information with Program Code

A set of abstract concerns commonly agreed upon seems to exist for defining software architectures, as manifested by the standard ISO/IEC 42010 [ISO11b]. These include the general structure of a system, usually expressed in components, interfaces, and their interconnection. They are often accompanied by abstract behaviour descriptions or quality aspects. During the development of software, the architecture is realised in the software artefacts, including the program code, configuration, and the use of existing platforms. The goal of the implementation is an executable system. The implementation of software architecture is driven by industry standards and platforms that define standard elements such as components and interfaces. Languages for architecture specification and for architecture implementation have common concerns (see, e.g., [MBG10]), typically at least the definition of components, interfaces, and their interconnections. However, they have different foci and include different types of architectural design and different details added to the architectural description.

The tool *Codeling* [Kon18, Kon16] creates a systematic mapping between architecture specification model elements, relations, and attributes and their implementation based on standardised or project-specific architecture implementation languages. These mappings specifically define places where arbitrary other code can be added. This kind of mapping allows to extract architecture specification models from program code and to propagate changes in these models back to the code.

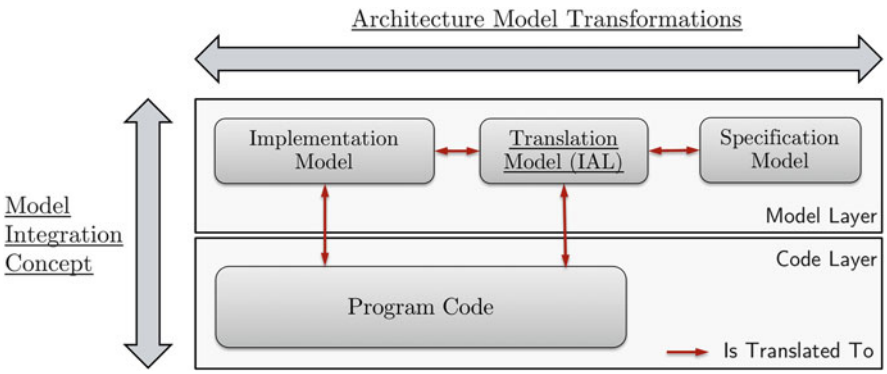

Fig. 9.9 The parts of Codeling for integrating architecture model information with code

Codeling comprises three parts. Figure 9.9 sketches an overview of these parts and their relations. The figure describes artefacts of the approach with rounded boxes and translations between these artefacts with arrows. The parts are used to bidirectionally translate between program code and a specification model expressed in an architecture specification language. The parts are underlined in Fig. 9.9. (1) An *Intermediate Architecture Language (IAL)* mediates between architecture implementation models and architecture specification models. The IAL has a small core with the common elements of architecture languages. The core is extended with a variety of stereotypes to represent, for example, different kinds of interfaces, component hierarchies, or quality attributes. Models expressed in the IAL are called *translation models*. (2) The *Model Integration Concept (MIC)* describes an approach for bidirectional formal mappings between program code structures and an implementation model expressed in a meta model of an architecture implementation language. As an example, a Java-type declaration that implements a specific marker interface might represent a component, and static final fields within this type declaration represent attributes of this component. In Codeling, the program code also contains information that is not part of an architecture implementation language but is only subject to a specification language. For example, many architecture implementation languages do not describe hierarchical architectures. The hierarchy information can be annotated in the program code. The translation model is enhanced with this information from the code using the Model Integration Concept. (3) Bidirectional architecture model transformations translate between implementation models, translation models, and specification models.

With the tool *Codeling*, architecture model specifications are integrated with program code. The models can be embedded into and reliably extracted from the code, leaving only the program code as single underlying model.

9.3.2 Application: Security Evolution Scenario

The running example within this section is CoCoME (see Sect. 4.2), a common case study for software architecture approaches. Figure 9.10 shows a subset of the CoCoME architecture expressed in the UML. In the running examples, three user roles interact with the system. *Cashiers* scan items at a cash desk and execute the sales process. *Store Managers* manage the store's inventory. They buy inventory and may see reports about their store's sales and inventory. *Enterprise Mangers* support the store managers. Therefore, they can see reports of sales, see the inventory of multiple stores, and trigger the exchange of inventory items between stores.

The excerpt shows two components: *ReportingServer* and *StoreServer*, both subcomponents of the component *Application*. These two components are user interfaces to the system and should provide their services only to authenticated users. The component *StoreServer* provides the interface *IStoreInventoryManager*, which should be accessible to store managers and enterprise managers. The component *ReportingServer* provides the interface *IReporting*, which should be accessible only to enterprise managers. For security reasons, customers are not allowed to use any of these interfaces.

The UML diagram in Fig. 9.10 is enhanced with Secure Information Flow (SIF) [RJ12] information. SIF annotations in UML diagrams define authorisation rules for structural elements in an architecture, which decide about the access of a partially ordered set of roles. In the running example, the following authorisation constraints are defined:

1. The component *StoreServer* provides the interface *IStoreInventoryManager*, which should be accessible to store managers and enterprise managers.
2. The component *ReportingServer* provides the interface *IReporting*, which should be accessible only to enterprise managers.

For security reasons, customers are not allowed to use any of these interfaces.

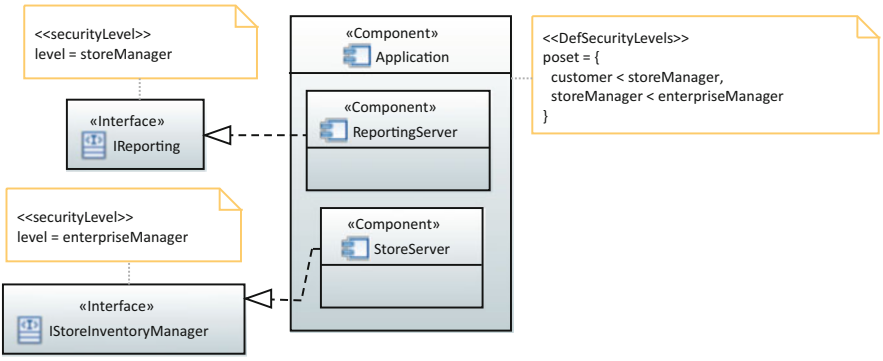

Fig. 9.10 Excerpt of the CoCoME architecture extended with SIF annotations in UML

Table 9.1 Overview of the mapping between UML meta-model elements, CoCoME meta-model elements, and program code structures

UML meta model element	CoCoME meta model element	Program code structures
Component with name "Server"	"Server" component	Type declaration with name "Server"
Composite component	Component with children	Package declaration with package or type declarations as subcomponents
Operation provided role	Provided interface	Implemented interface
Operation required role	Required interface	Interface instance given to type via constructor

In the context of the running example, Codeling is used to create a UML view upon the CoCoME architecture. A formal mapping between interconnected UML components and CoCoME program code already exists in Codeling. Table 9.1 gives an overview of the mapping between the CoCoME code, the corresponding architecture implementation language, and UML meta-model elements. The table only contains the mappings that are relevant for adding SIF information to the running example. To integrate SIF information on UML components, interfaces, and operations with Codeling (a), the IAL must be able to handle this information. Also mappings must be created (b) between the IAL and the CoCoME code and (c) between UML stereotypes for SIF and the IAL. In the following, these translations are described.

9.3.3 Security Aspects in the Intermediate Architecture Language

A translation model in Codeling is implemented using the IAL. Figure 9.11 shows the core of the IAL. Within Codeling, the IAL core is used to describe architectures with component types, which provide and require interfaces. Component instances represent single instances of placeholders for component instances that are dynamically created at run time. This core contains all relevant meta-model elements to describe the elements shown in Table 9.1. The architecture is the root element of the IAL. It comprises component types and interfaces. Component types provide and require interfaces. The IAL also contains run-time elements for component types, their provisions, and requirements.

The IAL core is extended with profiles [Lan+12] for describing SIF information. A profile extends a meta model with new classes and stereotypes. A stereotype application is an instance of a stereotype. When a stereotype is attached to a meta-modelled class, a stereotype application can be attached to the instances of that class. Such extended class instance can then use the attributes and references of the stereotype.

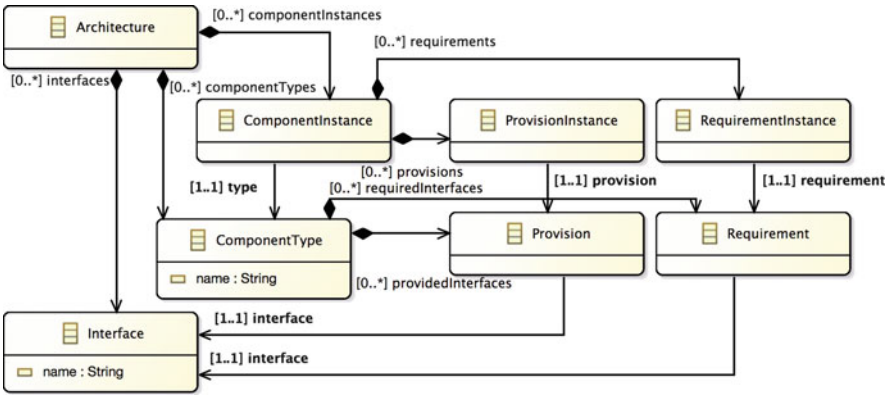

Fig. 9.11 The core of the IAL

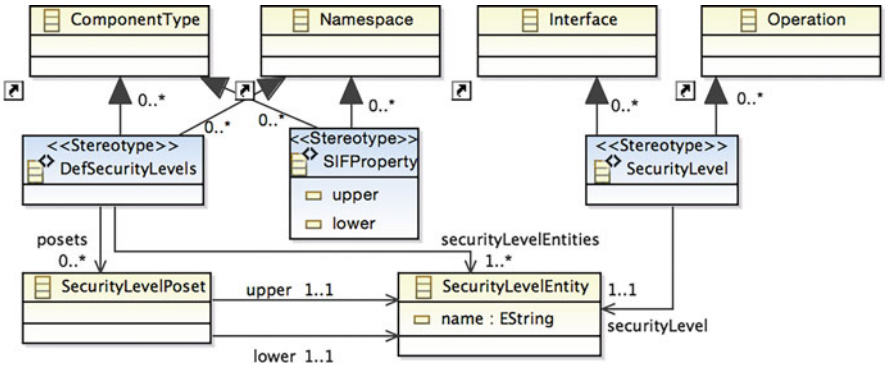

Fig. 9.12 SIF profile for the IAL

Figure 9.12 shows the profile for SIF information. This implementation is based on the definition of SIF from Ruhroth and Jürjens [RJ12]. The classes in the upper row of the figure are references to classes of the core or other profiles. The SIF profile defines three stereotypes. The stereotype *DefSecurityLevels* is applicable to component types or namespaces from the *namespaces* profile (not shown). With this stereotype applied, components or namespaces can declare partially ordered sets of *SecurityLevelEntities*, the *SecurityLevelPosets*. The entities correspond to roles in the system. The stereotype *SIFProperty* is also applicable to component types and namespaces. SIF properties describe the basic security predicates (BSPs). Each property takes two sets of names of BSPs as arguments the upper and lower BSPs [RJ12]. The stereotype *SecurityLevel* is applied to interfaces of the IAL core or operations of the profile for operation-type interfaces (not shown). With this stereotype, the minimum role necessary to use the interface or single operations of an interface is declared.

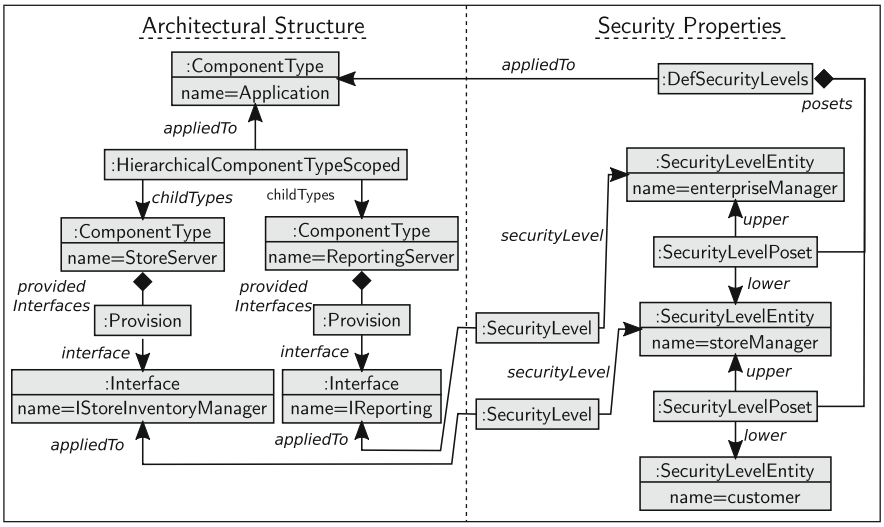

Fig. 9.13 Excerpt of the CoCoME architecture extended with SIF annotations in the IAL

In the running example, the CoCoME architecture, including the SIF information, is translated into an IAL model with the SIF profile applied. Bidirectional model transformations are used to create a formal mapping between the model types. Figure 9.13 shows an excerpt of the CoCoME architecture with the SIF information defined in Sect. 9.3.2. The left-hand side shows an excerpt of the CoCoME architecture expressed with the IAL. A composite component *Application* contains two subcomponents: *StoreServer* and *ReportingServer*. Each provides an interface. The right-hand side shows the SIF information attached to this architectural core. The composite component declares three security levels: *enterpriseManager*, *storeManager*, and *customer*. Their partial order is given with $enterpriseManager > storeManager$ and $storeManager > customer$. For using the interface *IReporting* of the component type *ReportingServer*, the user has to be an enterprise manager. For using the interface *IStoreInventoryManager* of the component type *StoreServer*, the user has to be a store manager or an enterprise manager. Customers are not allowed to access any of these interfaces.

9.3.4 Integrating Security-Architectures with Code Using the Model Integration Concept

Next, a mapping between SIF profile elements in the IAL and program code structures is necessary. These program code structures must work with the program code structures used for the translation of CoCoME architecture elements, briefly described in Table 9.1.

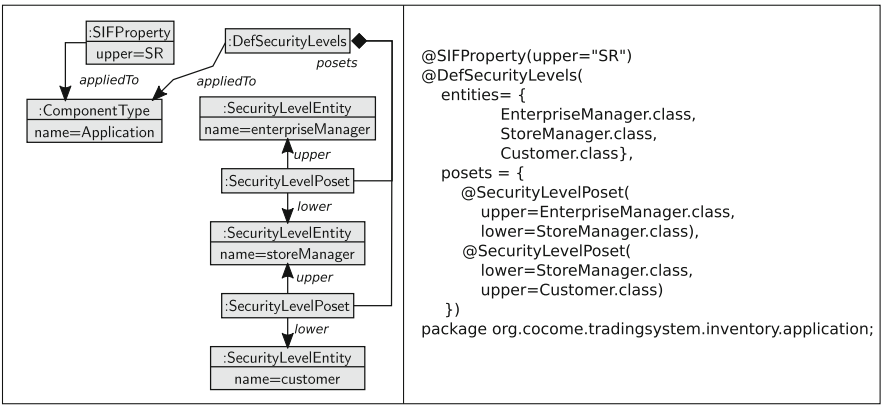

Fig. 9.14 Mapping between the *DefSecurityLevel* and the *SIFProperty* stereotypes and program code structures in CoCoME

The stereotypes *DefSecurityLevels* and *SIFProperty* are applicable to components and namespaces. In the context of this example, a definition of security levels and SIF properties should be applied to the composite component *application*. In the CoCoME program code, this composite component is represented by a Java package with the name `org.cocome.tradingsystem.inventory.application`. A feasible code structure for the given SIF information is annotations on the package. Therefore, a file `package-info.java` is created in the corresponding folder, applying a respective annotation to the package declaration. The corresponding annotation declaration is part of an external, reusable library that is generated for this purpose. Figure 9.14 shows this mapping. The SIF property annotation owns two members of the type String: `upper` and `lower`, corresponding to the respective stereotype attributes. In the example in Fig. 9.14, the value "SR" for the member `upper` denotes *Strict Removal*. This means that all confidential events are independent of events that are visible or "neither-nor" [Man03, RJ12]. The annotation for the definition of security levels and their partial order `DefSecurityLevels` is also attached to the package declaration. Its reference `entities` takes an array of types as parameter, which extend a specific marker interface. This marker interface denotes that the implementing type represents a security entity. This mechanism is used to use the type-safety features of the Java compiler to validate the member values at compile time. In addition, typical IDEs propose known security entities via their code completion features. The same mechanism is used for `SecurityLevelPosets`, where a lower and an upper entity are given as values.

In the example, only enterprise managers are allowed to access the interface `IReporting`. In the CoCoME program code, this interface is represented by a Java interface with the same name. A feasible code structure for the security level is an annotation on the interface declaration. Figure 9.15 shows the mapping as an

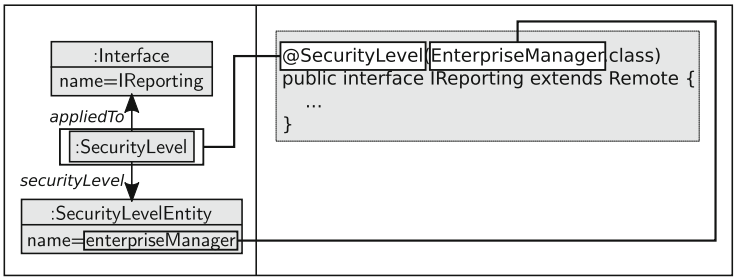

Fig. 9.15 Mapping between the *SecurityLevel* stereotype and program code structures in CoCoME

example. The default annotation member is a reference to a type that represents a security entity, that is it implements the corresponding marker interface.

9.3.5 Related Work

The relationship between models and code is subject to related work. The field of model-code co-evolution describes how models and code can evolve together. Works in this area usually focus on one specific type of model. For example, Langhammer [Lan17] describes an approach for the co-evolution of Palladio architecture models and Java program code. Langhammer describes rules that preserve a consistency relationship between the architecture model and the program code during changes in either side. Ruhroth et al. [Ruh+14b] present an approach for managing the consistency between certain security models and code. Their approach synchronises atomic change operations on models and corresponding operations on code. Our approach instead explicitly integrates arbitrary model information with program code.

Approaches for the co-evolution of models and code often do not consider the evolution of the underlying languages. Rocco et al. [Roc+14] explicitly describe language evolution as aspect of model-code co-evolution. When a system is modelled using meta models and corresponding code is generated, a challenge arises when the meta model is subject to evolution. Such changes can break the code generators. This is a case of model-code co-evolution: the meta model can be regarded as model, and the code generator can be regarded as code in the context of model-code co-evolution. The authors propose a co-evolution approach where model changes are propagated via well-defined transformations, which operate on the code and take the model difference as input. This approach can be used to handle architecture language evolution regarding model editors but not regarding the code that implements a system's architecture.

The synchronisation between models and between models and code is subject to the area of (in)consistency management [Fel+15]. These approaches assume

that two views upon a shared body of information overlap. When one view is changed in the overlapping part, these changes should be propagated to the other view. Consistency management deals with methods and tools to re-establish synchronisation. Existing consistency management approaches focus on coarse-grained program code structures, such as code files or classes, and relate them to model elements. Konersmann [Kon18] argues that a more fine-grained abstraction level is necessary and implements such consistency relationships in Codeling. Vitruv [KBL13] is a more general approach to keep different views consistent. It bases on coupling EMOF-specified meta models. For coupling the Palladio meta model for architectural specification with Java, see the PhD thesis of Langhammer [Lan17].

Already in 1995, Murphy et al. [MNS95] presented an approach for bridging the gap between program code elements and higher-level software models. In their approach, a mapping is created between higher-level model elements and program code elements. The approach of Murphy et al. is limited to mappings between model elements and program code files, neglecting the structures within the code files. Approaches need to address structures within the code files to add decision knowledge to specific architecture elements in the code.

9.3.6 Summary

This section presented the application of Codeling on security information. It is used to create a formal bidirectional mapping between security model information attached to architecture specification models and the program code that implements the security architecture. Therefore, the presented application addresses the first challenge of *diverse non- or semi-formal sources of security knowledge*. We have shown in this section that Codeling can be used to integrate model-based security information with program code, using a formal bidirectional mapping. The implementation allows to specify the security annotations in a model-based environment and in the program code. The program code takes the role of a single underlying model.

The program code structures that are used to represent the model information are also accessible at run time via introspection. Therefore, it is possible—and supported by Codeling—to create or extend a run-time environment so that the security constraints defined in the program code can automatically be verified. The approach can therefore be used not only for documentation and for relating security information to architecture model elements but also for monitoring the application security. This addresses the second challenge, because with this monitoring, developers and security experts are supported to react to context evolution, which may compromise the system's security design or compromise the system at run time.

9.4 Contextual Security Patterns

For the evolution of large and long-living software systems, it is essential to understand not just the existing parts of the software, like requirements, design/architecture, or code, but also how these elements could change over time and especially how the corresponding components (inter-) act or applied in run time. In particular, for maintaining the security of software-intensive systems, one has to consider not only changes to requirements, which result in adaptive, corrective, or perfective evolution steps in the system directly, but also changes to the context of this system. This context comprises the various parts of the execution platform, but also changes in attacker capabilities; changes to user role models, including defined use and misuse cases; new access policies; protocols; or run-time configurations.

An important factor for a successful architectural approach is the understanding that even with well-defined interfaces between components and subsystems, their inner behaviour (i.e. implementations) or usage profiles can change. This does not necessarily mean that the interfaces are accordingly modified, which can result in security problems, where especially the black-box-modelling is favoured. These factors need to be handled thoroughly and explicitly in design time, so the architects can foresee the possible outcomes of evolutionary changes and run-time differences before it is too late or any eventual costly and complex interferences are necessary. To address this issue, we provide a lightweight architectural documentation and analysis approach using security patterns enriched with explicit decision assumptions and prerequisites. In this section we introduce our approach.

9.4.1 Security Challenges in Software Evolution

Software security is a cross-cutting consideration with respect to various software life cycles (from requirements elicitation to maintenance) and with respect to other quality attributes (e.g. performance, reliability, etc.). Regarding the evolution software systems based on their security properties, there are several identified key challenges [Sei+16]. For our approach, the following issues are of great importance, which we categorise into two groups. System evolution is still an important factor as it is permanent and phase spanning. Changes within the known system boundaries can still be fuzzy, which need to be monitored thoroughly. However, context evolution results in more challenging issues, as the effects are not explicitly known or cannot be identified without further data in design time. We, again, list three main issues within the evolution of context of software-intensive systems as follows:

Threat Evolution Attackers' capabilities evolve very fast, and they are unpredictable. Therefore, continuous execution of security analysis are needed at run time to identify new vulnerabilities, which becomes a very costly process.

Deployment and Infrastructure There are a lot of factors regarding the resource environment and the allocation of the components that affect the correct appli-

cation of security solutions and patterns. Furthermore, fuzzy system boundaries worsen the situation due to uncertainty during design phase about the deployment and application of the system.

Application and Run-Time Configurations Configurations of a software system at run time and for its possible applications have strong influence on the system security. Early extraction is especially difficult, and there are no general approaches for multiple configurations with respect to security.

Hence, the effects of evolution can have severe results on the security as well, making irrelevant attacks relevant or making security decisions invalid. Preserving security during software evolution can be promoted by understanding and reasoning the architecture and made design decisions of the software system. However, security vulnerabilities are most often code related; still architectural misconceptions will create security vulnerabilities. Hence, an architectural security analysis can yield such risks and vulnerabilities in prior phases of a project and support its evolution. However, it is not comprehensive as code-related vulnerabilities need additional analyses. But architectural design decisions, such as using specific security design patterns, suit very well, and as the software architectures are a specific abstraction of the whole system, security needs to be addressed on the same abstraction level. Well-structured security patterns, to be decided and modelled at design time, suit very well to address and mitigate such vulnerabilities and risks. On the other hand, abstracting security properties often result in loss of rational knowledge and makes it complex to validate design decisions regarding security, as changes to software or its environment happen. Security, as well as other non-functional requirements such as performance, must therefore be addressed explicitly by the architects because these properties are determined not just by individual architecture components but also by their interaction, coordination, and usage at run time. We call these factors, on which the correct functioning of security patterns rely, the *context* of the software system. If decision-making process does not involve this explicit context information, security is doomed to degrade. In this section, we describe our approach addressing this problem by handling context information regarding the security properties explicitly on software architectures.

9.4.2 Contextual Security

Software security is a quality attribute that depends on many factors based on the contextual nature of the software. Most of these factors are usually unknown to software architects at design time, like the attacker behaviour or run-time configuration. Architects need to assume such aspects, if they ever do it in the first place, regarding their security decisions. Furthermore, like in any software architecture, where an abstraction is necessary to focus on only relevant aspects of the system, a lot of information becomes implicit, including assumptions about the possible security threats and applied patterns. This type of unpersisted information gets

usually neglected, especially as the software evolves and existing design decisions change, which would result in unnoticed security issues. Furthermore, due to the missing *security-related* information at design time, architects cannot foresee the probable security vulnerabilities that may arise first at run time, even if any related decision was made at design time. Hence, solutions addressing such security issues on architectural level need to consider the corresponding abstraction and has to be handled with limited amount of information, which still describes the *context* of the system in order to provide proper security for the data and function. Furthermore, in case the problems are repetitive, that is they happen to be recurring from time to time, even with contextual and application-specific differences, design patterns can be helpful in providing a generic solution, which needs to be made concrete later for the specific situation. One way to systematically deal with contextual security is the documented use of security patterns. Therefore, we extend descriptions of security patterns in their pattern catalogue with information on contextual security. This means we document which security-related assumptions a pattern needs to make on the context. This information is then used as structural logical constraints between the security patterns and threats, for whose mitigation they are designed to.

9.4.3 From Design Patterns to Security Patterns

Design patterns have their roots in civil engineering, where it is about the architecture of buildings and structures and not the software. According to Alexander [Ale77], "each pattern describes a problem which occurs over and over again in [...] environment, and then describes the core of the solution to that problem, in such a way that [...] can use this solution a million times over, without ever doing it the same way twice". Although this stands for a very different domain, the definition fits for software design patterns as well. In any case, patterns describe "a solution to a problem in a context" [Gam+95]. Furthermore, according to Schumacher et al. [Sch+05], a security pattern "[...] describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution to it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular such structure." This definition again underlines the importance of the context in which the problem and its solution resides, meaning the software and its context, including the run-time environment, user behaviour, etc.

It is important to mention, by analogy with design patterns of Gang of Four [Gam+95], that they are not invented on behalf but rather discovered/identified as a possible reusable security solution. Security patterns are specified using specific templates, like the design patterns. There is no standardised template, but in general they consist of at least a name, context, a problem statement, a solution, known

uses, and consequences. A few simple examples for security patterns in real-life applications would be the **role-based access control** or **application firewall** [Fer13].

9.4.4 Security Patterns as a Means for Contextual Software Security

By using security patterns and patterns in general, no reinvention of the wheel is necessary, which grants time and resources, and also a concise unambiguous documentation is established. This documentation based on security concerns (incl. threats and security patterns), structurally applied on the architectural level, can encompass the necessary contextual information crucial to the validity of the security solutions chosen by the software architects. By using security patterns as a means of mitigation against the modelled attacks or for resolving security issues in software architectures, architects also support different aspects of design decision-making process. It plays a crucial role at the design time, which reduces the further complexity and unnecessary complexity at run time. Furthermore, integrating such solutions into software architectures as rational knowledge base allows them to use this structured documentation as first-class software entity if they are working with model-driven software engineering methods [Völ+13]. Hence, the security patterns can be treated equally to the code within the entire life cycle of the system, become a primary element in implementation, and support architects or developers with automatic code generation, maintaining the system as it evolves or analysing/monitoring the run-time security state of the system. However, within our approach, as described in previous sections, security patterns are extended by explicitly using formally structured context prerequisites. It allows us to exploit this architectural documentation to check the correct application of security patterns in case of evolutionary changes and trace the impact on concrete architectural components responsible for the security solutions.

These ideas led to an architecture-based approach [TH16a] as an extension to architectural description languages (ADL) with security patterns, context prerequisites, as well as other security artefacts (e.g. modelled threats). Within this, the necessary profiles and stereotypes are provided for the integration of the models and catalogues into a specific ADL, the Palladio Component Model [Reu+16]. An overview of this approach can be seen in Fig. 9.16.

The abstract workflow of using security patterns enriched with contextual information can be summarised in a few points:

- The security expert creates the initial reusable, model-based security catalogue and documents the security patterns in combination with the possible threat mitigations. In it, the main security elements, that is attacks and security patterns, are logically combined via prerequisites, as can be seen in Fig. 9.17. No direct relation otherwise exists between them. Prerequisites, the architectural

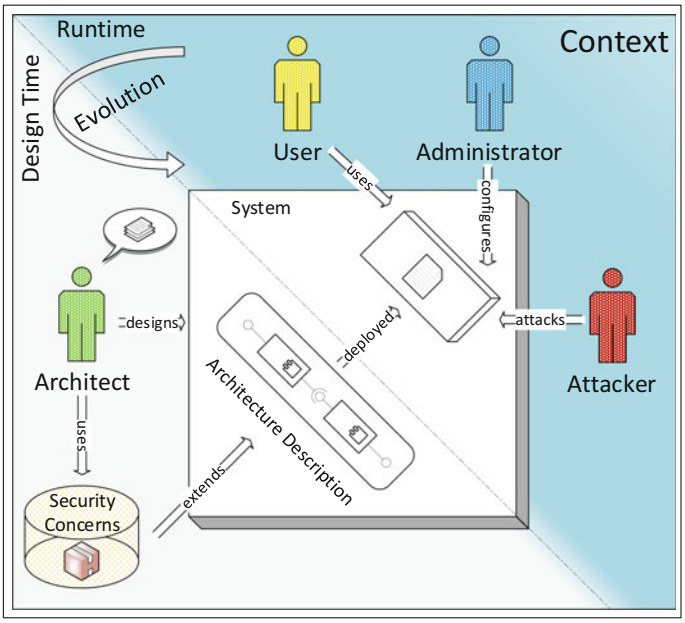

Fig. 9.16 Overview of the security pattern approach exploiting the contextual and run-time information on software architectures [TH16a]

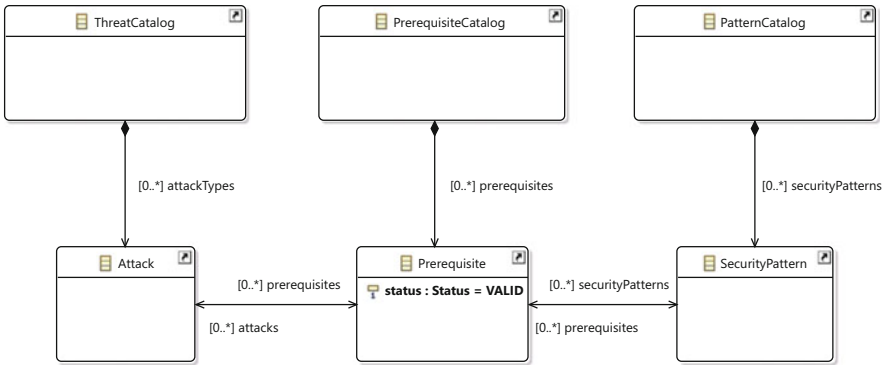

Fig. 9.17 Overview of catalogues containing the main security elements

representations of contextual assumptions, serve as success parameter for security patterns and are necessary for the correct and continuous application of the patterns.

- The software architect designs the software system and describes it using any architecture description language (ADL). The necessary knowledge of the systems context (spanning from usage profiles to possible configurations by the administrators or from deployment environment to even attacker behaviour)

can then be derived from the security catalogue. The integration of the security catalogue with the corresponding architecture description language happens with the well-known profiling and stereotyping. For this, the structural roles of the security patterns are mapped by the architect to the software components, which are again extended by the relevant prerequisites from the same catalogue based on the made security decisions.

- As the system evolves or any changes are necessary to make security design decisions, the architect can check the state of the explicit prerequisites to analyse whether a security pattern is still functional and efficient against any threat or whether an attacker is again capable of exploiting a vulnerability despite the existence of a mitigation security pattern.

The described roles and the relation between the elements of our approach are depicted in Fig. 9.18. An initial application of our approach is already conducted within the CoCoME case study, and its more detailed description can be found in Sect. 12.1.3.

As for validating the systems' security in case of any changes that reflect themselves in the made prerequisites about the security decisions, an analysis method, which is based on the propositional logic, is introduced [TH16a]. Further improvements have been made since then, and the method is consisted of two parts: (1) security pattern analysis and (2) trace impact analysis.

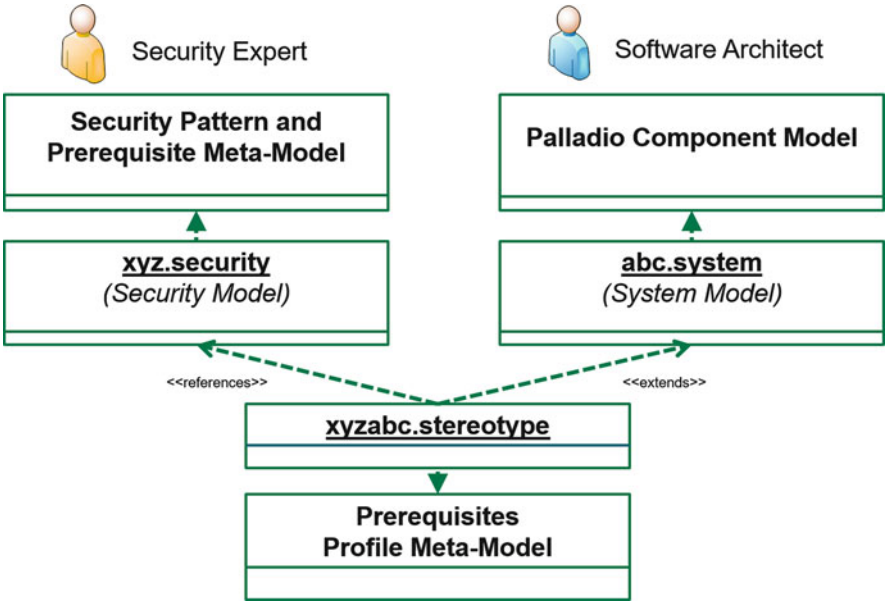

Fig. 9.18 Depiction of the roles and architecture elements for our approach

1. Security Pattern Analysis first checks whether all the necessary structural roles of security patterns are correctly applied on the architectural level. If a necessary role is missing in the first place, a security pattern cannot generally function correctly, and the vulnerabilities it should cover will be present. If all the necessary roles of a security pattern are present in the system's architecture, then contextual analysis is conducted. For this, it is checked whether any of the prerequisites of any possible attack on the system is covered by the security pattern. An attack can be successful and issue a risk if and only if all of its prerequisites are valid. Logic of the analysis anticipates to at least cover one of the required prerequisites for a possible mitigation of the corresponding attack. So if a security pattern is able to cover a required contextual prerequisite for an attack, it can be deemed to function correctly.
2. Trace Impact Analysis is conducted in case of any evolutionary changes, which result in changes to the secure state of the system. If an attack happens to be issuing a risk after the change happens and it is shown in security pattern analysis, the architect follows the roles of the security pattern in question over it stereotypes and changed prerequisites to trace to the architectural elements.

This analysis allows architects to react on possible evolutionary changes and different run-time scenarios with respect to security in early design phases, which becomes a complementary security measure to methods like code-based security analysis or penetration testing, where the code and concrete run-time environment have to be present.

9.4.5 Related Work

General-purpose (e.g. UML) or more specific (e.g. PCM) ADLs have often no direct support for security modelling. Nevertheless, there are several approaches and extensions addressing this gap, some of which also provide further support like analysis or simulations. Schneier [Sch11] introduced attack trees based on feature modelling to model threats, which are described based on the attackers capabilities. A tree-based structure is used to represent all possible attacks, with the main goal of an attacker placed in the root element and the different ways to achieve that goal exploited in the child nodes, which can be semantically enriched with values like probabilities or costs for validation purposes. However, this approach focuses only on the threat side and not on the architecture itself, including security patterns, or on its context, and due to neglected security patterns, it is not possible to easily handle security-related software evolution or any analysis thereof. An industrial approach to security modelling is "Security Development Lifecyle" (SDL) [Sho14], a practical process that is developed to accompany security-related decision-making. It is consisted of two catalogues: (1) STRIDE to model threats based on fix categories (e.g. tampering, denial of service, etc.) and (2) DREAD to evaluate the modelled threats based on possible impacts and a numeric scale. However,

considering only the attack side of the security leads eventually to inconsistencies between threat possibilities and applied security solutions.

An extension to UML is SecureUML [LBD02], which focuses solely on the system access. It specifies constraints for authorisation to define role-based access control and analyse discrepancies. Another extension is UMLSec [Jür05], [Ahm+17]. It introduces predefined profiles containing security-related stereotypes to cover security properties on architectural level, which are used to represent the component roles and the threat abilities that can exploit these roles. SecVolution [J B+15], on the other hand, builds upon it to support evolution. It provides a process model (consisted of a system and maintenance model) for security requirement elicitation, which combines the experiences gathered during development and possible evolution scenarios, which can support co-evolution. These extensions therefore focus on single principles of security (access control and information flow respectively) in an information system, and the analysis and evolution supports are either non-existent or can be limited in representing generic security information. This is why we see the need for a more expressive and adaptable model based solely on using security patterns and their analysis.

9.4.6 Summary

Software security is a very fragile quality attribute that is dependent on a lot of factors existing from run time, which are mainly unknown, to software architects during design time. So architects can only assume and document these assumptions if at all. After a brief introduction regarding the security patterns is given, this section presents in this matter the contextual security patterns approach, which mainly incorporates two sides of security (threats/attacks and solutions/mitigations) into an extension for ADLs and combine them via explicitly documented and accordingly formalised context assumptions called prerequisites. This extension handles security concerns on architectural level, in which the context-related information of security patterns and attacks are explicitly gathered and structurally documented. These prerequisites are used as a metric for model-based security analysis, which checks the validity of applied security patterns based on the software system state or run-time information. That way, software architects can further use analysis results to foresee the impact of evolutionary changes and trace them on system models and accordingly during software evolution, which could ease the process of maintaining the secure state of the underlying system.

9.5 Self-adaptive Security Maintenance at Run Time by Identifying Suspicious Behaviour

An observation made is that evolution in the system environment may lead to vulnerabilities or ineffective security mechanisms at run time. For example, a new attack pattern may be invented or a regulation might call for a more rigid privacy policy. The information system remains insecure or must be shut down until the security violation has been fixed. Finding and implementing a solution takes time. In cases where the system needs to be shut down, this is costly or may even be impossible for large and long-living systems. To get a security fix right, by considering all involved artefacts such as requirements, UML models, or code units, and respecting the system design to avoid architecture decay, this even more calls for careful acting. The design-time approaches, as discussed so far, focus on design-time artefacts and thus fall short on analysing vulnerabilities coming from the source code or the execution context. Moreover, there are security requirements that are hard to check statically, for instance when mechanisms like Java reflection are used [EL02, Mur+98, CM04]. Apart from that, systems that are requested to be available via the Internet and without downtime are more likely to be affected by an attack previously undocumented. In this cases, to avoid downtime or to narrow the attack surface, it is desirable to also detect new attacks, for example based on suspicious behaviour. This section presents work towards monitoring and adapting a long-living system at run time.

9.5.1 Overview

Maintaining a critical system needs expertise in the field of security. Although more and more violations can be prevented by technical means, the experience and expertise of security experts to deal with new attacks remain irreplaceable because many processes and approaches cannot be fully automated.

According to the 2017 Global Information Security Workforce Study, commissioned by the Information System Security Certification Consortium $(ISC)^2$, Europe will face a gap of 350,000 cyber-security professionals by the year 2022 [Int17]. For example, even organisations like the European Telecommunications Standards Institute (ETSI) only employ external security experts for a limited time [Hou+10]. Thus, security experts are few in number, and it is reasonable to support them to become as efficient as possible. Technical mechanisms for preserving security must be complemented by procedures and cognitive support for human experts who are willing to share their knowledge; they must be empowered to do so at the least effort possible.

Challenges include eliciting and modelling adaption requirements. A static view on the system's security is not sufficient. Therefore, multifaceted run-time information of the software system needs to be continuously monitored and analysed

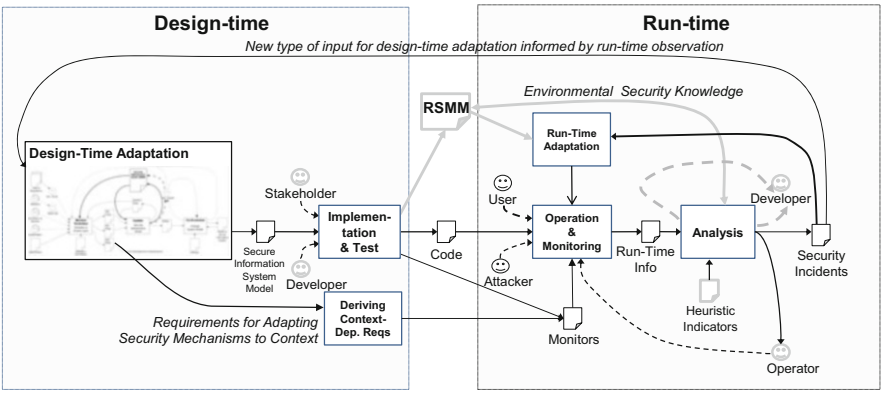

Fig. 9.19 Overview of the SecVolution run-time approach using the information flow syntax described in Fig. 9.2

with respect to given security requirements. Current infrastructure, configurations, and deployment information must be monitored, as well as relevant aspects of user behaviour. When an incident or a suspicious behaviour occurs, the adaption mechanism must make a decision as quickly as possible. After a system has been designed, it is implemented and gets deployed. To react to security breaches during run time, a specific security mechanism must be selected and put into operation and may be adapted to a certain extent.

Figure 9.19 presents the overview on the run-time security adaption approach. The rectangle captioned *Design-Time Adaption* captures the design-time process, as presented in Fig. 9.3. The outcome of this process is a model of the secure information system model. After the implementation and testing phase, executable code exists that is run and operated by the user. In contrast to the system design phase, an additional attack vector occurs from attackers challenging the system at run time. To bridge the gap between design time and run time, an extension of the security maintenance model, called *Run-Time Security Maintenance Model* (RSMM), is proposed. During the operation phase, the system is monitored. Monitoring data are continuously recorded and analysed. Supported by heuristic indicators, incoming monitor findings are assessed, with the assistance of the developer and operator. When a security incident is ascertained, it is decided if there needs to be a run-time or design-time adaption. Run-time adaptions are accomplished by adapting the running system. Moreover, security incidents discovered during run time can also trigger design-time changes.

In Sect. 5.4, an approach is presented to externalise tacit knowledge during run time. The focus is to gain insight of how users interact with the system to learn about which system requirements should be adapted, removed, or added. The focus of the run-time approach presented in this section is to assume a mostly static set of security requirements and check the system's compliance to this requirements, in conjunction with evolving security knowledge. While tacit knowledge in Sect. 5.4 is

used for the way a user interacts with a system and which features he uses in which way, in this section it can be seen as attack sequences an attacker can carry out.

Application: Running Example

As this motivating example, we consider an extension of CoCoME as introduced in Sect. 4.2. For our motivating example, we consider an extension of CoCoME with mobile shopping applications for a CoCoME online store. Mobile shopping applications for a CoCoME online store need to prevent attackers from exploiting entry points like personal data of the other customers, as well as internal business data. Thus, various security mechanisms are used, like cryptographic hashes, to secure authentication procedures (login).

Assume a mobile application (App) for the CoCoME online store that uses the SHA-1 hash-algorithm for the login protocol. This algorithm was considered secure until the year 2005, when a method was published to break the security mechanism [WYY05]. Since the security of the authentication depends directly on the security of the hash algorithm, the developers of the application can react to this change in the security knowledge by replacing the algorithm with another from the SHA-2 family (*design-time mitigation*). The SHA-2 family consists of six similar hash algorithms, each providing a different security level. After the replacement, the application can now choose an appropriate algorithm for a requested connection. More secure algorithms need more computation power and reduce the speed of feedback to the user. Thus, usability and acceptance of the mobile application are at stake, and the client will have to make a compromise between security and usability.

When it was decided to replace the hash function by SHA-2, the developers realised that this would take some time. Since the mobile application generates significant revenues for the company, the mobile application should not be deactivated while performing this update. Therefore, it was decided to take a calculated risk: The company did not want to lose too much business and was willing to accept a certain, limited risk of loss. Since regular user monitoring showed that most customers buy for less than 100 € per month, this limit seemed to be a reasonable compromise: The application was quickly modified to limit the maximal monthly turnover to 100 € per customer. As a result, a few customers might be prevented from spending more money. Most customers, however, would not notice the limit since they spend less money anyway. The company's business is not obstructed, and the turnover is only endangered to a small extent during the time of patching the authentication algorithm. When the new algorithm was in place, the limit could be removed. This strategy ensured that a sufficient degree of security was preserved at all times and with an optimal trade-off to limit negative impact on business.

As computing devices get more powerful, breaking hashes comes within the reach of attackers. Therefore, the weaker variants of the SHA-2 family will also be considered breakable at some future point in time. The system stays the same, but the increasing ability and knowledge of attackers compromise security. Given that the whole family of algorithms is available for implementation now, the system

can easily be adapted to prohibit the use of the insecure variants by an automated run-time adaption of the application.

9.5.2 Capturing Context for Security Adaption

As we briefly introduced in Sect. 9.5.1, a detailed view on the system's context is inevitable. Not only the code itself but also the execution context and information that can be gathered during run time, for example using monitoring, needs to be considered. They all belong to environmental aspects that can cause an adaption. Regarding the running example, if an access routine is executed, the corresponding assets may be at risk. Run-time monitoring can issue a warning at the conceptual level, and it can trigger heuristic reasoning. The new run-time extension of the security maintenance model, called *run-time Security Maintenance Model* (RSMM), constitutes as a formalisation of security-related knowledge at run time. When a concept is implemented, several components may be affected. For example, the asset of a password list can be stored in a database. It uses a granularity appropriate for design-time concepts (e.g. threats or assets). However, it is not sufficient to protect the database; instead, related access mechanisms, user interfaces, and supporting components need to be considered as potential entry points, too. Run-time monitoring [AJY11] and process mining [Aal11] can help spot executed parts of the implementation.

9.5.3 Leveraging Run-Time Information to Support Design-Time Security Adaption

A system during run time produces various kinds of data that may be relevant for assessing the system's security. Regarding the running example, monitoring CoCoME system generates various monitoring data: Not only internal server operations may be relevant for the system's security, but also the interaction that every customer with the system has is recorded. Not only call traces but also database transactions and application server messages can be put into an anomaly analysis.

Natural language analysis can play a role here. A family of heuristics can treat identifiers in source code as "expressions in natural language" (making use of results from work such as [DMJ08]). Through this assumption, certain identifiers are treated like words and can be mapped to security concepts such as *entry point* or *asset*. In an isolated environment, normal behaviour can be recorded. During run time, the monitored behaviour can be compared with the recorded one, also taking heuristic indicators into account to distinguish compliant behaviour from an ongoing security requirement violation. A procedure of selecting appropriate mechanisms to

monitor the desired security requirements is annotated in the model. For example, systems can be proactively monitored to predict potential violations [Zha+11]. The source code corresponding to the model is then instrumented accordingly.

9.5.4 Heuristics-Based Run-Time Assessment to Detect Security Requirement Violations

Run-time information consists of fine-grained representations of what happened during the execution of code and models. For example, log files or code can be monitored to trace the execution of software. A large amount of monitoring data must be managed for complex information systems.

A heuristic indicator associates a defined input (e.g. sequence of monitored data) with a conclusion. For example, a heuristic indicator may conclude from a sequence of repeated online orders that there is a case of misuse underway, trying to bypass the 100 € limit regarding the running example. This could be a violation of a corresponding requirement. Heuristics use shortcuts and unproven conclusions, but they are fast and can be used earlier than an algorithm with a supposedly higher recall and precision [TF97].

9.5.5 Adaption During Run Time

As we argued in Sect. 9.5.1 and illustrated as part of the running example, adaption during run time is a necessary kind of reaction when a security issue is detected that can be reacted upon with a restricted risk. The system model can additionally be annotated to support run-time adaptions in order to reflect implementation details into the model level. This information is used to decide which of the security requirements can be mitigated at design time and which one can only be treated at run time. Furthermore, this can be used to cope with code that is initially generated but then manually altered. The challenge here is to have tracing of security requirements beyond the design time, for which preliminary work exists [L M+10, AJY11].

If a violation of security requirements is detected, appropriate mitigation actions must be taken. Violations such as loss of privacy, information leaks, or attacks on specific assets may be mitigated through different actions. For example, the system can be reconfigured, for example to use alternative encryption mechanisms, or an adaption can limit the access for certain roles. For example, roles that have access over the Internet can only access the system via a virtual private network if data are at risk and transmission over insecure connections should be reduced. Mitigation actions may be inferred from measured behaviour and additional information [EAS14]. The definition of fail-safe components can support an immediate reaction with minimal reduction of features. Detected security violations are reported to

an expert system that retrieves priority and reaction knowledge from the modelled security knowledge. This escalation will use techniques for design-time mitigation.

9.5.6 Related Work

Regarding the question of when a system should be adapted to preserve security, [SDB14] presents how attacks on cyber-physical systems can be observed during run-time.

[ES10] introduces an approach to realise security adaptation at run-time using an ontology that takes context into account. This approach falls short of handling the automatic monitoring of the running system. Our results show that it is feasible to combine monitoring techniques with security adaptation techniques.

[Sal+12] gives an approach for modelling assets that can be used to model the requirements and (security) goals of a system. However, there is currently no security knowledge support. Our approach provides a seamless way of accompanying the development and maintenance process with context knowledge.

[Nhl+15] supports monitoring assumptions about security requirements at run time. However, this approach focuses on security of entities and does not address software development.

[Omo+12, Omo+13] focuses on privacy and the requirements-level within greenfield development of systems, while our goal is to cover security properties, to support also long-living systems (including legacy systems), to cover knowledge evolution, and to also cover system execution.

9.5.7 Summary

The SecVolution run-time approach has identified the following challenges:

- An evolution of the system environment may affect the system's security at run time.
- There are security properties that cannot be checked solely by regarding the system design. Security properties can depend on data that are stored in databases or can generally vary during run time, like access control configurations.
- Mitigating security incidents that arise during run time need to be acted upon also during run time. Investigating and adapting the system design to recover its security is not timely enough if the system needs to stay in service meanwhile.

The SecVolution run-time approach tackles these challenges. The run-time Security Maintenance Model (RSMM) bridges the gap between the design time and run-time development phases of a system. It connects artefacts like code that is based on the system design, as well as run-time relevant data like application server configuration. Run-time monitors are proposed depending on the security properties

required by the system design. Using techniques like process mining and heuristic indicators, raw monitoring data can be used to map running code to parts of the system design. By comparing anticipated and actual system behaviour, supported by heuristics, suspicious behaviour can be detected. By making a system run-time adaptable, ad hoc reactions to security incidents can be realised. By providing alternative components or fail-safe states, for example, controlled precautions can be already part of the system design to deactivate critical system parts or minimise the risk when a security incident occurs during run time.

9.6 Anomaly Detection for Evolving Software Controlled Production Systems at Run Time

9.6.1 Overview

Another area for maintaining security during software evolution is Cyber-Physical Systems. Cyber-Physical Systems are software systems that interact with their physical environment (e.g. embedded systems, automated production systems) and are connected to the Internet. This section focuses on Cyber-Physical Systems in the area of production systems that consist of physical and cyber components that are getting into connection with each other in situation-dependent ways [Mon14]. At the same time, production systems collect information of the state of the production process, and based on these information their process is controlled and analysed. These functionalities are business and safety critical and should be designed, developed, and certified with care [IEC05]. Because various components of the system depend on each other, a secure design implies that every part of the process automation equipment is required to operate within the boundaries of its specifications [ÅAGB11]. Therefore, production system operated in a cyber-physical environment must carefully detect violations of their specification during the whole evolution process, which includes, as Monostri et al. stated, a "special emphasis on security aspects" as a major challenge. One of the most relevant behaviour of production systems is the interdisciplinary behaviour resulting from the interaction of the software with its environment [Vog+15c, Lad+13a]. Therefore, unknown, unwanted, and undocumented changes in the interdisciplinary behaviour have to be detected in the system in order to continuously operate under the specification and to ensure a secured system.

The here presented anomaly detection approach of the FYPA^{2}C project within the priority program tries to find behaviour changes as potential indicators for newly arising risks to the security during run time. In this way, the interdisciplinary behaviour is directly considered as an information source for knowledge that can affect security. The behaviour is expressed in models of the machine state of every subcomponent of a production system. These models are learned during a phase in which the system is assumed to show a well-specified and secured behaviour. To identify anomalies, actual behaviour observed in control signals of the current

system operation is compared to previously learned behaviour specifications. These anomalies may increase the vulnerability of a system because they express unknown and undocumented behaviour changes that should be checked regarding the current security requirements. The anomalies can be intended or not intended during the evolution, which means that the change resulting in the anomaly might be reverted when it is not intended or transferred to the operator for adaptation. The adaptation is left in the hand of an experienced operator, but the operator is supported by given reasonable information about anomalies.

9.6.2 Detection Model: Machine State Petri Net

The model for anomaly detection used here describes the behaviour of a machine in terms of its state changes. The state of a discrete production system is described by a set of binary sensor and actor signals. Therefore, the behaviour of the system is described by the sequence of states that are observed in the production system. One state is characterised by a set of attribute-value pairs. One attribute describes one specific signal of the state, and the value of the signal is its elemental state. Consequently, the global state of a production system is expressed by the complete set of all elemental states. These set is, in case of the targeted discrete manufacturing systems like the Pick-and-Place Unit, deterministic and locally iteratively observable. The change between one elemental state to another is defined by an event. Therefore, the production system acts like an event-based system, whose events can be observed during the operation by monitoring its event bus, that is the digital sensor and actor signals.

The analysis of production processes is generally based on an analysis of models that reflect the behaviour of the production system [LFL16]. These models are systematically used as primary development artefacts, which are often iteratively developed and evolve during the production system's life cycle [Vog+15a]. The models serve as a formal specification of the dynamic interaction within the production system, for example to describe the signal behaviour. They can be learned by using learning techniques based on the observation of states (see Chap. 6 for details about learning methods). Different model types exist to express the behaviour of a production system. One of them is Petri Nets, which are a bipartite graph. This graph expresses the system states as places and the state changes as transitions. Places and transitions form the graph's nodes, which are linked with the arcs (edges).

Since the detection models used for detecting anomalies are based on the state changes, they are henceforth called *Machine State Petri Net (MSPN)*. A MSPN is a Petri Net $\langle P, T, F, M \rangle$, where P is the set of places, T is the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the set of directed arcs between places and transitions. Furthermore, M is the set of tokens allocated to the places and describing the state of the MSPN. Each transition $t \in T$ is annotated with one or several events $e_i(t)$. Every event describes a specific binary signal change and can be observed at the

system bus. All events annotated on a single transition occur nearly at the same time. The maximum time difference between these events is described by a threshold time T_{thresh}. The set of all events annotated on transition t is referred to as $\tilde{e}(t)$. Each transition has exactly one preplace and one postplace. Accordingly, an MSPN has the same properties as that of a state graph Petri Net (see, e.g., [DA10]). One property is that the number of tokens in the Petri Net remains constant. To describe the timing of an MSPN, each transition in T is annotated with a double $\langle d_{min}, d_{max} \rangle$. The included elements are:

- d_{min}: Minimum activation duration
- d_{max}: Maximum activation duration

A transition is called activated if and only if its preplaces are marked with a token. Furthermore, a transition t can fire if and only if:

1. It is activated.
2. The activation duration of t is between its annotated d_{min} and d_{max}.
3. All annotated events $\tilde{e}\,(t)$ occurred within a timing threshold T_{thresh}.

Such signal-based models can be, for example, learned during a phase in which the system is well specified and secured (cf. learning algorithms of Chap. 6).

9.6.3 Anomaly Detection Mechanism

For anomaly detection, the behaviour of the system is compared with the behaviour of its previously modelled MSPN. To do so, each event occurring in the real system is passed to the MSPN. The occurrence time t_{occ} of the kth event is henceforth referred to as $t_{occ}\,(e_k)$. If the events contradict a valid behaviour of the MSPN, an anomaly is detected. If not, the marking of the MSPN is updated according to the incoming events. The following events are defined for MSPN:

1. There is no activated transition that has the occurred e_k event annotated:
$$\nexists t : t \text{ activated} \wedge e_k \subseteq \tilde{e}\,(t) \tag{9.1}$$
This anomaly detects changes in terms of new introduced signals, that is when a new sensor is implemented in the system.
2. There is an activated transition t, and the occurring event is annotated on it. But the time difference between the current event and the last occurred event that is not part of $\tilde{e}\,(t)$ (i.e. the last event that triggered a firing) is smaller than the annotated minimum activation duration $d_{min}\,(t)$:
$$\exists e_{k-n} \notin \tilde{e}\,(t)\,, e_k \in \tilde{e}\,(t) : (t_{occ}\,(e_k) - t_{occ}\,(e_{k-n})) > d_{min}\,(t) \tag{9.2}$$
If this anomaly occurs, it gives a hint that some behaviour of the observed system is carried out faster as given by the model.

3. The time difference between the actual time t_{now} and the last occurred event e_k is bigger than the annotated d_{max} of all activated transitions:

$$t_{now} - t_{occ}(e_k) > d_{max}(t), \forall t : activated \tag{9.3}$$

In contrast to the previous one, this anomaly indicates that some behaviour is slowed down.

4. An event occurred that is part of $\tilde{e}(t)$ of an activated transition t, but not all other events of $\tilde{e}(t)$ occurred within the given time threshold:

$$\exists e_k \in \tilde{e}(t) : (t_{now} - t_{occ}(e_k)) > T_{thresh} \tag{9.4}$$

If this anomaly occurs, events that should occur (nearly) at the same time do not show this behaviour any more.

9.6.4 Example: Using the PPU Case Study

The presented anomaly detection method has been applied on the PPU case study plant and tested on the PPU during various runs. To apply the approach, I/O events of the PLC controlling the PPU have been observed. All sensor events (PLC inputs, e.g. triggers of light barriers detecting a workpiece), as well as all actuator setpoint events (PLC outputs, e.g. command motor on or off), have been passed to the corresponding MSPN. Technically, the events have been compared with a state automata transformed from the MSPN. Figure 9.20 shows an example of an MSPN

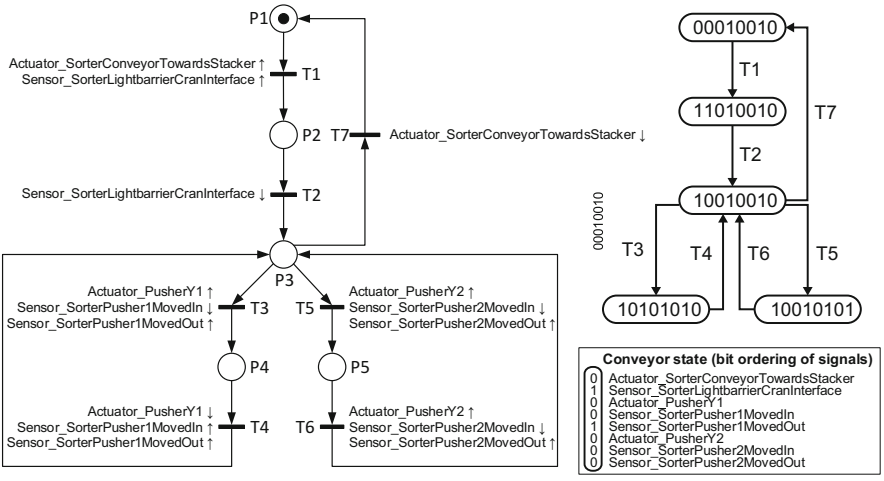

Fig. 9.20 MSPN of the PPU conveyor in Sc10 and corresponding state graph

describing the behaviour of the conveyor of the PPU in Scenario Sc10 (for scenario description, see [Vog+14b]). On the right side of Fig. 9.20, the constructed state graph is shown. For simplicity, the annotated timings are not shown here.

The evolution scenarios of the PPU have been executed consecutively from Sc1 to Sc12. MSPNs describing a specific scenario have been used for anomaly detection of the corresponding following scenario. For example, the shown MSPN from Sc10 has been used for anomaly detection in Sc11. Each evolution scenario could be detected with the anomalies defined above, as long as the evolution scenario resulted in a change of behaviour observable on the PLC I/Os. The change from Sc1 to Sc2, for example, could not be detected because it only includes an increase in the capacity of the output ramp. There is no change in the timing or order of any sensor or actuator events. Therefore, this change could not be detected. Most of the evolution scenarios of the PPU include the introduction of new sensors or actuators and could accordingly be detected by anomaly detection. In addition, further abnormal behaviour has been generated to test the anomaly detection mechanism. This includes arbitrary sensor triggering (e.g. manually triggering a light barrier), as well as stopping or slowing down workpiece transportation by removing or holding a workpiece. The anomaly detection mechanism was able to detect these anomalies during run time.

For further information regarding the application of the anomaly detection method for supporting evolution, see [LFL16, Lad+14b, Lad+15b].

9.6.5 Related Work: Finding Behaviour Anomalies

Methods for recognising behaviour changes are needed to support the evolution of systems that may change unintentionally or without model adaptation and analysis. One method includes observing the system behaviour on the software interface and comparing it with a model representing the last known behaviour. Such a method is called *anomaly detection*. The anomaly detection method described in this section is mainly oriented on fault detection known from fault diagnosis, for example [Ise06, HKW03, AA13, NF15, RLL10, AT12, LL11]. But, in contrast to fault detection, it is not assumed that a detected anomaly is faulty behaviour. A semi-automated process supporting to decide if a detected anomaly is intended or at least acceptable can be found in [Lad+14a]. It is assumed that the behaviour of the interdisciplinary system is fully discrete on its control interface, that is it can be observed in terms of input/output events of the software. This assumption holds, for example for discrete manufacturing systems [Chr06]. However, further methods also deal with continuous systems [Ise06] or hybrid systems [NF15]. The models to compare with are assumed to be time-based models having the corresponding events annotated on their transitions. The method introduced here describes a subset of the method introduced in [Lad+15a], where also a learning algorithm for automatic model generation is presented.

9.6.6 Summary

The here presented approach implements parts of the three-layered framework (Fig. 9.1) for production systems at run time. On the bottom layer, the interdisciplinary process of a production system is monitored in a non-invasive manner based on input/output signals of the production system. On the middle layer, this monitoring data are analysed regarding behaviour anomalies. Therefore, an anomaly detection method for Cyber-Physical Systems was presented that compares actual system behaviour at run-time with intended system behaviour expressed in signal-based models. If the observed behaviour contradicts the behaviour of the models, an anomaly as a potential risk is reported on a high-level model description to the top layer of the general framework. At this level, the conclusion regarding the potential risk and impact on the overall security and a suitable reaction to the anomaly can be made. The approach was evaluated on different scenarios of the PPU case study. Future work regarding anomaly detection includes to detect failures of the system based on an interdisciplinary a priori system model and finding anomalies of one production system by comparing its behaviour with the cyber-physical context of a distributed knowledge carrying network.

9.7 Conclusion

Preserving security in evolving software systems is challenging due to four main issues: First, security-relevant knowledge may only be available in a non- or semi-formal manner. Second, the impact of available knowledge to the security of the system at hand needs to be assessed. Third, as soon as the system is deemed insecure, a proper reaction to re-establishing security must be derived. Fourth, reactions may need to be performed automatically in a running software system.

In this chapter, we addressed these four challenges:

Diverse non- or semi-formal sources of security knowledge. The approach shown in Sect. 9.2 harnesses natural language processing to identify security requirements in given requirement descriptions, thus allowing to select a small portion of the overall requirements that deserve specific attention from experts. Security requirements can be captured systematically using a concept of nested ontologies that represent global and system-specific security knowledge. The approach in Sect. 9.3 can then be used to create a formal bidirectional mapping between security model information attached to architecture specification models and the program code that implements the security architecture. The approach in Sect. 9.4 extends the security knowledge by formalising and documenting contextual information from—if only—implicitly made assumptions about the security-related design decisions and from the system run time. These explicitly captured context prerequisites provide a formal relation between threats or vulnerabilities and security patterns on architectural level.

Assessing the impact of new security knowledge. The approach in Sect. 9.2 includes a concept of co-evolution rules that are triggered by specific changes to the security knowledge. The rules are designed in such a way that security weaknesses resulting from the changed knowledge can be detected and repaired. The approach in Sect. 9.3 relates security information to architecture model elements, which is also used for monitoring the application security. The approach in Sect. 9.4 uses the captured security prerequisites to assess the architectural validity of the security elements. In case of evolutionary changes to the software itself or in its context, prerequisites are used as a parameter for the architectural security analysis to check whether an attack type can then exploit a vulnerability or whether a security pattern still mitigates a specific threat. The approach in Sect. 9.5 proposes the use of run-time monitors for security properties required by the system design. Using techniques like process mining and heuristic indicators, raw monitoring data can be used to map running code to parts of the system design. The approach also proposes to compare anticipated and actual system behaviour to detect suspicious behaviour; however, it leaves open how this comparison is realised. A solution for the domain of production automation system is offered by the approach in Sect. 9.6, which expresses behaviour in learned models as a system specification, which is compared to actual system behaviour to find relevant violation at run time. These anomalies are provided to a human operator as high-level descriptions of suspicious behaviour.

Guiding architects and developers to (re)establish security. The approach in Sect. 9.2 proposes co-evolution steps to the human developer. A model-based security verification strategy is used to efficiently determine whether a particular co-evolution restores security requirements that were satisfied before the evolution step. The approach in Sect. 9.4 persists the extended knowledge in reusable extensible model-based catalogues. They are integrated into software architectures using tailored profiles and support architects in decision-making processes in case evolutionary changes impact the secure state of the system. The approach in Sect. 9.6 provides identified anomalies to a human operator as high-level descriptions of suspicious behaviour. By establishing an anomaly detection mechanism at run time, the approach guides human operators to find potential vulnerability in a complex, interdisciplinary environment in order to allow him to (re-)establish security by adapting the CPS or its environment.

Adapting the system to ensure and restore security. The approach in Sect. 9.5 makes a system run-time adaptable to realise ad hoc reactions to security incidents. By providing adequate precautions at design time, such as alternative components or fail-safe states, the system can be adapted at run time by switching between the available components or deactivating critical system parts. This way, the risk when a security incident occurs during run time is reduced.

In concert, these contributions allow to systematically capture, evaluate, and react to the evolving security knowledge. Based on these contributions, architects and developers are guided in addressing possible changes to the security knowledge and the resulting security loopholes in advance. Rather than in the ad hoc security engineering style of "fixing loopholes", security is managed in a systematic and by-design manner, thus allowing to better protect valuable assets in the face of a constantly changing environment.

9.8 Further Reading

Capturing and Leveraging Context Knowledge to Preserve Security Requirements During Design and Run Time At the time of writing, there is ongoing work on improving this part of the approach, especially focusing on the run-time phase and coupling design time and run time. Initial publications already exist [Bür+18, VKK17]. Moreover, research results are brought into the CARiSMA platform [Ahm+17].[1] Relevant results also have been produced by taking part in the ViSion project [AJ16].[2] The project is focused on privacy, which can be considered highly related to security. The contribution focuses on model-based privacy analyses of socio-technical systems.

Integrating Security Models with Program Code The integration of architecture models with code is subject to the work of Konersmann [Kon18]. It is based on the idea of embedded models by Balz [Bal11]. The tools for creating and executing translations between architecture-related program code and models are available on https://codeling.de. Konersmann et al. describe variants of this approach, for example for integrating deployment model information [KH16] or behaviour models [KG15] with program code and the use of integrated model information for locating and understanding errors [Kon14].

Anomaly Detection in Production Systems at Run Time Modelling the state of production system in signal-based Petri Nets has been presented by Ladiges et al. in [Lad+15a]. Further, anomaly detection is also defined for the material flow of a production system in [Lad+15c]. Malicious anomalies and their relation to production system is classified by Reichert et al. [Rei+17]. How to handle such anomalies within an ongoing evolution process is shown in [Lad+14a], and a fitting semi-automated decision process with a human in the loop targeting

[1] https://rgse.uni-koblenz.de/carisma/.

[2] https://cordis.europa.eu/project/rcn/194888_en.html.

anomalies of production system is presented in [Lad+14b]. Finally, Haubeck et al. [Hau+14a, HLF18] lay out how changes and their resulting anomalies can be managed within a knowledge carrying software.