Architectural Reasoning Support for Product-Lines of Self-adaptive Software Systems - A Case Study

Nadeem Abbas(✉) and Jesper Andersson

AdaptWise Department of Computer Science, Linnaeus University, Växjö, Sweden
{nadeem.abbas,jesper.andersson}@lnu.se

Abstract. Software architecture serves as a foundation for the design and development of software systems. Designing an architecture requires extensive analysis and reasoning. The study presented herein focuses on the architectural analysis and reasoning in support of engineering self-adaptive software systems with systematic reuse. Designing self-adaptive software systems with systematic reuse introduces variability along three dimensions; adding more complexity to the architectural analysis and reasoning process. To this end, the study presents an extended Architectural Reasoning Framework with dedicated reasoning support for self-adaptive systems and reuse. To evaluate the proposed framework, we conducted an initial feasibility case study, which concludes that the proposed framework assists the domain architects to increase reusability, reduce fault density, and eliminate differences in skills and experiences among architects, which were our research goals and are decisive factors for a system's overall quality.

1 Introduction

Software architecture provides the cornerstones for software system design and development. A high-quality architecture is a necessary condition if a software system should satisfy its requirements [6]. This condition becomes more vital when developing large and complex software systems.

While designing an architecture, software architects have to analyze and reason about design choices. The design choices are analyzed with respect to combinations of design parameters and their consequences. The design parameters affect the architecture decision process and include among others development time and cost, user goals, application requirements, and the operating environment. The architects select choices with outcomes that best matches the design parameters. The difficulty of architectural analysis and reasoning parallels the complexity growth in projects. To support architecture analysis and reasoning for complex systems, architects may use architectural reasoning frameworks [8,11].

We encountered several architectural analysis and reasoning challenges in our research on support for strategic reuse with Software Product Line Engineering (SPLE) [21] for Self-Adaptive Software Systems (SASS) [1]. Our goal

D. Weyns et al. (Eds.): ECSA 2015, LNCS 9278, pp. 20–36, 2015.
DOI: 10.1007/978-3-319-23727-5_2

is to develop assets that can be reused both vertically and horizontally [22] to support realization of self-management properties across products and product domains. The term "self-management" here refers to those characteristics which enable a software system to adapt itself in response to changes in its requirements, goals, environment and the system itself [10]. Self-configuration, self-healing, self-optimization, and self-protection are the four widely known self-management properties [18]. Realizing self-management properties is known to be a hard problem for a single self-adaptive system and becomes even more challenging when combined with reuse across products and products domains.

Self-adaptation combined with the product-line approach introduces variability along three dimensions. Domain variability, the first dimension, originates from the SPLE domain. It refers to differences among products in a product line. Run-time variability, the second dimension, comes from the self-adaptive software systems. It refers to run-time changes in a system's requirements, goals, environment, and the system itself [10]. The third dimension, cross-domains variability, stems from horizontal reuse, that is reuse across product domains. It refers to differences among products in two or more domains. The combination of three dimensions expands the design space architects have to consider, and consequently architectural reasoning and analysis become more complex.

In our work, we discovered that this increased complexity affected our primary goal, reusability, negatively. With increase in complexity, the importance of architects' skills and experience was elevated. We hypothesized that the lack of dedicated support for architecture reasoning with self-adaptation and strategic reuse was a primary reason. To that end, we adopted existing methods and techniques to develop an extended Architectural Reasoning Framework (eARF) [2].

The framework provides models and techniques that assist architects in analysis and reasoning in context of the variability described above. In this paper, we introduce the eARF elements and outline a workflow. The workflow provides step by step instructions to identify domain requirements along with their variability, extract design choices to realize requirements, analyze and reason about design choices, and finally map decisions to a reference architecture.

We have conducted an initial evaluation of the framework in a case study with final year master students [17]. The goal was to investigate its feasibility. The results indicate that use of the eARF framework provides better architectural analysis and reasoning support compared to the reference approach. It helps architects to design assets with increased reusability and reduced fault density. By providing architects with required knowledge encapsulated in the form of tactics and patterns, it also reduces the effect of differences in architects' skills and experience. The combined results indicate that our working hypothesis is correct. However, further evaluation is required to support the findings.

The remainder of this paper is organized as follows. Section 2 introduces the eARF and some of its artifacts and activities. Section 3 describes the case study. In Section 4, we analyze data and discuss results, which is followed by Section 4.4 that discusses threats to validity. Section 5 positions our work with respect to related work. We conclude and discuss future work in Section 6.

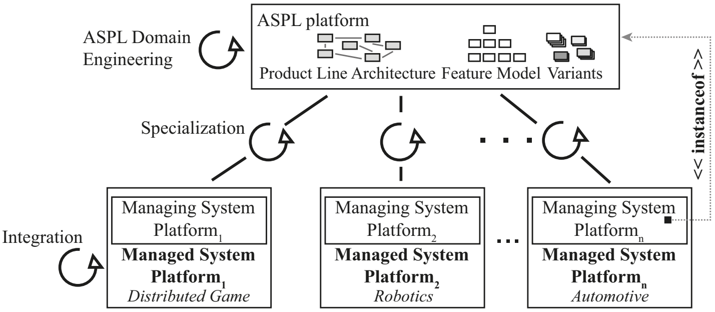

ASPL Domain Engineering
ASPL platform
Product Line Architecture
Feature Model
Variants
Specialization
<< instanceof >>
Integration
Managing System Platform$_1$
Managed System Platform$_1$
Distributed Game
Managing System Platform$_2$
Managed System Platform$_2$
Robotics
Managing System Platform$_n$
Managed System Platform$_n$
Automotive

Fig. 1. The ASPLe Processes

2 An Extended Architectural Reasoning Framework

Strategic reuse offers improved quality combined with reduced effort, which would contribute to self-adaptive software engineering practices. However, little or no work in this direction has been conducted in the self-adaptive software systems domain [24].

We have developed the Autonomic Software Product Line (ASPL) [1], an approach that supports both vertical and horizontal reuse [22] of assets across domains of self-adaptive software systems. The ASPL is a multi-product line approach involving three principle components and three development processes as shown in Figure 1. The first principle component is the ASPL platform, which is a horizontal platform for managing systems. It includes reusable assets that target cross-domain reuse, that is, it is independent from the managed system domain. The second principle component is the Managing System Platform, which is a vertical platform for a managing system domain. It is derived from the horizontal platform and specialized for a specific Managed System Platform, which is the third principle component.

The framework also defines three processes. The first process is a domain engineering process for the managing system domain. It is responsible for managing the horizontal ASPL platform and its reusable assets. Then in the middle, we have multiple instances of a specialization process. Each specialization process derives a vertical managing system platform for a specific application domain by specializing the horizontal ASPL platform. The third process integrates a specialized vertical managing system platform with a domain specific managed system platform. This approach is similar to a multi-product line strategy where the ASPL platform is reusable across products and product domains, which reduces complexity for both domain and application engineers.

Architects will reason about self-management properties and additional quality attributes in the three ASPL processes, thus we identified a need for adequate reasoning support, primarily due to the complex interactions of properties, attributes, variability, and uncertainty. A reasoning framework encapsulates quality attribute knowledge and techniques required to understand and analyze

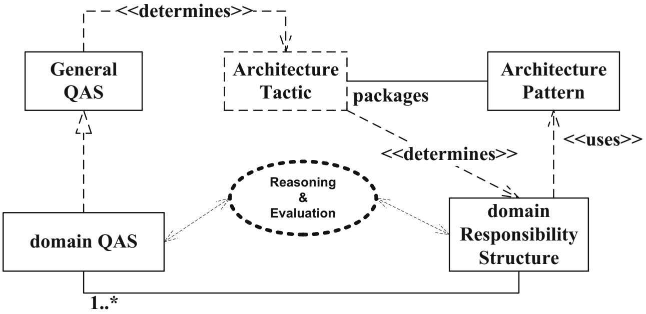


Fig. 2. Building Blocks of the eARF

a system's behavior for a specific quality attribute [7] and provides support for modeling, analysis, evaluation and interpretation [8]. We found that none of the existing reasoning frameworks provide sufficient reasoning support for realizing self-management properties. To that end, we have developed the extended Architectural Reasoning Framework (eARF).

The extended framework's structure is based on the architectural concepts defined by Diaz-Pace et al. [11]. Figure 2 outlines the extended framework and its elements: (1) Quality Attribute Scenarios (QAS), (2) domain QAS, (3) domain Responsibility Structure, (4) Architecture Tactics, and (5) Architecture Patterns. As compared to the reasoning framework proposed by Diaz-Pace et al. [11], the domain QAS and domain responsibility structure elements are the extended forms of a general QAS and responsibility structure, respectively, whereas the element "architecture patterns" is a new addition to the extended framework. We use an illustrative example to explain all these elements.

2.1 Illustrative Example

The PhotoShare Software Product Line (PSPL) contains service based products that allow users to upload, edit, and share photos. As shown in the feature model depicted in Figure 3, "uploading" and "sharing" are mandatory features, whereas "editing" is an optional feature. In addition to the mandatory features, the products are also required to guarantee performance from self-optimization. For example, a general self-optimization scenario is: *"From time to time, a PSPL product experiences increase in the number of picture upload requests that it can not handle adequately. The product can detect unacceptable latencies and adapt to self-optimize its performance"*. More details on PSPL may be found on the case study home page http://homepage.lnu.se/staff/janmsi/casestudydRS/.

Figure 4 depicts a workflow for how the eARF elements assist architects to realize self-management properties. The eARF artifacts used and produced are described below. We illustrate a scenario from the specialization process that prepares assets from the horizontal ASPL platform for integration with the vertical PSPL platform.

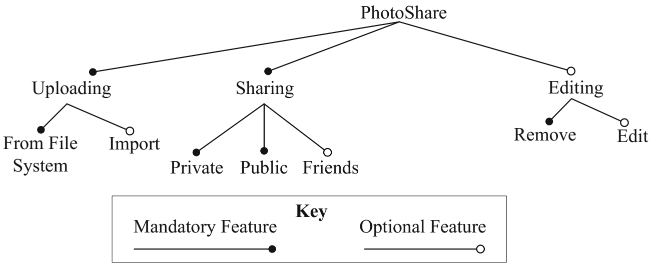


Fig. 3. PSPL – Feature Model

2.2 Domain Quality Attribute Scenarios

The identification and characterization of domain requirements and their variability is a prerequisite for architecture reasoning and design. The eARF uses quality attribute scenarios dQAS [3], an extension of QAS [6] with support to characterize domain variability, to specify a domain's requirements for self-management.

In ① in Figure 4, the domain requirements for self-optimization are elicited and specified. The requirements are analyzed for domain variability in activity ② and specified as domain scenarios. The ASPL platform provides a repository of reusable scenarios that may be adopted and reused. The PSPL architects reuse and adapt scenarios to reduce or expand their scope. For example, PSPL always schedule subscription users first; this domain specific constraint is specified by adapting scenarios from the ASPL platform. The domain analysts define new QAS and dQAS, if the platform contains no matching assets. We continue and design a domain Responsibility Structure when the dQASs are defined.

2.3 Domain Responsibility Structure

A domain Responsibility Structure (dRS) is an architectural model that consists of a responsibility part and a variability part. The first step, activity ③ in Figure 4, analyzes domain scenarios and identifies domain responsibilities [28] and variation points. The responsibilities and variation points are further analyzed for structure, associations and variants in activity ④. The responsibility part of a dRS is defined by mapping responsibilities to responsibility components. In this process, architects use tactics and patterns to support reasoning and decision making. Activity ④, completes the dRS by defining its variability part, i.e., variation points with variants, and connecting variation points to corresponding responsibility structures. The architects reuse and adapt features, variants, and variability points from ASPL into PSPL.

Self-management properties are similar to regular quality attributes. Their system-wide nature with tight coupling makes them difficult to modularize, which is also a known characteristic for quality attributes [6]. To assist the architects in reasoning about alternatives and decision making, the framework

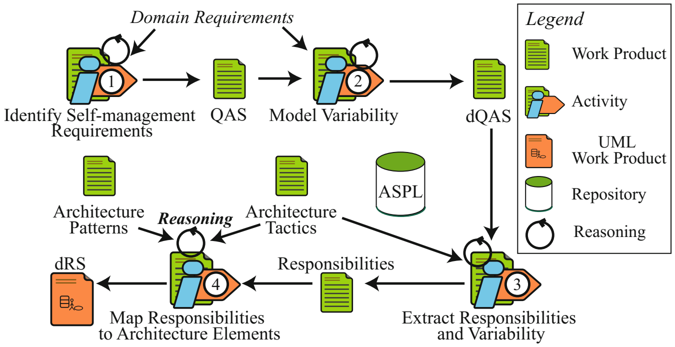


Fig. 4. Analysis and Design Workflow Using the eARF

provides design reasoning strategies, patterns and tactics for self-adaptive software systems. Examples of design strategies include "attribute driven design" [6] and "responsibility driven design" [28]. For the PSPL example we use the latter.

An architectural tactic encapsulates design decisions that may influence behavior of a system with respect to a particular quality attribute [6]. The framework promotes architectural tactics for the self-managing property. For the PSPL, we have several performance based self-optimization tactics, for example, resource demand, resource management, and resource arbitration [6].

The ASPL platform includes a set of tactics and patterns used to realize self-management properties, for instance, MAPE-K control loop tactic, and tactics for self-healing and for coordinating decentralized self-adaptation [26]. Tactics and patterns together assists architects to analyze and reason about a system's responsibilities and structure. Each tactic represents a design option, i.e., a variant, and the platform provides responsibility components and variants for the supported tactics. Tactics assist architects to identify variability and map it to a variability model for the dRS.

Figure 5 depicts a fragment of a dRS for self-optimization in the PSPL domain. Responsibilities and their variability were defined in activity ③. The resource management performance tactic, and the MAPE-K control loop pattern are used to identify and reason about sufficient allocation of responsibilities and possible variants for achieving self-optimization in the application domain.

The resulting dRS in Figure 5 contains a monitor element, the Response Monitor, from the MAPE-K pattern and a variation point with three variants: continuous, periodic, and event based. The PSPL products always include the event-based variant, while the other two variants are optional. The Planner element reuses two strategies from the resource management performance tactic: add threads, and add resources. Both strategies are mandatory for all products.

The Performance Manager subsystem in Figure 5 indicates that the target domain supports two performance manager variants; (1) centralized, and (2) decentralized. This is an example of how architects have used patterns in the process. The detailed design for one of the performance manager variants uses a

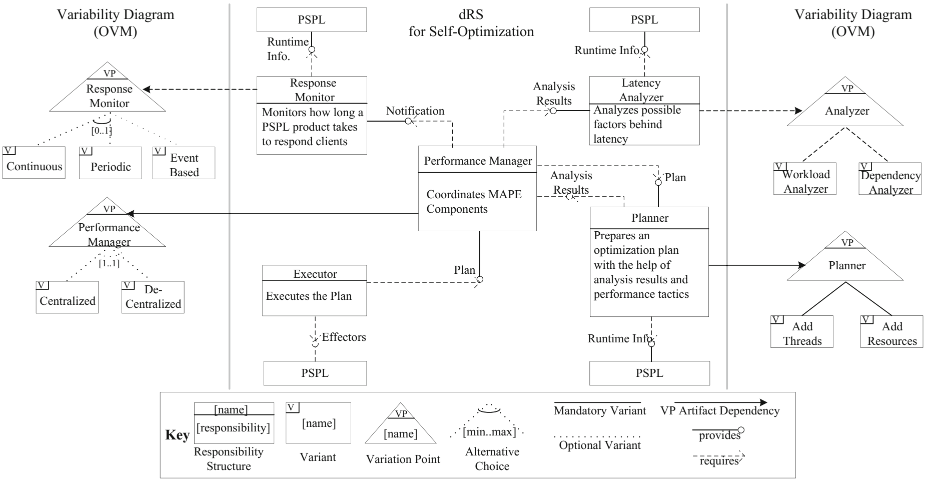


Fig. 5. Domain Responsibility Structure (dRS) for Self-Optimization

decentralization pattern for the managing systems [26], while the other variant uses a centralized feedback-loop.

The initial PSPL domain architecture for the self-optimization is ready. It will be further refined, detailed, and reconciled for additional self-management properties, and integrated with PSPL domain artifacts in the integration process. That is, however, not the focus for the work presented herein.

3 Evaluation

We conducted a case study to evaluate the eARF approach's feasibility. For a full account of the study, we refer to the study home page[1]. The primary goals for the study were: (1) to evaluate the eARF approach in comparison to a state-of-the-art reference approach, and (2) to collect user experiences for improving the eARF approach. For (1), we performed a case study with final year master students as a frame of reference [29]. The primary focus was on reusability [15], and fault density [13]. For (2), we collected qualitative data from interviews and questionnaires to understand the level of support for architectural reasoning.

We used the evaluation in (1) and user experiences from (2) to answer our hypothesis that the proposed reasoning framework provides better support for architectural reasoning and reusability in domains characterized by run-time variability, and reduces the effects of architects' skills and experience, and product's fault-density, in comparison to state-of-the-art practices.

[1] http://homepage.lnu.se/staff/janmsi/casestudydRS/

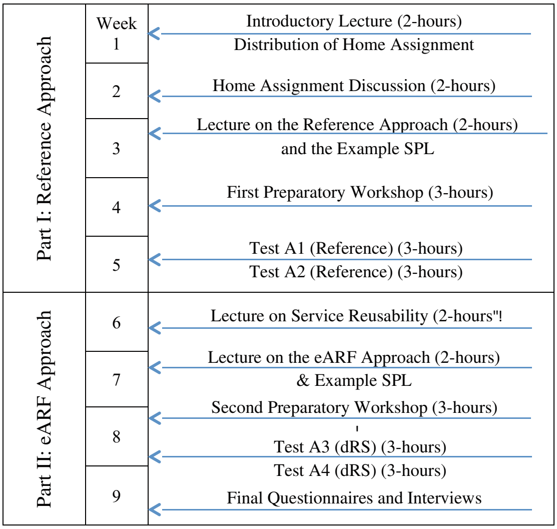

Part I: Reference Approach
Part II: eARF Approach
Week 1
Introductory Lecture (2-hours)
Distribution of Home Assignment
2
Home Assignment Discussion (2-hours)
3
Lecture on the Reference Approach (2-hours)
and the Example SPL
4
First Preparatory Workshop (3-hours)
5
Test A1 (Reference) (3-hours)
Test A2 (Reference) (3-hours)
6
Lecture on Service Reusability (2-hours"!
7
Lecture on the eARF Approach (2-hours)
& Example SPL
Second Preparatory Workshop (3-hours)
8
Test A3 (dRS) (3-hours)
Test A4 (dRS) (3-hours)
9
Final Questionnaires and Interviews

Fig. 6. Overview of the Nine Weeks Course in which the Case Study Took Place

3.1 Design and Planning

We follow a planning template suggested by Wohlin et al. [29]. The objective of the study is defined above. The eARF framework and a reference approach are the two cases studied. The reference approach consists of state-of-the-art practices for self-adaptive software system design, centered around MAPE-K feedback loop [18]. The Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) loop was first introduced by IBM [18] to add self-managing properties to software systems. It monitors and controls one or more underlying managed elements. The managed element might be a hardware or a software system. The reason for selecting the MAPE-K as a state-of-the-art reference approach is that, at present, it is the most widely used approach to realize self-adaptive software systems.

Test assignments, questionnaires and interviews are used as methods of data collection. We match the case study's objective and research questions with data collection methods to decide which data units we would use in our analysis, i.e., our selection strategy. Given our setting with a small number of subjects we adopt an "analyze-all" strategy, that is, data collected from all subjects is investigated and analyzed.

The case study was performed as a part of a nine weeks course, involving three researchers and 13 subjects. Most of the preparatory work, data collection, and analysis were performed by a doctoral student assisted by two senior lecturers, primarily in the role as reviewers and advisors. The subjects were final-year students on a two year master program in software engineering. As depicted in Figure 6, the case study was conducted in two parts, one for each case studied. The first part was concerned with the reference approach, and the second

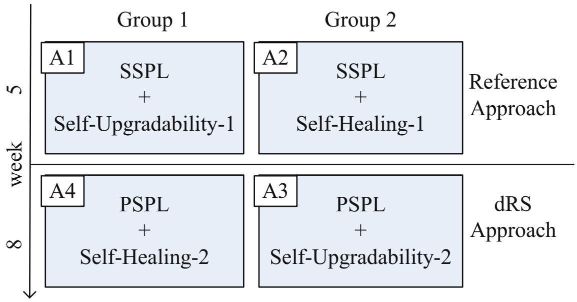


Fig. 7. Test Assignments - Design

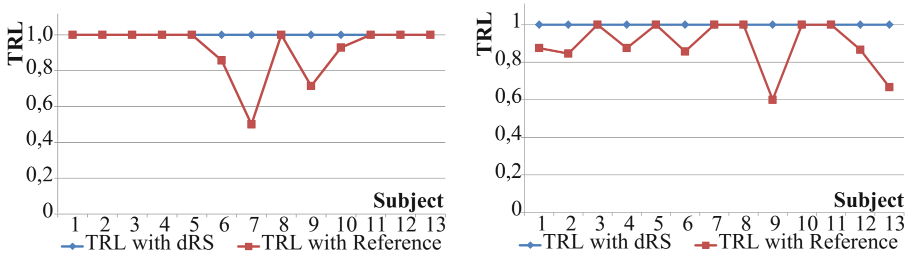
introduced the treatment, i.e., the eARF approach. The case study was completed with questionnaires and interviews conducted in final week of the part two.

3.2 Data Collection

The case study involves four test assignments that are performed by dividing subjects into two groups randomly based on blocked subject-object study classification [29]. A blocked subject-object study analyzes two or more objects (cases or units of analysis) using two or more subjects per object. In this type of study, each subject receives both treatments, i.e., the reference and the dRS approach. This allows paired comparison of the two approaches.

The test assignments target first two of the three variability dimensions addressed by the ASPL approach. Domain variability, the first dimension, comes from the product line engineering domain. Thus, two example SPLs, (1) Soft-Phones Software Product Line (SSPL) and (2) PhotoShare Software Product Line (PSPL) were designed for the test assignments. Details about these example SPLs are given at the case study home page. To cater for the second dimension, run-time variability, requirements for self-upgradability and self-healing were added to the product lines' scope. For each test, Figure 7 depicts a combination of an example SPL and a self-management property, for example, test A2 uses SSPL with self-healing as problem domain.

Each test assignment first introduces a problem domain followed by three tasks. Tasks 1 and 3 are design tasks. Task 1 has two parts: *a* and *b*. Each part requires subjects to extend an initial product design to support a given self-management property, either self-upgradability, or self-healing. Task 2 requires subjects to extend the core assets base by adding reusable artifacts from the Task 1. This represents the third variability dimension, which originates from reuse across multiple domains, i.e., horizontal reuse. Task 3 requires subjects to use the extended core assets base from Task 2 and design a new product. All tests were conducted as regular class assignments. No feedback was given to subjects on the first part prior to the completion of the second part.



(a) Total Reuse Level (TRL) in Task 1(a) (b) Total Reuse Level (TRL) in Task 3

Fig. 8. Total Reuse Level Achieved Using the Reference and the dRS Approach

Questionnaires and semi-structured interviews were used to collect data for the study's second objective. There were two types of questionnaires. The first type was a combination of "pre" and "post" test questionnaires designed to identify false positives and false negatives. The second type compares the two approaches with respect to their support for architectural reasoning. Each subject was interviewed at the very end of the data collection phase to clarify responses in questionnaires and collect details. The interviews were conducted after publishing the course result to assure that the subjects did not adjust their responses to get a better grade.

4 Analysis of Results

This section presents the data analysis and pinpoints findings that were observed to confirm or reject our hypothesis. The main objective was to analyze the eARF approach in comparison to the state-of-the-art reference approach with respect to three properties: (1) reusability, (2) fault density, and (3) support for architectural reasoning. We divide the analysis in three parts, one for each property.

4.1 Support for Reusability

To analyze the two approaches with respect to their support for reusability, we use quantitative data from the test assignments. The "reuse level" software metric [15] is used for this analysis. It is defined for hierarchically composed component system, and is well aligned with the way products are designed and composed in the test assignments. The metric is defined as:

$$\begin{aligned}
\text{Total Reuse Level} &= \text{External Reuse Level} + \text{Internal Reuse Level} \\
\text{External Reuse Level} &= E/L \\
\text{Internal Reuse Level} &= M/L
\end{aligned}$$

L – the total number of lower level items in the higher level item.

E – the number of lower level items from an external repository.

I – the number of lower level items not from an external repository.

M – the number of items not from an external repository but used more than once.

All reuse levels will be between 0 and 1, here 0 indicates no reuse. We assume that the products designed in the tests are the higher level items and compute the value of L by counting the number of lower level items used in a product. The core assets base from the example SPLs provides lower level items. We compute E by counting the number of items from the core assets, and I by counting items developed specifically for a product. M is computed by counting the items not belonging to the core assets base but used more than once.

We calculate total reuse level for task 1 and task 3 as depicted in Figure 8. There is no significant difference in the total reuse level for task 1(a). This is because this task presents subjects with an initial product design and asks them to extend it to support a self-management property. The approach is, however, at least as good as the reference approach for all subjects and sometimes the total reuse level is better. The results are similar for task 1(b), thus excluded for space consideration.

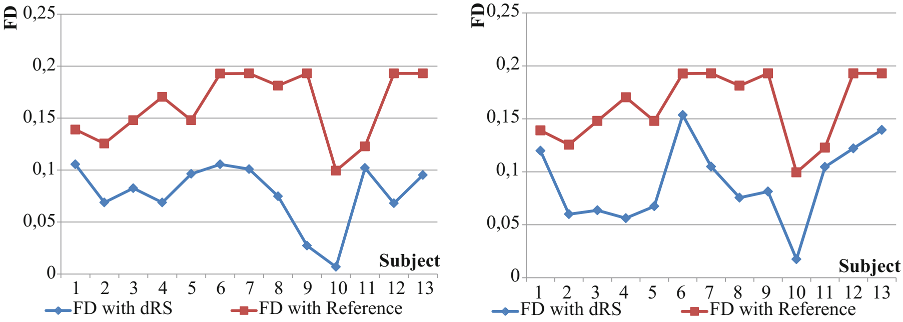
The difference between the two approaches in terms of the achieved total reuse level becomes more clear in task 3. In this task, a new product with support for a self-management property is designed from scratch. All subjects achieve maximum total reuse level of 1 with the eARF approach. In comparison only 46% were able to achieve the maximum total reuse level with the reference approach.

4.2 Fault Density

Fault density is the number of known faults divided by product size [25]. It is a de-facto measure of user perceived software quality [13]. We use it to analyze the eARF approach's support for producing high-quality architectural designs. We used quantitative data from task 1 and task 3 to calculate the fault density.

$$\text{Fault Density} = \text{Faults / Size} \tag{1}$$

To compute fault density using equation 1, we need to compute values of two input variables: *faults*, and *size*. As the tasks selected for this analysis resulted in design level artifacts, we were restricted to use methods at design level. We estimated faults for each task of each subject by comparing subjects' solutions (designs) with a reference solution, and counting one fault for each missing or



(a) Faults Density (FD) for Task 1(a)

(b) Faults Density(FD) for Task 3

Fig. 9. Faults Density Comparisons Using the Reference and the dRS Approach

incorrect element. The missing or incorrect element can be a component, service, interface, method, or a method parameter for test assignments A1 and A2 that use the reference approach. In the test assignments A3 and A4 that require the eARF approach, the element can be a responsibility component, responsibility definition, interface, variation point, or a variant.

The size of a software system can be measured in different ways. We used function point analysis (FPA) [4] method to measure size. As the test assignments involve design artifacts so we searched for an FPA based method that could be applied to design level artifacts. We could not find a method that suited the design artifacts created in the selected tasks. However, we found an FPA based high level analysis approach [20] that targets software requirements specification. We decided to use this approach. The approach uses independent assessors to rate each requirement on a scale: low, average, and high. We used three assessors, two doctoral students and a senior lecturer in software engineering. Following the procedure described by the approach, we computed three scores for *size* in terms of function points: minimum, expected, and maximum. We took average of these scores to get a size estimate. For details about the computation procedure, we refer to the original work by Peeters et al. [20] and the case study home page.

Figure 9 depicts the fault density for the approaches. It is clear that for both tasks the eARF approach results in a reduced fault density.

4.3 Support for Architectural Reasoning

We use qualitative data collected with a questionnaire to analyse the approaches with respect to their support for architecture analysis and reasoning. As the data is qualitative, the results are indicative.

In the final questionnaire, subjects were asked to answer six closed questions targeting support for architectural reasoning. A majority of the subjects rated the eARF approach as relatively better for the first two questions. In the remaining questions, we gave a statement and asked to select one of five options from

"Strongly Disagree" to "Strongly Agree", and a "Don't Know" option. More than 90% of the subjects "strongly agreed" or "agreed" to a statement that declared the eARF to be more assistive than the reference approach. The results were similar for the other statements, i.e., positive to the eARF approach. The analysis indicates that the eARF contributes positively to the design process and the subjects feel more confident with the support provided by the eARF approach.

4.4 Threats to Validity

We use the classification scheme suggested by Runeson et al. [23] to discuss threats to validity.

Construct validity ensures that a study actually relates to the problem that the study aims to address. The eARF focuses on realization of self-management properties characterized with domain and cross-domains variability. Accordingly, the study selected two SPLs with requirements for self-management properties as problem domains. A possible threat to validity is "use of unclear terminology" that causes subjects and researchers to interpret the used terms or concepts differently. We reduced this threat by dedicating three weeks in both parts of the study to lectures and preparatory workshops. Another possible threat is that subjects may guess what the researchers are looking for and adapt their answers accordingly. This threat was mitigated by presenting the activities as coursework. We also gave feedback and grades prior to the questionnaires and interviews.

Internal validity is a concern for explanatory studies where causal relations are examined [23]. This study is explorative, and thus less sensitive to this type of threat. Other potential threats are increased understanding and maturity of subjects. To mitigate these threats, we took three measures in the case study design: (1) a comprehensive knowledge base in the first part, (2) use of a new problem domain (PSPL) and requirements set in the second part, and (3) use of standards tools and methods such as UML.

External validity is concerned with the extent to which findings can be generalized and are relevant outside the study. This study uses final year master students as subjects [17]. The profile for such students is that they are knowledgeable but lack in professional skills and experience. One of the aspects we were interested in is, "how the experience and knowledge provided by eARF contributes to the design quality". It is important to note that professional architects will exhibit similar differences in knowledge and experience as the group of students usually have, and thereby, the effect of eARF would be generalizable. The size of the study's population is, however, small to generalize the results.

Another potential threat is that the framework and the test assignments are designed by the same group of researchers. There is a risk that the test assignments were designed in a way that favors the eARF approach. To reduce this threat, an independent senior researcher was requested to review the assignments. Moreover, a large portion of data collection and analysis was performed by a single researcher. The data analysis of the test assignments is based on objective data, except for the size measure. External reviewers were involved for that particular data point. With opinions from multiple sources the risk of a biased analysis is mitigated.

Reliability refers to the ability of other researchers to replicate the study. To support replication, a complete documentation for all activities in this study is available online at the case study home page.

5 Related Work

Reusable decision models support architectural reasoning by capturing architectural decisions and exchanging these within and between projects in the same or similar context. Olaf et al. [30] presented a proactive approach to model and reuse architectural knowledge. The approach need to be investigated in the context of self-management properties. Bass et al. [7] proposed use of reasoning framework which encapsulate knowledge and tools needed to analyze behavior of a system such as the modifiability reasoning framework [11]. A reasoning framework may help the architects to evaluate an architecture in its early stages, and save lot of effort and resources in the end. However, none of the existing frameworks target self-management properties and the three variability dimensions targeted in this study. An open source project, DiVA [12], provides a tool-supported methodology and framework for managing dynamic variability in adaptive systems. The project claims to contribute with a reasoning framework that takes a context and adaptation rules as input and does the reasoning to find and rank possible configurations for the given context. However, there are no details for the framework elements, which makes it difficult to compare with our work.

A related research theme that addresses development issues for reusable and dynamically reconfigurable core assets is Dynamic Software Product Lines (DSPL) [16]. The DSPL community has proposed several approaches to deal with run-time variability. The MADAM [14] middleware is one such approach that uses architecture models at run-time to reason about and control adaptations. In its current implementation, it uses a utility function to reason about and select the design options, this can be supplemented by encapsulating knowledge in the form of a reasoning framework. Thus the eARF framework has a potential to be integrated with the MADAM middleware to enhance reasoning support needed to deal with the three variability dimensions investigated in this study.

Liu et al. [19] proposed a dynamically reconfigurable reference architecture based approach to develop systems with evolvable run-time variability. The approach uses a dynamic update mechanism which at run-time updates the reference architecture by adding, removing, and modifying architectural elements. The authors described a process through which such updates are performed, however, there is no discussion about what triggers the dynamic update process, and how the design choices are analyzed and reasoned about at run-time.

Whittle et al. [27] presented RELAX, a requirements specification language for self-adaptive systems. It may help developers to identify variability in the requirements, by specifying requirements that a system could temporarily relax under certain conditions. However, RELAX does not support design and reasoning at the architecture level which is the focus for our work presented herein.

Cetina et al. [9] proposed a Common Variability Language (CVL) for runtime variability modeling. The CVL approach separates variability modeling from the base domain modeling. The split between variability modeling and base domain modeling in the CVL approach is similar to the split between variability part and responsibilities part in the extended responsibility structure presented in this study. However, the authors did not explicitly state and address the architectural reasoning support needed to make decisions and trade off at runtime.

Bachmann et al. [5] called for the provision of special methods that may assist in designing an architecture with quality attribute requirements. The authors require such methods to have three features, (1) knowledge encapsulation, (2) trade-offs, and (3) traceability from requirements to architecture. The eARF provides support for all these features through tactics, patterns, domain scenarios and the responsibility structures.

6 Discussion and Conclusions

Development of self-adaptive software systems with systematic reuse presents architects with the challenge of extensive architectural analysis and reasoning needed to analyze, reason about, and trade-off multiple design choices. The extended architectural reasoning framework supports the architects by providing them with models and techniques for reasoning, mapping, and structuring responsibilities with variability into a reference architecture. In addition, it provides architects with design knowledge and proven best practices encapsulated in the form of tactics and patterns.

We conducted a case study to investigate the feasibility of the proposed architectural reasoning framework. We conclude that the results from the study indicate that the framework offers better support for reuse and reduces fault density in comparison to the reference approach. We also collected qualitative data that indicates that the architects appreciate the structure and guidance provided by the eARF framework. This is also supported by the quantitative data where we see that skills and experience have less impact on the measured properties with the eARF approach.

The proposed extensions are the first steps towards a comprehensive design framework that leverages on reuse to engineer self-adaptive software product-lines across multiple domains. However, much work remains. For instance, the framework must include better support for reasoning. We have plans to further investigate tactics and patterns with the aim to establish a core of best practices for engineering self-adaptive software systems and include the practices as design advices for further increase of reusability and reuse levels.

Acknowledgments. Acknowledgments The research was funded by VINNOVA, the Swedish Agency for Innovation Systems and Innovative Product Development (Grant No. 2013-03492).

References

1. Abbas, N.: Towards autonomic software product lines. In: Proceedings of the 15th International Software Product Line Conference, SPLC 2011, vol. 2, pp. 44:1–44:8. ACM, New York (2011)
2. Abbas, N., Andersson, J.: Architectural reasoning for dynamic software product lines. In: Proceedings of the 17th International Software Product Line Conference Co-located Workshops, pp. 117–124
3. Abbas, N., Andersson, J., Weyns, D.: Modeling variability in product lines using domain quality attribute scenarios. In: Proceedings of the WICSA/ECSA 2012 Companion Volume, pp. 135–142. ACM, New York (2012)
4. Albrecht, A., Gaffney, J.E.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Transactions on Software Engineering **SE–9**(6), 639–648 (1983)
5. Bachmann, F., Bass, L., Klein, M., et al.: Designing software architectures to achieve quality attribute requirements. IEE Proceedings - Software **152**(4), 153–165 (2005)
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley Professional (2003)
7. Bass, L., Ivers, J., Klein, M., et al.: Encapsulating quality attribute knowledge. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005, pp. 193–194. IEEE Computer Society, Washington, DC (2005)
8. Bass, L., Ivers, J., Klein, M.H., et al.: Reasoning frameworks. Tech. rep. (2005). http://www.sei.cmu.edu/library/abstracts/reports/05tr007.cfm
9. Cetina, C., Haugen, O., Zhang, X., Fleurey, F., Pelechano, V.: Strategies for variability transformation at run-time. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 61–70. Carnegie Mellon University, Pittsburgh (2009)
10. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013)
11. Diaz-Pace, A., Kim, H.-W., Bass, L.J., Bianco, P., Bachmann, F.: Integrating quality-attribute reasoning frameworks in the ArchE design assistant. In: Becker, S., Plasil, F., Reussner, R. (eds.) QoSA 2008. LNCS, vol. 5281, pp. 171–188. Springer, Heidelberg (2008)
12. DiVA: Diva-dynamic variability in complex, adaptive systems. http://sites.google.com/site/divawebsite
13. Fenton, N.E., Neil, M.: Software metrics: roadmap. In: Proceedings of the Conference on The Future of Software Engineering, pp. 357–370. ACM, New York (2000)
14. Floch, J., Hallsteinsen, S., Stav, E., et al.: Using architecture models for runtime adaptability. IEEE Software **23**(2), 62–70 (2006)
15. Frakes, W., Terry, C.: Software reuse: Metrics and models. ACM Computing Surveys **28**(2), 415–435 (1996)

16. Hallsteinsen, S., Hinchey, M., Park, S., et al.: Dynamic software product lines. IEEE Computer **41**(4), 93–95 (2008)
17. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects-a comparative study of students and professionals in lead-time impact assessment. Empirical Software Engineering **5**(3), 201–214 (2000). http://dx.doi.org/10.1023/A:1026586415054
18. Kephart, J., Chess, D.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
19. Liu, J., Mao, X.: Towards realisation of evolvable runtime variability in internet-based service systems via dynamical software update. In: Proceedings of the 6th Asia-Pacific Symposium on Internetware, Internetware 2014, pp. 97–106. ACM, New York (2014)
20. Peeters, P., van Asperen, J., Jacobs, M., et al.: 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 edn. Netherlands Software Metrics Association (NESMA) (2005)
21. Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer-Verlag New York Inc. (2005)
22. Prieto-Diaz, R.: Status report: software reusability. IEEE Software **10**(3), 61–66 (1993)
23. Runeson, P., Höst, M., Rainer, A., et al.: Case Study Research in Software Engineering: Guidelines and Examples, 1st edn. Wiley Publishing (2012)
24. Weyns, D., Iftikhar, M., Malek, S., et al.: Claims and supporting evidence for self-adaptive systems: a literature study. In: 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systemsm, pp. 89–98 (2012)
25. Weyns, D., Iftikhar, M.U., Söderlund, J.: 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, pp. 3–12. IEEE Press, Piscataway (2013)
26. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013)
27. Whittle, J., Sawyer, P., Bencomo, N., et al.: RELAX: a language to address uncertainty in self-adaptive systems requirement. Requirements Engineering **15**(2), 177–196 (2010)
28. Wirfs-Brock, R., McKean, A.: Object design: roles, responsibilities, and collaborations. Addison-Wesley Professional (2003)
29. Wohlin, C., Runeson, P., Höst, M., et al.: Experimentation in Software Engineering, 1st edn. Springer, Heidelberg (2012)
30. Zimmermann, O., Gschwind, T., Küster, J.M., Leymann, F., Schuster, N.: Reusable architectural decision models for enterprise application development. In: Overhage, S., Ren, X.-M., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 15–32. Springer, Heidelberg (2008)