Linnaeus University Dissertations
No 318/2018

Nadeem Abbas

Designing Self-Adaptive Software Systems with Reuse

LINNAEUS UNIVERSITY PRESS

Designing Self-Adaptive Software Systems with Reuse

Linnaeus University Dissertations
No 318/2018

Designing Self-Adaptive Software Systems with Reuse

Nadeem Abbas

LINNAEUS UNIVERSITY PRESS

Designing Self-Adaptive Software Systems with Reuse
Doctoral Dissertation, Department of Computer Science, Linnaeus University, Växjö, 2018

ISBN: 978-91-88761-51-4 (print), 978-91-88761-52-1 (pdf)
Published by: Linnaeus University Press, 351 95 Växjö
Printed by: DanagårdLiTHO, 2018

Abstract

Abbas, Nadeem (2018). *Designing Self-Adaptive Software Systems with Reuse,* Linnaeus University Dissertations No 318/2018, ISBN: 978-91-88761-51-4 (print), 978-91-88761-52-1 (pdf). Written in English.

Modern software systems are increasingly more connected, pervasive, and dynamic, as such, they are subject to more runtime variations than legacy systems. Runtime variations affect system properties, such as performance and availability. The variations are difficult to anticipate and thus mitigate in the system design.

Self-adaptive software systems were proposed as a solution to monitor and adapt systems in response to runtime variations. Research has established a vast body of knowledge on engineering self-adaptive systems. However, there is a lack of systematic process support that leverages such engineering knowledge and provides for systematic reuse for self-adaptive systems development.

This thesis proposes the Autonomic Software Product Lines (ASPL), which is a strategy for developing self-adaptive software systems with systematic reuse. The strategy exploits the separation of a managed and a managing subsystem and describes three steps that transform and integrate a domain-independent managing system platform into a domain-specific software product line for self-adaptive software systems.

Applying the ASPL strategy is however not straightforward as it involves challenges related to variability and uncertainty. We analyzed variability and uncertainty to understand their causes and effects. Based on the results, we developed the Autonomic Software Product Lines engineering (ASPLe) methodology, which provides process support for the ASPL strategy. The ASPLe has three processes, 1) ASPL Domain Engineering, 2) Specialization and 3) Integration. Each process maps to one of the steps in the ASPL strategy and defines roles, work-products, activities, and workflows for requirements, design, implementation, and testing. The focus of this thesis is on requirements and design.

We validate the ASPLe through demonstration and evaluation. We developed three demonstrator product lines using the ASPLe. We also conducted an extensive case study to evaluate key design activities in the ASPLe with experiments, questionnaires, and interviews. The results show a statistically significant increase in quality and reuse levels for self-adaptive software systems designed using the ASPLe compared to current engineering practices.

Keywords: Variability, Uncertainty, Self-Adaptation, Software Reuse, Software Design, Methodology, Domain Engineering.

To beloved parents and wife

Acknowledgments

In the name of Allah, the Most Gracious and the Most Merciful. I bow down to Him with the humblest gratitude to thank for His countless blessings.

I would like to express sincere gratitude to my advisor, Jesper Andersson, for all the encouragement, support, and guidance. His determination, commitment, and high spirits helped me a lot to develop and flourish. There was a time when I lost belief in myself, but it was he who stood with me, restored the confidence and lead me to where I stand today. I would also like to thank Prof. Welf Löwe for accepting me as a Ph.D. student, sharing his knowledge and being supportive throughout the years.

I am deeply grateful to Prof. Danny Weyns who has been a great source of inspiration and learning. His invaluable knowledge and feedback helped me a lot to learn, improve and publish my work. Sincere thanks to Ass. Prof. Sarfraz Iqbal for guidance and proof-reading the thesis.

Special thanks to Anna Wingkvist, Sharafat Ali, Arianit Kurti, Jonas Lundberg, Mathias Hedenborg and Andreas Kerren for being around with a lot of positive energy and assistance. Many thanks to all the faculty and the administrative staff at the department of computer science, Linnaeus University.

I am very thankful to all my colleagues and friends Muhammad Usman Iftikhar, Amir Rasheed, Mirza Tassaduq Baig, Kostiantyn Kucher and Stepan Shevtsov for all the discussions and good time together.

I will always remain indebted and profoundly grateful to my parents for all the sacrifices they made for my upbringing. Sadly, they could not witness this milestone, but surely will be glad to know that their efforts have paid off. Special thanks to my mother and grandmother, who indeed would have loved staying me closer to home, yet sent me abroad for a better future. Special thanks to my uncle, cousins and all other family members for supporting and encouraging me whenever I felt low.

Last but not least, I am grateful to my beloved wife Maryyam and daughter Dua for all the prayers, support, and care. I would not have been able to achieve this goal without your backing throughout the years. You girls are my main source of energy, and I owe you a lot. Thank you for everything.

Växjö, Sweden
March 10, 2018

Contents

Chapter 1

Introduction

This chapter provides an overview of the thesis. First, we specify and motivate the problem statement. Then, we outline objectives for the desired solution. We map the problem statement to a set of research questions, list the thesis contributions, and define the research scope. The chapter ends with the thesis map and an overview of the chapters to follow.

1.1 Thesis Overview

Software systems are traditionally designed based on the assumption that there are no runtime variations in the systems' goals, the environments, and the systems themselves [Gar10]. Change is, however, an inevitable characteristic of systems, and requires software systems to be modified and adapted [C+09]. The adaptations are usually performed offline as software maintenance. However, for large and complex systems, the maintenance often exceeds system administrators' capabilities, and impacts system quality negatively, which calls for automated online capabilities.

With the development of new technologies, such as internet of things and cyber-physical systems, software systems are at the center of runtime variations. The runtime dynamics and growing complexity of software systems call for improved methods of software design and development [Bru+09; KM07]. Self-adaptive software systems have been proposed as a solution to develop software systems with self-managing characteristics. The self-managing or self-adaptive characteristics refer to capabilities of a software system to adapt and manage itself with no or minimal human intervention.

A Self-Adaptive Software System (SASS) is capable of modifying its behavior or structure in response to changes in its goals, environment, and in the system itself [De +13]. A SASS is conceptually composed of a managed subsystem and a managing subsystem. The managed subsystem models the application logic that provides a system's primary functionality. The managing subsystem models adaptation logic to monitor and manage the managed subsystem.

Designing SASS systematically and cost-effectively is known to be an engineering challenge [Bru+09; KM07] and requires well defined, disciplined, and systematic process support [De +13]. Software reuse [Kru92] is a proven approach to develop software systems in a controlled and cost-effective way. The reuse enables developers to improve quality and productivity at reduced cost and shorter

time-to-market [Gri93]. A vast body of knowledge to engineer self-adaptive systems has been established over the years. However, to the best of our knowledge, there is lack of systematic and repeatable process support to design and develop SASS with reuse. Thus, the problem this thesis addresses is the lack of process support to design and develop self-adaptive software systems with reuse.

The thesis presents an Autonomic Software Product Lines (ASPL) [AA15] strategy to address the problem. The ASPL is a multi product-lines strategy to design and develop self-adaptive systems. The basic concept of the ASPL is to establish a general platform and reuse it across several application domains. The strategy is realized in three steps. The first step defines and maintains an application domain-independent ASPL platform. The ASPL platform provides managing system artifacts for reuse across several application domains. It is established independent of application domains and is likely to have gaps between what is needed by a domain and what is offered by the platform. The second step addresses these gaps by transforming the ASPL platform into an application domain specific Managing System Platform. The third step integrates the managing system platform, derived in the second step, with an independently developed Managed System Platform. The integration is needed to align the managed and managing system platforms so that the artifacts from the two platform can be used to produce self-adaptive software systems.

Applying the ASPL strategy is challenging and requires process support. For instance, defining an ASPL platform (the first step) raises uncertainties due to lack of knowledge about target application domains [AAW18]. The principal cause of uncertainty in self-adaptive software systems is *runtime variability* [EM13]. The development with reuse, i.e., ASPL, introduces additional uncertainties caused by *domain variability* and *cross-domain variability* [AA15]. The domain variability originates from reuse within a single domain and refers to differences among systems within a domain. The cross-domain variability stems from reuse across several domains and refers to differences among systems across multiple domains. The three variability dimensions add to the design uncertainty. The uncertainty may lead to technical debt in the system, if not identified and addressed [EM13; Gar10]. To that end, we analyzed the variabilities and resulting uncertainties in context of the ASPL strategy. The analysis helped to discern factors causing uncertainty and to establish the Autonomic Software Product Lines engineering (ASPLe) methodology [AAW18].

The ASPLe is a methodology to design and develop self-adaptive software systems with reuse. It provides process support with step-wise activities and development artifacts to realize the ASPL strategy. The strategy involves two principal challenges, 1) variability and 2) uncertainty handling. For variability handling, the ASPLe provides process-level instructions and specially designed artifacts to identify, model, and manage variability. The explicit variability modeling enables system designers to better analyze and reason about design alternatives, and resolve uncertainties. For uncertainties caused by lack of knowledge, the ASPLe offers implicit support to mitigate such uncertainties by delaying requirements and design decisions [VBS01] till the point where complete or more knowledge is available. The ASPLe also advocates mitigating such uncertainties by collecting knowledge

from state-of-the-art analysis and design methods. To that end, it provides developers with an extended Architectural Reasoning Framework (eARF) [AJ15]. The eARF encapsulates SASS specific architectural knowledge and guides developers how to use this knowledge. It also includes an analytical framework [Abb+16], which provides rigorous and objective support to model and verify design options.

We developed three prototype product lines of self-adaptive software systems to demonstrate and validate the use of the ASPLe. An extensive case study was also conducted to evaluate the ASPLe with respect to support for reuse and uncertainty mitigation. The analysis of data from the case studies and prototypes shows that the ASPLe provides well defined and organized process support to design SASS with a significant increase in reuse and decrease in the number of faults. Based on the results, we conclude that the ASPLe helps developers to improve quality and efficiency with systematic reuse of artifacts and knowledge across several application domains.

The remainder of this chapter is organized as follows. Section 1.2 defines and motivates the problem, followed by Section 1.3 that specifies objectives for the desired solution. Research questions answered by the thesis are stated in Section 1.4. Section 1.5 summarizes the thesis contributions, followed by Section 1.6 that describes the research scope. The chapter ends with Section 1.7 that presents the thesis map and provides an overview of the subsequent chapters.

1.2 Problem Definition and Motivation

The problem addressed by the thesis is:

> *lack of process support to design and develop self-adaptive software systems with reuse*.

Software reuse, described in Section 2.2, has been long acclaimed as a useful method to build software systems efficiently and cost-effectively [Kru92]. It enables developers to resolve complexity and improve quality and productivity at reduced cost and shorter time-to-market [Gri93]. However, a systematic approach is required to achieve the goals and claimed benefits of development with reuse. A systematic approach to reuse requires much more than just code and library technology. It requires well planned, controlled and repeatable process support with purposefully designed tools and infrastructure [AA15; Fra94].

Self-adaptation is a common concern for a large class of systems and supporting its realization by generic reusable development artifacts would be a big step to improve quality, affordability, and productivity of software systems [HSF04]. Research has established a vast body of knowledge on engineering self-adaptive systems over the years. However, to the best of our knowledge, there is no or little work available that has considered systematic reuse of this knowledge. The research gap and benefits of development with reuse provided necessary motivation to investigate the design and development of self-adaptive software systems with systematic reuse.

The initial exploratory research we conducted showed that the development of SASS with reuse involves challenges of variability and uncertainty. Software

variability, described in Section 2.3, is a central concept in development with reuse [SVB05]. For development with reuse, software developers are required to specify requirements for variability and model reconfigurable software systems which can be extended or customized for reuse in a specific context. The current variability specification and modeling methods, such as feature modeling [Kan90], primarily support variability in functional requirements. There is lack of explicit support to identify and model variability in requirements for quality attributes [ESB07; MGA13]

Self-adaptive system properties referred as self-adaptation or self-management properties, such as self-healing and self-optimization, are quality attributes in essence. The development of SASS with systematic reuse requires specifying the self-adaptation properties with their commonalities and variabilities across several applications or application domains [AA15; AAL10]. The state-of-the-art variability modeling approaches lack explicit support for identifying and modeling self-adaptation properties with variability across several application domains. Moreover, knowledge about runtime variability of self-adaptation properties is often not available at design time. By analyzing and understanding an application domain, the designers may predict some, but not all variations that may occur at runtime. The lack of knowledge and hard to predict nature of self-adaptation properties lead to uncertainties in requirements engineering and design.

Uncertainty, described in Section 2.4, refers to a situation of specifying and modeling systems with imprecise or incomplete knowledge [MH05]. Uncertainty is closely related to software variability. It leads to variability in systems design when multiple design alternatives are available and designers are not sure about which one to select. High-quality software design is a necessary condition for a software system to satisfy its goals and requirements [BCK03]. The condition becomes more vital in the development of large and complex systems coupled with variability and uncertainty.

The combination of runtime variability and uncertainty broadens the design space. The designers are required to identify a number of design options and reason about them for several design parameters, such as business goals, application requirements, and operating conditions. The lack of knowledge about design parameters and their runtime variations makes designers less confident and uncertain in the architectural analysis and design. The problem of architectural analysis and design grows with the complexity of self-adaptive systems and calls for improved methods of design and development with systematic process support [Gar10].

1.3 Objectives for the Solution

We identified the following objectives for the solution:

O1 The solution should be a methodology to design and develop self-adaptive systems with systematic reuse.

O2 The methodology should be based on a well-defined strategy with separation of managed and managing subsystems concerns.

O3 The solution should be derived by exploring existing design principles and methods, such as design patterns and tactics [BCK03].

We set the objective O1 to devise a solution that provides developers of self-adaptive software systems with a well-organized, documented, and repeatable process support. The objective O2 was set to reduce complexity, improve reusability, and facilitate runtime adaptations [Tar+99] by maintaining the disciplined split between managed and managing systems concerns. The motivation for the objective O3 was to gain from the available knowledge and build the solution on top of the current engineering and design practices.

1.4 Research Questions

A primary goal of the thesis is to address the problem described in Section 1.2. To achieve the goal, we identified following research questions:

Q. 1 What is the current state-of-the-art in design and development of self-adaptive software systems with reuse?

- **(a)** What are the challenges confronted by system designers while designing SASS with systematic reuse?
- **(b)** How can the challenges identified in question 1(a) be addressed?

Q. 2 How can self-adaptive software systems be designed with systematic reuse in an application domain and across several application domains?

- **(a)** How can we design and develop generic application domain independent artifacts for reuse across several application domains of SASS?
- **(b)** How can the generic reusable artifacts be specialized for reuse in a specific application domain of SASS?

The first question aims to explore and understand the background and the current state-of-the-art in the SASS. The focus here is to investigate available knowledge, methods, and tools from *design with reuse* perspective. The two sub-questions, Q. 1(a) and Q. 1(b), are defined to identify known challenges and proposed solutions.

The second research question calls for a research effort that should result in a systematic approach to design and develop SASS with and for reuse. It requires developers to support reuse across several applications and application domains. Two crucial steps in development with systematic reuse are: 1) to design artifacts on purpose for reuse in several applications or application domains, and 2) specialize the reusable artifacts for reuse in a specific application or application domain. The research efforts needed to achieve the two steps are expressed as questions Q. 2(a) and Q. 2(b), respectively.

1.5 Thesis Contributions

The primary contributions of the thesis are:

1. Variability and uncertainty analysis in the context of the SASS design with reuse (Section 4.1.1).
2. The ASPL, a systematic reuse strategy to design and develop SASS (Section 4.1.2).
3. The ASPLe, a development methodology with process support to implement the ASPL strategy (Section 4.1.3).
4. The eARF, an extended reasoning framework with rigorous support for architectural analysis and decision making (Section 4.1.4).

Besides the above-listed contributions, the thesis also contributes with two secondary results:

1. Online learning and knowledge sharing mechanism to support evolution in software product lines; see [AAW11] for details.
2. An educational package for teachers and researchers to teach and experiment with self-adaptive systems. We see this as a valuable resource for future research and development in the field. The package can be downloaded at `http://homepage.lnu.se/staff/janmsi/sass-edu/`.

1.6 Research Scope

In this thesis, we study the development of self-adaptive software systems with reuse. The scope of reuse is limited to the managing subsystem level. We focus on the requirement and design phases of development. The implementation and testing phases are out of scope, and we plan these as future work. In requirements engineering and design, we target the problem of uncertainty and variability and provide repeatable process support to mitigate uncertainty and manage variability systematically.

1.7 Thesis Map and Organization

Figure 1.1 depicts the thesis research map. We follow Design Science Research Methodology (DSRM) [Pef+07] and Shaw's recommendations [Sha03] to structure and present the thesis. The DSRM, described in chapter 3, defines a nominal process composed of six activities: problem identification and motivation, objectives of a solution, design and development, demonstration, evaluation, and communication. These activities are depicted as regular rectangles in the research map, i.e., Figure 1.1. The arrows in the map show data and control flow between activities.

The activities shown as rounded rectangles are not originally part of the DSRM methodology. These activities, however, help in presenting and understanding a

research project. For instance, the background and research questions help in understanding a thesis context and research questions answered by the thesis. It is hard to understand and validate results without knowing the context and target research questions. Thus, we combined the background, research questions, and research methods activities with the DSRM to improve the thesis presentation and aid understanding.

As shown on the left of the research map, the thesis started with a review of journal articles and conference publications about self-adaptive software systems and related disciplines that form the *background* of the studied problem. Along with describing the background, we used the literature review to understand the problem domain, formulate the *problem statement, research questions*, and *objectives for the solution*. Other *research methods* used in the thesis are prototyping, case study, questionnaires, and interviews. We combined the prototyping and literature review methods to *design and develop* a solution that builds upon existing knowledge. The case study and other research methods listed in the research map were mainly used to *demonstrate* and *evaluate* the solution. The demonstration and evaluation are depicted as a single activity as we use them together to validate the solution. The last activity in the map, on the rightmost side, depicts a set of publications used to communicate the thesis problem, the solution elements, and studies performed to evaluate and validate the solution. We used the demonstration and evaluation results and feedback from publications to refine the objectives for the solution and the design and development activities to build the solution.

The rest of this thesis is organized as follows. Chapter 2 introduces theoretical foundations followed by an overview of the applied research methods in chapter 3. Chapter 4 reports the thesis results. Chapter 5 describes how we evaluate and validate the results. The publications made to communicate the thesis effort and results are introduced in chapter 6. Chapter 7 concludes the thesis with a discussion on the thesis results and future work.

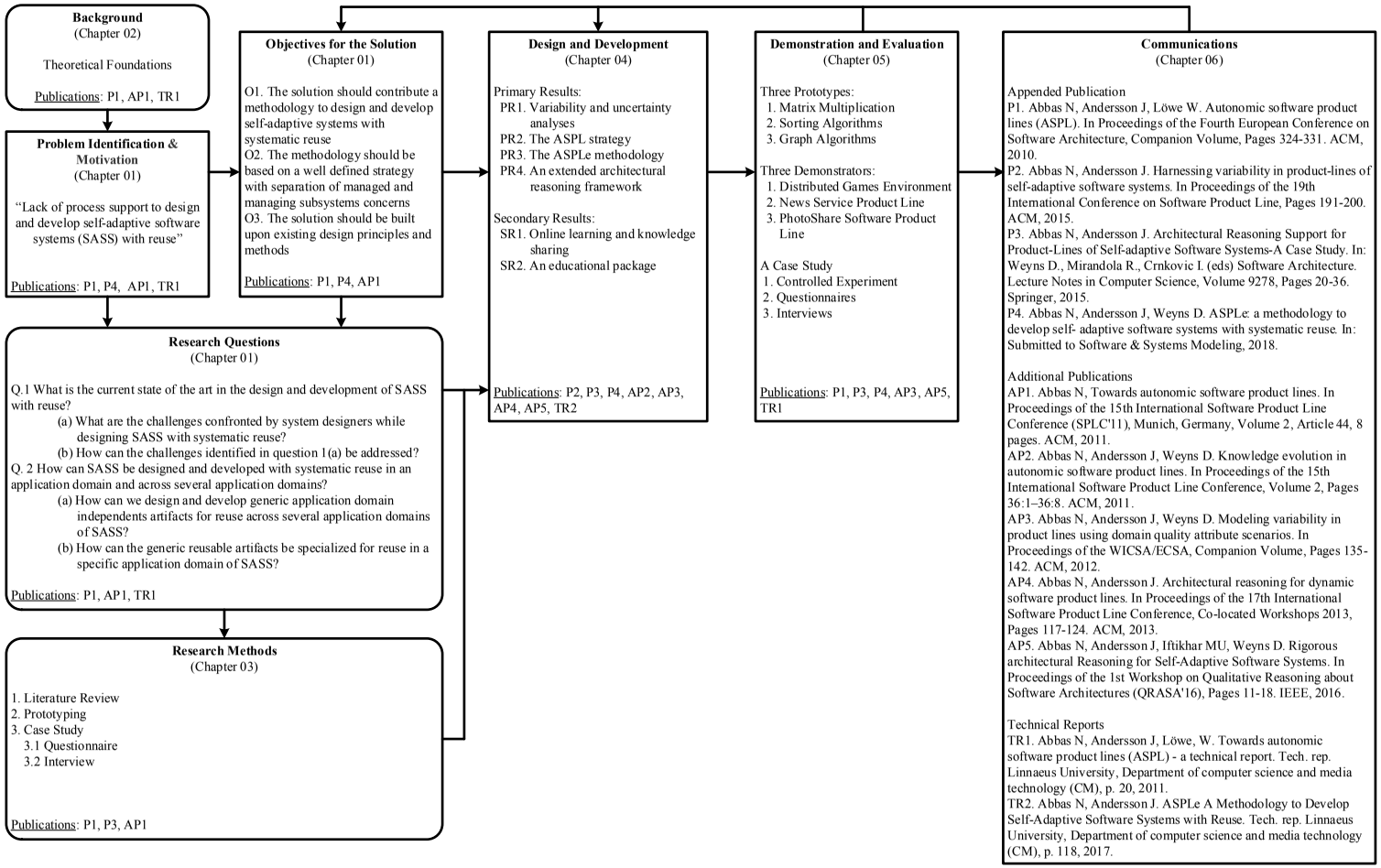

Figure 1.1: Research Map

Chapter 2

Theoretical Foundations

This chapter introduces the concepts of self-adaptive software systems, software reuse, variability, uncertainty, and software design. These concepts together form the thesis theoretical foundations.

2.1 Self-Adaptive Software Systems (SASS)

A Self-Adaptive Software Systems (SASS) is a software system that is capable of adapting its behavior and structure in response to its perception of the environment, goals and the system itself [De +13]. The principal motive to develop self-adaptive software systems was to fulfill software engineering's promise of developing systems that can retain flexibility throughout their lifecycle and are as easy to adapt in a field as are on a drawing board [O+99]. The complexity of current and emerging systems, such as pervasive computing and internet of things, has provoked the need for self-adaptive software systems even more.

As shown in Figure 2.1, a SASS is conceptually a combination of a managed system and a managing system [De +13]. The managed system is a software system responsible for primary application functionality. The managing system models adaptation logic to monitor and manage the managed system. Both the managed and managing systems are situated in an environment. The environment is an abstraction of an external world with which a self-adaptive system interacts, and in which effects of a system are observed and evaluated [Wey+13].

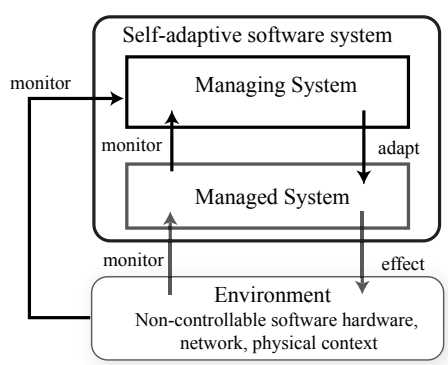

Figure 2.1: Conceptual Architecture of a Self-Adaptive Software System

According to the conceptual architecture of a SASS, both managed and managing systems monitor the environment. The managed system may affect the environment, but the managing system has no direct effect on the environment. However, the managing system may perform adaptive actions on the managed system, which in turn may affect the environment. The interactions between managed and managing systems are carried through the monitor and adapt interfaces.

2.1.1 Self-Adaptation Mechanisms

Plenty of research on SASS has been done, and much work is in progress. The state-of-the-art in self-adaptive systems distinguishes three adaptation mechanism types to design and develop SASS: internal vs. external, model-based vs. model-free, and closed vs. open adaptations.

The internal and external adaptations are distinguished based on the separation of adaptation and application logic. The internal adaptation mechanisms do not separate adaptation and application logic. Such mechanisms use programming language constructs such as exceptions, reflection, and conditional expressions to interweave application and adaptation logic [SGP13]. Due to tight coupling between adaptation and application logic, the internal adaptation approaches suffer from poor scalability, maintainability, and reusability of development artifacts.

The external adaptation mechanisms model external closed-loop type control mechanism, such as managing system shown in Figure 2.1, to analyze and adapt a managed system. Here, adaptation logic is separated from application logic. Due to the clear separation of concerns, external adaptation approach offers better support for scalability, reusability, and maintainability. Some of the most acknowledged external adaptation approaches include Rainbow [G+04], StarMX [AST09], MADAM [F+06], and MUSIC [Rou+09].

The model-based and model-free mechanisms are distinguished based on the use of the model(s) representing managed and managing systems and their environment. The model-based adaptation mechanisms use system models at runtime [VG14] to analyze and reason about adaptation decisions. Whereas, the model-free adaptations do not have predefined system models and, instead, use other adaptation mechanisms such feedback loop [KC03] or programming language constructs.

The closed vs. open adaptation mechanisms are distinguished based on support to incorporate new changes and adaptation actions at runtime. The closed type adaptation mechanisms support a fixed set of variations and adaptation actions; no new moves and strategies to adapt can be introduced at runtime. The open adaptation mechanisms are extendable, i.e., they allow changes in requirements and system behavior and support addition of new adaptation strategies at runtime.

2.1.2 SASS and Autonomic Computing

Autonomic Computing (AC) labels systems that can manage themselves given high-level objectives from administrators [HM08; KC03]. The concept of autonomic computing was envisioned by IBM to handle large-scale, complex systems that cannot be controlled and maintained efficiently by humans. The AC

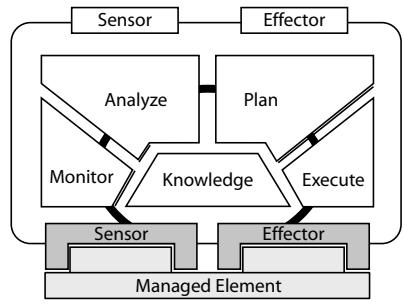

Figure 2.2: MAPE-K Feedback Loop

and SASS research communities resemble in their goals to produce systems with self-managing characteristics. Both communities often work together. For instance, the Monitor, Analyze, Plan, Execute and Knowledge (MAPE-K) feedback loop [KC03], a widely used mechanism to realize self-adaptation in SASS, has its origin in the autonomic computing. As shown in Figure 2.2, the MAPE-K loop consists of monitor, analyze, plan, execute, and knowledge components that work together to monitor and adapt a managed system. Below is a brief description of the MAPE-K components:

Monitor (M) component uses sensors to collect data from a managed system and its environment. The collected data are reported to the knowledge element.

Analyze (A) component analyzes the up-to-date data in the knowledge element. The analysis is performed to check whether adaptation is required or not.

Plan (P) component prepares an action plan to achieve system's goals based on the results of the analysis. The action plan is a workflow of adaptation actions.

Execute (E) component receives a plan and executes it, i.e., performs adaptive actions using effectors.

Knowledge (K) component is a central knowledge base accessible by the other MAPE components. The MAPE components use the knowledge base to coordinate and plan adaptive actions. In addition to the data reported by the MAPE components, the knowledge base may contain additional information such as architectural models, goal models, adaptation policies and change plans.

Despite the similarities, the SASS and AC differ in scope. The scope of the SASS community is limited to self-adaptation at application software level, whereas the autonomic computing studies self-adaptation at entire system level including network, operating systems and hardware layers [ST09]. In other words, the managed element in the SASS is a software system, while in the autonomic computing the managed element can be both software and hardware systems. Furthermore, the AC and SASS communities differ in the use of terminology for self-managing characteristics. The self-managing characteristics are referred as self-adaptation and self-management properties by the SASS and AC communities,

respectively. In this thesis, we use the terms self-adaptation and self-management interchangeably.

The self-adaptation properties are, in fact, self-oriented forms of traditional quality attributes and require software systems to adapt themselves at runtime in response to changes in the quality attributes. Kephart and Chess [KC03] described four such properties: 1) self-optimization, 2) self-healing, 3) self-configuration, and 4) self-protection. These properties are self-oriented forms of performance, availability, configurability and security quality attributes, respectively.

2.2 Software Reuse

Software reuse is a process of creating software systems from existing artifacts instead of developing them from scratch [Kru92]. In the late 1960s, McIlroy [McI68] proposed to *build large, reliable software systems in a controlled and cost-effective manner* - a challenge that still holds valid after about half a century [FK05].

Advances in reuse technology have leveraged software development with significant improvements in quality and productivity [FK05]. Software reuse is now acknowledged as one of the most efficient and effective ways to produce software systems. It enables developers to deliver high-quality systems with reduction in cost and time-to-market. However, a systematic approach is needed to achieve the claimed benefits of development with reuse.

2.2.1 Systematic Reuse

Griss [Gri96; Lan+] defined systematic reuse as an "institutionalized organizational approach to product development in which reusable assets are purposely created or acquired, and then consistently used and maintained to obtain high levels of reuse". We consider a systematic reuse approach as one that is based on a well-defined strategy and follows a repeatable and controlled process [Fra94; JGJ97]. Unlike ad-hoc reuse approaches [Pri93] with opportunistic reuse of lower level artifacts, such as code snippets, systematic reuse is planned for reuse of high-level artifacts such as requirements and design engineering artifacts. The artifacts are on purpose defined and maintained for reuse across several applications or application domains.

The systematic reuse approaches are anchored in three hypotheses: 1) redevelopment hypothesis, 2) the oracle hypothesis, and 3) organizational hypothesis [WL99]. According to the redevelopment hypothesis, most of the software development projects construct systems that are variants of existing systems [Par76]. Usually, these variants have more in common than what differs. The redevelopment hypothesis suggests that software developers should avoid redevelopment and find out ways for reuse by exploiting commonalities among software systems and their variants.

The oracle hypothesis is about predicting changes confronted by a system over its lifetime. It suggests that by anticipating and predicting changes, developers can constrain variability to some extent if not entirely. By defining variability,

developers may scope a product line and formulate a systematic reuse approach accordingly.

The organization hypothesis is about establishing an organization and controlling development activities. It suggests that by following a proper organizational structure, the software developers can take advantage of commonality and predictability from the redevelopment and oracle hypotheses, respectively. The three hypotheses, together, lay a strong foundation to formulate a systematic reuse strategy [AA15].

2.2.2 Software Product Lines Engineering (SPLE)

The SPLE [PBV05; WL99] is a systematic and widely used software reuse approach. It supports reuse by exploiting commonalities in a product line while managing differences (variability). A software product line for an application domain is a set of software applications that share features satisfying needs of a specific market segment, and are developed from a shared set of artifacts [N+07]. In this thesis, we use the terms software product line and application domain interchangeably. The fundamental principles of SPLE are *the use of common platforms* and *mass customization*. The mass customization is a process of producing systems in bulk and customizing them according to individual users needs. The production is supported by establishing platforms. A platform is a set of development artifacts designed and developed for reuse across several systems. Based on the concept of vertical and horizontal reuse [Pri93], we distinguish between vertical and horizontal platforms. A vertical platform is a collection of artifacts developed for reuse within a single application domain, whereas the horizontal platform is a collection of artifacts developed for reuse across several domains.

SPLE separates software development into two processes: 1) domain engineering and 2) application engineering. The domain engineering defines a vertical platform of reusable artifacts. The application engineering customizes artifacts from the common platform to derive individual application of a product line. The two processes with separation of concerns work together to achieve systematic reuse by exploiting commonality while managing variability in a planned, organized, and efficient way.

Pohl et al. [PBV05] define a SPLE framework that provides developers with guidelines to perform domain and application engineering processes. The framework divides the two processes into requirements, design, implementation, and testing subprocesses. There is no workflow defined for the subprocesses. However, the framework provides details on how to define and exploit commonalities and variabilities in each subprocess.

To support reuse across several applications or application domains, the SPLE and other reuse approaches are required to support variability. An overview of the variability is given below.

2.3 Variability

Variability is an ability of a software system or artifact to be efficiently extended, changed, customized or configured for (re)use in a specific context [VBS01]. It is a widely used concept in software reuse communities such as software product line engineering [PBV05; VBS01]. In software reuse literature, the term variability is often used to represent differences or variations among systems. SPLE develops a set of software systems by managing commonalities and variations in systems' artifacts such as requirements, architectures, components, and test cases [CAA09]. Svahnberg et al. [SVB05] described a framework to get required variability in place and manage it. The framework is composed of four activities, 1) variability identification, 2) constraining variability, 3) variability implementation and 4) variability management. The framework activities are introduced as follows.

2.3.1 Variability Identification and Modeling

The variability identification is concerned with the identification, i.e., where variations may occur and where support is needed. The identification is often performed as a part of requirements engineering. However, it can be identified or refined in later development phases such as design, implementation, and testing. A number of feature model-based approaches, such as FODA [Kan90], FORM [Kan+98], and FeatureRSEB [GFd98], have been proposed to identify and model variability in terms of features. A feature is a prominent or distinctive end-user visible aspect, quality or characteristic of a system(s) [Kan90]. It abstracts a set of functional and quality requirements that specify a logical unit of system behavior [Bos00].

Variability modeling is the explicit representation of variability [Sin+04]. Variability can be modeled either as a part of traditional development artifacts, such as requirements specification and component diagram, or as a separate variability model. Pohl et al. [PBV05] argued to model variability as a separate Orthogonal Variability Model (OVM). Modeling variability as a separate OVM helps to maintain separation of concerns with a reduction in complexity and improvement in consistency and maintainability.

2.3.2 Constraining Variability

Once variability has been identified and modeled, it needs to be constrained to keep it within manageable limits [SVB05]. Variability is constrained by defining a set of variation points, variants, variability dependencies and constraints [LSR07; PBV05]. A variation point refers to a system attribute which may vary, for instance, at design or runtime. A variant is an alternative that can be bound or rebound to a variation point.

2.3.3 Variability Implementation

The variability implementation deals with identifying suitable variability realization techniques to implement variation points defined to constrain the variability. Conditional compilation [GA01], Aspect-oriented Programming [Kic+97], and Open Services Gateway Initiative [OSG07] are few examples of the variability

realization mechanisms [AMA17]. Svahnberg et al. [SVB05] argued that constraining variation points enables more informed decisions for how to implement the variability.

2.3.4 Variability Management

The variability management activity manages feature models, variation points, variants and variability constraints defined to model and constrain variability. System variability may evolve due to changes in the system's requirements or operating environments. Thus, the artifacts and techniques used to model, constrain and implement variability must be adapted, for instance by adding new or pruning old, no longer used variation points, variants, and corresponding variability mechanisms.

2.3.5 Variability in the Development of SASS with Reuse

Software development with systematic reuse involves variability across two dimensions, 1) domain variability and 2) cross-domain variability. Domain variability originates from vertical reuse and refers to differences among systems in an application domain [AA15]. An example of domain variability is the differences in performance requirements among different systems in a domain. The second dimension, cross-domain variability originates from reuse across several application domains and refers to differences among systems in different domains.

The development of self-adaptive software systems with systematic reuse introduces a third variability dimension, runtime variability. Runtime variability comes from self-adaptation and refers to the variability needed to incorporate runtime variation in a system's requirements, goals, environment, and the system itself [De +13]. It enables a self-adaptive software system to add, change, or remove variants dynamically at runtime with minimal or no human intervention [COH14; Hel+09].

The three variability dimensions complicate the design space architects have to consider and add to uncertainty in the design and development of SASS with reuse.

2.4 Uncertainty

Uncertainty is an inherent property in complex systems with effects on all system development activities. Walker et al. [Wal+03] define uncertainty as *"any deviation from the unachievable ideal of complete determinism"*, that is, it refers to things which are not or imprecisely known at a specific point in time [MH05]. From a software design perspective, uncertainty affects decision making and lead to suboptimal or invalid design decisions.

Hastings and McManus [MH05] presented a framework, shown in Figure 2.3, to manage uncertainties and their effects. Primary sources of uncertainty are lack of knowledge and lack of definition. These sources lead to risks, such as failure, degradation, and cost/schedule deviation. The risks can be handled using mitigation techniques such as margins, redundancies, and proper design strategies. The uncertainties are not necessarily bad things; these may add value if identified and

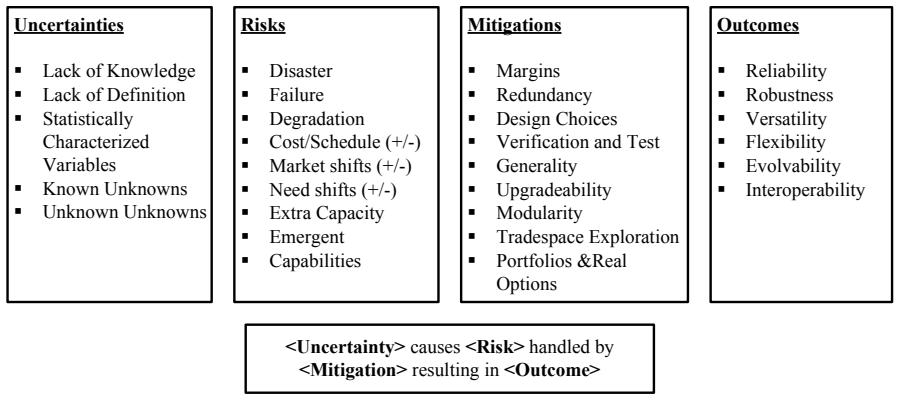

Figure 2.3: Framework for Managing Uncertainties (reproduced from [MH05])

mitigated appropriately. Below is an overview of the uncertainty in software engineering in general and in the development of self-adaptive systems in particular.

2.4.1 Uncertainty in Software Engineering

Until recently, uncertainty has been treated as a second-order concern in software engineering [EM13; Gar10]. However, challenges, such as complexity, in modern systems require uncertainty to be addressed as a first-order concern. A software system designed without considering uncertainty is likely to suffer from risks, such as technical failures, degradations, cost and schedule deviations.

Two primary sources of uncertainty are 1) lack of knowledge and 2) lack of definition [MH05]. The lack of knowledge refers to a state of having incomplete or imprecise data needed to model a system architecture and other design artifacts. The lack of definition points to a situation in which system attributes, such as functional or quality requirements, are not decided or declared precisely.

At the beginning of a software development project, the system attributes are often undefined, unknown or known only partially. As the development proceeds towards design and later stages of development, more knowledge about user requirements, business goals, and target environments become available. Thus, uncertainties caused by both the lack of knowledge and definition can be mitigated by collecting or creating (defining) knowledge. However, collecting and defining knowledge are challenging and lead to problems. For instance, defining too much about a system too early may result in a collection of imprecise or false information. Moreover, some of the required knowledge may not be available at design time and may require design decisions to be delayed until runtime.

2.4.2 Uncertainty in the Development of SASS with Reuse

Uncertainty is an intrinsic characteristic of complex systems and self-adaptive software systems are no exception to it [EM13; Gie+14; PM14]. Runtime variability

is a principal factor that leads to uncertainty in the design of self-adaptive systems. It has its roots in several areas of concern, including:

1. Functional and non-functional requirements
2. Operating environments
3. Interconnected systems
4. Market forces

The knowledge about runtime variations in these areas is either not available or partially available at design time. Due to this lack of knowledge, system developers are less able to specify requirements and model design decisions [MH05]. For instance, runtime variations in a system's operating environment cannot be predicted or known entirely and precisely at design time. Even if predicted, there remain uncertainties about when a prediction will come true and how will it impact a system. Such uncertainties caused by runtime variations challenge developers in each phase of development, particularly in requirements and design phases.

Esfahani and Malek [EM13] studied uncertainty in SASS and characterized several sources of uncertainty that challenge the confidence with which the adaptation decisions are made. The sources are identified based on FORMS [WMA10], a reference architecture for SASS, and cover a wide range of factors, such as interfaces between managed and managing systems, human in the loop, and differences between models used for decision making and actual systems. Mahdavi-Hezavehi et al. [MAW17] proposed a classification framework for uncertainty and its sources for architecture-based self-adaptive systems. The framework classifies uncertainty with respect to several dimensions and sources such as environment, goals, resources, and adaptation functions.

Several others researchers, such as [Esf11; Gie+14; PM14; Whi+10], have studied uncertainty in self-adaptive systems domain. However, none has investigated uncertainty in the context of this thesis.

2.5 Software Design

Software design is a process of transforming requirements specifications into software architecture [Bos00; TMD09] and adding details to the architecture's components to a level where implementation is straight-forward. The two levels of design, architectural and detailed design, are both concerned with decision making and aim at best decisions to provide for the requested requirements. In this thesis, we focus on software design at the architectural level and thus, describe software architecture and architectural design activities below.

The software architecture is a *"structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationship among them"* [BCK03]. The externally visible properties are the assumptions that other elements make about an element. The assumptions are generally modeled in the form of *provide* and *required* interfaces. The provide and required interfaces enable an element to interact with other elements and provide or request services from other elements [Bac+02].

Designing an architecture in principle is a decision-making activity where design decisions are modeled as architectural elements. The architectural level decisions are concerned with large parts or the whole system and require support for architectural reasoning [Dia+08]. The architectural reasoning is a complex activity of identifying design alternatives, analyzing and reasoning about the options, make trade-offs, and eventually decide, create, and structure architectural elements. The architectural reasoning for specific quality attributes can find the necessary support from a reasoning framework [Bac+05]. Unlike most functional requirements, quality attributes are system-wide and as such difficult to localize and realize in isolation [BCK03]. The design decisions made for one quality attribute typically affect one or more of the other quality attributes. In these situations, the architects must decide on trade-offs among decisions for multiple quality attributes to find a balance. The performance quality attribute, for instance, is negatively affected by almost all other quality attributes. Moreover, a quality attribute can be satisfied through many design options. For example, the availability attribute can be satisfied through multiple design options derived from the availability tactics [BCK03]. Tactics are "reusable knowledge" that focuses on system capabilities that can be used to achieve a certain quality goal. The dependencies among quality attributes and numerous design options make architectural reasoning for quality attributes a challenging task.

Traditionally, software design is confined to a distinct "design phase" in the software development lifecycle [Roy87]. The development cycle begins with requirements engineering followed by design phase. A complete design is produced at the end of the design phase and is handed to programmers for implementation. Iterative and incremental processes call for more agility and try to avoid the big design up front. Architectural and detailed design are both considered continuous activities that architects and designers perform throughout the system's lifecycle.

Chapter 3

Research Methods

This chapter introduces the design science research methodology (DSRM) [Pef+07]. We also present the research methods we used in the thesis, such as literature review, prototyping, case study, questionnaire, and interview.

3.1 Design Science Research Methodology

The design science research methodology provides researchers with a system of principles, practices, and procedures for conducting design science research in computer science, information systems, and related disciplines. The methodology was presented by Peffers et al. [Pef+07]. It provides researchers with a nominal process model for organizing research, and a mental model for structuring and presenting the research outputs.

As depicted in Figure 3.1, the DSRM process model consists of six activities:

1. Problem identification and motivation
2. Defining objectives for a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

We group the activities into three phases: 1) problem identification, 2) solution development, and 3) validation [Off+09]. The problem identification phase consists of activities 1 and 2 that urge researchers to define a specific research problem, justify its significance, and establish a set of requirements for the desired solution. The solution development phase includes only one activity "design and development" that models and develops a solution to the problem. The validation phase groups demonstration and evaluation activities. These activities are used to demonstrate and evaluate the contributed solution.

The DSRM structures activities in sequential order from activity one to six. However, there are no restrictions to change the sequence. Moreover, other research methods, such as literature review and case study, can be used for the DSRM activities. For instance, literature review and expert interviews can be used for the problem identification and motivation [Off+09].

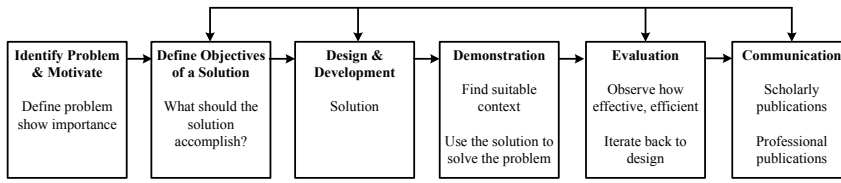

Figure 3.1: DSRM Process Model

We followed the DSRM methodology to plan, structure and organize the research project and thesis into units, including problem definition, motivation, solution, and validation. We started with problem definition, defined objectives for the desired solution and a set of research questions. Next, we designed and developed solution elements. The solution elements were validated through demonstration and evaluation. We communicated the results through a set of publications described in chapter 6. The validation output and the publications feedback helped us to revise the solution elements and objectives.

3.2 Literature Review

The literature review is a type of the review method used to explore and describe published materials about an area of study [GB09]. It enables researchers to identify what has been done previously, what needs to be done, and what are the challenges [Gra13]. Furthermore, it allows researchers to learn about a field and build upon the current knowledge while avoiding duplication.

The literature review begins by defining goals and forming a set of research questions. Next, it requires establishing a search strategy to explore bodies of literature and explore relevant literature. The search strategy may include inclusion and exclusion criteria to filter irrelevant contents and focus on specific publication or areas of research. The material found as a result of the search strategy is analyzed for the research questions. The analysis results are documented and communicated to strengthen evidence-based research [KDJ04].

We used the literature review to get an up-to-date understanding of the problem and establish the basis of the contributed solution. We used automated search to find state-of-the-art publications including journal articles, conference proceedings, technical reports, and books. Table 3.1 lists the search engines, journals and conference proceedings that were mainly used to search the literature. The literature review was performed as a continuous activity throughout the thesis to keep with the progress in the field,

3.3 Prototyping

Prototyping is an iterative development method to build an early type of a system, test and explore the early type, and rework on it until an acceptable version is achieved [Hoy+87]. The prototyping method is used for a variety of purposes

Search Engines	Journals and Conference Proceedings
IEEE Xplore	Proceedings of Symposiums on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)
ACM Digital Library	Proceedings of the Self-Adaptive and Self-Organizing Systems (SASO) Conference Series
Springer Link	Proceedings of the Systems and Software Product Line Conference (SPLC) Series
Google Scholar	Proceedings of the International Conference on Software Engineering (ICSE) Series
ScienceDirect	Proceedings of the Dagstuhl Seminars on Software Engineering for Self-Adaptive Systems (SEfSAS) Series
Elsevier	Proceedings of the International Workshop on Dynamic Software Product Lines (DSPL) Series
ISI Web of Science	ACM Transactions on Autonomous and Adaptive Systems (TAAS), ACM Transactions on Software Engineering and Methodology (TOSEM), and IEEE Transactions on Software Engineering (TSE)

Table 3.1: Search Engines, Journals and Conference Proceedings used for Literature Review

including exploration, evaluation, validation, and acceleration of the development process [Flo84; Hoy+87].

We used prototypes to explore the problem area and to demonstrate and evaluate the solution. At first, we developed three prototype systems:

1. Matrix-Multiplication
2. Sorting
3. Graph Algorithms

Matrix-Multiplication is a product line of matrix multiplication applications. It implements matrix multiplication algorithm with four variants: 1) Inline, 2) Baseline, 3 Recursive and 4) Strassen. Sorting prototype forms a product line of data sort algorithms with two variants, mergesort, and quicksort. Graph Algorithms is a product line of graph algorithms with five variants, 1) Depth First Search, 2) Breadth First Search, 3) Connected Components, 4) Strongly Connected Components, and 5) Transitive Closure.

The above prototypes were used to explore the initial concept of the ASPL [AAL10; AAW18], a part of the solution, and underlying challenges. Three experiments were conducted to evaluate and test the ASPL for variability handling and online learning and knowledge sharing mechanism. See [AAL11] for further details about the prototypes and the experiments.

The prototypes helped us to identify a need for explicit process support for variability and uncertainty handling in the design and development of SASS with reuse. We addressed the need by reforming the ASPL concept into a systematic ASPL strategy [AA15] with well-defined process support [AAW18; AJ15].

The revised ASPL strategy and process support, the ASPLe methodology, were validated by designing three prototypes. The prototypes were designed as product lines of self-adaptive systems using process support offered by the ASPLe. Table 5.1 lists three application domains, one for each prototype, and required self-adaptation properties. The application domains and self-adaptation properties were used as problem domains to demonstrators and validate the use of the AS-

PLe in practice. The publication TR2 [AA17] describes the example application domains and how we used them to demonstrate the ASPLe methodology.

3.4 Case Study

A case study is a research method used for in-depth investigation of contemporary phenomena in their natural context [R+12]. The investigated phenomena are commonly known as *cases* of a study. Based on research purpose, there are three main types of case studies:

1. Exploratory case study
2. Descriptive case study
3. Explanatory case study

An exploratory case study explores one or more phenomena of interest to find out what is happening, seek new insights, and provoke ideas for future research. A descriptive case study examines one or more cases to understand and describe their current status. The primary objective of such case studies is to characterize studied cases. An explanatory case study examines data or phenomena deeply and thoroughly to explain a situation or problem, generally but not necessarily, in the form of a causal relationship.

We performed a case study for two purposes:

1. to explore and validate the ASPLe methodology, principal contribution of the thesis.
2. to collect user experiences and feedback for improvement of the ASPLe.

We used a planning template defined by Wohlin et al. [W+12] to design and plan the case study activities. The case studies involved both quantitative and qualitative data. The data were collected using mixed data collection methods including test assignments, questionnaires, and interviews. The mixed methods were used to strengthen the case study findings by collecting data using different ways at different occasions, i.e., triangulation. Below is an overview of the data collection methods.

3.4.1 Questionnaire

The questionnaire is a research method used to collect data by asking a set of questions in a pre-determined order [Gra13]. There are two types of questions, 1) open questions, and 2) closed questions. The open questions have no definite answer and allow respondents to answer in detail. The closed questions limit the answers to a set of pre-defined replies, such as yes/no, and multiple-choices. Both types have merits and demerits, for instance, the open questions have potential to collect more detailed and in-depth data, but are hard to analyze.

We used the questionnaire method with closed type questions to collect and analyze the data to compare the eARF, a core part of the ASPLe, with a reference approach. The data were collected from subjects who participated in the case

study and used both the eARF and the reference approach to design self-adaptive software systems with reuse. We asked closed questions to get data suitable for comparison. However, using closed questions prevented respondents from providing details. We addressed this limitation with interviews.

3.4.2 Interview

An interview is a data collection method in which one or more persons, the interviewers, attempt to inquire and record information from other persons, the interviewees [Gra13]. It enables researchers to ask direct questions, observe interviewees behavior and body language, and adapt questions. The nonverbal cues often help to understand verbal responses better.

There are three primary types of the interview method: 1) structured, 2) semi-structured, and 3) unstructured interviews [Gil+08]. A structured interview is like a verbally administered questionnaire in which predefined questions are asked with little or no variations. There is minimum interaction, other than questions and responses between interviewer and interviewees and there is no scope for follow-up questions. A semi-structured interview is based on predefined questions but allows the interviewers to rephrase, add or remove, and adapt the questions based on interviewees' responses. It enables researchers to reflect, probe and ask follow-up questions. An unstructured interview is performed with no predefined questions and little or no organization. It begins with an opening question and lets the conversation to develop based on interviewees' responses.

In this research project, we used the interview method, as a part of the case study, to explore and evaluate the eARF and the design support provided by the ASPLe. We did semi-structured interviews with subjects who participated in the case study to clarify the questionnaires data and to probe and collect additional details.

Chapter 4

Results - Thesis Contributions

This chapter presents the thesis claimed contributions. We classify contributions into two groups, 1) primary results and 2) secondary results. The primary results group major contributions of the thesis. The secondary results group intermediary outcomes of the thesis. To report the results, we use a common structure comprising:

- The research questions addressed
- the objectives satisfied
- a description of results
- the research methods used
- the publications that communicate the results

4.1 Primary Results

4.1.1 PR1: Variability and Uncertainty Analyses

Research Questions The variability and uncertainty analyses answer research questions Q. 1(a) and Q. 1(b). The Q. 1(a) is answered by identifying and analyzing challenges raised by variability and uncertainty in the design of SASS with reuse. The results of the analyses provide insight to address the challenges, i.e., answers Q. 1(b).

Objectives The analyses contribute to objective O1.

Research Methods We used literature review and prototyping methods for both variability and uncertainty analyses. Beginning with literature review, we explored the state-of-the-art journals and conference proceedings in software reuse, product lines, self-adaptive software systems and autonomic computing research communities. The literature review was used to identify principal issues, existing work, and proposed solutions. The findings drawn from the literature review were tested and refined by developing six prototypes.

Results Variability handling and uncertainty mitigation are two principal challenges in the design and development of self-adaptive software systems with reuse. We analyzed both variability and uncertainty to identify their root causes, gain better insight and to provide developers with systematic process

support to address the challenges. Both the analyses were performed from system designer's perspective.

The development of self-adaptive systems with and for reuse involves three types of variability [AA15; AJ15]:

1. Domain variability
2. Cross-domain variability
3. Runtime variability

We analyzed each variability type for reuse at three levels, 1) single system, 2) vertical platform, and 3) horizontal platform [Pri93]. The analysis was structured and performed according to variability handling framework defined by Svahnberg et al. [SVB05].

The analysis showed that reuse at single system level involves runtime variability, i.e., there are no domain and cross-domain variabilities. The reuse at vertical platform level involves runtime and domain variabilities, whereas the reuse at horizontal platform level involves all three types of variabilities. The managed and managing subsystems of a self-adaptive system form two distinct domains. Thus, the domain and cross-domains variabilities for the managed and managing system domains can be identified and specified separately, but cannot be constrained independently due to dependencies between the managing and managed system domains. The runtime variability is hard to predict, identify and constrain. It is caused due to runtime variation in systems' environment, goals, and system themselves, which may change at runtime. We do not know precisely when, where and how a change may occur. For the three reuse levels, runtime variability is easier to identify, constrain and manage at a single system level. This is because systems' requirements, environments, and user goals with runtime variations are better known at the single system level than at the vertical and horizontal platform levels. The horizontal platform is most challenging because here the developers do not know application domains (managed systems) and their application requirements, goals, and environments precisely.

Uncertainty in the design of self-adaptive systems with reuse was analyzed using Ishikawa fishbone diagram [Ish86] and uncertainty handling framework depicted in Figure 2.3. The uncertainty analysis revealed two primary factors causing uncertainty in the design of SASS with reuse:

1. Runtime variability
2. Development for reuse

The first factor, runtime variability, has roots in several areas of concern including functional and non-functional requirements, operating environments, interconnected systems, and market forces. The knowledge about runtime variations in these areas of concern is not available at design time. The lack of knowledge leads to uncertainties in system design. Uncertainties induced by lack of knowledge can be addressed with time and effort, for instance by collecting knowledge [MH05], however, collecting complete and precise

knowledge about runtime variability is not feasible. Runtime variations at best can be predicted, however, even if predicted, there are no guarantees whether the predictions will come true or not. The analysis showed that a suitable strategy to mitigate uncertainties caused by runtime variability is to delay design activities such as design decisions [VBS01] until the time, e.g., runtime, when complete or more knowledge becomes available.

The second factor, development for reuse, requires designers to design artifacts for reuse in several applications or application domains. While designing such artifacts for reuse, knowledge about target application or application domains is usually missing or available only partially. The lack of knowledge leads to uncertainties that may lead to technical debt.

Based on the combined results of the variability and uncertainty analyses, we conclude that software variability and uncertainty have a bidirectional cause-effect relationship. The two-way cause-effect relationship means uncertainty may lead to variability and vice versa. The uncertainty leads to variability in systems design when multiple design alternatives are available, and a designer is not sure about alternatives that need to to be selected. On the other side, the different types of variability may lead to uncertainty. For instance, consider an application domain that differs in its requirements for user authentication. Some of the domain applications require simple user-id/password authentication, while others require a one-time password with an option to switch between email and SMS based authentication at runtime. Such differences in product requirements both at design and runtime result in uncertainty in architectural analysis and decision making.

Papers Details about the variability and uncertainty analyses are given in publications P2 [AA15] and P4 [AAW18], respectively.

4.1.2 PR2: Autonomic Software Product Lines (ASPL) Strategy

Research Questions The ASPL provides the basis to answer the research questions 2(a) and 2(b).

Objectives The ASPL satisfies the objective O2 by defining systematic reuse strategy with separation of managed and managing subsystem concerns.

Research Methods We used literature review and prototyping methods to define and validate the ASPL strategy. The literature review was used to explore existing methods and tools that support development with reuse. The prototyping was used for preliminary evaluation of the ASPL. The results derived from the prototypes and the knowledge gained from literature review helped to reform the initial idea of the ASPL into a systematic reuse strategy.

Results The ASPL is a multi product-line based strategy to design and develop self-adaptive systems. The essence of the ASPL strategy is the separation of the managed and the managing system concerns. The ASPL exploits this separation by establishing an application domain independent platform for managing system artifacts and reusing the platform to realize self-management properties across a set of managed system domains.

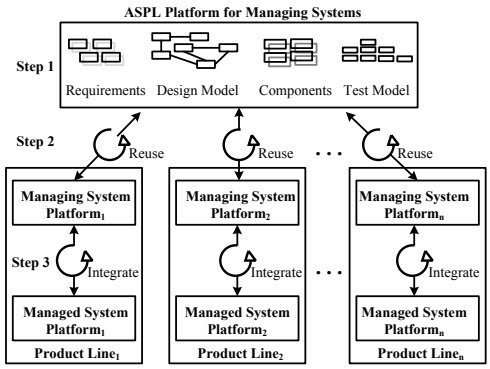

Figure 4.1: The ASPL Strategy

As shown in Figure 4.1, the ASPL strategy is composed of three steps summarized below:

Step 1: Establish an ASPL Platform To support reuse across several domains, the first step is to establish a horizontal ASPL platform. The ASPL platform targets adaptation logic and provides application domain-independent artifacts for managing systems. To support reuse across several domains, the platform artifacts are systematically defined independent of any application domain.

Step 2: Derive Managing System Platform(s) from the ASPL Platform The second step of the ASPL strategy is to transform the horizontal ASPL platform into a vertical (application domain-specific) managing system platform. To support reuse across several domains, any number of managing system platforms may be derived from a single ASPL platform. Each of the derived platforms targets adaptation logic in a specific domain and provides reusable artifacts to realize managing systems in that domain.

Step 3: Integrate Managing and Managed System Platform(s) The third step of the ASPL is to integrate the managing system platform, derived from step 2, with an independently developed managed system platform. This step is required because a managing system platform targets adaptation logic only. The application logic is developed separately in the form of a managed system platform. As the managed and managing system platforms are developed separately, there exist mismatches between the platforms. The mismatches are analyzed and addressed in this step so that the artifacts from the managed and managing platforms can be used to derive a product line of self-adaptive systems.

Papers The publication P1 [AAL10] and TR1 [AAL11] introduce the initial concept of the ASPL. The publication P2 [AA15] and TR2 [AA17] describe the reformed ASPL strategy.

4.1.3 PR3: Autonomic Software Product Lines engineering (ASPLe) Methodology

Research questions The ASPLe answers the research questions 2(a) and 2(b).

Objectives The ASPLe contributes to the objectives O1 and O2.

Research Methods We used literature review, prototyping, and case study methods to define and validate the ASPLe methodology. The literature review helped us to comprehend domain engineering and application engineering processes [PBV05]. The two processes, particularly the domain engineering, form the basis of the ASPLe. We developed prototypes and performed a case study to evaluate the ASPLe and collect user experiences to improve the ASPLe.

Results The ASPLe is a software development methodology with process support to design and develop self-adaptive systems with systematic reuse. Aligned with the ASPL strategy, the ASPLe is composed of three principal processes: 1) ASPL Domain Engineering, 2) Specialization, and 3) Integration. We provide an overview of the processes below.

ASPL Domain Engineering Process The ASPL Domain Engineering (ADE) process defines activities, work-products, and roles to establish a horizontal ASPL platform. It is composed of requirements engineering, design, implementation, and testing subprocesses. The ASPL requirements engineering scopes the ASPL platform and specifies application domain-independent requirements for self-adaptation. The requirements are then mapped to a reference architecture by the ASPL design subprocess. The reference architecture models high-level design decisions with variation points and variants, i.e., variability, to support reuse across multiple domains. The ASPL implementation subprocess provides guidelines to transform reference architecture into reusable code components or libraries. The ADE ends with ASPL testing subprocess, which provides guidelines to produce reusable test artifacts to validate and verify the reusable code components produced by the ASPL implementation.

Specialization Process The specialization process defines activities, work-products, and roles to transform a horizontal ASPL platform into a vertical managing system platform. Following the ADE process structure, the specialization process is also composed of requirements, design, implementation, and tests specialization subprocesses. Instead of developing artifacts from scratch, each specialization subprocess searches for reusable artifacts in the ASPL platform and specializes the found artifacts according to needs of a given application domain. For instance, the requirements specialization subprocess searches the ASPL platform to find requirement specifications that match requirements of a given application domain. The found requirement specifications are analyzed and customized according to needs of the given domain. All the subsequent specialization subprocesses follow the same workflow.

Integration Process The integration process defines activities, work-products and roles to align and integrate a managing system platform with a separately developed managed system platform. In line with the ADE and specialization processes, the integration process is composed of requirements, design, implementation, and testing subprocesses. The general workflow for each integration subprocesses is same. Each integration subprocess begins with analysis activity and provides developers with guidelines to identify mismatches in the managed and managing system platforms. The analysis activity is followed by an integration activity that addresses the identified mismatches, for instance, by adding, removing or modifying requirements and other development artifacts. The integration process comes to an end with an assurance that the artifacts in the managed and managing system platforms can be used together to derive a product line of self-adaptive systems.

Papers The publication P2 [AA15] introduces the ASPLe. The publications P4 [AAW18] and TR2 [AA17] provide the detail description of the ASPLe.

4.1.4 PR4: extended Architectural Reasoning Framework (eARF)

Research questions The eARF answers question Q. 1(b) by providing architectural knowledge and reasoning support to address uncertainty and variability. The eARF also partially answers question Q. 2 by complementing the ASPLe methodology with reasoning support.

Objectives The eARF satisfies the objective O3 by building upon existing design principles and methods.

Research methods We used literature review and case study methods to define and evaluate the eARF.

Results The extended Architectural Reasoning Framework is a body of knowledge composed of proven best design methods, architectural practices, and templates for requirements and design artifacts. The uncertainties and variabilities induced by systematic reuse and self-adaptation complicate architectural reasoning in the ASPLe design processes. To that end, the eARF complements the ASPLe with knowledge and reasoning support in the form of self-management properties driven architectural tactics, patterns, and design methods. Initially, the eARF was composed of five elements: 1) Quality Attribute Scenario (QAS), 2) domain QAS, 3) domain Responsibility Structure, 4) Architecture Patterns, and 5) Architecture Tactics. The eARF was later enhanced with an analytical framework to address the lack of rigor for architectural analysis and reasoning [Abb+16]. Below is an overview of the eARF elements.

Quality Attribute Scenario (QAS) A quality attribute scenario (QAS) is a quality attributes requirements specification template defined by Bass et al. [BCK03]. It consists of six elements: stimulus, source, environment, artifact, response and response measure.

domain QAS (dQAS) A dQAS is an extended form of the QAS. The QAS provides a basic structure for characterizing quality attributes. However, it lacks in explicit support to specify quality attributes with variability at an application domain level. To that end, we extended the standard QAS template to the domain Quality Attribute Scenario [AAW12].

domain Responsibility Structure (dRS) A dRS is an architectural representation of design decisions made to realize a self-management property [AA13; AJ15]. It consists of two parts: 1) responsibility part, and 2) variability part. The responsibility part is defined by extracting a set of responsibilities from a dQAS and mapping the responsibilities to architectural elements. The elements are called responsibility components. The variability part is defined by defining variation points and variants for the responsibility components.

Architectural Tactics The tactics are the design options being using for years to realize quality attributes [BCK03]. The eARF recommends the use of tactics to identify and reason about design options for self-management properties specified in the form of dQASs.

Architectural Patterns An architectural pattern expresses a fundamental structural organization schema for software systems [BCK03]. The responsibilities extracted for a self-management property are likely to fall in monitoring, analysis, planning, and execute categories. Thus, the eARF recommends the use of Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) feedback loop [KC03; Wey+13] as a principal pattern to structure and model the responsibilities as architectural elements of the dRS.

Analytical Framework The analytical framework provides designers with rigorous analytical means to verify the architectural models, such as dRS. It describes activities to transform dQASs to architecture models, specify desired properties, and evaluate the models for the properties using a model checker [Abb+16].

Papers The eARF is introduced in additional publication AP4 [AA13] and is validated and explained with more details in the appended publication P3 [AJ15]. The QAS and dQAS elements of the eARF are described in the additional publication AP3 [AAW12]. The additional publication AP5 [Abb+16] introduces and exemplifies the analytical framework element.

4.2 Secondary Results

4.2.1 SR1: Online Learning and Knowledge Sharing

Research questions The online learning and knowledge sharing answer the questions Q. 2(a) and Q. 2(b) partly by providing support mechanisms for knowledge evolution and reuse.

Objectives The online learning and knowledge sharing contribute to satisfying the objective O1.

Research methods We used the prototyping method to formulate the online learning and knowledge sharing mechanisms.

Results The knowledge available at design time of self-adaptive systems may become obsolete at runtime due to changes in design parameters, such as environment and business goals. Thus, a dynamic knowledge evolution mechanism is required to update the knowledge. To that end, we defined two mechanisms: 1) online learning and 2) knowledge sharing.

Online Learning Online learning is a dynamic knowledge evolution mechanism that updates knowledge-base used by managing systems to analyze and plan adaptation actions. It consists of three sequential phases: 1) instrumentation, 2) monitored execution, and 3) learning. The *instrumentation* phase begins when an instrumentation-monitor detects an active managing system. The instrumentation-execute preempts the managing system and takes control of adaptations in an underlying managed system. It executes a learning strategy, for instance, reinforcement learning [KLM96], and triggers monitored execution phase. The *monitored-execution* phase uses a monitor component to record outcomes of the learning strategy. The recorded properties and measures are used in the learning phase to update the knowledge-base. The *learning* phase analyzes the knowledge-base in comparison with the recorded data and updates the knowledge-base based on results of the analysis.

Online Knowledge Sharing Online knowledge sharing is a mechanism to exchange knowledge and learn from each other's experiences at runtime. It enables software systems to self-optimize and improve quality faster by capitalizing on knowledge derived by other software systems. Based on communication mode, we distinguished two kinds of knowledge sharing: 1) direct exchange and 2) indirect exchange. In the direct exchange, software systems share knowledge directly with their partners, such as similar applications in an application domain. In the indirect exchange, software systems share knowledge with each other through a middleware, such as a knowledge broker or manager. Different communication styles such as "push", "pull", and "broadcast" can be used for both the direct and indirect knowledge exchange mechanisms.

Papers The additional publication AP2 [AAW11] describes the online learning and knowledge sharing mechanisms.

4.2.2 SR2: Educational Package

Research questions The educational package does not answer any of the research questions directly. It is a by-product of the thesis effort.

Objectives The educational package does not contribute to any objectives of the solution. However, it adds value to the thesis by providing resources for teaching and experimenting with self-adaptive systems.

Research methods The educational package was defined as a result of the case study design exercises that were defined and used to evaluate primary results of the thesis.

Results The educational package is a set of academic resources that can be used in a classroom environment to teach and experiment with self-adaptive software systems. We used the package in a two years master degree program, but it can be adopted for undergraduate programs as well. The package consists of following items:

Lecture Notes A set of lecture slides prepared to introduce and describe the ASPL strategy and the ASPLe methodology. The slides are organized into two lectures, each lecture prepared for a 3 hours time slot.

Reading Assignment A home assignment that requires students to study a set of articles and textbook chapters distributed as reading material. As a solution, students are required to send a summary with critical reflection on the reading material.

Technical Report A technical report that extensively describes and demonstrates the ASPLe methodology.

Example Application Domains Four example application domains that require three self-management properties: 1) self-healing, 2) self-optimization, and 3) self-upgradability.

Workshops and Test Assignments The educational package contains a set of workshops and test assignments to demonstrate and test the use of the ASPLe methodology in practice. The workshops are designed to refresh design skills and to prepare the students for test assignments. There are two test assignments provided with solutions. Each assignment specifies tasks with requirements for self-adaptation and requires to analyze and map requirements to design artifacts using the ASPLe methodology.

Reusable Artifacts The package provides a set of application domain independent artifacts created for reuse in several applications and application domains. Following the ASPLe methodology, the reusable artifacts are collected as ASPL Platform for managing systems.

Questionnaires A set of questionnaires with closed type questions. The questionnaires are designed to measure learning outcomes and collect user experiences and feedback on the use of the ASPLe methodology.

Papers The technical report TR2 describes and demonstrates the ASPLe methodology. The educational package can be downloaded at `http://homepage.lnu.se/staff/janmsi/sass-edu/`.

Chapter 5

Validation

This chapter introduces the activities we conducted to validate the contributions. In line with the design science research methodology [Pef+07], we used demonstration and evaluation methods for validation.

5.1 Validation by Demonstration

A demonstration exhibits the operation or use of an artifact, such as program, process, methodology, or the like, to prove that the artifact works and solves a problem [Pef+07]. The main resource required for the demonstration method is *knowledge of how to use the validated artifact*. The demonstration method is simple and easy to use, however, lacks in rigor and formal evaluation.

We used the demonstration method to show that the ASPLe methodology works in practice and actually provides developers with support to handle variability and mitigate uncertainties in the development of self-adaptive systems [AA17]. Beginning with the first ASPLe process, ASPL Domain Engineering, we demonstrated the use of the ASPL requirements and design subprocesses to develop an example ASPL platform. The scope of the example platform was constrained to two self-adaptation properties, self-upgradability and self-optimization. For each property, we defined application domain independent requirements and design artifacts using the ASPL requirements and design processes.

Next, we demonstrated the use of the Specialization process to derive managing system platforms for example application domains listed in Table 5.1. Each of the managing system platforms was established with reuse from the example ASPL platform. We used the requirements and design specialization subprocesses to reuse the requirements and design artifacts from the example ASPL platform and specialize them for self-adaptation properties required by the example application domains.

Then, we demonstrated the use of the Integration process to integrate each managing system platform, derived using the specialization process, with an independently developed managed system platform. The requirements and design artifacts in the managed system platforms were produced using conventional methods, such as use case scenarios and architectural views. We performed the requirements and design integration processes to integrate the requirements and design artifacts, respectively. See [AA17] for further details about how we performed the Integration and other ASPLe processes.

Application Domains	Required Properties
Distributed Game Environment	Self-Upgradability
News Service Product Line	Self-Optimization, Self Healing
PhotoShare Product Line	Self-Upgradability, Self-Healing

Table 5.1: Example Application Domains

5.2 Validation by Evaluation

An evaluation is a systematic study performed to observe and measures how well a proposed solution addresses a problem [Pef+07]. The evaluation studies are classified into two distinct groups: 1) formative and summative evaluations, 2) outcome and process evaluations [Rob93]. A formative evaluation aims to help in the development of an artifact, for example, program, process or method. A summative evaluation focuses on assessing results and effectiveness of the artifact being evaluated. An outcome evaluation measures to what extent the evaluated artifact meets its claimed goals or objectives. A process evaluation observes an artifact being evaluated and answers the question "what happens, and how?".

Different research methods, such as experiments, case study, questionnaires, and interviews, can be used to perform an evaluation. We used the case study method to evaluate the ASPL strategy and the ASPLe methodology, which are primary contributions of the thesis. The case study's primary objective was to assess the ASPLe design subprocesses and architectural analysis and reasoning support provided by eARF part of the ASPLe. We compared the design support provided by the ASPLe and the eARF with a state of the art reference approach. The comparison was made for two principal goals, *maximizing total reuse* and *mitigating uncertainties* in system design. The data, for comparison, were collected using mixed methods including controlled experiment, questionnaires, and interviews. The data were analyzed using hypothesis testing, descriptive statistics, and graphical visualizations. The results of the analysis show that the ASPLe provides systematic process support to design self-adaptive systems with reuse and mitigate uncertainties. In comparison to the reference approach, the ASPLe and the eARF enabled the developers to model self-adaptive systems with a statistically significant increase in total reuse and decline in fault-density. The decrease in fault-density indicates that the ASPLe provides better support to mitigate uncertainties in the design of self-adaptive software systems with reuse. See the publication P4 for details about the case study design, data collection, analysis, and results.

We used the case study for two purposes. First, to assess the ASPLe for its target goals, maximizing total reuse and mitigating uncertainty. Second to get feedback from subjects and use the feedback to refine and enhance the ASPLe. Hence, we classify our case study evaluation as a mix of formative, summative and outcome evaluation.

Chapter 6

Publications

This chapter introduces publications used to communicate the thesis findings. The publications are classified into three groups:

1. Appended publications
2. Additional publications
3. Technical reports

The appended publications report primary results to address the research questions. The additional publications communicate secondary findings that helped to achieve the primary results. The technical reports provide a detailed description of the solution elements, the ASPL and the ASPLe.

6.1 Overview of Appended Publications

Appended Publication - P1

Abbas N, Andersson J, Löwe W. Autonomic software product lines (ASPL). In Proceedings of the Fourth European Conference on Software Architecture (ECSA'10), Copenhagen, Denmark, Companion Volume, Pages 324-331. ACM, 2010.

Summary: This publication introduces the initial concept of the Autonomic Software Product Lines (ASPL). It presents the ASPL as a dynamic software product line with variability handling mechanism. The variability handling mechanism enables a software product line to adapt itself at runtime in response to variations in its context, resources, and goals.

The variability handling mechanism comprises three activities:

1. Offline Training
2. Context-Aware Composition
3. Online Learning

We describe and exemplify all the activities using two scenarios: 1) open world, and 2) closed world. The closed world scenario represents a class of systems with a fixed set of resources and contexts, whereas the open world scenario represents a class of systems with variable resources and settings.

Appended Publication P2

Abbas N, Andersson J. Harnessing variability in product-lines of self-adaptive software systems. In Proceedings of the 19th International Conference on Software Product Line (SPLC'15), Nashville, Tennessee, Pages 191-200. ACM, 2015.

Summary: This publication reports variability analysis in the context of designing and developing self-adaptive software systems (SASS) with reuse. It argues variability handling as a key to accomplish systematic reuse. The development of SASS with reuse involves variability across three dimensions: 1) domain variability, 2) cross-domains variability and 3) runtime variability. We analyze the variability dimensions for four activities: 1) identify variability, 2) constrain variability, 3) implement variability, and 4) manage variability. The analysis pinpoints opportunities and challenges. The challenges are primarily caused due to the complex interaction of variabilities across three dimensions and require a systematic approach to address them. To that end, this publication introduces the ASPLe methodology as a framework. The framework consists of three processes that form the basis of a systematic approach to address the challenges identified as a result of the variability analysis.

Appended Publication P3

Abbas N, Andersson J. Architectural Reasoning Support for Product-Lines of Self-adaptive Software Systems-A Case Study. In: Weyns D., Mirandola R., Crnkovic I. (eds) Software Architecture. Lecture Notes in Computer Science, Volume 9278, Pages 20-36. Springer, 2015.

Summary: This publication emphasizes the role of software architecture in the development of large and complex software systems. It reports a problem of architectural analysis and reasoning in the context of this thesis and presents the extended Architectural Reasoning Framework (eARF) to address the problem. It describes elements of the eARF and defines a workflow to use these elements. We use an example application domain to illustrate the use of the eARF for analysis and design. Moreover, the publication reports a feasibility case study to validate the reasoning framework. Based on results of the case study, we conclude that the eARF provides better support to analyze and reason about design alternatives while designing SASS with reuse.

Appended Publication P4

Abbas N, Andersson J, Weyns D. ASPLe: a methodology to develop self-adaptive software systems with systematic reuse. In: Submitted to Software & Systems Modeling, 2018.

Summary: This publication describes uncertainty, variability, and lack of process support as principal problems in the development of self-adaptive software systems with reuse. It motivates the need for a separation of concerns based strategy to mitigate complexity and uncertainty and presents the ASPL strategy as an enhanced form of the ASPL approach introduced in the publication P1. We argue that implementation of the ASPL strategy involve

uncertainties in system analysis and design, and requires well-organized process support. To that end, the publication presents the ASPLe methodology with a focus on design activities. The ASPLe encapsulates systematic process support to implement the ASPL strategy and address underlying challenges of uncertainty and variability handling. We demonstrate the use of the ASPLe methodology for an example application. We also report an extensive case study performed to evaluate the ASPLe.

6.2 Additional Publications

Additional Publication AP1

Abbas N, Towards autonomic software product lines. In Proceedings of the 15th International Software Product Line Conference (SPLC'11), Munich, Germany, Volume 2, Article 44, 8 pages. ACM, 2011.

Abstract: "We envision an Autonomic Software Product Line (ASPL). The ASPL is a dynamic software product line that supports self-adaptable products. We plan to use reflective architecture to model and develop ASPL. To evaluate the approach, we have implemented three autonomic product lines which show promising results. The ASPL approach is at initial stages and requires additional work. We plan to exploit online learning to realize more dynamic software product lines to cope with the problem of product line evolution. We propose online knowledge sharing among products in a product line to achieve continuous improvement of quality in product line products".

Additional Publication AP2

Abbas N, Andersson J, Weyns D. Knowledge evolution in autonomic software product lines. In Proceedings of the 15th International Software Product Line Conference (SPLC'11), Munich, Germany, Volume 2, Pages 36:136:8. ACM, 2011.

Abstract: "We describe ongoing work in knowledge evolution management for autonomic software product lines. We explore how an autonomic product line may benefit from new knowledge originating from different source activities and artifacts at runtime. The motivation for sharing run-time knowledge is that products may self-optimize at runtime and thus improve quality faster compared to traditional software product line evolution. We propose two mechanisms that support knowledge evolution in product lines: online learning and knowledge sharing. We describe two basic scenarios for runtime knowledge evolution that involves these mechanisms. We evaluate online learning and knowledge sharing in a small product line setting that shows promising results".

Additional Publication AP3

Abbas N, Andersson J, Weyns D. Modeling variability in product lines using domain quality attribute scenarios. In Proceedings of the WICSA/ECSA 2012, Helsinki, Finland, Companion Volume, Pages 135-142. ACM, 2012.

"The concept of variability is fundamental in software product lines and successful implementation of a product line largely depends on how well domain

requirements and their variability are specified, managed, and realized. While developing an educational software product line, we identified a lack of support to specify variability in quality concerns. To address this problem, we propose an approach to model variability in quality concerns, which is an extension of quality attribute scenarios. In particular, we propose domain quality attribute scenarios, which extend standard quality attribute scenarios with additional information to support specification of variability and deriving product specific scenarios. We demonstrate the approach with scenarios for robustness and upgradability requirements in the educational software product line".

Additional Publication AP4

Abbas N, Andersson J. Architectural reasoning for dynamic software product lines. In Proceedings of the 17th International Software Product Line Conference (SPLC'13) Co-located Workshops 2013, Tokyo, Japan, Pages 117-124. ACM, 2013.

Abstract: "Software quality is critical in today's software systems. A challenge is the trade-off situation architects face in the design process. Designers often have two or more alternatives, which must be compared and put into context before a decision is made. The challenge becomes even more complex for dynamic software product lines where domain designers have to take runtime variations into consideration as well. To address the problem, we propose extensions to an architectural reasoning framework with constructs/artifacts to define and model a domain's scope and dynamic variability. The extended reasoning framework encapsulates knowledge to understand and reason about domain quality behavior and self-adaptation as a primary variability mechanism. The framework is demonstrated for a self-configuration property, self-upgradability on an educational product-line".

Additional Publication AP5

Abbas N, Andersson J, Iftikhar MU, Weyns D. Rigorous architectural Reasoning for Self-Adaptive Software Systems. In Proceedings of the 1st Workshop on Qualitative Reasoning about Software Architectures (QRASA'16), Venice, Italy, Pages 11-18. IEEE, 2016.

Abstract: "Designing a software architecture requires architectural reasoning, i.e., activities that translate requirements into an architecture solution. The architectural reasoning is particularly challenging in the design of product-lines of self-adaptive systems, which involve variability both at development time and runtime. In previous work, we developed an extended Architectural Reasoning Framework (eARF) to address this challenge. However, evaluation of the eARF showed that the framework lacked support for rigorous reasoning, ensuring that the design complies with the requirements. In this paper, we introduce an analytical framework that enhances eARF with such support. The framework defines a set of artifacts and a series of activities. Artifacts include templates to specify domain quality attribute scenarios, concrete models, and properties. The activities support architects with transforming requirement scenarios to architecture models that comply with

required properties. Our focus in this paper is on architectural reasoning support for a single product instance. We illustrate the benefits of the approach by applying it to an example client-server system and outline challenges for future work".

6.3 Technical Reports

Technical Report TR1

Abbas N, Andersson J, Löwe, W. Towards autonomic software product lines (ASPL) - a technical report. Tech. rep. Linnaeus University, Department of computer science and media technology (CM), p. 20, 2011.

Summary: This publication provides a detailed description of the initial concept of the ASPL strategy. It describes software product line engineering and autonomic computing as a background of the ASPL. The publication presents the need for runtime variability handling mechanisms as a fundamental motivation for the ASPL strategy. We define the ASPL as a dynamic software product line [Hal+08] and describe three core activities that work together to provide for self-adaptation. Each of the three activities is discussed using FORMS primitives [WMA12].

We also report three experiments conducted to evaluate the ASPL approach. The experiments were performed using three prototypes systems: 1) Matrix-Multiplication, 2) Sorting, and 3) Graph Algorithms. Moreover, we discuss and position the ASPL approach in connection with related works.

Technical Report TR2

Abbas N, Andersson J. ASPLe A Methodology to Develop Self-Adaptive Software Systems with Reuse. Tech. rep. Linnaeus University, Department of computer science and media technology (CM), p. 118, 2017.

Summary: This publication complements the appended publication P4 with an extensive description of the ASPLe methodology and its process activities. An introduction to the ASPL strategy and the ASPLe methodology is presented followed with a detailed description of the three ASPLe processes: 1) ASPL Domain Engineering, 2) Specialization, and 3) Integration. An example application domain that requires a self-upgradability property is used to demonstrate each of the three processes. In line with the thesis research scope, the scope of the described processes is limited to the requirement and design engineering.

Chapter 7

Conclusion

This chapter summarizes the thesis, discusses its findings and contributions, points at some limitations, and outlines directions for future research.

7.1 Concluding Remarks

This thesis studies development of self-adaptive software systems with systematic reuse. Self-adaptation has been recognized as an essential property to manage complexity and runtime variations in software systems [De +13]. There exist a vast body of knowledge on engineering self-adaptive systems. However, there is lack of process support to design and develop self-adaptive software systems with reuse. The thesis targets this lack of process support as a principal problem. We identified two research questions, restated below for clarity.

Q. 1 What is the current state of the art in design and development of self-adaptive software systems with reuse?

- **(a)** What are the challenges confronted by system designers while designing SASS with systematic reuse?
- **(b)** How can the challenges identified in question 1(a) be addressed?

Q. 2 How can self-adaptive software systems be designed with systematic reuse in an application domain and across several application domains?

- **(a)** How can we design and develop generic application domain independent artifacts for reuse across several application domains of SASS?
- **(b)** How can the generic reusable artifacts be specialized for reuse in a specific application domain of SASS?

The thesis answers Q. 1(a), by identifying and analyzing variability and uncertainty in the context of software reuse and self-adaptive software systems. The analysis shows that the following three variability dimensions are the primary causes of the lack of knowledge and induced uncertainties:

1. Domain variability
2. Cross-domain variability
3. Runtime variability

The analysis results show that managing variability and mitigating uncertainty in this context is an overwhelming challenge for system developers. To that end, the thesis contributes separation of concerns based ASPL strategy and process support with purposefully defined activities and work-products to address the challenge. This answers question Q. 1(b).

The ASPL strategy also provides a theoretical foundation that answers Q. 2(a) and Q. 2(b) partially. It describes how the development artifacts can be designed and specialized for reuse across several domains of self-adaptive systems. It is necessary but not sufficient to answer Q. 2 completely. The Autonomic Software Product Lines engineering (ASPLe) methodology answers the practical part of Q. 2(a) and Q. 2(b). The methodology provides process support to realize the ASPL strategy and consists of three steps. The first step is a development of an application domain-independent platform for managing systems. The second step is a transformation of the domain-independent platform to domain-specific platforms for reuse across several domains. The third step is the integration of the domain-specific platforms with independently developed managed system platform.

We specified three objectives to steer the thesis effort and validate the solution, i.e., the ASPL and the ASPLe.

- The objective O1 directed the solution to be a methodology with organized and repeatable process support.
- The objective O2 required the solution to be founded on a well-defined strategy with separation of managed and managing system concerns.
- The objective O3 specified use of existing practices and knowledge, such as variability management mechanisms and design patterns, to develop the solution.

We developed three example application domains and conducted a comprehensive case study to demonstrate and validate the solution elements, the ASPL, ASPLe, and eARF. In the case study, we used controlled experiment, questionnaire, and interview methods to evaluate the solution. The case study data were analyzed using hypothesis testing, descriptive statistics, and graphical visualizations. The use of the ASPLe to produce example systems and results of the case study show that the contributed solution elements satisfy all the three objectives.

The thesis meets the objective O1 by providing a well-documented and demonstrated methodology, the ASPLe. The case study data show that the ASPLe helps to mitigate uncertainties, manage variabilities and design self-adaptive systems with a higher degree of total reuse and reduced fault density. The reduced fault-density and higher degree of reuse enable developers to produce systems with improvement in quality and reduction in cost and time to develop and market.

The thesis meets the objective O2 by defining the separation of concerns based strategy, the ASPL, to address the problem. The ASPL maintains a clear separation between managed and managing systems The ASPLe methodology, which provides process support for the ASPL, also preserves the separation of managing and managed systems in its processes. For the separation of concerns, about 95% subjects who participated in the case study rated the ASPLe better or equal to current engineering practices.

The thesis fulfills the objective O3 by building the solution elements on top of the current practices for software reuse and design of self-adaptive software systems. For instance, the ASPLe employs domain engineering, selection, specialization, integration, and application engineering concepts from research on software reuse [Kru92; PBV05]. The eARF recommends the use of proven best design practices, such as architectural patterns and tactics, to analyze and reason about design decisions. The ASPLe design processes use the responsibility driven design approach [WM03] to extract responsibilities from requirements and map them to design components. Furthermore, the MAPE-K feedback loop [KC03; Wey+13] is used as a principal architectural pattern by the ASPLe to identify and structure design components.

Based on combined analysis of quantitative and qualitative data from the case study and the demonstrators, we conclude that the thesis contributes significant research effort to address the problem of designing self-adaptive software systems with systematic reuse. Although there remains work to make the ASPLe a comprehensive methodology, yet it contributes significantly to improve the current state of design practices for the development of self-adaptive software systems with reuse.

7.2 Future Work

Future work comprises several steps to address limitations and to consolidate the thesis contributions. The ASPLe currently lacks process support for implementation and testing. Thus, the first step we plan is to extend the ASPLe with such support. There exist several approaches, such as [AST09; Bru+09; G+04; KD07; Kic+97; VG14], to design and implement self-adaptive software systems. However, there is only a little work done for runtime verification and validation of self-adaptive systems [De +13]. Furthermore, testing methods used for conventional systems are not easy to adopt for SASS because the interfaces between managing and managed systems are often quite different from that of traditional systems [HVG15]. Thus, we conjecture that adding testing process support to the ASPLe will be challenging.

As a second step, we plan to augment the requirements and design process support offered by the ASPLe. For the design part, we intend to develop tool support for architectural analysis and reasoning. The tool support is aimed to provide designers with design alternatives for given requirements, model and assess the alternatives, and select best-fit alternatives. A potential challenge here will be scalability and trade-off to model and reason about alternatives for multiple self-adaptation properties. For the requirements part, we plan to study and develop a more formalized language for expressing elements and fragments of dQAS. The dQAS formalization is likely to support automation of the reasoning process. However, a challenge here will be preserving sufficient expressiveness, while providing machine-readable specifications that can be interpreted automatically.

The third step, we aim for is to explore patterns and tactics to develop as a library of best practices to design and develop self-adaptive systems with reuse. The

library will be incorporated into the eARF and can be used by other frameworks, methods, and tools for design support.

The fourth step, we plan is to further evaluate the ASPLe and the eARF for other self-adaptation properties, such as self-configuration and self-protection, both in controlled environments and in real-world settings.

Bibliography

[AA13] Nadeem Abbas and Jesper Andersson. “Architectural Reasoning for Dynamic Software Product Lines”. In: *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*. SPLC ’13 Workshops. Tokyo, Japan: ACM, 2013, pp. 117–124. ISBN: 978-1-4503-2325-3. DOI: 10.1145/2499777.2500718. URL: http://doi.acm.org/10.1145/2499777.2500718.

[AA15] Nadeem Abbas and Jesper Andersson. “Harnessing Variability in Product-lines of Self-adaptive Software Systems”. In: *Proceedings of the 19th International Conference on Software Product Line (SPLC)*. SPLC ’15. Nashville, Tennessee: ACM, 2015, pp. 191–200. ISBN: 978-1-4503-3613-0. DOI: 10.1145/2791060.2791089. URL: http://doi.acm.org/10.1145/2791060.2791089.

[AA17] Nadeem Abbas and Jesper Andersson. *ASPLe – A Methodology to Develop Self-Adaptive Software Systems with Reuse*. Tech. rep. Linnaeus University, Department of computer science and media technology (CM), 2017, p. 118.

[AAL10] N. Abbas, J. Andersson, and W. Löwe. “Autonomic Software Product Lines (ASPL)”. In: *Proceedings of the 4th European Conference on Software Architecture: Companion Volume*. ACM. 2010, pp. 324–331.

[AAL11] Nadeem Abbas, Jesper Andersson, and Welf Löwe. *Towards Autonomic Software Product Lines (ASPL) - A Technical Report*. Tech. rep. Linnaeus University, Department of computer science and media technology (CM), 2011, p. 20.

[AAW11] Nadeem Abbas, Jesper Andersson, and Danny Weyns. “Knowledge evolution in autonomic software product lines”. In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC ’11. Munich, Germany: ACM, 2011, 36:1–36:8. ISBN: 978-1-4503-0789-5. DOI: http://doi.acm.org/10.1145/2019136.2019177. URL: http://doi.acm.org/10.1145/2019136.2019177.

[AAW12] Nadeem Abbas, Jesper Andersson, and Danny Weyns. “Modeling variability in product lines using domain quality attribute scenarios”. In: *Proceedings of the WICSA/ECSA 2012 Companion Volume*. WICSA/ECSA ’12. Helsinki, Finland: ACM, 2012, pp. 135–142. ISBN: 978-1-4503-1568-5. DOI: 10.1145/2361999.2362028. URL: http://doi.acm.org/10.1145/2361999.2362028.

[AAW18] Nadeem Abbas, Jesper Andersson, and Danny Weyns. “ASPLe: A Methodology to Develop Self-Adaptive Software Systems with Systematic Reuse”. In: *Submitted to Software & Systems Modeling* (Mar. 2018).

[Abb+16] Nadeem Abbas et al. “Rigorous Architectural Reasoning for Self-Adaptive Software Systems”. In: *1st Workshop on Qualitative Reasoning about Software Architectures*. IEEE, 2016, pp. 1–8.

[AJ15] Nadeem Abbas and Andersson Jesper. “Architectural Reasoning Support for Product-Lines of Self-adaptive Software Systems - A Case Study”. English. In: *Proceedings of the 9th European Conference on Software Architecture (ECSA)*. Ed. by Danny Weyns, Raffaela Mirandola, and Ivica Crnkovic. Vol. 9278. LNCS. Springer, 2015, pp. 20–36. ISBN: 978-3-319-23726-8. URL: http://dx.doi.org/10.1007/978-3-319-23727-5_2.

[AMA17] Loreno Freitas Matos Alvim, Ivan do Carmo Machado, and Eduardo Santana de Almeida. “A Preliminary Assessment of Variability Implementation Mechanisms in Service-Oriented Computing”. In: *Mastering Scale and Complexity in Software Reuse*. Ed. by Goetz Botterweck and Claudia Werner. Cham: Springer International Publishing, 2017, pp. 31–47. ISBN: 978-3-319-56856-0.

[AST09] R. Asadollahi, M. Salehie, and L. Tahvildari. “StarMX: A framework for developing self-managing Java-based systems”. In: *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS ’09. ICSE Workshop on*. May 2009, pp. 58–67. DOI: 10.1109/SEAMS.2009.5069074.

[Bac+02] Felix Bachmann et al. *Documenting software architecture: Documenting interfaces*. Tech. rep. CMU/SEI-2002-TN-015. Pittsburgh, PA 15213-3890: Software Engineering Institute, Carnegie Mellon University, 2002.

[Bac+05] F. Bachmann et al. “Designing software architectures to achieve quality attribute requirements”. In: *Software, IEE Proceedings* - 152.4 (Aug. 2005), pp. 153–165. ISSN: 1462-5970. DOI: 10.1049/ip-sen:20045037.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. 2nd. Addison-Wesley Professional, 2003.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN: 0-201-67494-7.

[Bru+09] Yuriy Brun et al. “Engineering Self-Adaptive Systems Through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems* 5525 (2009), pp. 48–70.

[C+09] B. Cheng, R. de Lemos, H. Giese, et al. “Software engineering for self-adaptive systems: A research roadmap”. In: *Software Engineering for Self-Adaptive Systems* (2009), pp. 1–26.

[CAA09] L. Chen, M. Ali Babar, and N. Ali. “Variability management in software product lines: a systematic review”. In: *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University. 2009, pp. 81–90.

[COH14] R. Capilla, Ó. Ortiz, and M. Hinchey. “Context Variability for Context-Aware Systems”. In: *Computer* 47.2 (Feb. 2014), pp. 85–87. ISSN: 0018-9162. DOI: 10.1109/MC.2014.33.

[De +13] Rogério De Lemos et al. “Software engineering for self-adaptive systems: A second research roadmap”. In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[Dia+08] Andres Diaz-Pace et al. “Integrating Quality-Attribute Reasoning Frameworks in the ArchE Design Assistant”. In: *Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures*. QoSA ’08. Karlsruhe, Germany: Springer-Verlag, (2008), pp. 171–188. ISBN: 978-3-540-87878-0. DOI: 10.1007/978-3-540-87879-7_11. URL: http://dx.doi.org/10.1007/978-3-540-87879-7_11.

[EM13] Naeem Esfahani and Sam Malek. “Uncertainty in Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by Rogério de Lemos et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 214–238. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_9. URL: http://dx.doi.org/10.1007/978-3-642-35813-5_9.

[ESB07] Leire Etxeberria, Goiuria Sagardui, and Lorea Belategi. “Modelling variation in quality attributes”. In: *First International Workshop on Variability of Software-Intensive Systems (VaMos 2007)*. Vol. 1. 2007, pp. 51–59.

[Esf11] N. Esfahani. “A framework for managing uncertainty in self-adaptive software systems”. In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Nov. 2011, pp. 646–650. DOI: 10.1109/ASE.2011.6100147.

[F+06] J. Floch, S. Hallsteinsen, E. Stav, et al. “Using architecture models for runtime adaptability”. In: *Software, IEEE* 23.2 (Mar. 2006), pp. 62–70. ISSN: 0740-7459. DOI: 10.1109/MS.2006.61.

[FK05] W.B. Frakes and Kyo Kang. “Software reuse research: status and future”. In: *Software Engineering, IEEE Transactions on* 31.7 (July 2005), pp. 529–536. ISSN: 0098-5589. DOI: 10.1109/TSE.2005.85.

[Flo84] Christiane Floyd. “A Systematic Look at Prototyping”. In: *Approaches to Prototyping*. Ed. by Reinhard Budde et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 1–18. ISBN: 978-3-642-69796-8. DOI: 10.1007/978-3-642-69796-8_1. URL: http://dx.doi.org/10.1007/978-3-642-69796-8_1.

[Fra94] W. Frakes. “Systematic software reuse: a paradigm shift”. In: *Proceedings of 1994 3rd International Conference on Software Reuse*. Nov. 1994, pp. 2–3. DOI: 10.1109/ICSR.1994.365817.

[G+04] D. Garlan, S.W. Cheng, A.C. Huang, et al. “Rainbow: Architecture-based self-adaptation with reusable infrastructure”. In: *Computer* 37.10 (2004), pp. 46–54.

[GA01] Critina Gacek and Michalis Anastasopoules. “Implementing Product Line Variabilities”. In: *SIGSOFT Softw. Eng. Notes* 26.3 (May 2001), pp. 109–117. ISSN: 0163-5948. DOI: 10.1145/379377.375269. URL: http://doi.acm.org/10.1145/379377.375269.

[Gar10] David Garlan. “Software Engineering in an Uncertain World”. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. FoSER ’10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 125–128. ISBN: 978-1-4503-0427-6. DOI: 10.1145/1882362.1882389. URL: http://doi.acm.org/10.1145/1882362.1882389.

[GB09] Maria J. Grant and Andrew Booth. “A typology of reviews: an analysis of 14 review types and associated methodologies”. In: *Health Information & Libraries Journal* 26.2 (2009), pp. 91–108. ISSN: 1471-1842. DOI: 10.1111/j.1471-1842.2009.00848.x. URL: http://dx.doi.org/10.1111/j.1471-1842.2009.00848.x.

[GFd98] Martin L Griss, John Favaro, and Massimo d’Alessandro. “Integrating feature modeling with the RSEB”. In: *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE. 1998, pp. 76–85.

[Gie+14] Holger Giese et al. “Living with Uncertainty in the Age of Runtime Models”. In: *Models@run.time: Foundations, Applications, and Roadmaps*. Ed. by Nelly Bencomo et al. Cham: Springer International Publishing, 2014, pp. 47–100. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_3. URL: http://dx.doi.org/10.1007/978-3-319-08915-7_3.

[Gil+08] Paul Gill et al. “Methods of data collection in qualitative research: interviews and focus groups”. In: *British dental journal* 204.6 (2008), p. 291.

[Gra13] David E Gray. *Doing research in the real world*. Ed. by Jai Seaman. Sage, 2013.

[Gri93] M. L. Griss. “Software reuse: From library to factory”. In: *IBM Systems Journal* 32.4 (1993), pp. 548–566. ISSN: 0018-8670. DOI: 10.1147/sj.324.0548.

[Gri96] Martin L. Griss. *Systematic Software Reuse: Architecture, Process and Organization are Crucial*. http://martin.griss.com/pubs/fusion1.htm. 1996.

[Hal+08] S. Hallsteinsen et al. “Dynamic Software Product Lines”. In: *IEEE Computer* 41.4 (2008), pp. 93–95.

[Hel+09] Alexander Helleboogh et al. “Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines”. In: *Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL@ SPLC 2009)*. Carnegie Mellon University. 2009, pp. 18–27.

[HM08] M.C. Huebscher and J.A. McCann. “A Survey of Autonomic Computing - Degrees, Models, and Applications”. In: *ACM Computing Surveys (CSUR)* 40.3 (2008), p. 7.

[Hoy+87] C Graf Hoyos et al. “Software Design with the Rapid Prototyping Approach: A Survey and some Empirical Results”. In: *Cognitive Engineering in the Design of Human-Computer Interaction and Expert Systems* 2 (1987).

[HSF04] Svein Hallsteinsen, Erlend Stav, and Jacqueline Floch. “Self-adaptation for Everyday Systems”. In: *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*. WOSS ’04. Newport Beach, California: ACM, 2004, pp. 69–74. ISBN: 1-58113-989-6. DOI: 10.1145/1075405.1075419. URL: http://doi.acm.org/10.1145/1075405.1075419.

[HVG15] J. Hänsel, T. Vogel, and H. Giese. “A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models”. In: *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. Sept. 2015, pp. 134–139. DOI: 10.1109/SASOW.2015.27.

[Ish86] Kaoru Ishikawa. *Guide to quality control*. Tokyo, Japan: Asian Productivity Organization, 1986.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997. ISBN: 0-201-92476-5.

[Kan+98] K.C. Kang et al. “FORM: A feature-oriented reuse method with domain-specific reference architectures”. In: *Annals of Software Engineering* 5.1 (1998), pp. 143–168.

[Kan90] K.C. Kang. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[KC03] J.O. Kephart and D.M. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (2003), pp. 41–50.

[KD07] J.O. Kephart and R. Das. “Achieving Self-Management via Utility Functions”. In: *IEEE Internet Computing* (2007), pp. 40–48.

[KDJ04] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. “Evidence-Based Software Engineering”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 273–281. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999432.

[Kic+97] Gregor Kiczales et al. “Aspect-oriented programming”. In: *ECOOP’97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242. ISBN: 978-3-540-69127-3.

[KLM96] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. “Reinforcement learning: A survey”. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[KM07] J. Kramer and J. Magee. “Self-Managed Systems: an Architectural Challenge”. In: *Future of Software Engineering, 2007. FOSE ’07*. May 2007, pp. 259–268. DOI: 10.1109/FOSE.2007.19.

[Kru92] Charles W Krueger. “Software reuse”. In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[Lan+] Caroline Lange et al. “Systematic reuse and platforming: Application examples for enhancing reuse with model-based systems engineering methods in space systems development”. In: *Concurrent Engineering* (), p. 1063293X17736358. DOI: 10.1177/1063293X17736358. eprint: https://doi.org/10.1177/1063293X17736358. URL: https://doi.org/10.1177/1063293X17736358.

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN: 3540714367.

[MAW17] Sara Mahdavi-Hezavehi, Paris Avgeriou, and Danny Weyns. “A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements”. In: *Managing Trade-Offs in Adaptable Software Architectures*. Elsevier, 2017, pp. 45–77.

[McI68] M. D. McIlroy. “Mass-produced software components”. In: ed. by P. Naur and B. Randell. Garmisch, Germany, 1968, pp. 138–150.

[MGA13] Sara Mahdavi-Hezavehi, Matthias Galster, and Paris Avgeriou. “Variability in Quality Attributes of Service-based Software Systems: A Systematic Literature Review”. In: *Inf. Softw. Technol.* 55.2 (Feb. 2013), pp. 320–343. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.08.010. URL: http://dx.doi.org/10.1016/j.infsof.2012.08.010.

[MH05] Hugh McManus and Daniel Hastings. “A Framework for Understanding Uncertainty and its Mitigation and Exploitation in Complex Systems”. In: *INCOSE International Symposium* 15.1 (2005), pp. 484–503. ISSN: 2334-5837. DOI: 10.1002/j.2334-5837.2005.tb00685.x. URL: http://dx.doi.org/10.1002/j.2334-5837.2005.tb00685.x.

[N+07] LM Northrop, PC Clements, et al. *A Framework for Software Product Line Practice, Version 5.0*. 2007. URL: http://www.sei.cmu.edu/productlines/frame_report/index.html.

[O+99] P. Oreizy, M.M. Gorlick, R.N. Taylor, et al. “An architecture-based approach to self-adaptive software”. In: *Intelligent Systems and their Applications* 14.3 (1999), pp. 54–62.

[Off+09] Philipp Offermann et al. “Outline of a Design Science Research Process”. In: *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*. DESRIST ’09. Philadelphia, Pennsylvania: ACM, 2009, 7:1–7:11. ISBN: 978-1-60558-408-9. DOI: 10.1145/1555619.1555629. URL: http://doi.acm.org/10.1145/1555619.1555629.

[OSG07] OSGi Alliance. *OSGi Service Platform Release 4*. [Online]. Available: http://www.osgi.org/Main/HomePage. [Accessed: Jun. 17, 2009]. 2007.

[Par76] D. L. Parnas. “On the Design and Development of Program Families”. In: *IEEE Transactions on Software Engineering* 2.1 (1976), pp. 1–9.

[PBV05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York, Inc., 2005.

[Pef+07] Ken Peffers et al. “A Design Science Research Methodology for Information Systems Research”. In: *Journal of Management Information Systems* 24.3 (2007), pp. 45–77. DOI: 10.2753/MIS0742-1222240302. eprint: http://www.tandfonline.com/doi/pdf/10.2753/MIS0742-1222240302. URL: http://www.tandfonline.com/doi/abs/10.2753/MIS0742-1222240302.

[PM14] Diego Perez-Palacin and Raffaela Mirandola. “Uncertainties in the Modeling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. Dublin, Ireland: ACM, 2014, pp. 3–14. ISBN: 978-1-4503-2733-6. DOI: 10.1145/2568088.2568095. URL: http://doi.acm.org/10.1145/2568088.2568095.

[Pri93] R. Prieto-Diaz. “Status report: software reusability”. In: *Software, IEEE* 10.3 (May 1993), pp. 61–66. ISSN: 0740-7459. DOI: 10.1109/52.210605.

[R+12] Per Runeson, Martin Höst, Austen Rainer, et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st. Wiley Publishing, 2012. ISBN: 1118104358, 9781118104354.

[Rob93] C. Robson. *Real World Research: A Resource for Social Scientists and Practitioner-Researchers*. Blackwell Publishers Inc., 1993. ISBN: 9780631176893. URL: https://books.google.se/books?id=gNO6QgAACAAJ.

[Rou+09] Romain Rouvoy et al. "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments". In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty Cheng et al. Vol. 5525. Lecture Notes in Computer Science. 10.1007/978-3-642-02161-9_9. Springer Berlin / Heidelberg, 2009, pp. 164–182. ISBN: 978-3-642-02160-2. URL: http://dx.doi.org/10.1007/978-3-642-02161-9_9.

[Roy87] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0-89791-216-0. URL: http://dl.acm.org/citation.cfm?id=41765.41801.

[SGP13] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. "An Analysis of Language-Level Support for Self-Adaptive Software". In: *ACM Trans. Auton. Adapt. Syst.* 8.2 (July 2013), 7:1–7:29. ISSN: 1556-4665. DOI: 10.1145/2491465.2491466. URL: http://doi.acm.org/10.1145/2491465.2491466.

[Sha03] Mary Shaw. "Writing Good Software Engineering Research Papers: Minitutorial". In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 726–736. ISBN: 0-7695-1877-X. URL: http://dl.acm.org/citation.cfm?id=776816.776925.

[Sin+04] Marco Sinnema et al. "Covamof: A framework for modeling variability in software product families". In: *Software Product Lines*. Springer, 2004, pp. 197–213.

[ST09] M. Salehie and L. Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4.2 (2009), p. 14.

[SVB05] M. Svahnberg, J. Van Gurp, and J. Bosch. "A Taxonomy of Variability Realization Techniques". In: *Software: Practice and Experience* 35.8 (2005), pp. 705–754.

[Tar+99] Peri Tarr et al. "N degrees of separation: multi-dimensional separation of concerns". In: *Proceedings of the 21st international conference on Software engineering*. ACM. 1999, pp. 107–119.

[TMD09] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN: 0470167742, 9780470167748.

[VBS01] J. Van Gurp, J. Bosch, and M. Svahnberg. "On the notion of variability in software product lines". In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'2001)*. IEEE. 2001, pp. 45–54.

[VG14] Thomas Vogel and Holger Giese. "Model-Driven Engineering of Self-Adaptive Software with EUREMA". In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (Jan. 2014), 18:1–18:33. ISSN: 1556-4665. DOI: 10.1145/2555612. URL: http://doi.acm.org/10.1145/2555612.

[W+12] Claes Wohlin, Per Runeson, Martin Höst, et al. *Experimentation in Software Engineering*. 1st. Springer-Verlag Berlin Heidelberg, 2012. ISBN: 978-3-642-29044-2.

[Wal+03] W.E. Walker et al. "Defining Uncertainty: A Conceptual Basis for Uncertainty Management in Model-Based Decision Support". In: *Integrated Assessment* 4.1 (2003), pp. 5–17. DOI: 10.1076/iaij.4.1.5.16466. eprint: http://dx.doi.org/10.1076/iaij.4.1.5.16466. URL: http://dx.doi.org/10.1076/iaij.4.1.5.16466.

[Wey+13] Danny Weyns et al. "On patterns for decentralized control in self-adaptive systems". In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 76–107.

[Whi+10] Jon Whittle et al. "RELAX: a language to address uncertainty in self-adaptive systems requirement". In: *Requirements Engineering* 15.2 (2010), pp. 177–196. ISSN: 1432-010X. DOI: 10.1007/s00766-010-0101-0. URL: http://dx.doi.org/10.1007/s00766-010-0101-0.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-based Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-69438-7.

[WM03] Rebecca Wirfs-Brock and Alan McKean. *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003.

[WMA10] D. Weyns, S. Malek, and J. Andersson. "FORMS: a formal reference model for self-adaptation". In: *Proceeding of the 7th international conference on Autonomic computing (ICAC '10)*. New York, NY, USA: ACM, (2010), pp. 205–214.

[WMA12] Danny Weyns, Sam Malek, and Jesper Andersson. “FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems”. In: *ACM Trans. Auton. Adapt. Syst.* 7.1 (May 2012), 8:1–8:61. ISSN: 1556-4665. DOI: 10.1145/2168260.2168268. URL: http://doi.acm.org/10.1145/2168260.2168268.