Capturing Ambiguity in Artifacts to Support Requirements Engineering for Self-Adaptive Systems

Juan C. Muñoz-Fernández[1,2], Alessia Knauss[3], Lorena Castañeda[4], Mahdi Derakhshanmanesh[5], Robert Heinrich[6], Matthias Becker[7], and Nina Taherimakhsousi[4]

[1] Facultad de Ingeniería, Universidad Icesi, Colombia, jcmunoz@icesi.edu.co,
[2] CRI, Université Paris 1 Panthéon - Sorbonne, France.
[3] Chalmers University of Technology, Sweden.alessia.knauss@chalmers.se
[4] University of Victoria, Canada. {lcastane, ninata}@uvic.ca
[5] MHP – A Porsche Company, Germany. mahdi.derakhshanmanesh@mhp.com
[6] Karlsruhe Institute of Technology, Germany, heinrich@kit.edu
[7] Software Engineering Group, Fraunhofer IEM, Germany.
matthias.becker@iem.fraunhofer.de

Abstract. Self-adaptive systems (SAS) automatically adjust their behavior at runtime in order to manage changes in their user requirements and operating context. To achieve this goal, a SAS needs to carry knowledge in artifacts (e.g., contextual goal models) at runtime. However, identifying, representing, and refining requirements and their context to create and maintain such artifacts at runtime is a challenging task, especially if the runtime environment is not very well known. In this short paper, we present an early concept to requirements engineering for the implementation of SAS in the context of uncertainty. Especially the wide variety of knowledge materialized in artifacts created during software engineering activities at design time is considered. We propose to start with a list of ambiguous requirements - or under-specified requirements -, leaving the ambiguity in the requirements, which will in the later steps be resolved further as more information is known. In contrast to conventional requirements engineering approaches, not all ambiguous requirements will be resolved. Instead, ambiguities serve as key input for self-adaptation. We present five steps for the resolution of the ambiguity. For each step, we describe its purpose, identified challenges, and resolution ideas.

Keywords: runtime requirements; self-adaptive systems; artifacts

1 Introduction

Requirements engineering (RE) activities for self-adaptive systems (SAS) trespass the limits of requirements analysis phases taking place also at design time and runtime [11]. When developing a SAS, designers need to define structural

and behavioral adaptation points explicitly (where, when, what, how). In traditional RE approaches, a particular set of artifacts (e.g., requirements lists, goal models, feature models, and statecharts) is created and managed by requirements engineers. To achieve adaptation, a SAS must be aware of this knowledge gathered at design-time to be able to adapt itself and carry the knowledge, e.g., in the form of models at runtime [2]. Furthermore, based on this knowledge, a SAS can decide about the best suitable adaptation at runtime. In related work, adaptive requirements have been introduced to explicitly define variation points in requirements to leave room for adaptation [10].

We propose to focus on the reuse of the knowledge contained by the set of artifacts usually created and managed at design-time and to enable their facilitation at runtime systematically. We define an *artifact* as a machine-readable document relevant to requirements and context. A draft of the artifacts are created by requirements engineers at design time, and they will be updated at runtime.

We start from a set of elicited requirements tolerating some degree of ambiguity. We propose to keep on purpose the potentially contained ambiguity captured in the requirements artifacts and to design and implement the software without (entirely) resolving the ambiguity. We define the ambiguity as of under-specification of requirements. Usually, resolving ambiguities is an essential part of the traditional RE activities. In a SAS, however, we claim that embracing ambiguity throughout software design and implementation is a major step towards defining and extending relevant context attributes for self-adaptation. Nevertheless, only the ambiguity that leads to feasible variability points should be maintained. For example, for autonomous vehicles, the requirements will be refined after the vehicle is developed (e.g., using continuous experimentation). A SAS shall resolve contextually relevant ambiguity itself or with human help at runtime. Runtime testing will then serve to guarantee that the detailed requirements are implemented properly. This resolution requires an implementation of the system without fully hard coding all choices, thereby leaving space for runtime variability.

Two aspects enable a SAS to adapt to emerging situations at runtime within safe and secure bounds. On the one hand, the combination of knowledge about ambiguity in requirements and its context (problem space). On the other hand, foreseen adaptation points in the software itself (solution space).

2 Guiding Artifact Centric RE Activities for SASs

To guide software engineers during requirements engineering and design of SAS, we propose an iterative and incremental approach comprising five steps. Table 1 summarizes the steps and its associated artifacts. The steps are described in the rest of this section.

Step 1: Capture Requirements

This first step starts with a traditional requirements elicitation activity at design time resulting in a set of requirements. Some of these requirements can be

Table 1. Guiding Activities and Excerpt of Related Artifact Types and Examples

Step	Activity	Artifact
1	Capture requirements	Specification (e.g., early goal models)
2 (Iter.)	Identify variability points	Model (e.g., late goal models)
3 (Iter.)	Identify context	Context model (e.g., ontologies)
4 (Iter.)	Identify situations	Situations model (e.g., reasoning support)
5 (Iter.)	Design variability (solution space)	Architecture models (e.g., component models)

ambiguous. The assumption here is that such ambiguities will be either resolved by the SAS or with human participation at runtime. Therefore, it is crucial that the requirements elicited in this step be documented or mapped to a machine-readable notation (e.g., goal models). In later steps when more information on the context is available, goal models can be extended to contextual goal models [12] and use cases can be extended to adapt cases [7].

Challenges: The consideration of leaving ambiguity in requirements: Some types of ambiguity might need to be resolved, as they will not lead to any adaptation needs (e.g., logical inconsistencies, stakeholders' interpretations) while other types of ambiguity are necessary for the SAS variability at runtime.

Ideas: Define different categories of ambiguity and their effect on adaptation capabilities of a SAS through empirical investigations.

Step 2: Identify Variability Points

The identification of ambiguities is an iterative process comprising three cases: (1) If a requirement is ambiguous in our accepted sense, a new variation of the requirement is produced for each interpretation and is evaluated again. (2) If a requirement is not ambiguous, then its variabilities are identified, and all considerable behaviors are defined as an alternative to the requirement. (3) If there is no variability, then the requirement is considered as solid with only one expected behavior. That behavior determines a system's state, and it is marked as valid or not. It is important to store the older versions of a requirement as well as invalid behaviors as system's knowledge. The initially defined requirements can be associated with a special type of requirements called *architecturally significant requirements* (ASRs). ASRs should provide relevant information for designers based on quality attributes, architecting family of related products and specific technologies required, as proposed by Bass et al. [1]. The identification of ASRs is important to satisfy all requirements adequately.

Existing work on variability in software engineering focuses on software product lines (SPL) [14], in which variability is used to produce new versions of a software product whereas goal-oriented modeling use variability to define behaviors of the system [4]. A Dynamic software product line (DSPL) target a single system with multiple and dynamic bindings [3]. REFAS [8] includes support the sub-specification with concern levels and aggregation relations. The specification is centered on constraints that are applicable at design time and runtime.

Challenges: Techniques and methods are required to provide (1) runtime variability assurance (i.e., reaction to a variability at runtime), (2) unexpected

variability and ambiguities management (i.e., not being prepared for certain variability), and (3) alternatives to model the sub-specification of variability and adaptation requirements, including the horizontal and hierarchical relations to be evaluated only at runtime.

Ideas: (1) The development of an evaluation framework with techniques and metrics to assess the variability of the requirements. This framework should consider the restrictions of a runtime environment such as response time, performance, and availability. (2) The development of runtime techniques to validate the variabilities of the requirements according to the states of the running system. (3) The development of runtime management infrastructures to sense, analyze and act upon unforeseen variabilities (i.e., discard or accept) and ambiguities (i.e., generate its variabilities) on the requirements such as the DYNAMICO reference model. [15]. (4) The runtime consideration for the definition of relevant relations that constraint the adaptation (e.g., constraints to adapt between two scenarios).

Step 3: Identify Context which Influences System Behavior

After having identified variation points, the requirements engineer has to understand the influence factors for variation points. We propose to represent these factors through *context-attributes* [6], which means that ambiguous parts of requirements can be refined further through context attributes (i.e., relevant information for the system that can be sensed and monitored). The context data can be obtained from the system and information about the external environment.

Challenges: (1) the identification of context attributes that are relevant to the variation points – this is partially done by the system and requires techniques to support this process – and (2) how to reduce invalid or not desired adaptation possibilities from an initial situation.

Ideas: Modeling constraints between variables for particular values and evaluate them at runtime. With the results of this evaluation, the system can reduce the adaptation possibilities of the model.

Step 4: Identify Context Situations and Variabilities

During this step, context is analyzed with the purpose of identifying the situations that are relevant to runtime events. For this purpose, the context already identified is traced to each requirement while specifying the meaning for each of them. Changes in the context have different reactions and implications for requirements depending on the interpretation of the situations and requirements.

Existing work on *context situations* in software engineering include techniques to manage context information, such as context models [16, 13] and using machine learning techniques to keep context information up-to-date [5].

Challenges: (1) Information on the scenario that might not be sensed by the system is not available. (2) Situations (or scenarios) can be subjective. (3) The generation of all possible situations poses a risk to the system's performance. (4) The influence of values from context variables is oversimplified. They are directly linked to some of the requirements.

Ideas: (1) The implementation of runtime models to represent situations, as well as required infrastructures to support the evolution of such representations.

(2) Strategies to guarantee the system's performance while generating diverse situations, including mechanisms and metrics to stop the system before an explosion of scenarios and computing possible scenarios offline or when the system is idle. (3) The definition of influence from context variables mediated by scenarios or other runtime context conditions to support more complex situations. Thus, the semi-automatic or automatic derivation of possible situations only from those influence factors. (4) The support of complex expressions to relate the context with the requirements closer to the real system configuration.

Step 5: Design Variation Points in the Solution Space

The intention of this step is to provide a software solution that (i) can satisfy the goals according to a given variability definition and (ii) is extendable when new goals emerge. The realization must especially support adaptation at runtime.

Existing work includes external and internal approaches [9]. *External approaches* use explicit feature models linked to artifact elements using (1) hard links/references or (2) an expression language where elements in artifacts are annotated using Boolean expressions over features. *Internal approaches* present the expressiveness of an artifacts language used to express variability (e.g., dynamic binding in Java, preprocessor in C and conditional "tags" in XML). Other techniques for dealing with variability include tailoring of artifacts (e.g., scripts in ANT or MAKE) and model transformation languages (e.g., ATL or QVT).

Challenges: Knowledge about variability is not fully (re)used at runtime. DSPL approaches support this reuse, but the relation between design-time and runtime variability is not well established.

Ideas: To have different views on the variability data: some for design-time (suited for requirements engineers) and some for runtime (suited for software: adaptation manager components). These views imply a language that requires the integration of knowledge (meta-data) from both – design time and runtime, i.e., make explicit why a variation at a certain variation point is an alternative. Such knowledge can also facilitate automated decision-making at runtime.

3 Concluding Remarks

SAS are expected to deal with uncertainty from their requirements and operating context, resulting in situations that require adaptations of the system. Some of these situations already occur at requirements level.

In this short paper, we sketched an artifact-centric approach for RE in the domain of SAS comprising five guiding, incremental, and iterative steps. For each step, we described its purpose, estimated associated challenges and potential ideas on how to resolve these challenges. The overall goal of the presented concept is to bridge design-time and run-time RE activities to manage ambiguities better.

Future work has to investigate the presented challenges and corresponding evaluation of results. As for the methodology, our proposal is to use empirical research methods, but combining quantitative and qualitative experiments with selected project cases from industry.

4 Acknowledgments

We thank T. Vogel, M. Tichy, and A. Gorla, organizers GI-Dagstuhl Seminar 14433 in which this paper was conceived. This work was supported by Vinnova grant 2014-06229.

References

1. L. Bass, J. Bergey, P. Clements, P. Merson, I. Ozkaya, and R. Sangwan. A Comparison of Requirements Specification Methods from a Software Architecture Perspective. Technical report.
2. N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, and E. Letier. Requirements Reflection: Requirements as Runtime Entities. In *ICSE'10*, pages 199–202, 2010.
3. R. Capilla, J. Bosch, P. Trinidad, A. Ruiz Cortés, and M. Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, 91:3–23, 2014.
4. B. Gonzales-Baixauli, J.C.S. Prado Leite, and J. Mylopoulos. Visual Variability Analysis for Goal Models. In *RE'04*, pages 198–207, Sept 2004.
5. A. Knauss, D. Damian, X. Franch, A. Rook, H. A Müller, and A. Thomo. ACon: A learning-based approach to deal with uncertainty in contextual requirements at runtime. *Information and Software Technology*, 70:85–99, 2016.
6. A. Knauss, D. Damian, and K. Schneider. Eliciting Contextual Requirements at Design Time: A Case Study. In *EmpiRE'14*, pages 56–63. IEEE, 2014.
7. M. Luckey, B. Nagel, C. Gerth, and G. Engels. Adapt Cases: Extending Use Cases for Adaptive Systems. In *SEAMS'11*, pages 30–39. ACM, 2011.
8. J.C. Muñoz-Fernández, G. Tamura, R. Mazo, and C. Salinesi. Towards a Requirements Specification Multi-View Framework for Self-Adaptive Systems. *CLEIej*, 18(2), 2015.
9. K. Pohl, G Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
10. N.A. Qureshi and A. Perini. Engineering Adaptive Requirements. In *SEAMS'09*, pages 126–131. IEEE, 2009.
11. N.A. Qureshi, A. Perini, F. Bruno, K. Irst, N.A. Ernst, and J. Mylopoulos. Towards a Continuous Requirements Engineering Framework for Self-Adaptive Systems. In *RE@RunTime*, pages 9–16, 2010.
12. A. Raian. *Modeling and Reasoning about Contextual Requirements: Goal-Based Framework.* PhD thesis, University of Trento, Italy, 2010.
13. Q.Z. Sheng and B. Benatallah. Contextuml: a uml-based modeling language for model-driven development of context-aware web services. In *In: The 4th Int. Conf. on Mobile Business*, pages 206–212, 2005.
14. R. Tawhid and D.C. Petriu. Product Model Derivation by Model Transformation in Software Product Lines. In *ISORCW'11*, pages 72–79, 2011.
15. N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. In *Self-Adaptive Software Systems*, volume 7475 of *LNCS*, pages 265–293. Springer, 2013.
16. N.M. Villegas. Context Management and Self-Adaptivity For Situation-Aware Smart Software Systems. Phd Thesis, Univ. Of Victoria, 2013.