The Journal of Systems and Software 167 (2020) 110626

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

ASPLe: A methodology to develop self-adaptive software systems with systematic reuse

Nadeem Abbas[a,*], Jesper Andersson[a], Danny Weyns[b]

[a] Linnaeus University, Sweden
[b] Linnaeus University, Sweden and KU Leuven, Belgium

ARTICLE INFO

Article history:
Received 5 November 2019
Revised 24 March 2020
Accepted 1 May 2020
Available online 6 May 2020

Keywords:
Software reuse
Domain engineering
Self-Adaptation
Uncertainty
Variability
Software design

ABSTRACT

More than two decades of research have demonstrated an increasing need for software systems to be self-adaptive. Self-adaptation manages runtime dynamics, which are difficult to predict before deployment. A vast body of knowledge to develop Self-Adaptive Software Systems (SASS) has been established. However, we discovered a lack of process support to develop self-adaptive systems with reuse. The lack of process support may hinder knowledge transfer and quality design. To that end, we propose a domain-engineering based methodology, Autonomic Software Product Lines engineering (ASPLe), which provides step-by-step guidelines for developing families of SASS with systematic reuse. The evaluation results from a case study show positive effects on quality and reuse for self-adaptive systems designed using the ASPLe compared to state-of-the-art engineering practices.

1. Introduction

Advances in technology trigger market changes that increase the importance of software as a value enabler. The ubiquity and connectivity of software systems cause context variability that introduces additional complexity on systems during their life-cycles. Software systems are consequently required to be more open and flexible concerning a system's environment and its goals, which affect the system's design. Lack of knowledge and so-called known-unknowns (McManus and Hastings, 2005) on a system's goals, behavior, and environment are root-causes for uncertainty in the design process. The design time uncertainty can sometimes only be mitigated at runtime when all goals and environment characteristics are discernible (Garlan, 2010; De Lemos et al., 2013).

Self-adaptation (Cheng et al., 2009; De Lemos et al., 2013) is a risk mitigation strategy for uncertainties induced by runtime changes. The basic idea of self-adaptation is to let a system gather new knowledge at runtime to resolve uncertainties, reason about itself, its context and goals, and adapt to realize its goals, or gracefully degrade if necessary. From an architectural point of view, a self-adaptive system is composed of two subsystems; the managed subsystem, which is responsible for a system's core functionality, and the managing subsystem, which is responsible for managing the managed system by adapting it. Research and practice have established a large body of knowledge for how to engineer self-adaptive software systems (Weyns, 2019). However, to the best of our knowledge, no work in this field has defined a methodology[1] that systematically reuses this knowledge.

Hence, the research problem we address in this paper is the *design and development of self-adaptive software systems with systematic reuse*. Software reuse is a long acclaimed method to build systems efficiently and cost-effectively (Krueger, 1992). It enables developers to resolve complexity and improve quality and productivity at reduced cost and shorter time-to-market (Griss, 1993). The development costs are amortized on several application projects, and the widespread usage helps to test artifacts and improve quality. The proven benefits of development with reuse motivated us to investigate a reuse-based methodology for self-adaptive systems. The specifics of self-adaptation in combination with systematic reuse, however, require changed perspectives and modified reuse strategies.

We analyzed the specifics of SASS and reuse in the previous work (Abbas and Andersson, 2015) and proposed an Autonomic Software Product Lines (ASPL) strategy. ASPL is a multi-product

* Corresponding author
E-mail address: nadeem.abbas@lnu.se (N. Abbas).

[1] With methodology, we mean the methods and processes that are required to engineer self-adaptive systems.

https://doi.org/10.1016/j.jss.2020.110626

lines strategy that exploits a disciplined split between the managed and managing subsystems. It consists of three steps. The first step establishes an application domain-independent ASPL platform that provides reusable artifacts for managing systems. The second step derives an application domain-specific managing system platform from the ASPL platform. The third step integrates the managing system platform with a separately developed managed system platform. Practical experiences with applying ASPL strategy showed a lack of dedicated processes to support software engineers in applying the strategy when developing self-adaptive systems.

To that end, this work contributes an engineering methodology called Autonomic Software Product Lines engineering (ASPLe in short). ASPLe provides developers with process support to implement ASPL strategy, i.e., develop product lines of self-adaptive systems with *reuse at the managing system level*. ASPLe is composed of three processes: 1) ASPL Domain Engineering, 2) Specialization, and 3) Integration. These processes describe roles, work-product, activities, and workflows to realize ASPL strategy. Each of the three processes covers requirements, design, implementation, and testing subprocesses. The focus of this paper, however, is the design subprocesses.

We evaluated ASPLe's design processes in a case study. The case study results show that ASPLe is an improvement compared to current design practices. In particular, the results show that the engineering methodology helps developers to reduce fault-density and increase reuse. We link the improvements to the structured management of variability for different sub-domains, which simplifies and thus increases the quality of developers' activities.

The remainder of this paper is organized as follows. Section 2 positions our research to related work. We describe the study's context, identify a research gap, and specify objectives to address the gap in Section 3. Section 4 introduces an example application that we use to illustrate ASPLe. Section 5 describes ASPLe focusing on design processes. Section 6 reports a case study conducted to evaluate the ASPLe. We conclude and outline future work in Section 7.

2. Related work

Research in self-adaptive systems distinguishes between internal and external adaptation mechanisms (Garlan et al., 2004; Salehie and Tahvildari, 2009; Weyns et al., 2013a). Internal adaptation mechanisms rely on programming language constructs, such as exceptions, reflection, dynamic linking, and conditional expressions, to realize self-adaptation. Here, application logic and adaptation logic are mixed. Due to tight coupling between the two logics, internal adaptation leads to poor maintainability and reusability for development artifacts. ASPLe methodology enforces a clear separation of application and adaptation logic with reusable process support to model an external adaptation mechanism. In this context, Andersson et al. (2009) refers to a disciplined split between application and adaptation logic.

External adaptation mechanisms rely on the principles of feedback loop control, such as the managing system shown in Fig. 1. External adaptation separates application and adaptation logic; thus, adaptation logic may be reused across self-adaptive systems (Garlan et al., 2004). Examples of approaches that apply external adaptation include Rainbow (Garlan et al., 2004), StarMX (Asadollahi et al., 2009), MADAM (Floch et al., 2006), MUSIC (Rouvoy et al., 2009), and ENTRUST (Calinescu et al., 2018).

Rainbow framework (Garlan et al., 2004) provides architecture based reusable infrastructure for developing SASS. It is similar in infrastructure to ASPLe. The two approaches are also identical in the use of architectural styles and patterns to encapsulate and reuse architectural knowledge. However, ASPLe supports both vertical and horizontal reuse, whereas Rainbow's primary target is vertical reuse. Furthermore, Rainbow does not provide process support, which is a primary facet of ASPLe.

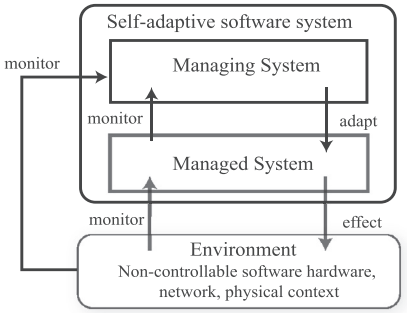


Fig. 1. SASS - conceptual architecture.

Other frameworks, including StarMX (Asadollahi et al., 2009), MADAM (Floch et al., 2006), and MUSIC (Rouvoy et al., 2009) are similar to ASPLe in the use of patterns and tactics. However, none of these frameworks provide any process support.

ENTRUST (Calinescu et al., 2018) uses a combination of 1) design-time and runtime modeling and verification, and 2) industry-adopted assurance processes to develop trustworthy self-adaptive software and assurance cases arguing the suitability of the software for its intended application. It exploits reusable artifacts, such as model templates, an assurance argument pattern, and an execution platform. However, compared to ASPLe, ENTRUST was not devised driven by systematic reuse.

A related research theme that studies the development of dynamically reconfigurable software systems is Dynamic Software Product Lines (DSPLs) (Hallsteinsen et al., 2008). A DSPL uses principles and mechanisms from software product lines, such as variability management, to model dynamic adaptive behaviors as runtime variability. In contrast, ASPLe offers process support to engineers emphasizing systematic reuse. ASPLe can, in fact, be used to develop DSPLs.

Garlan (2010) examined uncertainty from a general software engineering perspective and proposed self-adaptive systems as a solution. However, an impediment to efficient self-adaptive systems development is the poor understanding of uncertainty (Esfahani and Malek, 2013; Mahdavi-Hezavehi et al., 2017), and lack of explicit process support to analyze, reason, and decide with uncertainty as a factor. ASPLe focuses on supporting design with uncertainty, offering a reasoning framework that focusses on variability and runtime adaptations, providing a mindset that actively guides designers to work with and mitigate risks that arise from uncertainties.

Krupitzer et al. (2015) presented reusable templates based approach to support the development of self-adaptive systems with reuse. Their approach is similar to ASPLe in the separation of adaptation logic and application logic and the use of the MAPE-K loop to realize adaptation logic. However, they did not consider uncertainty, which is a principal challenge to realize the development of the self-adaptive systems with reuse. Furthermore, their work lacks process support, which is a key characteristic of the ASPLe methodology.

In summary, a vast body of research on engineering self-adaptive systems exists. However, state of the art falls short in two important aspects: 1) existing work offers limited support for dedicated processes to engineer self-adaptive systems, and 2) existing work has not paid sufficient attention to systematic reuse. Both these aspects are crucial to facilitate knowledge transfer and enhance the quality design of self-adaptive systems.

3. Context and motivation

The research context for this work is twofold: *self-adaptive software systems* and *software reuse*. We have conducted an in-depth analysis of how these could be combined, presented critical challenges that arise, and a strategy called ASPL to address the challenges in previous work (Abbas and Andersson, 2015). We expand on this work below, reiterating the context, identify challenges that define a research gap, and specify a set of objectives for the proposed solution.

3.1. Self-adaptive software systems

A Self-Adaptive Software System (SASS) is a software system that adapts its behavior and structure in response to changes (Cheng et al., 2009; De Lemos et al., 2013). There are two common ways to look at self-adaptive systems (Weyns, 2019). The *first* is to view them as systems with the ability to adapt their structure or adjust their behavior in response to the perception of the environment, systems themselves, and their goals. The *self* prefix indicates that a system adapts autonomously without or with minimal interference of humans. The *second* is to view self-adaptive systems as a mechanism used to realize adaptation logic typically by means of a closed feedback loop. This way of looking at self-adaptive systems separates a part of the system that deals with domain concerns (goals for which the system is built) and a part that deals with adaptation concerns.

Fig. 1 shows the conceptual architecture of a self-adaptive system. The *managed subsystem* represents core application functionality. The *managing subsystem* comprises adaptation logic, which identifies changes that require adaptations to maintain the managed system's goals. The managing subsystem observes changes in the managed subsystem or the environment and performs adaptive actions on the managed subsystem via effectors.

Self-adaptation is a common concern for several systems (Cheng et al., 2009; De Lemos et al., 2013). Supporting development of self-adaptation by generic reusable development artifacts would be a big step to improve quality, affordability, and productivity of software systems (Hallsteinsen et al., 2004; Abbas, 2018). 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 only a little work available that has considered systematic reuse of this knowledge. The benefits of development with reuse motivated us to investigate the design and development of self-adaptive systems with systematic reuse.

3.2. Software reuse

Krueger (1992) defines software reuse as a *"process of creating software systems from existing software rather than building software systems from scratch"*. He enumerates *abstraction, selection, specialization*, and *integration* as essential activities for any reuse technique. Developing strategies and techniques that consider and support these activities is key to successful software reuse. Prieto-Diaz (1993) distinguishes vertical and horizontal reuse. As shown in Fig. 2, vertical reuse is reuse within the same application domain and is supported by a vertical platform. Horizontal reuse refers to reuse across several domains and is supported by a horizontal platform.

Frakes and Kang (2005) state that domain engineering is an optimal choice for reuse as it improves both quality and productivity. Domain engineering develops domain-specific reusable artifacts. These artifacts are then reused by application engineering processes (Pohl et al., 2005) to produce individual applications.

Software Product Lines (SPL) is a widely adopted systematic reuse strategy. A software product line is a set of software systems that share common features, developed with platform reuse and mass-customization (Pohl et al., 2005). It defines a platform that provides artifacts for reuse across multiple products in a domain. The platform artifacts are then customized for individual applications' needs. The fundamental difference between traditional software development and SPL is the shift of focus from a single system to domain engineering.

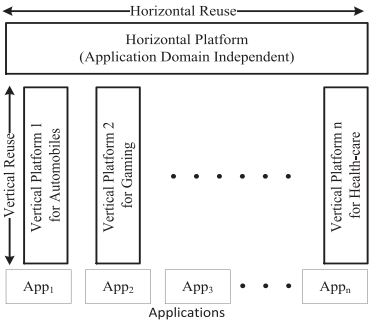


Fig. 2. Horizontal and vertical reuse.

We have seen some examples of reuse strategies in self-adaptive software research projects, such as the Rainbow (Garlan et al., 2004) and MADAM/MUSIC frameworks (Floch et al., 2006; Rouvoy et al., 2009), and ENTRUST (Calinescu et al., 2018). A particularly relevant line of research in this context is dynamic software product lines (DSPL). DSPL (Hallsteinsen et al., 2008) leverages concepts from software product lines to establish a runtime variability mechanism, which has capabilities similar to that of a self-adaptive software system. However, neither the framework approaches nor work on DSPL has contributed with *process support* to design and develop self-adaptive systems with systematic reuse (Andersson et al., 2012).

3.3. Research gap

The combination of software reuse and runtime variability for self-adaptation creates a complex landscape. A significant cause for this complexity is *uncertainty*. Uncertainty refers to things that are not known or known imprecisely, such as requirements and operating environments, at a specific point in time (McManus and Hastings, 2005). It is an inherent property of complex systems with effects on system design and other development activities. Existing reuse research and practices address uncertainty in reuse, primarily with domain modeling that reduces the scope for reuse, thus the uncertainty. Extensive work in the self-adaptive systems domain has characterized uncertainty and proposed specific methods and techniques that mitigate risks associated with uncertainty (Ramirez et al., 2012; Esfahani and Malek, 2013; Perez-Palacin and Mirandola, 2014; Mahdavi-Hezavehi et al., 2017).

Separation of Concerns (SoC) is a fundamental design principle to reduce complexity and improve reusability (Tarr et al., 1999). The separation of managing and managed subsystems, as depicted in Fig. 1, implies that we may develop the subsystems independently, and establish two separate platforms, one for each subsystem. Moreover, managing systems have adaptation-functionality that is often common across several application domains (Abbas and Andersson, 2015). For instance, applications that want to optimize their performance through resource management or arbitration tactics (Abbas and Andersson, 2015) share a self-optimization property at some level of abstraction. The commonality among managing systems suggests that a horizontal platform can be established and reused to derive managing system platforms for several application domains.

However, the development of managed and managing systems through separate platforms injects uncertainty into several development activities. There are two primary uncertainty sources when we develop the managed and managing systems as separate platforms. The first source is *lack of knowledge*. Developers of a horizontal platform for managing systems have generic application domain-independent requirements. At this level, knowledge about target applications and application domains is either not available or available only partially. Thus, developers are uncertain about stakeholders' goals and deployment environments. Both goals and environments may also change during development and further after deployment. All these unknowns about the target application domains introduce uncertainty to horizontal platform development.

The second source of uncertainty is the *runtime variability* in managed systems and their environment. Full knowledge about runtime variations in a system's requirements, environment, and the system itself is often not available at design time. Due to this lack of knowledge, system analysts and designers face uncertainty in requirements and design specifications. Moreover, the runtime variations are hard to anticipate at design time, and even if anticipated, there are no guarantees that the variations are characterized accurately.

Autonomic Software Product Lines strategy (Abbas and Andersson, 2015) addresses variability, reuse, and uncertainty challenges. Two fundamental principles of ASPL are 1) strict separation of managed and managing system concerns and 2) stepwise specialization of reusable assets into product-specific assets. These principles help reducing complexity and mitigating uncertainty. As shown in Fig. 4(a), ASPL strategy is composed of the following three steps:

Step 1. Establish a Horizontal ASPL Platform
The first step of ASPL is to establish a horizontal platform that provides application domain-independent artifacts for reuse in several managing subsystems. The artifacts span the range from requirements engineering to testing and are developed on purpose for reuse across several domains.

Step 2. Derive a Vertical Managing System Platform
The second step transforms the horizontal ASPL platform into a vertical managing system platform. As depicted in Fig. 4(a), *n* number of application domain-specific managing system platforms can be derived with reuse from a single ASPL platform, where each derived platform targets adaptation logic for a specific application domain.

Step 3. Integrate Platforms
The third step integrates a managing system platform derived in the second step with a separately developed managed system platform. The managed system platform includes development artifacts for application logic. The managed system platform is developed using a software product line engineering approach, such as SPLE (Pohl et al., 2005).

While instrumental for the development of self-adaptive systems, a fundamental research gap needs to be tackled for a successful and repeatable application of the ASPL strategy. In particular, the strategy needs to be complemented with a software development methodology. The methodology is needed to provide engineers with process support in the form of well-defined roles, activities, and work-products.

3.4. Research objectives

To bridge the above described research gap, this article presents Autonomic Software Product Lines engineering methodology (ASPLe). We set the following three objectives to steer and direct our research effort to formulate the ASPLe methodology.

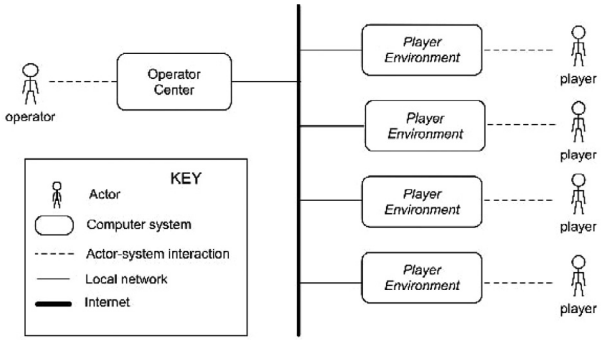


Fig. 3. Distributed game environment.

O1. Define process support to establish a horizontal ASPL platform.
O2. Define a stepwise refinement process for *selection, specialization,* and *integration* of managing system artifacts from the ASPL platform with separately developed managed system artifacts.
O3. Provide explicit support for architectural analysis, reasoning, and decision making, which is needed to harness variability and mitigate uncertainty in the design of self-adaptive systems with reuse.

4. Distributed game environment – running example

Distributed Game Environment (DGE) is a prototype software product line that we developed for research and educational purposes (Quan, 2013). The DGE products are multi-player board games deployed in a distributed setting. As shown in Fig. 3, each DGE product consists of two subsystems: Operator Center (OC) and Player Environment (PE). A PE represents a client-side used by a human player to play games. An OC represents a server-side operated by a human operator to perform administrative tasks such as add, remove, and update games.

PEs' updates are triggered and controlled by a human operator. The operator uses the OC to push updates to PEs. Alternatively, a PE can request an update from the OC. All such requests need to be handled by an operator and may take more time than expected to respond. The DGE management wants to introduce a self-upgradability property to improve the upgrade process by enabling a PE to update itself at runtime with minimal human intervention. The DGE requirements for self-upgradability are:

1. New updates should be introduced through an updates repository, a directory or a folder in a file system that stores updates.
2. The OC should get a notification as soon as a new update appears in the repository.
3. The OC should analyze new updates and push them to target PEs. Such updates are called push-type updates.
4. When a new update appears, a PE gets an update notification from the OC within 5 to 60 s.
5. A PE may accept or ignore push-type updates. If accepted, an update delivery and execution should not take more than 10 min.
6. The vital updates called critical push-type cannot be ignored by PEs. Such updates must be executed within 120 s.
7. PEs can view available updates and request the OC for an update. The updates requested by a PE are called pull-type updates.
8. The OC responds an update request within 5 to 60 s.

The DGE domain is currently composed of four products: P1, P2, P3, and P4. The products differ in requirements for self-upgradability. Product P1 requires only push-type updates, which can be either accepted or postponed by PEs. P2 should support

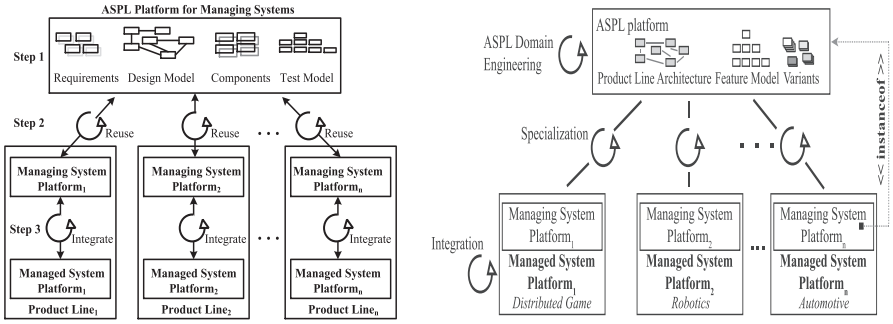


Fig. 4. ASPL strategy and ASPLe methodology.

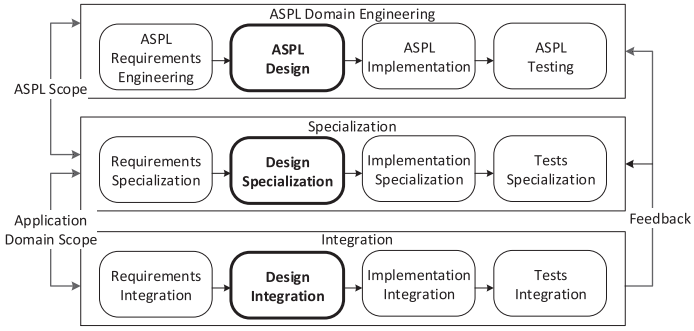


Fig. 5. ASPLe processes.

both push-type and critical push-type updates. P3 should provide for all three update types, i.e., push, critical push, and pull-type. Finally, P4 only supports pull-type updates.

5. ASPLe methodology

We now present ASPLe methodology that provides process support to design and develop product lines of self-adaptive systems with systematic reuse at the managing system level. As depicted in Fig. 4(b), ASPLe is aligned with ASPL strategy and comprises three processes: 1) ASPL Domain Engineering, 2) Specialization, and 3) Integration. A difference between ASPL and ASPLe is that the ASPL is a strategy that outlines an overall plan of developing self-adaptive systems with reuse. It envisions a plan of actions in the form of three steps, described in Section 3.3, but does not provide process support to realize these steps. The process support is provided by the ASPLe methodology. By process support, we mean documented and repeatable guidelines and assistance in the form of design and development activities, roles, and work-products.

The work presented in this article targets design processes highlighted in Fig. 5. We use process modeling concepts and notations from Software Process Engineering Meta-model (SPEM) (OMG, 2008) to describe the processes.

We start by introducing an extended reasoning framework that is pivotal in each design process of ASPLe. Then, we present ASPLe processes starting with the ASPL Domain Engineering followed by the Specialization process, and finally, the Integration process.

5.1. extended Architectural Reasoning Framework (eARF)

A reasoning framework (RF) encapsulates architectural knowledge and methods to realize quality attributes (Bass et al., 2005). Existing RFs do not provide support for self-adaptation properties, such as self-healing and self-optimization (Kephart and Chess, 2003). To that end, we enhanced a reasoning framework presented by Diaz-Pace et al. (2008). The enhancement resulted in the extended Architectural Reasoning Framework (eARF) (Abbas and Jesper, 2015). The primary enhancements include requirements and design work-products with explicit support for variability specification and modeling, and self-adaptive systems specific architectural patterns, tactics, and evaluation methods.

Fig. 6 shows eARF's roles, activities, work-products, and workflow. The eARF is used by domain analysts and designers to analyze requirements and map them to design decisions. Its workflow involves four work-products: i) dQAS, ii) dRS, iii) Architectural Tactics, and iv) Patterns. The dQAS and dRS are respectively described in Sections 5.1.1 and 5.1.2. The architectural tactics and patterns (Bass et al., 2003) encapsulate proven design decisions to realize quality attributes with self-adaptation. Both these work-products provide reusable knowledge to domain analysts and designers, which assist them with identifying requirements and design alternatives, reasoning about the alternatives, and eventually making and modeling design decisions. eARF supports a stepwise mapping of self-adaptation requirements to design decisions. The first part of the eARF's workflow, activities ① and ②, specifies requirements in two steps. In activity ①, a domain analyst identifies the scope, i.e., which self-adaptation requirements and their variants should be specified and realized. A domain analyst consults tactics from eARF's knowledge base to determine and specify requirements with variability. Requirements are specified in activity ② by defining domain Quality Attribute Scenarios (dQASs). The scenarios describe how system may react (adapt) to internal or external stimuli.

Activities ③ and ④ map requirements to design alternatives. A domain designer applies responsibility-driven design (Wirfs-

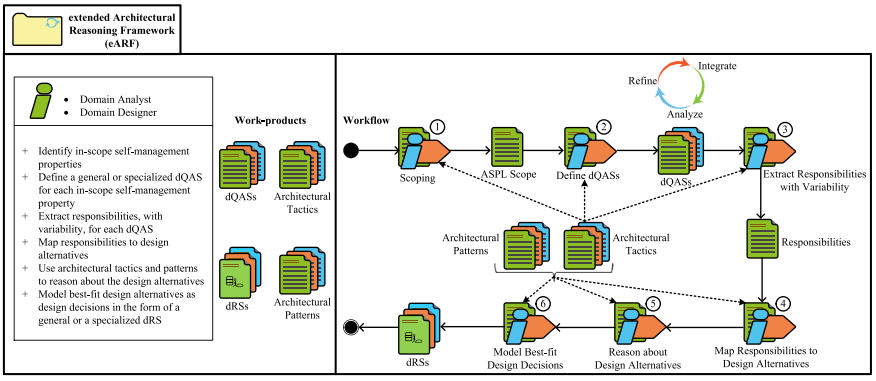


Fig. 6. extended architectural reasoning framework.

Table 1
dQAS.

dQAS Elements	
Source	the origin of a Stimulus.
Stimulus	a condition that triggers self-adaptation.
Artifact	the affected parts of a system.
Environment	operating environment under which a Stimulus arrives.
Response	how does a system respond to the Stimulus.
Response Measure	how the Response is monitored and measured
Variants	different forms of a self-adaption property
Valid Configurations	how the Variants can be combined.
Fragment Constraints	constraints on the selection of fragments of a standard QAS elements

Brock and McKean, 2003) approach, in activity ③, to extract responsibilities from scenarios. It is important to note that responsibilities are extracted with variants to support several adaptation scenarios. Responsibilities and variants are modeled as architectural elements in activity ④. Tactics and patterns from the knowledge base are important inputs to the activities ③ and ④. Tactics provide knowledge to identify design alternatives (responsibilities), and patterns are used to structure architecture elements in activity ④. The current design decisions are evaluated for fitness concerning quality attribute values in a separate activity ⑤. eARF provides informal and formal techniques to reason about and verify design decisions (Abbas et al., 2016). In the final activity ⑥, best-fit decisions are modeled as responsibility structures with variation points and variants in the form of a domain Responsibility Structure (dRS). eARF is used iteratively in subsequent iterations, which is difficult to illustrate in the sequential workflow in Fig. 6. In particular, domain analysts and designers use eARF incrementally and iteratively. In each iteration, new quality attributes, scenarios, and dRSs are added and integrated with existing work-products, the work-products are analyzed until all requirements are specified, and resulting dRSs include sufficient responsibilities to provide for all requirements.

5.1.1. domain Quality Attribute Scenario (dQAS)

Specifying requirements is a prerequisite for architectural analysis and design. eARF provides a template, dQAS (Abbas and Jesper, 2015; Abbas et al., 2012), to specify a domain's requirements for self-adaptation with variability. The dQAS extends a quality attribute scenario (QAS) (Bass et al., 2003) with three elements: Variants, Valid Configurations, and Fragment Constraints. Table 1 lists and briefly describe all the dQAS elements. Domain analysts may specify fragments in the first six elements. Fragments serve as variation points and are used to specify variability. A dQAS specifies domain requirements with variability and can be specialized to derive multiple product-specific scenarios. Derivation of product-specific scenarios is constrained using the Valid Configurations and Fragment Constraints elements.

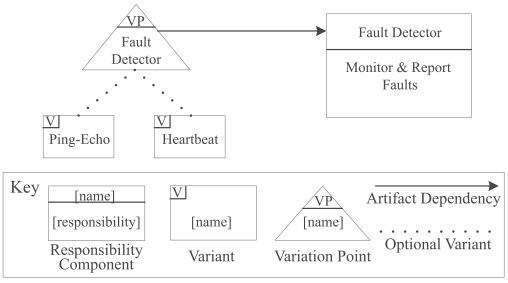


Fig. 7. An excerpt from a dRS for self-healing.

5.1.2. domain Responsibility Structure (dRS)

A dRS is an architectural representation of design decisions. It is modeled by translating requirements into responsibilities and representing them as responsibility components. The responsibility components use *provided* and *required* interfaces for orchestration. Each responsibility component has two compartments; the upper compartment specifies a unique identifier, and the bottom compartment specifies one or more responsibilities. Fig. 7 shows a fault detector responsibility component extracted from a dRS for Self-Healing. The fault detector component is responsible for monitoring, detecting, and reporting silent node failures.

The dRS models variability separately from the responsibility components in an Orthogonal Variability Model (OVM) (Pohl et al., 2005). The OVM provides separation of concerns and reduces complexity; it captures domain and cross-domain variability (Abbas and Jesper, 2015; Abbas and Andersson, 2015) for responsibility components by defining variation points and variants. A variation point represents a design decision that may vary from one domain to another, or one product to another product, either at design or runtime. The OVM in Fig. 7 models a fault detector variation point

with two variants, illustrating an open design decision. At some point in time, one of the variants is selected and bound to the responsibility component.

5.2. ASPL domain engineering (ADE)

ASPL domain engineering defines roles, activities, and work-products to establish a horizontal ASPL platform. As shown in Fig. 5, ADE is composed of four subprocesses. The first subprocess is requirements engineering. Its purpose is to scope the ASPL platform and specify application domain-independent requirements for self-adaptation in the form of general dQASs. The general dQASs are input work-products to design subprocess, which defines a set of application domain-independent general dRSs. The implementation subprocess describes activities to develop reusable code components or libraries to realize self-adaptation properties. The testing subprocess defines reusable test artifacts for the platform.

Defining boundaries is critical to the success of any project. ASPL scope definition specifies in-scope self-adaptation properties and tactics supported by the ASPL platform. A domain analyst defines the scope as a part of the ASPL requirements subprocess. The analyst consults tactics from eARF to get additional scope details such as variants, dependencies, and constraints. An in-scope property and associated tactics together form a structure which, in many ways, is similar to a feature model (Kang, 1990). Fig. 8 shows a feature model for self-upgradability. The feature model scopes an example ASPL platform that supports only one adaptation property, self-upgradability. The features are derived based on self-upgradability tactics (Abbas and Andersson, 2013; Miedes and Munoz-Escoi, 2012) and organized into three groups: 1) update detection, 2) update delivery, and 3) update introduction. These features group variants to find, deliver, and execute new updates.

ASPL design subprocess provides guidelines for developing architectural artifacts to realize requirements specified as general dQASs. Its objective is to define an application domain-independent reference architecture. The architecture is reused and specialized to support domain-specific product lines of self-adaptive systems. Designing an application domain-independent architecture presents designers with uncertainties due to the lack of concrete knowledge about target application domains and their runtime environments. An example is trade-offs, where the goal is to balance the design and satisfy two or more self-adaptation properties that designers typically cannot resolve before they have domain-specific or even application-specific knowledge at hand. eARF assists designers to mitigate this and other uncertainties.

Fig. 9 shows ASPL design process roles, work-products, activities, and workflow. A domain designer models general dRSs, reference architectures, in four activities performed for each in-scope self-adaptation property. The designer selects a general dQAS for a property and identifies a set of responsibilities and variants using responsibility-driven design approach (Wirfs-Brock and McKean, 2003) in activity ①.

Responsibilities identified in activity ① can be realized through several design options. Activity ② identifies design options with the help of tactics from eARF. A domain designer identifies suitable tactics for a responsibility where each tactic presents design options. In this activity, the designer may also define alternative design strategies that are new or derived from one or more tactics. The options are further analyzed and integrated with existing design decisions in activity ③. The integrate-analyze-refine cycle is repeated for each responsibility-variant to provide guarantees for all combinations.

Design options are evaluated using the eARF analytical framework (Abbas et al., 2016). Domain designers choose an architectural analysis method and perform analysis to identify design options that best match the desired quality attribute level for in-scope self-adaptation properties.

The verified design options are modeled as responsibility components in a general dRS in activity ④. Defining a dRS requires domain designers to further reason about the architectural structure and responsibility components' interfaces. eARF provides patterns such as the MAPE-K feedback loop (Kephart and Chess, 2003) and others (Weyns et al., 2013a; 2013b) for responsibility components organization.

To exemplify the ASPL design subprocess, the following is a description of how we performed this process for an example ASPL platform that supports self-upgradability. Beginning with activity ①, we analyzed an ASPL requirement artifact, a general dQAS for self-upgradability, which was defined using the ASPL requirements engineering subprocess. Each element of the general dQAS was analyzed, and a set of responsibilities and their variants were identified. Table 2 lists al the identified responsibilities and variants. For instance, in the *Source* element, we identified two responsibilities, 1) updates provider and 2) updates consumer. The updates provider serves as a source of new updates. An update is introduced by a system administrator, developer, or the system itself. Thus, three variants, i) system administrator, ii) system developers, and iii) software (sub)system, of the updates provider were identified. In the second activity ②, we identified design options for all responsibilities and their variants.

Next, we verified the design options in activity ③, and modeled best-fit options in a dRS in activity ④. Fig. 10 shows the resulting application domain-independent dRS for self-upgradability. We zoom in on the *Updates Provider* element and the three variants discussed above. We can also see how designers have used the MAPE-K feedback loop pattern (Kephart and Chess, 2003) to structure the dRS with the *monitor, analyze, plan,* and *execute* elements.

The ASPL platform produced by ADE is application domain-independent. Reusing it requires further specialization to adapt assets for a specific application domain. We describe the specialization process below.

5.3. Specialization process

The second step of the ASPL strategy is to specialize a domain-independent ASPL platform for a specific domain, i.e., transform a horizontal ASPL platform into a vertical platform. Below we give an

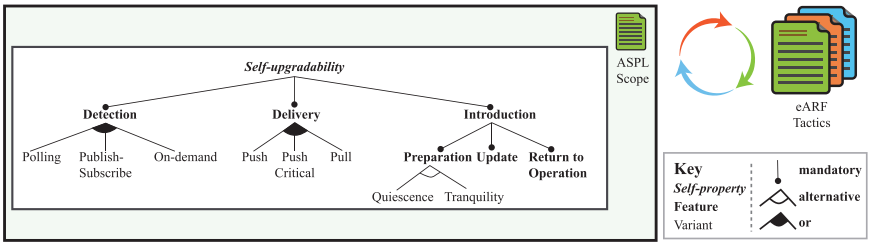


Fig. 8. ASPL scope definition - feature model.

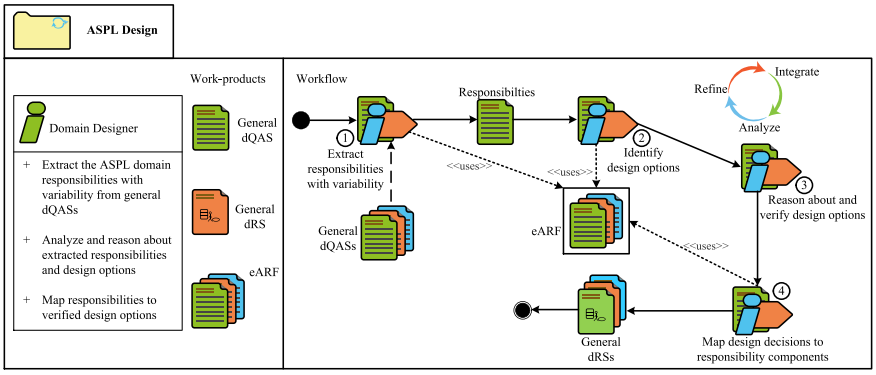


Fig. 9. ASPL design process package.

Table 2
Responsibilities extracted from the general dQAS for self-upgradability.

dQAS Elements	Responsibilities and Design Choices
Source	1. Updates Provider: serves as a source for the updates; may have the following variants: i) software (sub)system ii) system administrators iii) software developers 2. Update Consumer: requests for updates with the following variants: i) software (sub)system ii) end-user
Stimulus	1. Update Provider: (same as in the source element) 2. Update Consumer: (same as in the source element)
Artifact	1. Following responsibilities were identified for the Updates Manager Artifact: a) Monitor Updates: monitor updates provider for new updates i) Periodic polling ii) Event-based iii) On-demand b) Analyze: analyze new updates c) Plan: plan delivery and execution of the new updates i) Push-type ii) Critical Push-type d) Execute: execute the updates i) Quiescence ii) Rewriting Binary Code iii) Use of Proxies iv) Intrusion and Cooperation 2. The Update Manager cab be realized in two different ways: i) Centralized Control: a central component is responsible for all responsibilities. ii) Distributed Control: responsibilities are distributed among several components 3. Target System: uses updates, i.e., a software system on which updates are performed.
Environment	Runtime operating environment with normal work-load
Response	1. Updates Monitor: monitors the source variants for new updates; may have the following variants: i) Periodic polling ii) Event-based iii) On-demand 2. Analyzer: analyzes new updates 3. Update Manager: notifies updates to Target Systems. The notification leads to the following two response variants: i) Target Systems may accept and performs updates ii) Target Systems may postpone updates 4. Target System has the following two responsibilities i) request Update Manager for updates ii) download updates from the Update Manager and execute them
Response Measure	No new responsibility identified in this element.
Variants	No responsibilities but three variants for how an update is planned and performed are identified in this element: 1) Push, 2) Critical Push, and 3) Pull

overview of the specialization process with a focus on the design specialization process.

The starting point for the specialization process is the analysis of the ASPL scope and an application domain scope. The analysis is performed to identify self-adaptation properties supported by the ASPL platform. For supported properties, requirements specialization subprocess provides a workflow to identify and specialize (reuse) general dQASs that are available in the ASPL platform. If a general dQAS is identified, it is further analyzed and either constrained, for instance, by removing fragment variants or extended by adding fragment variants. The domain requirements for unsupported properties are specified by defining new dQASs.

The specialization process has a feedback mechanism that considers new and modified artifacts produced by the specialization

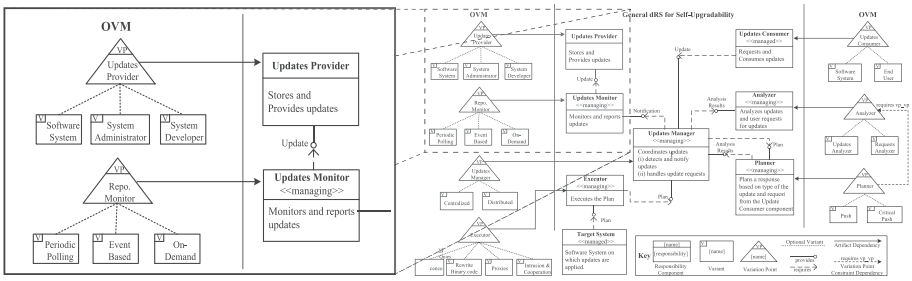


Fig. 10. A general dRS for self-upgradability.

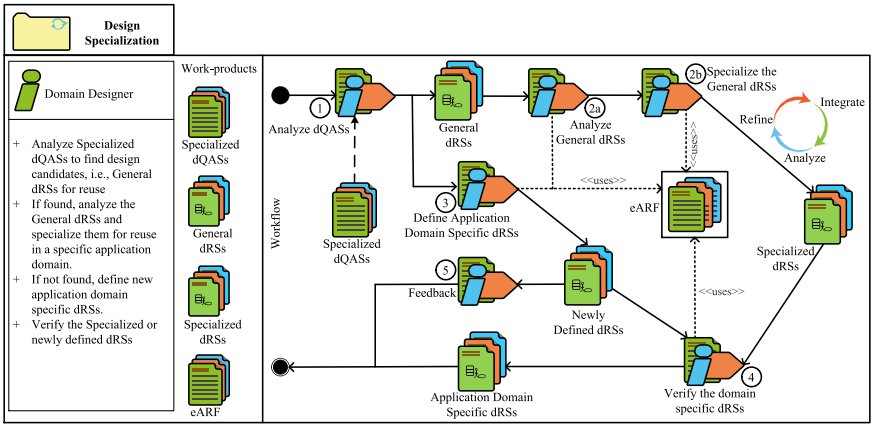


Fig. 11. Design specialization process package.

process for inclusion in the ASPL platform. For example, assume that an application domain requires parallel processing support for self-optimization, which is unsupported in the current ASPL platform. In this case, the specialization process develops artifacts for parallel processing and considers them for inclusion in the ASPL platform so that these artifacts can be reused in other fitness domains.

Requirements specialization is followed by design specialization. Fig. 11 depicts roles, work-products, and workflow of the design specialization process. Design specialization is in many aspects similar to the ASPL design process besides the parallel paths for reused and new design artifacts. The process begins by analyzing a set of specialized dQASs for self-adaptation properties required by an application domain. A domain designer analyzes the specialized dQASs and searches the ASPL platform for corresponding general dRSs, in activity ①.

If a general dRS is found, a designer analyzes it for missing responsibilities or responsibilities that require modifications. The analysis takes place in activity (2a), and the designer modifies the general dRS in activity (2b).

If a designer cannot find a general dRS, the alternative workflow-path creates a new application domain-specific dRS in activity ③. The new dRS is defined in a similar fashion as a general dRS is defined in the ADE. eARF supports activities (2a), (2b) and ③ with architectural knowledge.

All new and specialized elements in dRSs are integrated, analyzed, and refined to form a coherent design in ④. Modified and new dRSs defined in the activity ③ are considered for inclusion in the ASPL platform in activity ⑤.

To exemplify the design specialization process, the following is a description of how we performed this process to derive a vertical managing system platform for the DGE. The vertical platform was derived from the example ASPL platform that we developed earlier using the ASPL design process. Beginning with activity ①, we analyzed reusable requirements and design artifacts provided by the example ASPL platform. The ASPL platform supported self-upgradability; thus, a general dRS for self-upgradability was found. We analyzed the general dRS in activity (2a) and identified some gaps, mainly in the OVM part of the dRS. We addressed the gaps in activity (2b) to match the DGE's self-upgradability requirements. For example, there were three optional variants defined for the "updates provider" component in the general dRS. However, the DGE required only one of these variants; thus, we removed the other two variants. Fig. 12 depicts the resulting specialized dRS produced as a result of the activity (2b). The specialized dRS does not require further verification, activity ④, as no new elements were added to the dRS. Since no additions were made, thus nothing needs to be considered for inclusion in the ASPL platform as activity ⑤.

The specialization process corresponds to the ASPL strategy's second step and defines a vertical managing system platform. Following the ASPL strategy's third step, the managing system platform needs to be aligned and integrated with a separately developed managed system platform. To that end, ASPLe defines an integration process.

5.4. Integration process

The integration process defines activities, work-products, and roles to align and integrate separately developed managing and managed system platforms and establish a product line of self-adaptive systems. Developers use integrated platforms to derive self-adaptive systems using an application engineering process, such as the one described by Pohl et al. (2005).

The integration process analyzes managed and managing system platforms for data and behavioral mismatches. Most mis-

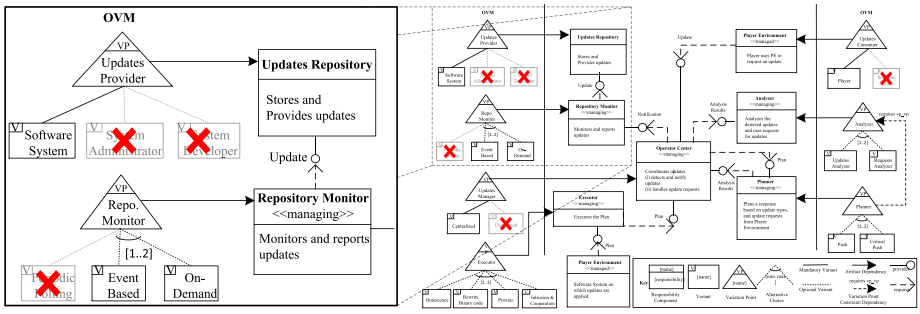


Fig. 12. The DGE domain-specific specialized dRS for self-upgradability.

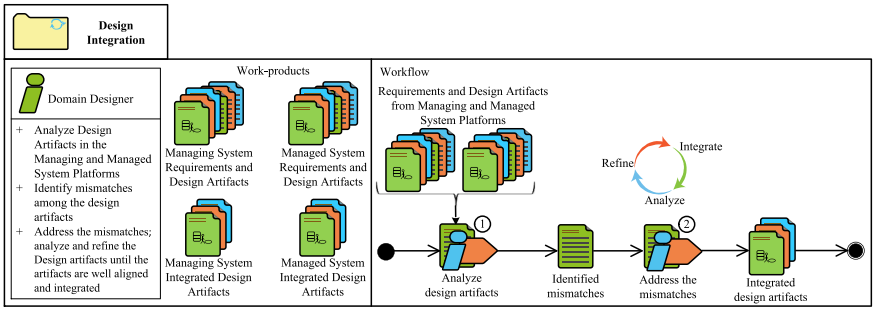


Fig. 13. Design integration process package.

matches are found in artifacts containing elements that cross platform boundaries, such as monitor and adapt interfaces. The integration process begins with the requirements integration subprocess. The requirements integration subprocess analyzes requirement specification artifacts in a managed system platform and connects them to specialized dQASs in a managing system platform. It results in integrated requirement artifacts that are used as key input by design integration process to analyze and address design mismatches.

Fig. 13 shows roles, work-products, activities, and workflow of the design integration process. The process begins with activity ① that analyzes design artifacts in managing and managed system platforms. The analysis pinpoints gaps in design artifacts. It also identifies architectural mismatches (Garlan et al., 1995), for example, interface incompatibilities.

Next, activity ② addresses mismatches by iteratively integrating, analyzing, and refining design artifacts. Refinement examples include the reconsideration of design decisions, for example, adding, removing, and changing components, their interfaces, variation points, and variants. The activity ② produces a set of adapted design artifacts that can be reused to design self-adaptive systems comprising both managed and managing subsystems.

To exemplify the design integration process, the following is a description of how we performed this process to integrate design artifacts of the separately developed managed and managing system platforms for our running example, the DGE. Beginning with activity ①, we analyzed design artifacts in both managing and managed system platforms. In the managing system platform, we analyzed the self-upgradability dRS shown in Fig. 12 and identified only one gap, a missing interface. The operator center requires a probe interface with PEs to retrieve runtime information. We addressed the gap in activity ② by adding the missing interface.

The managed system platform for the DGE was developed separately (Quan, 2013). In this platform, we analyzed the "player environment update" architectural view, depicted in Fig. 14(a). The OC and PE subsystems were connected by "data flows" without explicit interfaces. We accounted the lack of explicit interface as an architectural mismatch and addressed it in activity ② by adding interfaces to the OC and PE subsystems. Further, we annotated the OC as *managing*, the PE as *managed* subsystem components, and "Bundle Storage" as a *data store* component. Fig. 14(b) depicts the updated architectural view with the integration activity changes in focus.

6. Evaluation

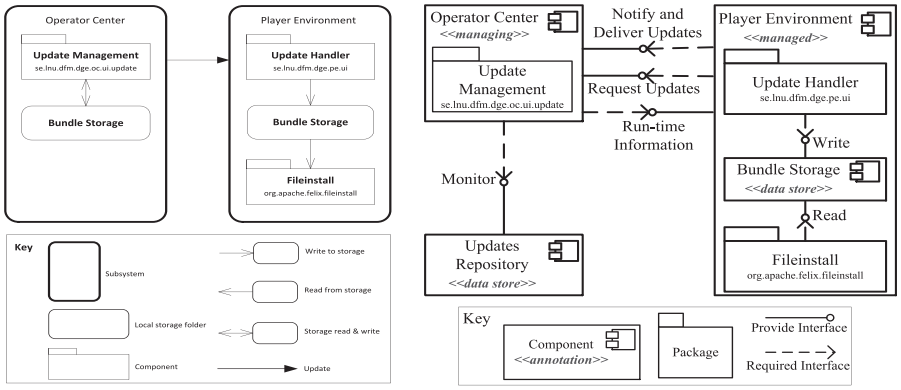
Evaluating ASPLe methodology as a whole is challenging, in fact, it must be evaluated in parts. Hence, for this work, we limited evaluation to ASPLe's design subprocesses. We extended a pilot case study (Abbas and Jesper, 2015) to do the evaluation. The extension was done to address the limitations we observed in the pilot study. While extending the study, we did not change its design and activities. We simply added more subjects, in total 22, and did rigorous hypotheses testing to strengthen the findings.

Both the pilot study and its extended version were done for the same purpose using the same design, activities, and data collection methods. Thus, from this point onward, we report both studies as a single integrated case study.

6.1. Design and planning

The case study was done as a part of a nine-week course. The course was aimed to provide theoretical knowledge of self-adaptive software systems and practical hands-on experience to design and implement such systems. The subjects were students of a final-year master's program in software engineering. The subjects had a solid knowledge of software design and implementation obtained from a mix of courses (15 ECTS on average) and practical experience between 0 and 5 years (1 year on average).

We designed and planned the case study activities using a template for the organization of case studies provided by Wohlin et al. (2012). As an essential part of the study, we planned the evaluation as a paired comparison of a reference approach and a treatment. We used ASPLe methodology as a treatment, and



(a) Before Integration (b) After Integration

Fig. 14. DGE managed system - self-upgradability architectural view.

Plan, Design and Prepare			
Part I: Reference Approach		**Part II: ASPLe Approach**	
Week 1	Introductory Lecture (2 hours) & Distribution of Home Assignment	**Week** 6	Introductory Lecture on Service Reusability (2 hours)
2	Home Assignment Discussion (2 hours)	7	Introduction to the ASPLe, eARF, and an Example SPL (2 hours)
3	Introduction to the Reference Approach and an Example SPL (2hours)	8	Second Preparatory Workshop (3 hours)
4	First Preparatory Workshop (3 hours)	9	Assignment A3 (3 hours)
5	Assignment A1 (3 hours)		Assignment A4 (3 hours)
	Assignment A2 (3 hours)		Final Questionnaire (30 minutes) and Interviews (≈ 20 - 30 minutes per interview)
Data Analysis			

Fig. 15. An overview of the case study.

Monitor-Analyze-Plan-Execute-Knowledge feedback loop, or *MAPE-K* in short (Kephart and Chess, 2003), as a reference approach. MAPE-K is a widely used model to design managing subsystems for self-adaptive software systems. From our experience of teaching self-adaptation, MAPE-K is an intuitive and straightforward approach to apply (Weyns et al., 2013a; Iglesia and Weyns, 2015). It defines four activities with clear responsibilities and a supporting *knowledge* component.

We divided the case study activities into two parts, one for the ASPLe and one for the reference approach. Fig. 15 provides an overview of the case study activities. We structured both parts in the same way. We began with introductory lectures, did preparatory workshops, and asked subjects to do test assignments. The lectures introduced the cases, while the workshops prepared subjects for the assignments. In the end, we interviewed subjects and asked them to answer a questionnaire.

6.2. Hypothesis formulation

The main objective of the case study was to compare ASPLe with the MAPE-K reference approach for design support to *maximize software reuse* and *mitigate uncertainties*. Uncertainties often lead to subpar design decisions that in turn introduce faults to a system (Ziv et al., 1996). This is a well-known cause and effect relationship. Thus, we used fault density as a measure of uncertainty mitigation in hypothesis formulation and testing.

The main objective was split into two sub-objectives that we formulated as hypotheses H_{01} and H_{02}. For the formulation of the hypothesis, we follow the guidelines of Wohlin et al. (2012), and Juristo and Moreno (2010). In particular, we mapped each sub-objective to a null hypothesis (H_0) that should be tested, and an alternative hypothesis (H_a) that should be accepted if and only if the null hypothesis is rejected. In the hypotheses, we use μ to denote average, while *total_reuse(input: approach)* and *fault_density(input: approach)* represent functions to compute "total reuse" and "fault density", respectively.

- H_{01}: There is no difference in *total reuse* achieved for a self-adaptive system design produced using the reference approach, and a design produced using ASPLe.

$$H_{01} : \mu\ total\_reuse(Reference) = \mu\ total\_reuse(ASPLe) \quad (1)$$

$$H_{a1} : \mu\ total\_reuse(Reference) < \mu\ total\_reuse(ASPLe) \quad (2)$$

- H_{02}: There is no difference in *fault density* for a self-adaptive system design produced using the reference approach, and a design produced using ASPLe.

$$H_{02} : \mu\ fault\_density(Reference) = \mu\ fault\_density(ASPLe) \quad (3)$$

$$H_{a2} : \mu\ fault\_density(Reference) > \mu\ fault\_density(ASPLe) \quad (4)$$

6.3. Independent and dependent variables

Hypothesis testing requires that we define independent and dependent variables, and select appropriate metrics to measure treatments' effects on dependent variables (Wohlin et al., 2012).

6.3.1. Independent variables

Independent variables are the variables that are controlled and manipulated to study treatments' effects. We identified two independent variables:

a) Test Assignment: The first independent variable we used is the "Test Assignment". It has four instances, A1, A2, A3, and A4; see Section 6.4.1 for details.

b) Approach: The second independent variable is the "Approach" to solve the assignments. It has two instances, ASPLe, and the MAPE-K reference approach. We controlled this variable to study the effect on the dependent variables.

6.3.2. Dependent variables

Dependent variables are variables studied to understand treatments' effects on a subject's performance. These are often derived directly from hypotheses (Wohlin et al., 2012). We derived two dependent variables *total-reuse* and *fault-density* from the hypotheses H_{01} and H_{02}, respectively, and selected appropriate metrics to measure treatments' effect.

a) Total-Reuse: This variable was used to study and compare software reuse support provided by ASPLe and the reference approach. We used the Total Reuse Level (TRL) metric (Frakes and Terry, 1996) to measure the total-reuse variable. The rationale for this decision is that the TRL is the best match for reuse aspects we want to study, that is, design with and design for reuse.

$$\text{Total Reuse Level (TRL)} = \text{External Reuse Level} + \text{Internal Reuse Level} \quad (5)$$

$$\text{External Reuse Level} = E/L \quad (6)$$

$$\text{Internal Reuse Level} = M/L \quad (7)$$

L – the number of lower-level items in a higher-level item.
E – the number of lower-level items reused from an external repository.
M – the number of lower-level items not from an external repository but used more than once.

The reuse level variables may take values in a range from 0 (no reuse) to 1 (maximum reuse). In line with (Frakes and Terry, 1996), we consider products (self-adaptive systems) designed in the case study as higher-level items and artifacts that compose a product, such as design components, as lower-level items. Thus, we computed L by counting a product's components. We counted correctly reused components from an external repository to compute E. We labeled components suggested by ASPLe or the MAPE-K reference approach, such as monitor and analyze components, as external repository artifacts. We used a checklist to determine if an artifact was reused correctly. M was computed by counting internally developed lower-level artifacts (not belonging to an external repository) that were used more than once.

b) Fault-Density: We used fault-density as an indirect measure of uncertainty mitigation. Fault density, defined in Eq. (8), is the number of *faults* divided by the *size* of a software system (Fenton and Neil, 2000). A fault is the manifestation of errors in a software system (610.12-1990, 1990). We compared subjects' solutions with a reference solution to identify and count faults. We labeled an incorrect step or missing element as a fault.
We used a high-level function point analysis method described by Peeters et al. Peeters et al. (2005) to calculate a system size. We computed three scores for the size in function points, *minimum, expected*, and *maximum*, and took their average as an estimate.

$$\text{Fault Density} = \text{Faults} \;/\; \text{Size} \quad (8)$$

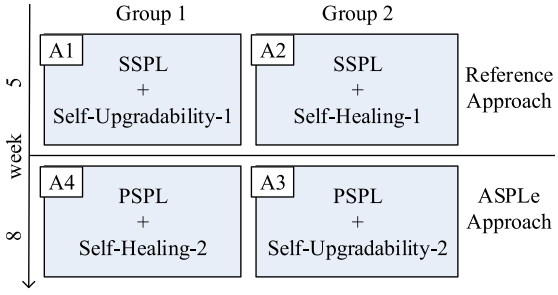


Fig. 16. Test assignments - design.

6.4. Data collection methods

Data in the case study were collected using three methods: *test assignments, questionnaires*, and *interviews*. All material of the study, including test assignments, questionnaires, and interviews data, can be downloaded from the study homepage[2].

6.4.1. Test assignments

We designed four test assignments, which were labeled as A1, A2, A3, and A4, for the case study. The assignments A1 and A2 were performed using MAPE-K (Kephart and Chess, 2003) as a reference approach, whereas the assignments A3 and A4 were performed using ASPLe as a treatment.

Each assignment contained three tasks. The focus of all tasks was on activities in the design phase. In Task 1, subjects were requested to extend two products with reuse from a set of core assets to realize a self-adaptation property. Task 2 requested subjects to extend core assets with reuse candidates developed in Task 1. Task 3 challenged subjects to design a new self-adaptive product with reuse from the extended set of core assets.

From ASPLe perspective, Tasks 1 and 3 covered the design specialization activities ③ and ④, eARF activities ③ – ⑥, and partially covered ASPL design activities ① – ④ and the design specialization activity ①. Task 2 covered the feedback to ASPL platform activity ⑤ in the design specialization process.

To do the assignments, we divided subjects into two groups according to the block subject-object study classification setup (Wohlin et al., 2012). The block subject-object classification ensures that each subject receives both the treatment and the reference approach, which enables a paired comparison of the two approaches. Two example product lines, Soft-Phones Software Product Line (SSPL) and PhotoShare Software Product Line (PSPL), were used as example application domains for the assignments. Both SSPL and PSPL required self-healing and self-upgradability properties. For each assignment, Fig. 16 depicts a combination of an example SPL and a self-management property, for example, assignment A2 uses SSPL with self-healing as a problem domain.

6.4.2. Questionnaires and interviews

Besides test assignments, we collected data using two questionnaires and additional semi-structured interviews. The first questionnaire has two variants, 1) pre-test and 2) post-test. Both variants have a mix of open and closed type questions (Gray, 2013), and each has six questions in total. We used the two variants to identify false positives and false negatives in the assignment data. The identification was done by requiring subjects to answer the pre-test variant before, and the post-test variant after each test assignment, and then by comparing and analyzing responses for the two variants.

[2] https://people.cs.kuleuven.be/~danny.weyns/material/2020JSS/index.htm.

Table 3
Statistical tests results.

Variable	Task ID	Approach	Shapiro-Wilk (Normality Test) p-value	Test Selected for Hypothesis Testing	Selected Test's p-value
Total Reuse Level (TRL)	Task 1(a)	Reference	0.573	Paired-samples *t*-test	0.000
		ASPLe/eARF	0.467		
	Task 1(b)	Reference	0.706	Paired-samples *t*-test	0.000
		ASPLe/eARF	0.924		
	Task 3	Reference	0.845	Wilcoxon test	0.002
		ASPLe/eARF	0.009		
Fault Density (FD)	Task 1(a)	Reference	0.010	Wilcoxon test	0.000
		ASPLe/eARF	0.039		
	Task 1(b)	Reference	0.017	Wilcoxon test	0.000
		ASPLe/eARF	0.492		
	Task 3	Reference	0.098	Paired-samples *t*-test	0.000
		ASPLe/eARF	0.042		

Table 4
Descriptive statistics for part-C of final questionnaire.

Comparison Criteria	ASPLe is Better	Reference is Better	Both are Equal
Knowledge	36.36%	36.36%	27.27%
Comprehension	18.18%	18.18%	63.64%
Application	22.73%	22.73%	54.55%
Separation of Concerns	45.45%	4.55%	50%
Being Intuitive and Helpful	54.55%	22.73%	22.73%

The second "final questionnaire", appended as Appendix A, was used for cross verification of assignment data after subjects completed their assignments. It was designed as a self-completing questionnaire with closed type questions structured in four parts: A, B, C, and D. Parts A and B collected subjects background data and data focusing on their understanding of ASPLe. Part C required respondents to compare and rate approaches on Likert scales according to comparison criteria listed in Table 4. The first three criteria, *knowledge, comprehension,* and *application*, were derived from Bloom's taxonomy of learning (Bloom et al., 1956). These are used to compare ASPLe and the MAPE-K reference approach with respect to challenges in understanding and learning. The fourth criterion focuses on the separation of concerns, which Tarr et al. (1999) pin-point as a key design principle to resolve complexity. The final criterion, "being more intuitive and helpful", targets respondents' subjective opinions on design support provided by ASPLe and the reference approach.

Part D included four statements that asked respondents to compare the two approaches on a scale with five options spanning a range from "strongly disagree" to "strongly agree" and a sixth "don't know" option.

At the end of the case study, we interviewed subjects to clarify questionnaire responses and collect additional details. Interviews were done in a semi-structured way. Twenty-one out of the twenty-two subjects participated in the interviews. The interviews were recorded and transcribed before the analysis.

6.5. Data analysis

We analyzed the case study data in two parts. As described below, first, we analyzed assignments data, and then we analyzed questionnaires and interviews data.

6.5.1. Assignments data analysis

We analyzed the data of the assignments to test the hypotheses. For each assignment, we analyzed task 1(a), task 1(b), and task 3. We excluded task 2 as it did not involve a design effort. The analysis focused on the dependent variables, *total-reuse*, and *fault-density*. Table 3 summarizes statistical tests results for both variables.

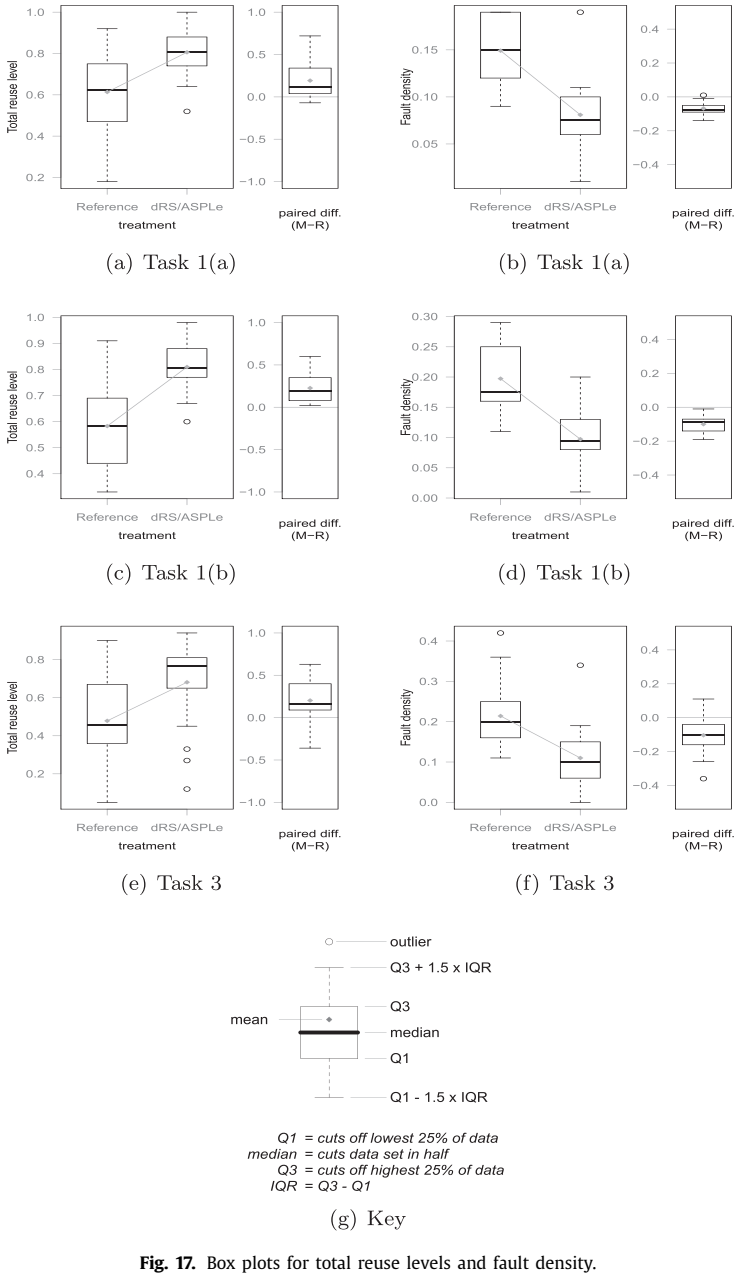
A hypothesis can be tested using statistical tests such as *t*-test, paired *t*-test, and ANOVA (Wohlin et al., 2012; Gray, 2013). Statistical tests differ in statistical power, which is a probability of a test to find an effect if there is an effect to be found. Paired-samples *t*-test and Wilcoxon tests are recommended for studies where two approaches, such as ASPLe and the MAPE-K, are compared, and every subject has used both approaches (Wohlin et al., 2012). The choice of selecting a test mainly depends on the distribution of data. Thus, to choose an appropriate test, we first determined the distribution of the assignments data. To that end, we used a commonly used normality test, Shapiro-Wilk (Razali and Yap, 2011; Shapiro and Wilk, 1965), with a significance level (α) of 0.05. Then, based on the results of the Shapiro-Wilk test, we selected Paired-samples *t*-test for normally distributed data and Wilcoxon test for other distributions. This makes the selected tests best match to the design and objective of our case study.

For all tasks, both the tests rejected both the null hypotheses, H_{01} for *total reuse* and H_{02} for *fault density*, with a significance level α of 0.05. Thus, we may accept the alternative hypotheses. This is a strong indication that the architectural analysis and reasoning support provided by ASPLe helps improving total reuse and lowering fault density compared to the reference approach. The box-plots in Fig. 17 provide a visual representation of the hypotheses testing results.

The box-plots labeled as "paired-diff" plot paired differences between ASPLe and the reference approach. A paired difference Z_i is defined as $Zi = Ti - Ri$ for $i = 1, \ldots, n$. Ti and Ri represent measurements of subject i for ASPLe and the reference approach, respectively, where n is the number of subjects. Subjects differ less in their values for fault density than their values for TRL, for all the three tasks. The variance measures a treatment's relative contribution. A small variance indicates that a treatment contributed uniformly to all subjects' performance.

6.5.2. Questionnaires and interviews data analysis

Questionnaires and interviews data were analyzed for triangulation (Gray, 2013). We analyzed parts C and D of the final questionnaire. Parts A and B collected subjects' specific data and thus did not directly contribute to the evaluation. Table 4 specifies descriptive statistics for the part C. Subjects rated the two approaches as equal for *knowledge, comprehension,* and *application* criteria. About 45% of the subjects rated ASPLe as a better approach for the separation of concerns criterion compared to the only 5% who fa-



(a) Task 1(a) (b) Task 1(a)

(c) Task 1(b) (d) Task 1(b)

(e) Task 3 (f) Task 3

(g) Key

Fig. 17. Box plots for total reuse levels and fault density.

vored the reference approach. The respondents also ranked ASPLe as more intuitive and helpful.

Pie charts in Fig. 18 depict questionnaire data for the first two statements in part D.

Statement 1: "As compare to the reference approach, the dRS approach provides more intuitive approach to transform quality attribute (self-management properties) requirements into responsibilities and components designed to fulfill these responsibilities?"

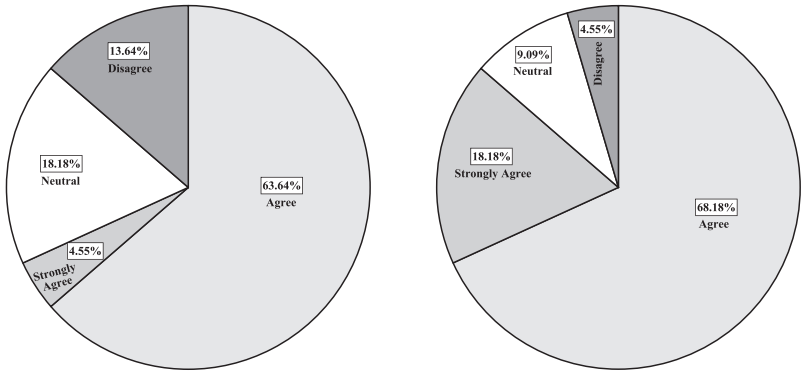
Statement 2: "As compare to the reference approach, the dRS approach provides better over-all support to realize domain quality attributes with self-adaptive characteristics, such as self-healing, self-optimization, self-upgradability, etc."

For both statements, a majority of the respondents, 68% and 86% respectively, either agreed or strongly agreed in favor of the dRS. dRS, described in Section 5.1.2, is a core component of the reasoning framework, eARF, provided by ASPLe. For the third statement, which compares the support for designing quality attributes with runtime changes, 50% agreed or strongly agreed in favor of ASPLe, 31% remained neutral, and 18% disagreed. For the fourth statement, which compares architectural analysis and reasoning support, 80% favored ASPLe.

Statistics we derived from the final questionnaire's parts C and D indicated which approach is favored by subjects, but the questionnaire data did not provide any rationale. For example, why subjects ranked one approach as better or equal for a particular criterion. We analyzed interview data to find rationale for subjects' decisions.

The interview data were qualitative, thus, we used a qualitative content analysis approach (Gray, 2013) to analyze it. We defined a set of keywords and phrases (Table 5) and used them as selection criteria to code and categorize the data. The keywords in *italics* were identified and added to the list during the coding activity.

We read all interview transcripts and marked words, phrases, and sentences having one or more keywords. We classified the



(a) Statement 1 (b) Statement 2

Fig. 18. Pie charts for questionnaire data .

Table 5
Code categories and keywords for qualitative content analysis.

Categories	Sub-categories	Keywords/Phrases
1. ASPLe	a) Learning	knowledge, comprehension, understanding, application, use, *difficult/difficulties, confusion*
2. Reference	b) Analysis and Reasoning Support	tactics, patterns, separation of concerns, (functional) requirements, quality attributes, responsibilities, analysis, design, components, variability, intuitive, helpful, *architecture, design decisions, reasoning*

marked data, first, into ASPLe and Reference categories to support a paired comparison, and then into one subcategory for learning, and one for analysis and reasoning support.

We examined the learning subcategory to understand why subjects rated the two approaches equally for the knowledge, comprehension, and application criteria. Examination revealed that ASPLe uses MAPE-K loop (Kephart and Chess, 2003), the reference approach, as a principal architectural pattern. Thus, the subjects did not see additional contributions from ASPLe as significant and rated two approaches on equal scales.

We used the "analysis and reasoning support" subcategory to analyze the subjects' responses for parts C and D of the final questionnaire. For part C, we used it to analyze statistics for the "separation of concerns" and "being intuitive and helpful" criteria items. For part D, we used it to understand the statements data, for instance, "what made subjects to agree, disagree, or stay neutral for the given statements. The analysis highlights the following four key characteristics of ASPLe, which motivated a majority of the subjects, see Table 4 and Fig. 18, in favor of ASPLe.

i) Step-by-step and well-defined processes.
ii) A disciplined split between managed and managing subsystems, and a clear separation of responsibilities, i.e., concerns.
iii) Provisioning of self-management property specific architectural patterns and tactics that serve as a reusable design library.
iv) Explicit support to identify, model, constrain, and manage variability.

6.6. Threats to validity

Various factors, such as design, plan, investigated variables, and metrics, may cause threats to the validity of a study. We use a scheme suggested by Runeson et al. (2012) to discuss threats and measures we took to mitigate these threats.

6.6.1. Construct validity

Construct validity considers a case study's design, plan, and execution (Gray, 2013; Runeson et al., 2012). If one aligns a case study's elements, such as objectives, data collection and analysis methods, and metrics, with the investigated research problem, the threat to construct validity is smaller.

Our case study was designed and executed by researchers who contributed ASPLe parts under evaluation. There is a threat that the researchers design a case study that favors ASPLe. We addressed this threat by inviting an independent senior researcher who reviewed and approved the case study's design and execution plan.

The case study extends a previous evaluation (Abbas and Jesper, 2015). The variations in the two evaluations' elements and activities may put the validity at risk. To mitigate the risk, we used the same activities and data collection methods for the extension. Moreover, we combined data from the two studies and re-analyzed the combined data using the same analysis methods.

Another threat is that subjects may guess which answers researchers are looking for and adapt their responses accordingly. We mitigated this threat by presenting the case study as regular coursework. Moreover, we informed the subjects that the case study results will not affect their grades.

6.6.2. Internal validity

Internal validity refers to threats caused by causal relations among investigated factors. While investigating two factors with a cause-effect relationship, for instance, horizontal reuse and uncertainty, researchers may ignore the third factor, for instance, runtime variability. To mitigate such threats, we did not weigh the effect of the reuse and runtime variability factors separately or differently. This is because the case study objective was to evaluate the uncertainty mitigation support provided by ASPLe, and not which factor causes more uncertainty. To further improve the internal validity, we applied data triangulation (Gray, 2013). We collected both qualitative and quantitative data from multiple sources, including test assignments, questionnaires, and interviews, and analyzed the data using multiple analysis methods such as hypothesis testing, graphs, and qualitative content analysis.

6.6.3. External validity

External validity refers to what extent the case study findings are generalizable. We mitigated this threat by involving a larger number of subjects in the evaluation. Furthermore, we strengthened the external validity through triangulation by collecting data from multiple sources. Nevertheless, since ASPLe

is only evaluated for two cases, additional studies will be required to strengthen its generality. To support such studies, we have documented the case study objective and other details at the study homepage https://people.cs.kuleuven.be/~danny.weyns/material/2020JSS/index.htm. Additional context and extended viewpoint descriptions help readers to understand the study, which is a support when transferring the findings to other settings (Runeson et al., 2012).

6.6.4. Reliability

Reliability threats refer to conditions that may impede achieving the same results if another researcher replicates the study. One such threat is shallow and incomplete case study documentation. To that end, we have documented design, planning, data collection and analysis methods, and the data of our study that can be downloaded from the study homepage.

6.7. Discussion

The evaluation focuses on the three objectives specified in Section 3.4. The first and second objectives specify that ASPLe should provide systematic process support for the development of self-adaptive systems with reuse. The third objective specifies that ASPLe should provide architectural analysis and reasoning support to manage uncertainty and variability in the design of self-adaptive systems with reuse.

A comprehensive assessment of a complex process like ASPLe is not feasible. Our evaluation focuses on the design subprocesses and, more specifically, the eARF core component, its activities, and work-products. The results of the evaluation support our claim that ASPLe satisfies all its objectives. The evaluation primarily evaluates the design processes and not the whole ASPLe. However, the design processes involve activities, such as decision making and trade-offs, which are the most affected activities by uncertainty and variability.

The first objective for ASPLe, O1, is concerned with the separation of concerns (SoC), a design property that reduces design complexity and increases reusability. The results indicate that ASPLe leverages separation of concerns between managed and managing systems. About 95% of subjects, according to the data in Table 4, rated ASPLe as better or equal to the reference approach for SoC. It is essential to keep in mind that the MAPE-K reference approach centers around monitoring, analysis, planning, execution, and knowledge. Still, only 5% ranked it as better than eARF. The subjects argued that they found the reference approach very abstract, and it was difficult to distinguish between concerns such as analysis and planning.

The second objective, O2, focuses on improving reuse and product quality with a process. Based on statistical analysis, we claim that ASPLe provides process support for reuse and product quality. The analysis shows that ASPLe delivers self-adaptive systems designed with a higher degree of total reuse and with reduced fault density. We find additional support for our claim in questionnaire data (Table 4) where a majority, 54% of the subjects, rated ASPLe as more intuitive and helpful than the reference approach.

We have statistically significant results in favor of ASPLe for the third objective, O3, which is concerned with architectural reasoning and decision support. ASPLe provides improved reasoning and decision support in the form of eARF, which reduces design faults with statistical significance. eARF's additional architectural knowledge and explicit support for variability modeling help mitigating uncertainties in reasoning and decision making.

The questionnaires and interviews provide additional results regarding the comprehension and application of ASPLe. The data in Table 4 rates both approaches as equal. An analysis of the interview data shows that subjects who rated ASPLe as difficult argued that ASPLe requires more descriptive material, while the reference approach was self-descriptive.

7. Conclusion

The primary goal of this work is to provide support for software engineers to design and develop self-adaptive systems with reuse and to mitigate uncertainties. To that end, we present and evaluate a novel methodology called ASPLe. ASPLe defines roles, activities, work-products, and workflows to repeatedly support the development of self-adaptive software systems with improved quality and enhanced reuse. It exploits a strict separation of concerns, which makes assets more reusable and reduces impediments to reuse self-adaptation assets. Furthermore, ASPLe contributes a stepwise approach to select, specialize, and integrate reusable artifacts to develop self-adaptive software systems. Moreover, ASPLe includes support for architectural analysis, reasoning, and decision making to reduce the impact of design time and runtime uncertainties.

We have conducted a case study that combines quantitative and qualitative methods to evaluate a critical path through ASPLe with a focus on design activities. The quantitative evaluation based on data from a controlled experiment shows a statistically significant increase in software reuse and a decrease in uncertainty in systems designed using ASPLe. We conducted a series of questionnaires and interviews to provide additional data for triangulation. The data further supports our claim that the ASPLe design support is better than the MAPE-K reference approach.

Although the evaluation indicates that the design support improves on the existing state of practice, several challenging tasks remain before ASPLe is a complete methodology. Currently, ASPLe supports requirements and design activities, and we plan to extend this with support for implementation and testing, where we conjecture that the testing support will be the most challenging. The methodology will also require more evaluation both in controlled environments and in real-world settings. The evaluation also indicates that ASPLe and its process components will benefit from improved descriptions, more examples, and tutorials, which we also put on our roadmap.

ASPLe methodology is defined on purpose to support the development of self-adaptive software systems with reuse. However, we believe that it can be adopted to develop other systems with similar needs such as Dynamic Software Product Lines (Hallsteinsen et al., 2008), Self-Managing Internet of Things (Weyns et al., 2018), and Cloud-Native Applications (Toffetti et al., 2017). Thus, we plan and invite others to investigate and extend, if needed, the ASPLe process support for the development of other related systems.

Declaration of Competing Interest

None.

Appendix A. Final Questionnaire

- This questionnaire is designed to measure learning outcomes and to collect feedback on the course 4DV610. It has nothing to do with examination or grades in the course. So feel free to express your opinion while answering the questionnaire.
- To measure learning outcomes, we use first three levels of Bloom's taxonomy[3], as defined below:

  Knowledge is the lowest level of learning outcomes and is defined as remembering of previously learned material. This

[3] Forehand, Mary. "Bloom's taxonomy". Emerging perspectives on learning, teaching, and technology (2010)

may involve the recall of a wide range of material, from specific facts to complete theories, but all that is required is the bringing to mind of the appropriate information.

Comprehension is one step higher than the knowledge, and is defined as the ability to grasp the meaning of material. This may be shown by translating material from one form to another (words to numbers), by interpreting material (explaining or summarizing), and by estimating future trends (predicting consequences or effects).

Application is one step higher than the comprehension level, and refers to the ability to use learned material in new and concrete situations. This may include the application of such things as rules, methods, concepts, principles, laws, and theories.

- For Multiple choice questions, please encircle the choice(s) which best matches to your answer.

Part A - Background Questions

1. Which degree program are you currently enrolled?
   a. Master b. —— (Please specify, if other)
2. What is your major subject?
   a) Computer Science/Software Technology b) —— (Please specify, if other)
3. How many credits have you earned so far?
   a. 30 or Less b. 60 or Less c. 120 or Less d. More than 120
4. Out of these credits, how much come from the courses primarily focusing on *software design* and *architecture*?
   a. 7.5 or less b. 15 or less c. 30 or less d. More than 30
5. How do you rate your *knowledge* of Software Design and Architecture on a scale from 1 to 6?
   1. [Poor] 2. 3. 4. 5. 6. [Excellent]
6. How do you rate your *comprehension* of Software Design and Architecture on a scale from 1 to 6?
   1. [Poor] 2. 3. 4. 5. 6. [Excellent]
7. How do you rate your *application skills* of Software Design and Architecture on a scale from 1 to 6?
   1. [Poor] 2. 3. 4. 5. 6. [Excellent]
8. Do you have working experience in industry with focus on software design and engineering?
   a) No b) 6 Months or Less c) 1 Year or less d) 1 to 5 Years e) More than 5 Years

Part B - domain Responsibility Structure (dRS) Approach

9. Which one of the following statements best describes the domain Responsibility Structure (dRS)?
   a) The dRS is a design pattern that offers a reusable solution to a commonly occurring problem within a given context in software design.
   b) The dRS is a set of (responsibility) components and connectors that together form reference architecture to realize quality attributes, such as self-management properties.
   c) The dRS is a framework that provides a reusable platform to develop software systems.
   d) Don't know.
10. The core of the dRS is a *self-adaptation* (MAPE-K) which is used as principle runtime variability mechanism, supported through available architectural knowledge in the form of *architectural patterns* and *tactics*.
    a) True
    b) False
    c) Don't know
11. The dRS approach uses to model domain variability. (Select only one option)
    a) Feature Diagram
    b) Traditional Software Development Diagrams such as Use Case, Sequence Diagrams, etc.
    c) An Orthogonal Variability Model (OVM)
    d) Don't know.

Part C: In this part we rate and compare the two approaches that were used during the two class assignments to design or redesign products with given self-adaptation requirements. To distinguish the two approaches, let's call the approach used in the first assignment as ***the reference approach***, and the other used in the second assignment as ***the dRS approach***. Please use "Ref Scale" for the reference approach, and "dRS Scale" for the dRS approach, wherever applicable.

12. How do you rate the two approaches from gaining *knowledge* point of view, on a scale from 1 to 6?
    Ref. Scale: 1. [Very Simple] 2. 3. 4. 5. 6.[Too Complex]
    dRS Scale: 1. Very Simple 2. 3. 4. 5. 6. Too Complex]
13. How do you rate the two approaches from *comprehension* point of view, on a scale from 1 to 6?
    Ref. Scale: 1. [Very Simple] 2. 3. 4. 5. 6. [Too Complex]
    dRS Scale: 1. [Very Simple] 2. 3. 4. 5. 6.[Too Complex]
14. How do you rate the two approaches from *application* point of view, on a scale from 1 to 6?
    Ref. Scale: 1.[Very Simple] 2. 3. 4. 5. 6.[Too Complex]
    dRS Scale: 1.[Very Simple] 2. 3. 4. 5. 6.[Too Complex]
15. How do you rate the two approaches from *separation of concerns* point of view, on a scale from 1 to 6?
    Ref. Scale: 1.[Poor] 2. 3. 4. 5. 6. [Excellent]
    dRS Scale: 1.[Poor] 2. 3. 4. 5. 6.[Excellent]
16. How do you rate the two approaches from being more intuitive and helpful in terms of available documentation for analysis and design process, on a scale from 1 to 6?
    Ref. Scale: 1.[Poor] 2. 3. 3. 4. 5. 6. [Excellent]
    dRS Scale: 1.[Poor] 2. 3. 4. 5. 6.[Excellent]

For Questions 17 to 20: We compare the two approaches by making a statement and require you to encircle only one of the given options that best matches to your opinion.

17. As compare to the reference approach, the dRS approach provides more intuitive approach to transform quality attribute (self-management properties) requirements into responsibilities and components designed to fulfill these responsibilities?
    (a) Strongly Disagree
    (b) Disagree
    (c) Neutral
    (d) Agree
    (e) Strongly Agree
    (f) Don't Know
18. As compare to the reference approach, the dRS approach provides better over-all support to realize domain quality attributes with self-adaptive characteristics, such as self-healing, self-optimization, self-upgradability, etc.
    (a) Strongly Disagree
    (b) Disagree
    (c) Neutral

(d) Agree
(e) Strongly Agree
(f) Don't Know

19. As compare to the reference approach, the dRS approach provides better over-all support to realize domain quality attributes characterized with runtime changes in their requirements.
(a) Strongly Disagree
(b) Disagree
(c) Neutral
(d) Agree
(e) Strongly Agree
(f) Don't Know
20. As compare to the reference approach, the dRS approach provides better architectural support in the form of *architectural patterns* and *tactics*?
(a) Strongly Disagree
(b) Disagree
(c) Neutral
(d) Agree
(e) Strongly Agree
(f) Don't Know

CRediT authorship contribution statement

Nadeem Abbas: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Writing - original draft. **Jesper Andersson:** Supervision, Conceptualization, Methodology, Writing - original draft. **Danny Weyns:** Validation, Formal analysis, Writing - review & editing.

References

Abbas, N., 2018. Designing Self-Adaptive Software Systems with Reuse. Linnaeus University, Department of computer science and media technology (CM).

Abbas, N., Andersson, J., 2013. Architectural reasoning for dynamic software product lines. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops. ACM, New York, NY, USA, pp. 117–124. doi:10.1145/2499777.2500718.

Abbas, N., Andersson, J., 2015. Harnessing variability in product-lines of self-adaptive software systems. In: Proceedings of the 19th International Conference on Software Product Line (SPLC). ACM, New York, NY, USA, pp. 191–200. doi:10.1145/2791060.2791089.

Abbas, N., Andersson, J., Ifikhar, M.U., et al., 2016. Rigorous architectural reasoning for self-adaptive software systems. In: 1st Workshop on Qualitative Reasoning about Software Architectures. IEEE, pp. 1–8.

Abbas, N., Andersson, J., Weyns, D., 2012. Modeling variability in product lines using domain quality attribute scenarios. In: Proceedings of the WICSA/ECSA 2012 Companion Volume. ACM, New York, NY, USA, pp. 135–142. doi:10.1145/2361999.2362028.

Abbas, N., Jesper, A., 2015. Architectural reasoning support for product-lines of self-adaptive software systems - a case study. In: Weyns, D., Mirandola, R., Crnkovic, I. (Eds.), Proceedings of the 9th European Conference on Software Architecture (ECSA). Springer, pp. 20–36.

Andersson, J., Baresi, L., Bencomo, N., et al., 2012. Software engineering processes for self-adaptive systems. In: Software Engineering for Self-adaptive Systems 2. In: Lecture Notes in Computer Science, vol. 7475. Springer, pp. 51–75.

Andersson, J., et al., 2009. Modelling dimensions of self-adaptive software systems. LNCS. Software Engineering for Self-Adaptive Systems, vol. 5525. Springer.

Asadollahi, R., Salehie, M., Tahvildari, L., 2009. 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, pp. 58–67. doi:10.1109/SEAMS.2009.5069074.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison-Wesley Professional.

Bass, L., Ivers, J., Klein, M.H., et al., 2005. Reasoning Frameworks. Technical Report. Software Engineering Institute, Carnegie Mellon University.

Bloom, B.S., Engelhart, M.D., Furst, E.J., et al., 1956. Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain, vol. 19. New York: David McKay Co Inc.

Calinescu, R., Weyns, D., Gerasimou, S., et al., 2018. Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Trans. Softw. Eng. 44 (11), 1039–1069. doi:10.1109/TSE.2017.2738640.

Cheng, B., de Lemos, R., Giese, H., et al., 2009. Software engineering for self-adaptive systems: a research roadmap. Softw. Eng. Self-Adapt. Syst., 1–26.

De Lemos, R., Giese, H., Müller, H.A., et al., 2013. Software engineering for self-adaptive systems: a second research roadmap. Software Engineering for Self-Adaptive Systems II. Springer, pp. 1–32.

Diaz-Pace, A., Kim, H., Bass, L., et al., 2008. Integrating quality-attribute reasoning frameworks in the arche design assistant. In: Becker, S., Plasil, F., Reussner, R. (Eds.), Quality of Software Architectures. Models and Architectures. In: Lecture Notes in Computer Science, vol.~5281. Springer Berlin Heidelberg, pp. 171–188. DOI: 10.1007/978-3-540-87879-7_11.

Esfahani, N., Malek, S., 2013. Uncertainty in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., et al. (Eds.), Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24–29, 2010 Revised Selected and Invited Papers. Springer Berlin Heidelberg, pp. 214–238. doi:10.1007/978-3-642-35813-5_9.

Fenton, N.E., Neil, M., 2000. Software metrics: roadmap. In: Proceedings of the Conference on The Future of Software Engineering. ACM, New York, NY, USA, pp. 357–370. doi:10.1145/336512.336588.

Floch, J., Hallsteinsen, S., Stav, E., et al., 2006. Using architecture models for runtime adaptability. Softw. IEEE 23 (2), 62–70. doi:10.1109/MS.2006.61.

Frakes, W., Terry, C., 1996. Software reuse: metrics and models. ACM Comput. Surv. 28 (2), 415–435. doi:10.1145/234528.234531.

Frakes, W.B., Kang, K., 2005. Software reuse research: status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536. doi:10.1109/TSE.2005.85.

Garlan, D., 2010. Software engineering in an uncertain world. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. ACM, New York, NY, USA, pp. 125–128. doi:10.1145/1882362.1882389.

Garlan, D., Allen, R., Ockerbloom, J., 1995. Architectural mismatch: why reuse is so hard. IEEE Softw. 12 (6), 17–26. doi:10.1109/52.469757.

Garlan, D., Cheng, S., Huang, A., et al., 2004. Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer 37 (10), 46–54.

Gray, D.E., 2013. Doing Research in the Real World. SAGE Publications Ltd..

Griss, M.L., 1993. Software reuse: from library to factory. IBM Syst. J. 32 (4), 548–566. doi:10.1147/sj.324.0548.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. IEEE Comput. 41 (4), 93–95.

Hallsteinsen, S., Stav, E., Floch, J., 2004. Self-adaptation for everyday systems. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems. Association for Computing Machinery, New York, NY, USA, pp. 69–74. doi:10.1145/1075405.1075419.

Iglesia, D., Weyns, D., 2015. MAPE-K Formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. Auton. Adapt. Syst. 10 (3), 15:1–15:31. doi:10.1145/2724719.

I.S.610.12-1990, 1990. IEEE Standard Glossary of Software Engineering Terminology. Technical Report. The Institute of Electrical and Electronics Engineers 345 East 47th Street, New York, NY 10017, USA doi:10.1109/ieeestd.1990.101064.

Juristo, N., Moreno, A.M., 2010. Basics of Software Engineering Experimentation, first ed. Springer Publishing Company, Incorporated.

Kang, K., 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report. DTIC Document.

Kephart, J., Chess, D., 2003. The vision of autonomic computing. Computer 36 (1), 41–50.

Krueger, C.W., 1992. Software reuse. ACM Comput. Surv. (CSUR) 24 (2), 131–183.

Krupitzer, C., Roth, F.M., Vansyckel, S., et al., 2015. Towards reusability in autonomic computing. In: 2015 IEEE International Conference on Autonomic Computing, pp. 115–120. doi:10.1109/ICAC.2015.21.

Mahdavi-Hezavehi, S., Avgeriou, P., Weyns, D., 2017. A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In: Mistrik, I., Ali, N., Kazman, R., et al. (Eds.), Managing Trade-Offs in Adaptable Software Architectures. Morgan Kaufmann, Boston, pp. 45–77. doi:10.1016/B978-0-12-802855-1.00003-4.

McManus, H., Hastings, D., 2005. A framework for understanding uncertainty and its mitigation and exploitation in complex systems. INCOSE Int. Symp. 15 (1), 484–503. doi:10.1002/j.2334-5837.2005.tb00685.x.

Miedes, E., Munoz-Escoi, F.D., 2012. Dynamic Software Update. Technical Report. Instituto Universitario Mixto Tecnologico de Informatica, Universitat Politecnica de Valencia, Campus de Vera s/n, 46022 Valencia (Spain).

OMG, 2008. Software & Systems Process Engineering Metamodel Specification (SPEM). Technical Report. OMG. http://www.omg.org/spec/SPEM/2.0.

Peeters, P., van Asperen, J., Jacobs, M., et al., 2005. The Application of Function Point Analysis (FPA) in the Early Phases of the Application Life Cycle a Practical Manual: Theory and Case Study, 2.0 ed. Netherlands Software Metrics Association (NESMA).

Perez-Palacin, D., Mirandola, R., 2014. Uncertainties in the modeling of self-adaptive systems: a taxonomy and an example of availability evaluation. In: 5th ACM/SPEC International Conference on Performance Engineering. ACM, New York, NY, USA, pp. 3–14. doi:10.1145/2568088.2568095.

Pohl, K., Böckle, G., Van Der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer-Verlag New York, Inc..

Prieto-Diaz, R., 1993. Status report: software reusability. Softw. IEEE 10 (3), 61–66. doi:10.1109/52.210605.

Quan, N., 2013. Distributed Game Environment: A Software Product Line for Education and Research. Master's thesis.

Ramirez, A.J., Jensen, A.C., Cheng, B.H.C., 2012. A taxonomy of uncertainty for dynamically adaptive systems. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Press, Piscataway, NJ, USA, pp. 99–108.

Razali, N.M., Yap, B.W., 2011. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. J. Stat. Model. Anal. 2 (1), 21–33.
Rouvoy, R., Barone, P., Ding, Y., et al., 2009. Music: middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Cheng, B., de Lemos, R., Giese, H., et al. (Eds.), Software Engineering for Self-Adaptive Systems. In: Lecture Notes in Computer Science, vol. 5525. Springer Berlin / Heidelberg, pp. 164–182.
Runeson, P., Höst, M., Rainer, A., et al., 2012. Case Study Research in Software Engineering: Guidelines and Examples, first ed. Wiley Publishing.
Salehie, M., Tahvildari, L., 2009. Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst.(TAAS) 4 (2), 14.
Shapiro, S.S., Wilk, M.B., 1965. An analysis of variance test for normality (complete samples). Biometrika 52 (3/4), 591–611.
Tarr, P., Ossher, H., Harrison, W., et al., 1999. N degrees of separation: multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering. ACM, pp. 107–119.
Toffetti, G., Brunner, S., Blchlinger, M., et al., 2017. Self-managing cloud-native applications: design, implementation, and experience. Future Gener. Comput. Syst. 72, 165–179. doi:10.1016/j.future.2016.09.002.
Weyns, D., 2019. Software engineering of self-adaptive systems. In: Cha, S., Taylor, R.N., Kang, K. (Eds.), Handbook of Software Engineering. Springer International Publishing, Cham, pp. 399–443. doi:10.1007/978-3-030-00262-6_11.
Weyns, D., Iftikhar, U., Söderlund, J., 2013. Do external feedback loops improve the design of self-adaptive systems? A controlled experiment. In: Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE Press, pp. 3–12. http://dl.acm.org/citation.cfm?id=2487336.2487341.
Weyns, D., Ramachandran, G.S., Singh, R.K., 2018. Self-managing internet of things. In: Tjoa, A.M., Bellatreche, L., Biffl, S., et al. (Eds.), SOFSEM 2018: Theory and Practice of Computer Science. Springer International Publishing, Cham, pp. 67–84.
Weyns, D., Schmerl, B., Grassi, V., et al., 2013. On patterns for decentralized control in self-adaptive systems. In: Software Engineering for Self-Adaptive Systems II. Springer, pp. 76–107.
Wirfs-Brock, R., McKean, A., 2003. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley Professional.
Wohlin, C., Runeson, P., Höst, M., et al., 2012. Experimentation in Software Engineering, first ed. Springer-Verlag Berlin Heidelberg.
Ziv, H., Richardson, D., Klösch, R., 1996. The Uncertainty Principle in Software Engineering. University of California. Technical Report. Irvine UCI-TR-96-33.

Nadeem Abbas is a senior lecturer within the Department of Computer Science and Media Technology, Linnaeus University, Sweden. His main research interests include the development of self-adaptive software systems with systematic reuse, and development of mechanisms to manage variability and uncertainties involved in the development of self-adaptive software systems.

Jesper Andersson is a senior lecturer and department chair within the Department of Computer Science and Media Technology, Linnaeus University, Sweden. His research focuses mainly on Software Engineering techniques and methodologies supporting the development of smarter systems. His other research interests include software architecture, self-adaptive software systems, software reuse, and software ecosystems.

Danny Weyns is a professor in the Department of Computer Science, Katholieke Universiteit Leuven, Belgium; he is also part-time affiliated with Linnaeus University, Sweden. His main research interest includes software engineering of self-adaptive systems, with a particular focus on formalisms and runtime models to realize and assure the goals of adaptive systems relying on principles from software architecture and control theory.