Chapter 14
Future Research

Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin

The overall result of the priority program, as discussed in the previous chapter, shows that the SPP 1593 projects covered a wide range of aspects regarding evolution. However, the entire field is still much broader than the usual set of projects that an SPP can cover in a 6-year period. Deep results have been achieved, and various central ideas have been put forward, as discussed in the previous chapters. In many cases, these results are related to "the system" or "family of systems" and various additional artefacts, including a special sort of meta information: the specification(s) of the system. Such specifications define the functions and qualities of the system like performance and security but also internal qualities like maintainability and evolvability.

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

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

W. Hasselbring
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany
e-mail: hasselbring@email.uni-kiel.de

B. Vogel-Heuser
Technische Universität München, Institute of Automation and Information Systems, Garching, Germany
e-mail: vogel-heuser@tum.de

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

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_14

14.1 General Aspects: Interaction of Multiple Qualities

As has been put forward, one needs to consider—ideally all—software qualities during the entire lifetime of a software system in an integrated way to prevent their degradation during the software system's evolution. In addition, a new dimension has been used: the deployment and operation of the system. Thus, in principle, better informed design and implementation decisions during the development can be done using configuration and runtime information. In order to propose a grand view, the SPP's specific notion of *knowledge-carrying software* has been put forward. Still this notion of *knowledge-carrying code* is worthwhile to consider in future research and development methods. The basic idea was to create a lasting relationship between the code of the system and its specifications and other related documentation. This will be the basic mechanism to build a tightly coupled network of the code and specifications, which need to be maintained by the development actions during evolution.

The challenge here is to define and manage the various qualities and their mutual interdependencies. For example, performance and security interact in many cases in a negative way—which, of course, is not the desired outcome. This means that a naïve implementation of secure communication in distributed system entails complex mathematical operations to encode and decode the messages between the components of the distributed system. Usually, by performing these frequently, the related operations will significantly "steal" execution cycles and as a consequence degrade the performance for other functions of the system. Thus, by introducing security, additional computing power is needed to run the entire system. This is especially an issue in embedded systems that normally use SoCs (System on a Chip) with limited computing power for economic reasons. Even in such cases, a clever software architecture or cheap encoding/decoding coprocessor can be instrumental to help out in such a dilemma. In case of performance and reliability, the use of redundancy can provide a suitable solution to enhance both qualities in many situations.

This is one aspect. In addition and more important in the context here, the evolution of a system will change the (source)code of the system. Having security and performance as quality goals, the complexity of the system grows. For example, the system has a cache for storing and retrieving intermediate results for faster access. Such cache structures pose challenges for security, for example Meltdown [Lip+18] and Spectre [Koc+19], which is of course not directly related to this area but should be taken as an indicator for the increase of complexity when auxiliary structures like caches are introduced. A change in functionality will possibly impact the system's structure in more than one place since the cache (not only the change in the function) has to be taken into consideration as well. This can easily be done in a non-strict and insufficient way, which can degrade one or both of the two qualities.

This is only a simple example and should be taken as a hint that the interdependencies between qualities, when viewed from the evolution perspective, will pose challenges. This is probably even more relevant if one wants to formulate general

rules applicable at the whole range of software qualities and sorts of software systems. Thus, a main issue here is not only to address these interdependencies but also to consider the possible influences of the change introduced in the evolution steps. The range of qualities and aspects as a whole needs more attention. This is certainly easier to say than to pursue actually. Managing a specific individual pair or subsets of qualities and their related multilateral interaction, especially in view of evolution, is a challenge in itself. In addition, the overall management of all relevant qualities in software development is beyond the range of the usual single DFG research project.

14.2 Specific Areas

In addition to this large area of further and future research, a number of topics have been hinted at in the previous section on lessons learned, which need further elaboration.

14.2.1 The Use and Extension of NLP Techniques

One specific topic is the use of natural language processing/understanding in the software development process in order to support the detection of issues in the documentation and communication between stakeholders, including developers. The problem of understanding real-world utterances and generating useful hints from those natural language fragments is still an unsolved problem at the general level. However, the area we address here has a number of characteristics, which might improve the situation in a way that allows for better results in understanding natural language comments and statements contained in the code, specifications, documentation, and dedicated communication between stakeholders. For example, integrated development environments and dedicated social media, like Slack,[1] provide such communication channels, which help the stakeholders to keep track. The additional gain as compared to the general problem is due to the at least semi-structured documentation in specifications and code, which reduces the ambiguity in natural language. Given the known model world of the software system in question, it helps to disambiguate natural language statements more easily, and the missing explicit knowledge (tacit knowledge) might be inferred in an easier way than in the general case. In summary, the combination of natural language understanding in the context of models and code (knowledge carrying code) is one area to explore.

It should be noted that this kind of analysis causes concerns in the area of the involved persons' privacy. This dimension is seen differently in the various

[1] https://slack.com/.

geographic and political areas of our planet but in any case introduces a possible bias in communication, which might influence the communication negatively concerning the degree of interesting findings related to the software systems' properties and, of course, not related to the involved persons representing the various stakeholders' views. In the same way, sentiment analysis of the spoken words can be helpful or also be detrimental.

14.2.2 The Organization of the Whole Range of Software Qualities

This area sketched above can be seen as a special case of the situation wherein a software development code, specifications, models, and natural language descriptions have to be integrated. When assessing the integrity of the various specifications/presentations, it is useful to derive information which helps to pursue further development steps. This is especially the case when inconsistencies between those presentations/specifications can be spotted. As has been discussed in the previous chapter, it is desirable to have more than one representation of the system at different levels of abstraction and detail. Given this is done in a structured and systematic way, better insight and consequently better informed development decisions can be made. Such an approach is often referred to as view-based software engineering and is addressed in the previous chapter to coordinate these views at the meta level; that is, the general level about type of actions, type of changes, etc. is still quite difficult.

There are numerous works available regarding views in Software Engineering [MG10]. The systematic development of structures supporting evolution at the meta/type level needs to be addressed in an actual useful way for stakeholders and specifically developers. This not only is a technical discussion but also entails empirical work. This has been addressed in the SPP, and some findings are available. More work needs to be done to obtain valid data, how useful such structures—including at the meta level—should be created to foster better understanding and design actions/design decisions.

This is also problematic since the usual independent multi-factor studies using test persons of the necessary skill levels and the size of studies in terms of entire projects and the involved software system itself is actually impossible in practice. A new approach needs to be developed to provide insight on how empirical results can be obtained here in the situation of large projects.

A part of this aspect is a specific *diversity* aspect of software development: *variants* and *versions*. It is obvious that in many cases, not the *single-purpose* software system is developed, but various variants, for example depending on context and/or configuration, are developed and deployed in a given operating environment. In software product line development, these variants are usually created by specifying a *family of software systems*, and a given instance is created based on a set of particular choices.

In addition, over time evolution of a software system or a family of software systems not only removes errors and inconsistencies but also adds new functionality and features. These additions usually create *new versions* of the software system/family of systems. The research (c.f. previous chapter) revealed that it is still conceptually not very well defined how these various strands of development can be related to each other and as a consequence also how the management of these variants and version can be improved as well.

It could be the case that a proper organization of the views mentioned above will possibly yield a better understanding. Hence, this means that structuring the development work entails a better understanding of the purposes of variants, product lines and versions. When done at the meta level, useful general insight can be generated, which extends the Software Engineering knowledge in a very important area.

14.2.3 Agility in Software Development

In industrial practice, evolution appears to be the norm, *and* the speed of a given development cycle from perceiving a specific need/inconsistency between expectation and actual behaviour has increased in a dramatic way during the last 10 years. The agile development agenda has addressed the inevitable learning curve during a specific software development endeavour in a way that sometimes some qualities of a given software system are not addressed explicitly any more. One could view, for example, the use of AB testing as an implicit elicitation of requirements: various parts of the user community get different versions of the software system or service to use, and the actual usage and resulting use effects (e.g. user satisfaction but also generated traffic, monetary value, etc.) determine which of the versions will be the base for the next evolutionary development steps.

This implies a frequent generation and deployment of a new version or time frames of the system. It is not uncommon that any new deployment of a system is done in time frame in the order of magnitude of 10 s while a given to be tested new feature takes from inception to deployment a time frame of about 1 or 2 weeks. Given such circumstances, the careful elicitation, documentation, and analysis of requirements; the design of solution elements in the form of a changed software architecture; and the subsequent high-performance implementation and deployment are not done explicitly in every aspect any more.

This specific way of working and also the related infrastructure of tools are referred to as *continuous delivery/continuous integration*. In general, the approach to automate the process to integrate all artefacts, automate the acceptance test, and deploy the solution—if passing all tests—is good for all software development processes. Of course, these are badly needed for continuous delivery. However, not all software development endeavours are of this kind. Embedded systems in high-integrity application areas, for example health care, pharmaceuticals, transportation,

critical infrastructures often require certification and approval processes, which have to be signed off by human experts in the end.

Much of the research presented here and in other documentation and publications of the SPP's projects rely on explicitly documented qualities, design, design decisions, etc. Thus, one can see in continuous delivery development profiles a mismatch between requirements from practice and the assumptions of research in Software Engineering. How can developers in such situations and beyond produce good quality solutions in terms of robust and correct software systems/families of software systems?

The answer to this question is to scale up the involved persons' productivity by using automated tools. In many cases, an initial investment into (non-executable) specifications/description of software properties and related qualities seems to be a good one. The problem comes when the system evolves and grows substantially. Many of the more advanced (formal) methods and tools are not really scalable in the sense that truly incremental approaches are available. Usually formally verifying a software system of realistic industrial size is not performed within 10 s and—if feasible at all—will last days if not weeks. An incremental approach which starts small and grows with system's growth only in small steps according to changes/additions is still not a solved task. The main problem here is that a small change to a software system is not—in a mathematical sense—continuous and has only a small effect. Advancements in *incremental verification/regression verification* would also be beneficial for other fields in computer science as well.

As a slightly smaller version of the incremental verification procedures sketched above, better, automated *detection of inconsistencies* might be also helpful. Not so much the ubiquitous stack trace or lengthy/indigestible logfiles or memory dumps but specific identification of erroneous behaviour and related potential causes will be helpful. This has to be incremental and also related to the many software qualities. For example, a security analysis could detect whether the change in the software introduces the potential of a new attack vector which wasn't there before the change was applied.

For both areas, verification and inconsistency management of modular approaches would be beneficial for increasing their respective scalability in order to tackle real-sized problems in the industry.

The application of the approaches to the case studies in this SPP shows the applicability of the various approaches in somewhat realistic settings. This means that for the two application domains, the various attempts and approaches showed their usefulness in these settings. However, this also showed that there is more need for connecting models in the respective application domain and the related code in, for example, microcontrollers and PLCs (xPPU) in such a way that the changes in the environment, general context, and the application domain area can easily be traced/connected to the implementation.

This seems also to be a prolific research area to join forces in other areas like business management (CoCoMe) and mechanical engineering (production automation PPU). It is easy to imagine the additional benefit when the actual problem grows in size and complexity well beyond the two case studies of this SPP.

14.2.4 Summary

The priority program investigated several facets of the notion of knowledge-carrying software. In particular, the relationship between requirements, architecture, code, and runtime data was investigated in several projects. Design decisions leading from requirements to architecture and to code were seen to make the transition between requirements, architecture, and code easier. Also, the decisions' relationship to reusable artefacts such as design patterns was investigated. In summary, at the end of the funding period of the priority program, we see that the idea of knowledge-carrying software is quite broad and carries further. It has a good chance to act as a conceptual frame to also include data which drive software functionality increasingly.

This leads also to an emerging area of research which addresses the question on how a software system and all the related artefacts in additional documentation of the system and the design process can be of help when it comes to the question on how and why the result of applying a software product yields a specific result. This ability to derive such an explanation of a result becomes more difficult in the context one might call Data-Driven Software Engineering.

This term addresses software products which use additional sources of information, for example streams of data to compute their results. Examples are systems using artificial neural nets to process data, categorize and assess it, and produce some output which might affect already our purchase decisions or other similar important aspects of our life.

Such systems existed for a long time, but at the start of the SPP they had not much practical use. After the start of the SPP, we saw a rise in the production and use of such systems in the area of data-driven software. Data determine the functionality of the system in a similar way as the implemented code does. For example, machine learning can improve the quality of the functions, or in the case of multi-tenant systems, customer-specific data help create customer-specific services based on the same service implementation.

For evolution, this poses new challenges. Firstly, it needs to be ensured that data privacy is preserved during evolution. This means that analyses on the accessibility of data need to be applied during evolutionary development—ideally in an incremental way. It also includes that one can provide evolving configuration files related to data access rights for such analyses. Secondly, the use of machine learning to drive software functionality leads to an increase need of means to make

the resulting decisions and operations of a software system understandable. In such cases, looking into the code of the software system does not help explain why a certain function performed in the way it did. This creates an urgent need to assess the influence of evolving data sets on the software system and its services and functions.