Chapter 13

Lessons Learned

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

The lessons learned while investigating new methods and techniques for knowledge-carrying software in SPP 1593 are summarised below. Some lessons indicate promising areas for future work, which will be discussed in Chap. 14.

Extracting tacit usage knowledge during runtime raises various challenges; see Chap. 5. Potentially *wrong* usage behaviour might transition into a pattern that is of interest and relevant for a new feature or functionality of an application. This learning phase needs to be a core element in the detection of usage behaviour, making it an important reference that points to tacit knowledge. Likewise, it is important to distinguish tacit knowledge from any kind of *noise* effects. We learned that only a limited set of new features can be detected, while the quality

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, Spezifikation von Softwaresystemen,
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, Lehrstuhl für Automatisierung und Informationssysteme,
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

of insights highly depends on the application in question. We acknowledge that requirements which lead to new features might be relevant for security. Identifying tacit knowledge in the form of security knowledge is a difficult task for which a good understanding of security and the domain of the software is necessary. Natural language processing can support the requirement engineer during this task.

Software product lines require continuous evolution over time, as discussed in Chap. 7. In particular, a software product line comprises numerous diverse yet inter-related and thus co-evolving artefacts (e.g. a feature model, implementation artefacts and test cases), where the evolution of one software product line artefact may also result in changes to other software product line artefacts, resulting in successive versions of an the entire software product line. While conceptually coping with respective software product line co-evolution scenarios, we observed that existing strategies for handling variant-rich software systems may also be adopted for handling version-rich software, as well as combinations of both dimensions of variability. In particular, efficient family-based analysis techniques based on explicitly available variant knowledge attached to software product line artefacts, so-called presence conditions, can be adapted to handle both variant- and version-rich software systems, as well as arbitrary combinations thereof. However, we encountered three main differences in handling versions as compared to variants:

- All variants of a product line coexist “in space,” while different versions emerge successively “over time”.
- Variant information is usually available already during requirement engineering, thus being traceable throughout all subsequent engineering phases and respective artefacts down to the code level, whereas new versions are often introduced ad hoc, thus being only barely documented. Hence, handling, analysing, and integrating new versions of an SPL usually requires additional up-front effort (e.g. model differencing, model merging) to fully understand the impact of SPL artefact evolution.
- For encoding variant information, it is usually sufficient to use Boolean variables and respective operators, whereas version information additionally requires some ordinal data types (e.g. Integers) to express (partial) ordering among versions. The latter may complicate automated tool support for automated reasoning.

Model-based round-trip engineering in embedded systems has to tackle the problem of on-site changes in mechanics, electrics, and software code. During the operation of such embedded systems, changes are induced by faulty devices, faulty code, or even spare parts no longer available. Such changes are often not documented or not known by the embedded system supplier. To enable a proper round-trip engineering, such changes need to be evaluated and included into the model. Merge mechanisms are required to merge different bug fixes that were conducted concurrently, even on different sites. Employing meta-models for coupling models of multiple disciplines is a promising idea. However, achieving consensus for constructing and evolving these meta-models remains a great challenge.

Strategies on how to efficiently address performance for variants and versions have been developed throughout the last years, and we have presented selected and complementary approaches based on the two SPP 1593 cases studies in Chap. 8:

- Regarding performance analysis of variants, we presented three strategies that tackled the challenge of combinatorial explosion of variants. We experienced that reducing the number of variants to be analysed and exploiting commonalities lead to promising results. Additional case studies were used to evaluate the approaches.
- Regarding the performance analysis of versions, we presented three strategies that focused on the management of versions, which evolve as different types of artefacts throughout the software's life cycle. We experienced that the amount and diversity of existing types of artefacts, modelling formalisms, evaluation techniques, and tool challenges impose a challenge for the widespread adoption of the strategies in practice. Moreover, the non-availability of suitable case studies and data sets involving all aspects to evaluate the research is a challenge.

We have observed a research gap with respect to the combination of strategies for variants and versions. First, the developed strategies seem to focus on either the analysis of versions or on the analysis of variants. Second, both types of strategies would benefit from an exchange of techniques used.

An approach to combine informal and formal techniques into an integrated approach is presented in Chap. 9. Informal requirements in natural language are used as the starting point for analysing security relatedness. At a clearly defined interface of security knowledge, the formal part takes over and applies existing and new techniques to the formal analysis and co-evolution. It was demonstrated how a socio-technical system, such as a large and long-lasting software product, can be monitored in full and be improved. Persons and their natural ways of communication were analysed. They met with adapted formal techniques at a novel interface: a layered ontology with a generic security part was extended evolutionarily towards a project-specific representation of knowledge. This way, feasible approaches to keep knowledge alive—and the software useful—are introduced.

As discussed in Chap. 10, embedded software in cyber-physical systems may evolve implicitly when its context—the surrounding physical world—changes. Even when the functional requirements remain stable, it may happen that the surrounding physical world changes, which may drive the software's evolution. This context consists not only of electronics and mechanics but also of human users and legal entities, all of which may drive the software's evolution. For cyber-physical systems, it is not sufficient to restrict knowledge to software:

- In general, it is useful to regard existing development artefacts (program code, hardware descriptions, design models and decisions, test cases, requirements, etc.) as valuable assets and to transfer these artefacts into knowledge-carrying software. This holds for any type of software in cyber-physical systems—embedded software and information systems—to allow for long-lasting maintenance of this knowledge and the software.

- Observing the dynamic behaviour of software at its interfaces and generating models from these observations is a successful approach to generate new knowledge about the software. Observation of dynamic behaviour may also achieve consistent co-evolution of running systems and its specification (models @ runtime). However, the applicability is restricted to recurring observable behaviour—ideally cyclic, repetitive behaviour. Causality cannot be addressed via observation, but regressions in quality characteristics may be identified among evolving software versions.
- Runtime observation via monitoring may be combined with other methods for knowledge-carrying software such as model-based and model-driven engineering. Model-based variant and version management offers techniques for mastering the evolution of knowledge at design and runtime of systems, both in sequential and parallel development processes. Multi-disciplinary variant and version management in multi-SPL settings remains a great challenge.
- Continuous integration is still a challenge for embedded systems: the application of continuous integration in the software part of embedded systems linked to mechanic and electric changes is possible but is mainly focused on testing. The design principles and the production of the mechanic and electrical parts of an embedded system differ from software, and therefore smaller changes are hard to integrate continuously, besides on the simulation level. One reason is the required manufacturing process of mechanics and electrics, compared to software.

Formal verification is not commonly applied in the industrial practice of software development for the automation domain. One of the main reasons is that, in practice, sufficiently accurate specifications or models of the intended behaviour are not available. Coming up with adequate specifications often requires an effort that companies are not willing to invest at present; it would not be economically beneficial. Moreover, when software (or some other component of a system) evolves, formal specifications must co-evolve—which increases the required overhead for formal specification even further. Formally verifying the behaviour of cyber-physical systems is, thus, often infeasible due to the absence of formal requirement specifications. However, the research on formal methods within the priority program focused on the application of formal modelling, specification, and verification to evolving software, in particular in the automation domain. The identified idiosyncrasies of this application scenario are explicated in Chap. 11:

- Formal methods can benefit from the fact that during evolution, a history of software artefacts is available. For *regression verification*, an old revision serves as formal specification for the new one. The verification essentially checks (behavioural) equivalence between the two programs. Experience from our experiments shows that regression verification (using a state-of-the-art model checker with invariant generation) can be successfully applied to software systems with a considerable state space. We have experienced that, in some cases, the two software revisions do not suffice as input for regression verification. In general, environment models are needed to define those parts of the plant behaviour relevant to establish equivalence. These models are generally difficult

to obtain and validate. While modern automatic model checkers employ a number of optimisations and scale way better than earlier tools, applying the approach to systems with larger state spaces is sometimes possible, but not always, depending on how similar states in related runs of the two programs are. Since the new version is usually intended to behave differently in some situations and equivalently in others, there must be means to define the notion of equivalence. For conditional and relational equivalences, some form of temporal specification is required, formulated in a relational behavioural specification language. The design principles of usable and expressive languages is still an active research topic. Regression verification against some previous executable system version—considered as an executable specification—is a promising approach, which requires further improvements with respect to scalability.

- Generalised test tables are a formal specification language building on the well-known and accepted paradigm of test tables (used in industry to specify test cases). We have seen that they are expressive enough to specify interesting properties of realistic systems and have applied them for the (partial) behavioural specification of production plant systems on the signal level. Specifying relevant properties of the entire system on this low level was shown to be possible yet rather technical and unintuitive. On the other hand, generalised test tables are very well suited for the specification and verification of individual general-purpose function blocks, for example within a software library for the automation domain. We have adapted this easy-to-comprehend specification technique to be also usable as specification language for the temporal relational specifications in regression verification.
- The software of automated production systems can often not be analysed in isolation, separated from its application context (the physical entities of the plant, manufacturing management system, etc.). It is important to model different viewpoints (requirements, process, and system) simultaneously for precise and successful model checking of multi-disciplinary systems. A challenge for this verification is to model the continuous processes of a cyber-physical system faithfully, sufficiently, and precisely and to validate these models. The experiments show that probabilistic modelling and subsequent probabilistic model checking are a feasible way to deal with the arising problems.

As an overarching lesson, we can report that shared case studies—see Chaps. 4 and 12—are an invaluable means in collaborative research programs such as SPP 1593 for communication and evaluation of research results. However, implementing and maintaining such shared case studies require significant effort and resources.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.