Chapter 3
Addressed Challenges

Reiner Jung, Lukas Märtin, Jan Ole Johanssen, Barbara Paech, Malte Lochau, Thomas Thüm, Kurt Schneider, Matthias Tichy, and Mattias Ulbrich

Software evolution is a necessity for present-day software development and the operations of enterprise software systems and embedded systems, including production lines. Evolution is driven by changing and new requirements originating from user needs, alterations in the underlying hardware, and environmental changes, such as cloud computing for enterprise systems and modifications of production lines and processes. Current methods and processes in software system engineering are not well suited to handle these drivers of change, as knowledge about the software is predominantly stored in informal documents and not linked with other artefacts. Furthermore, most parts of a software system are only represented in the form of a source code, which carries knowledge only on what to do but not on why to do it.

We address these shortcomings with new ways and forms to describe and specify artefacts used in the development and operation of software systems. Hence, we must use and collect knowledge concerning the software system and its context at runtime and apply it at design time to enrich the evolution. We support the discovery, extraction, and handling of knowledge with novel methods and processes to foster

R. Jung (✉)
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany
e-mail: reiner.jung@email.uni-kiel.de

L. Märtin
Institute for Programming and Reactive Systems, Technische Universität at Braunschweig, Braunschweig, Germany
e-mail: l.maertin@tu-braunschweig.de

J. O. Johanssen
Technische Universität München, Institut für Informatik I1, Garching, Germany
e-mail: jan.johanssen@tum.de

B. Paech
Universität Heidelberg, Mathematikon - Institut für Informatik, Heidelberg, Germany
e-mail: paech@informatik.uni-heidelberg.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_3

evolution and make it more reliable and the software maintainable, performant, and secure. To enable these methods and processes, we provide and use new platforms and environments.

During our research, we assessed our methods and processes with two case studies based on the Common Component Modeling Example (CoCoME), resembling a software system for a supermarket chain, and the extended Pick and Place Unit (xPPU), illustrating an industrial plant automation system. CoCoME, which is introduced in Sect. 4.2, includes a fast set of evolution scenarios for the enterprise domain, like adding a webshop or including credit card payments. Similarly, the PPU case study, introduced in Sect. 4.3, provides evolution scenarios originating from industrial production plants. Our aim to incorporate knowledge in software and processes tailored for software and system evolution faces a diverse set of challenges from different perspectives. Firstly, the discovery and externalization of knowledge about requirements, the recording and representation of design decisions, and the learning from past experience in evolution form the human perspective, including that of developers, operators, and users. Secondly, performance and security induce the software quality perspective. Thirdly, round-trip engineering, testing, and co-evolution define the technical perspective. And fourthly, formal methods for evolutionary changes provide the foundation and define the formal perspective. This chapter introduces the challenges we discuss and address in this book, which were researched during the priority programme for managed software evolution:

Tacit Knowledge (Sect. 3.1) The key to evolution is an understanding of changing needs and derived requirements thereof. Unfortunately, stakeholders are often unaware of all aspects and assumptions underlying their needs and requirements. This *tacit knowledge* must be externalised in order to understand requirements and successfully evolve software systems.

Design Decisions (Sect. 3.2) To accommodate changing requirements, software engineers change the software architecture and apply different design patterns.

M. Lochau
Technische Universität Darmstadt, Fachbereich Elektrotechnik und Informationstechnik, Fachgebiet Echtzeitsysteme, Darmstadt, Germany

T. Thüm
Institute for Software Engineering and Automotive Informatics, TU Braunschweig, Brunswick, Germany

K. Schneider
Leibniz Universität Hannover, Fachgebiet Software Engineering, Hannover, Germany

M. Tichy
Institut für Softwaretechnik und Programmiersprachen, Universität Ulm, Ulm, Germany

M. Ulbrich
Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

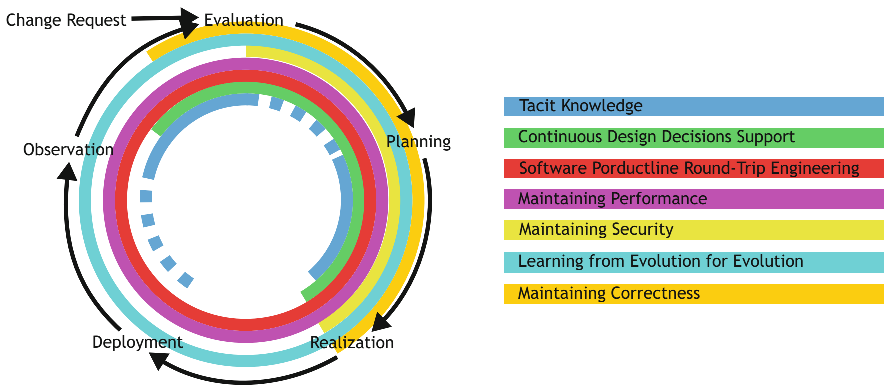

Fig. 3.1 Design for future—addressed challenges

These design decisions could conflict with decisions made in previous iterations of the evolution process, eroding the architecture and harming evolvability. Therefore, it is necessary to support the documentation of and access to design decisions.

Software Product Line Round-Trip Engineering (Sect. 3.3) As depicted in Fig. 3.1, software evolution is a circular process where introducing changes occurs often. Today, software systems are not only subject to reoccurring changes; they also exist in different variants. This is especially the case in embedded systems. Providing a consistent view on versions and variants of product lines introduces new challenges to software evolution.

Maintaining Performance (Sect. 3.4) Being able to predict and forecast performance is necessary for software systems to ensure a timely execution and control over resources. Feature sets from software product lines can result in large numbers of variants, which cannot be evaluated for performance in a timely manner. Furthermore, runtime measurements address only one version of one variant. Both aspects are central challenges in maintaining performance.

Maintaining Security (Sect. 3.5) Keeping a software system secure is a great challenge by its own. It is affected not only by changing requirements within the software system but also by its changing environment. These changes are covered in non-formal documents. Supporting security experts and developers in deriving formal information from non-formal documents and supporting security evaluation throughout evolution are the challenges we motivate in this section.

Learning from Evolution for Evolution (Sect. 3.6) The previous challenges address certain challenges within the evolution cycle. However, we also need to learn and transfer knowledge from one evolution step to another and from one project to another to grow our knowledge on software evolution and improve our processes and software quality. Therefore, we face the challenges of how to process semantically rich changes in past evolution steps and develop methods to understand and exploit this knowledge.

Maintaining Correctness (Sect. 3.7) Software evolution may erode functionality and cause unwanted behaviour alterations in software. While non-formal processes and methods help to mitigate unwanted changes, they cannot detect and correct them. Therefore, we need formal approaches using models to verify software systems, be able to test them for changes and to know how to distinguish wanted from unwanted changes.

3.1 Tacit Knowledge

Long-living software systems face challenges during requirements identification and update due to various reasons. First of all, software systems and the requirements that describe their functional behaviour and non-functional performance change over time. The technical development and the availability of new software and hardware components affect and change existing requirements or even make them obsolete. On top of this, a substantial part of the relevant requirements for software systems remains *tacit*. This means that important knowledge carried by requirement analysts, software users, or other stakeholders remains in their minds. In general, *tacit knowledge* can be described as knowledge that is internalised by a person while its active verbalisation, that is the externalisation of this knowledge, is difficult [PS09].

In contrast to the goal of a complete representation of a software system [Dav93], this results in an incomplete set of requirements. As a consequence, the associated software system remains incomplete as well, which is expressed in different facets. First, software systems are exposed to any type of intrusion. They are in particular vulnerable to attacks in case of a lack of security-related requirements. Second, users of a software system are presented with an unsatisfying set of functionalities that does not match their needs.

Stakeholders of a software system, such as the requirements analyst or the software user, might not be aware of an urgent demand for action or of the associated knowledge that would help to understand a situation in question. Thus, they are unable to verbalise those tacit requirements. This is the reason why an automatic identification and extraction of this tacit knowledge represents an important source of knowledge for the development of long-living and continuously evolving systems.

We envision that tacit knowledge is particularly exposed to being automatically captured, processed, and externalised during the *design time* and *runtime* of a software system: On the one hand, during the design time of a software system, requirements are elicited and described using natural language. In doing so, the use of certain words might indicate implications for the functionality that they describe. For instance, the way a functional requirement is described can pose requirements towards non-functional aspects. On the other hand, during the runtime of a software system, users unconsciously provide insights into the way they interact with the system. Ignoring certain functionalities of a software system or repetitively applying the same usage patterns might hint towards a particular functional requirement that the developers or requirement analysts were previously not aware of.

We want to focus on *tacit knowledge* as an instance of knowledge that is preserved in a software system or its underlying design. Describing tacit knowledge and its building blocks requires defined models. This demand is manifested in one of the SPP1593 themes by calling out for customised meta-models that enable the ongoing development of software systems. We strive for a basic representation of tacit knowledge in the form of ontologies and taxonomies, which can be utilised to detect and describe tacit knowledge. Furthermore, tacit knowledge naturally emerges in an unstructured format produced by heterogeneous instances and actors. Its building blocks remain incomplete and potentially irrelevant, only until they are mapped with existing requirements that relate to the same entity. Addressing the extraction of tacit knowledge requires a platform that can be deployed in a continuously changing environment to visualise building blocks of tacit knowledge. This is one of the SPP1593 themes, that is establishing platforms and environments that enable access to design- and runtime information when it is needed.

By extracting tacit knowledge from both design- and runtime observations, we aim for the following goals. First, we want to enable the creation of software releases that match the requirements of both customers and users and their expectations. Second, we intend to improve and maintain the quality of development for long-living systems through the co-evolution of adequate non-functional mitigation activities. Third, we aim for increasing the software system's usability and an adaption towards the needs of users.

Understanding tacit knowledge poses challenges in their identification due to multiple reasons. First, we expect different kinds of tacit knowledge that can potentially arise during software evolution. Second, the availability of various sources of tacit knowledge plays an important role. Eventually, when it comes to the extraction of tacit knowledge, we see challenges in reducing the mismatch between developers' and security experts' mental model, as well as reducing the mismatch between developers' and users' mental model. Aside from the identification and extraction of tacit knowledge, we face challenges in working with tacit knowledge and its explicit counterpart. In particular, the detection of deviations between specified and derived security knowledge or deviations between expected and observed user behaviour demand attention in their analysis. We summarise this collection of challenges under the following two main challenges and address them in Chap. 5.

Challenge 1 How to identify and extract tacit knowledge to reduce the mismatch between stakeholders' mental models during software evolution?

Challenge 2 How to detect deviations between explicitly elicited requirements and implicitly derived requirements?

3.2 Design Decisions

Continuous Software Engineering (CSE) is a software engineering process in which developers continuously change the software while keeping it in a releasable state [KB17]. CSE means to develop, release, and learn from software in very short rapid cycles [Bos14]. It incorporates agile practices and involves activities such as continuous integration, delivery, and deployment [SAZ17, Joh+18b]. The emergence of CSE is driven by a growing need for flexibility and rapid adaption in the current software environment [FS17]. Thus, CSE provides many techniques for a continuous change. This can also be exploited for continuous design decision support.

Software developers and architects continuously make design decisions while they develop software. When they evolve software, it is important for them to reflect and build on former decisions. Otherwise, they might make inconsistent decisions and are likely to contribute to the erosion of the software architecture or introduce other quality problems. Reflecting on former decisions is particularly important for long-living software systems where many decisions build on one another. Documenting design decisions is important since many different developers are involved at different times and cannot communicate directly.

Design decisions can be made in either a rational or a naturalistic way. Rational decision-making means that developers weigh alternatives and arguments, whereas naturalistic decision-making means that they reuse past experiences to solve a decision problem [ZCM07]. It is often assumed that decision-making in software design is a deterministic and rational process [Fal+11] since software development is an engineering activity. However, this is not so in practice as, for example, Hesse et al. empirically show that naturalistic decision-making is dominant over rational decision-making in the Firefox open-source project [Hes+16]. In naturalistic decision-making, developers do not consider all alternatives and arguments. This is risky as humans tend to overlook what is missing and are subject to cognitive biases [Raz+16]. Thus, developers might anchor on those solutions that first come to mind, omitting more relevant alternative solutions. If the arguments for the decision are not documented, other developers might not understand the decision or might not be convinced. Thus, support for rational decision-making is important. Rational decision-making requires the management of decision knowledge.

Design decision knowledge is the knowledge about design decisions, the problems they address, solution approaches and their alternatives, their context, and their justifications (also called rationale). Decision knowledge vaporizes quickly; that is, if developers do not document decisions immediately, the design decisions are never documented and thus not available later [JB05]. Decisions are often discussed informally and captured partly and distributed: for example, in code, issue comments [Hes+16], commit messages, pull requests [Bru+14], chat messages [Alk+17a, Alk+17b], wikis, and emails; this knowledge is difficult to access later. Thus, developers need support to capture decision knowledge or evolve it from naturalistic decisions and to access it efficiently.

Our long-term vision is an *on-demand decision documentation* as part of the *on-demand developer documentation* suggested by Robillard et al. [Rob+17]. We envision that developers continuously capture and reflect decision knowledge during CSE. Benefits of a continuous capture and reflection on decision knowledge are an improved decision-making process through explicit criteria, the prevention of knowledge vaporization, and consistent future changes.

Our goal is to support developers in this continuous capture and reflection, in particular by performing rational decision-making. The following three developer tasks should be lightweight, that is they should require as little effort as possible: *rational decision-making*, *documentation of decision knowledge*, and its *exploitation*.

There are two major challenges for this support: intrusiveness and inconsistency. It is a challenge to minimize the intrusiveness of a continuous design decision support and to document and maintain decision knowledge consistent with the other artefacts and with former decision knowledge. We summarise and express these challenges under the following two paragraphs and provide solutions in Chap. 6.

Challenge 3 How to integrate rational design decision-making, documentation, and exploitation in software engineering practices? Tool support to manage decision knowledge can be characterized by its intrusiveness in the software development process [Dut+06]. Tools that fit into the development context are less *intrusive* and will more likely be used [KCD09]. Such tools do not require additional effort (e.g. for installing or starting a separate tool) and are thus also lightweight. Rational decision-making, documentation of decision knowledge, and its exploitation should be non-intrusive in the context of the CSE process.

Challenge 4 How to ensure consistency between decision knowledge and artefacts? Consistency means that design decisions are documented and linked to and realized in the artefacts they relate to. To exploit decision knowledge, it is important that the design decisions are *consistent* with former design decisions and with the artefacts, for example with the requirements, architectural software design, and code.

3.3 Software Product Line Round-Trip Engineering

Modern software systems tend to become more and more long living and, therefore, have undergone continuous evolution to ever new versions in order to meet constantly changing requirements. For instance, the initial version of the PPU case study only comprises a stack with multiple slides for sorting different work pieces according to their types, as well as a crane and a stamp. Later on, the PPU undergoes several evolution scenarios in order to adapt to changing requirements and platforms (e.g. the ramp is later replaced by a standard ramp to support application scenarios without sorting). As a consequence, all PPU (software) artefacts (potentially) affected by those changes have to be adapted to support the new versions.

In addition, modern software systems are highly configurable, thus comprising many different variants being custom-tailored to specific needs. For instance, the modular architecture of the PPU supports many different variants in order to adapt to different environments, platforms, and customers' requirements. Such a collection of similar yet well-distinguished variants of the same core product is frequently called a product family. Software product line engineering (SPLE) is an established methodology for handling the additional complexity caused by the increasing variability of modern (software) systems by means of variability-aware engineering and quality-assurance techniques. To this end, SPLE aims at systematically exploiting knowledge about commonality and variability among all kinds of engineering artefacts (e.g. design and test models, implementation code, and test cases) in a family of similar products.

Finally, modern software systems are, in most cases, an integral part of larger socio-technical systems, thus requiring accurate quality assurance to reduce the risk of fatal errors. Model-based testing is a widely used black-box testing technique for automated quality assurance, where a test model serves as a behavioural specification of the expected behaviour of the (potentially inaccessible) implementation code to be tested. For instance, the PPU behaviour is specified using statechart models, which can be used to automatically derive test cases covering a predefined set of test goals for systematically investigating the different runs of the PPU.

Although very promising concepts and tools exist in recent research for tackling all those three kinds of engineering challenges separately, a comprehensive approach integrating the different solutions into one conceptual framework is still an open issue. In particular, a corresponding round-trip engineering methodology has ensured an effective and efficient quality assurance of evolving, variant-rich software systems in a systematic and consistency-preserving way. To this end, a structured process for artefact co-evolution is required for all possible kinds of evolution scenarios of engineering- and quality-assurance artefacts involved.

Our vision is to define a comprehensive methodology for round-trip engineering and model-based testing of evolving, variant-rich software systems. To realize this vision, we first have to extract and integrate variant/version information in an automated way from evolving model-based product-line engineering and quality-assurance artefacts. Based on this additional information, we pursue to define criteria for detecting and avoiding inconsistencies between those different design-, implementation- and quality-analysis artefacts.

To achieve our goals, we have to address several challenges with respect to the three guiding themes of the SPP, namely *Knowledge Carrying Software*, *Methods and Processes*, and *Platforms and Environments for Evolution*. In particular, we address two essential challenges.

Challenge 5 How to automatically extract and integrate variant/version information in model-based SPL engineering and quality assurance?

Challenge 6 How to avoid inconsistencies in different design-, implementation- and quality-analysis artefacts?

By addressing these research questions, we contribute to the different guiding themes of the SPP in various ways.

3.4 Maintaining Performance

Performance is a key quality characteristic of software systems, describing its properties with respect to timeliness and resource usage. Typical performance measures include response times and throughput of a software system. Insufficient performance has a negative impact on the service quality of software systems, which in turn affect key business indicators such as revenue. Performance issues in enterprise applications and web services can limit employee productivity and cause customers to switch to other services. In production systems, insufficient performance can limit production output and may reduce the quality of products, harm employees, and damage facilities and products. Therefore, performance needs to be addressed throughout the entire software life cycle from development to operations via suitable performance analysis methods, techniques, and tools.

Researchers have developed a wide range of performance analysis methods in the past, which allow to assess single versions and variants of a software product. However, today's software is often highly configurable and evolves frequently. Different *versions* replace each other over time, while multiple *variants* co-exist at the same time. Especially in the context of product lines, which play an important role in production systems and handheld devices, variants can be numerous as all potential feature combinations must be evaluated separately. For example, different variants of the Pick-and-Place Unit (PPU) can be configured by choosing from the defined relationship of mandatory, optional, and alternative features, such as alternative cranes and stamps, as well as a set of supported workpieces. Furthermore, modern software is often developed with agile development methods and processes that create new versions for every feature, resulting in a high frequency of changes. For example, Common Component Modeling Example (CoCoME) includes a definition of design and runtime evolution scenarios such as the addition of new features or platform migrations based on changing requirements and runtime reconfigurations. Therefore, the number of versions and variants, as well as the different and evolving types of artefacts (models, code, measurements, etc.) pose challenges on performance analysis strategies.

Our vision is to address the performance of variants and versions in an efficient way throughout the software life cycle. This supports software engineers and administrators as they can predict software performance at design time and evaluate it at runtime.

Performance is influenced by design, configuration, implementation and deployment. Therefore, performance analysis must be part of the design process. At runtime, the effects of these influence factors become apparent and allow to further understand their performance impact. We envision to use knowledge derived from runtime observations to enrich and improve performance assessments.

Performance evaluation of potential variants is excessively time consuming, for example the PPU feature tree allows for X variants, which limits the ability to apply performance prediction approaches. However, performance is a key element also in software product lines. Our vision is to reduce the necessary effort through a smart selection of variants, modularization, reuse and knowledge gained during runtime of previous versions of the variants.

Our goal is to provide continuous support for addressing performance concerns for versions and variants via respective performance analysis strategies. They must be able to provide answers to performance questions by engineers in a timely manner.

Variants can comprise minor deviation from each other or result in very different software systems. Each difference in the architecture can influence the performance of a component, as the communication changes between components. Unfortunately, to test and evaluate every potential variant is time consuming and impede development due to long evaluation cycles. In Chap. 8, we want to address this challenge.

Challenge 7 How to efficiently analyse the performance of all variants of a software system?

While variants are different software assemblies that exist in parallel, versions reflect differences over time as the software evolves. During the evolution, engineers need to address performance either due to current performance issues that they have to solve or in order to fulfil performance requirements in the future. This leads us to the challenge.

Challenge 8 How to exploit evolving artefacts for the performance analyses of software throughout its life cycle?

3.5 Maintaining Security

The security of software systems is a highly important quality aspect. This is motivated by the fact that today an increasing amount of personal data are handled by software. A vast amount of people not affiliated with security or inner workings of information technology (IT) is trusting that the data are processed securely.

In detail, in many cases this means compliance with the most common security requirements like integrity, authenticity, availability, and privacy.

Moreover, an increasing amount of systems exist that tend to collect data of a whole human life span and/or collect data throughout the day. For example, cloud storage services like Dropbox can store not only a theoretically unlimited amount of data but also an unlimited amount of revisions. Social networks like Facebook are able to record a whole life. Smartphones or smartwatches are with us the whole day and continuously mine data through quite a few sensors and also pre-analyse data like determine a person's position by combining GPS data, names of available Wi-Fi spots, and assigned IP addresses.

On top of that, a growing number of information processing, mostly Internet-connected systems, is pervading our daily lives, like most *smart* or *IoT* devices, such as smart light bulbs, smart light switches, or simply smart speakers/assistants like Google Home or Amazon Echo. There are hardly any instances where these systems do not rely on servers or services that are Internet based. In a world of interconnected systems, your system is also connected to an unpredictable number of attackers.

There actually is a big number of systems that were developed or deployed a long time ago, and there will be even more in the future. As a result, data that pervades all of our lives is in the hands of an opaque mesh of systems connected through the Internet. And even if one person wants to avoid her data being stored in such services, it is a desperate situation when her friends store, for example, photos or other personal data in their cloud services.

Today we must experience that current systems fail to keep their promise. Hacks, vulnerabilities, and data breaches had already happened in a magnitude that has never been seen before. Examples are Heartbleed (OpenSSL), Krack (WPA2), 68 million password hack (Dropbox), PlayStation network hack (77 million customer data), and the CPU bugs leading to the Spectre/Meltdown attacks that affect nearly every processor in end-user systems rolled out since 1995.

The vision is to incorporate security relevant knowledge accompanying the ordinary system design. Ordinary system development runs through different levels of abstractions, and so there are possibilities for wrong decisions at early development stages. Especially caused by the fact that most systems tend to be interconnected and new attacks come up rapidly, not only system development should be accompanied in early stages like design decisions but also the system's context needs to be touched, like current security knowledge and knowledge about attacks and mitigations.

To achieve this goal, knowledge needs to be gathered (semi-)automatically. The knowledge must include new attacks (or new attack vectors), mitigations, precautions, and best practices relevant for a given system and domain. Even when a *secure system design* has been obtained, the runtime behaviour of the system is also important. On the one hand, there is a number of security requirements that cannot be checked fully at design time, at least when they rely on runtime data, for example consider a deployment context or access-control-related user data. On the other hand, as argued before, there might be a high risk that a system with an initially secure design is attacked during runtime using an unforeseen attack. In this case, one might want to detect this via anomaly detection techniques. At least, one might want to react at runtime by adapting a system. To reach this goal, continuous monitoring of the system seems inevitable. The result shall be detection of unwanted behaviour regarding the security design and also adapting the system to mitigate threats.

Challenge 9 How can security knowledge, available via diverse non-formal sources, be incorporated and utilized for a long-living system design?

Challenge 10 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 runtime?

3.6 Learning from Evolution for Evolution

Learning is the process of changing one's behaviour through knowledge acquisition. New knowledge is generated during both the *design and construction phase* and the *operation phase* of a long-living software system. Making this knowledge accessible in *Knowledge Carrying Software* is one of the guiding themes of the priority programme. Knowledge can be learned and applied through the whole evolution cycle. But much knowledge is either implicit and never documented or missing completely.

There are multiple reasons for this; for example, tight time and cost restrictions can prevent software engineers from creating documentation in the first place. Bad requirements engineering practices can also lead to this outcome. Furthermore, creating formal documentation, for example in the form of models, is a complex task that might be perceived as tedious and cumbersome. Creating this kind of formal documentation also requires a high level of expertise. Documentation might also be wrong or become out of date. Oftentimes tests are also used to ensure correctness of software and to document it, but for practical reasons tests cannot cover the entire behaviour of a system. Thus, knowledge is often not documented and scarcely available.

Missing knowledge about the system and its environment greatly hampers the evolution of long-living software systems. Reasons for this are that detailed knowledge of a software system is an essential prerequisite for an effective software evolution and for ensuring the correctness of a software system.

Our vision is to semi-automate the learning of knowledge and its application for the evolution of model-based long-living software systems. We can then use this knowledge to support software engineers who would not otherwise have access to this knowledge. This support comes in the form of ensuring correctness and recommendations about future evolutions, as well as assessing the effort required for changes. For this we need to identify evolutions in the past and present, assess their impact on the systems, and use the gained knowledge to derive future evolutions, for the same system or different systems. Our automatically extracted knowledge will enable the development off *Knowledge Carrying Software*. This extracted knowledge can be about past evolutions or about the current behaviour of the system, for example in the form of automatically learned behavioural models. Our results will be implemented in software tools that can be used as *Platforms and Environments* for evolution. One example is the SiLift tool (cf. Sect. 10.1.1) for identifying historical evolution steps, which is the foundation for other software tools in this chapter.

Our goal is to automatically create knowledge about a system or its past evolutions. This knowledge shall then be used to support future evolutions of that system or similar systems. Knowledge about past evolutions is contained in artefacts stored in software repositories. These past evolutions need to be extracted and then processed so that the engineer can readily use this knowledge. Similarly, knowledge about the current system might be derived from the actual running system, for example to create models about a system's functional or non-functional behaviour. The derived knowledge shall then be utilized by recommending, selecting, or deriving evolutions of the system or similar systems such that those systems correctly realize changed functional requirements or improve their non-functional behaviour due to the evolution.

Realizing those goals poses several challenges. We group these challenges into those concerning the analysis of *past* and *future* evolutions.

Challenge 11 How to identify and process semantically rich changes from past software evolutions?

Past changes in model-based systems come in the form of models under version control (e.g. git). These models and their versions can be numerous and describe the system under different viewpoints. However, simple graph differences on the abstract syntax level are too fine-grained and lack the semantics of changes on higher level representations. Consequently, the first challenge is to identify past software evolutions by computing and grouping the corresponding model differences and give those evolutions semantics on the modelling language level.

Those semantically rich software evolutions can then be used to drive future software evolutions—leading us to the second challenge.

Challenge 12 How to exploit past software evolutions to improve future software evolutions?

This second challenge has multiple variants. One variant is related to the co-evolution of different viewpoint models of the system. Here, a system can exploit past evolutions to recommend co-evolutions of viewpoints if a user changes a single viewpoint.

Another variant is to use knowledge about past evolutions to establish a knowledge-carrying network. This network could exchange experiences of past evolutions between similar systems characterized by their behaviour and context and use this knowledge to support the engineer in evolving systems. A final variant addresses maintainability of long-living systems. Here, knowledge about past evolutions could be used to estimate the maintainability of information systems and automated production systems.

3.7 Maintaining Correctness

Evolution is usually driven by the need to change a particular part of the system, for example in order to repair a malfunction or to add or improve features. The challenge is to ensure that other aspects of the system that are not targeted by the change are not modified. Unfortunately, system evolution might invalidate properties a system had achieved before and is a threat to the system's safety, security, performance, maintainability, and other system properties. In particular, evolution may threaten the trust that an earlier version of the system has gained in earlier testing phases or by formal verification. Also, if a system has run flawlessly for a decade, this generates some amount of (informal) trust in the correctness of the system.

The goal of *formal verification* within the context of software evolution is to prove that system properties are not lost due to introduced changes. The properties to be maintained can either be formulated explicitly as formal specification (or modelling) artefacts, or they can be present implicitly in form of the code that drives the existing system.

Knowledge about the system is present both in specification artefacts and in the code of the program run on the system. If formal verification is able to prove that a new revision also has these explicit or implicit properties in the code, then verification serves as a preservation means for the trust into systems - and management of knowledge. The task for the formal analysis of a system evolution step can be partitioned into two disjoint sub tasks:

1. *Analysis of system aspects that are intended to be retained.*
   This analysis is used to establish that defined parts of the system behave as before the change in defined cases. It transfers all properties of the retained part of the system behaviour onto the new revision without requiring to explicitly state them.
2. *Analysis of system aspects that are intended to be changed.*
   Almost every evolution step (if it is not a pure software refactoring) contains an intentional change for some part of the observable behaviour. The above analysis does not help in this case; we cannot (solely) rely on the old revision as specification for the intended behaviour after the evolution step, but we need to specify the intended properties of the system explicitly.

Both aspects of formal verification for evolving systems are challenging in themselves, and it is interesting to observe how they can accompany an evolutionary process spanning over evolutionary steps.

It is important to observe that for the analysis of automated production systems, these cannot be reduced to their software alone—instead, it is imperative that models of the contextual hardware are taken into consideration as well: Interdisciplinary modelling is important to make the context and environment part of the verified system. In Chap. 11, we focus on the preservation of safety properties throughout the evolution of automated production systems. Similar techniques for proving the preservation of properties are in principle also thinkable for security, performance, or other properties—but have not been investigated within this programme. The aspects of embedding formal evolution analyses into a user-friendly development process is outlined in Sect. 10.2.

We envision a software evolution process that is naturally and fully accompanied by (automatic) formal verification steps, thus guaranteeing that desired system properties are always maintained during evolution. The engineers responsible for designing and implementing an evolution step will be provided with expressive and usable specification languages with which they can specify which parts of the systems should remain untouched and which parts should expose a different behaviour. These specification techniques allow the engineer to specify desired behaviour both incrementally (as differences to behaviour of the earlier version) and interdisciplinary (concerning not only the software but also the context and the hardware). While a formal verification is the more far-reaching goal, the obtained specification artefacts can also serve as oracles for testing as a more conventional technique of verification.

To realise this vision, appropriate specification languages and techniques and according verification techniques are required that enable the application of formal verification within the evolutionary process. The first goal is therefore to provide the right specification and verification techniques for a formal verification for evolution. The specification techniques must allow for a multi-disciplinary approach going beyond the software and comprising also the hardware and must take special needs of the applications into account. They must also operate incrementally. The corresponding automatic verification techniques must be powerful enough to discharge typical verification conditions within reasonable time and fully automatically.

The two research questions for this research field arise naturally from the partitioning of the analysis tasks described earlier in this section. They correspond to the duality of the nature of an evolution step requiring that some chosen system properties are retained while others may change (in a chosen fashion).

Challenge 13 How to model, specify, and verify that a system retains desired behaviour during evolution?

Challenge 14 How to model, specify, and verify intentionally changed behaviour during system evolution?