A Software Lifecycle Process for Context-Aware Adaptive Systems *

Marco Mori
IMT Institute for Advanced Studies Lucca
Lucca, Italy
marco.mori@imtlucca.it

ABSTRACT

It is increasingly important for computing systems to evolve their behavior at run-time because of resources uncertainty, system failures and emerging user needs. Our approach supports software engineers to analyze and develop context-aware adaptive applications. The software lifecycle process we propose supports static and dynamic decision making mechanisms, run-time consistent evolution and it is amenable to be automated.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.9 [**Software Engineering**]: Management

General Terms

Design, Verification

1. RESEARCH PROBLEM AND MOTIVATION

As ubiquitous computing systems are becoming increasingly popular, software engineers have to deal with different variability dimensions such as the heterogeneity of the underlying communications, executing environment and changing user needs. In addition even the system may be modeled as a source of changes taking into account the possible software failures [2].

In the software engineering arena, adaptive systems are studied in order to define methodologies and theories to efficiently and consistently support their evolutions at run-time [31, 7]. Context plays a key role for the evolution since it represents the portion of the environment that is beyond the control of the system but it may influence the system behavior. Designers define a set of alternative behaviors at design time while the actual evolution decisions are taken at run-time when the context information are available. Since the context is not always predictable it is necessary to support also dynamic adaptations by providing mechanisms to update the logic of evolution [28]. To this end, the context needs to be explicitly modeled as proposed in contemporary research of context-aware systems [5, 18, 25].

We propose context-aware adaptive systems to deal with the uncertainty shown by the environment, the user and the system itself. Indeed, their main ability is to perform run-time adaptations driven by the "context" thus promoting software evolution to the norm rather than considering it as an exception. Context is characterized by foreseen and unforeseen variations to which correspond foreseen and unforeseen system evolutions. In the foreseen case the system evolves in order to keep satisfied a fixed set of requirements while in the unforeseen case the system evolves in order to respond to requirements variations that are unknown at design-time.

A consensus is emerging in the SE community that these kind of applications demand for a different software engineering process where the traditional distinction in phases and their characterization as static activities versus dynamic ones is disappearing [14, 23]. A challenging research problem is to define a software lifecycle process to enable software engineers to design context-aware adaptive applications resilient to context and user needs variations. This process should support consistent system evolutions, thus maintaining the system goal satisfaction in the face of new environmental conditions. To this end, evolutions should be both functional and non-functional. Run-time and consistent evolutions are achieved exploiting traceability links among system models at the different software engineering phases. Requirements, design and implementation artifacts have to be preserved at run-time along with a unified context model that supports the management of the evolution logic.

2. BACKGROUND AND RELATED WORK

The notion of context has been exploited in the software engineering field in order to define context-aware requirement models, context-aware architectural models and context-aware implementation models. In the literature different approaches address requirement engineering problems for context-aware adaptive systems. In [3] the authors have proposed a framework to explain the relations among the main concepts concerning requirement engineering for context-aware adaptive services. The approaches presented in [12] and [8] describe two different attempts to deal with the requirement elicitation problem for context-aware adaptive systems. They define requirement artifacts by means of

*My research is under the supervision of prof. Paola Inverardi, Università dell'Aquila (paola.inverardi@di.univaq.it)

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

first-class context entities. In [36] and [34] the authors propose two different formal representations of context-aware requirements by including a notion of uncertainty. In [30] the authors define a complete framework for requirement engineering to distinguish activities at design-time from activities at run-time. They provide a mechanism to evolve the requirement specification at run-time driven by the user. In [32] it is claimed that requirements represent the adequate level to enact the system evolution, thus systems should be requirement-aware. The authors also show how to achieve the traceability from requirements to the design in order to actually achieve the reconfiguration.
Architectural representations for context-aware system have been surveyed in [5, 11]. Finally at implementation level Hirschfeld et al. [17] propose a new programming technique called Context-oriented Programming (COP) in order to adapt the behavior of software entities to the current execution context. They surveyed different mechanisms to treat the context explicitly and to achieve the consequent adaptation for the implementation artifacts at run-time.
Several frameworks address the problem of achieving system evolution at different granularity level. The Rainbow framework [13] enables architecture-based self-adaptation for software systems. It proposes adaptation rules to specify how to reconfigure system components whenever certain situations arise. The framework supports non-functional adaptation whereas it does not consider requirements management at run-time. In addition there is no notion of consistency check for the system evolution and there is no explicit definition of context. The PLASTIC approach [4] applies reconfiguration strategies at implementation level by exploiting an explicit definition of context. The approach supports non-functional reconfiguration to statically defined Java artifacts driven by current context variations. Nevertheless a notion of consistency check is still missing. A common aspect for the above mentioned frameworks [13, 4] is that whatever is the grain of reconfiguration, they do not support evolutions arising at run-time. They only consider evolution strategies that are statically analyzed at design-time; thus making the system unable to achieve reconfigurations required by unanticipated user need variations arising at run-time.
In [35] the authors present a three-layered conceptual model to support the architectural reconfiguration of self-managing systems. This model entails a Component layer, a Change Management layer and a Goal Management layer. The Goal layer identifies the action to perform while the Change layer executes the required action and interacts with the Component layer to add/remove/reconnect components. Through the Goal Management layer, requirements are managed at run-time by generating new plans to perform whenever the deployed ones are not suitable for the current context situation. This feature supports reconfigurations required by new requirements arising at run-time. Even if the framework provides functional and non-functional evolutions there is no definition of context and a definition of consistency check for the composition of components is still missing.
To the best of our knowledge the frameworks presented in the literature do not consider a complete software lifecycle process for context-aware adaptive systems. They address only a few of the issues arising from the three software engineering phases. Moreover most approaches are not based on an explicit context model to support system evolution in all phases of the software lifecycle process.

3. APPROACH AND UNIQUENESS

My research work concentrates in defining a framework to support a general lifecycle process for the development and the evolution of context-aware adaptive systems. We focus on context-aware requirements, context-aware design models and context-aware implementations mechanisms.
We have identified two different types of evolutions each one addressing a different nature of context variability. On one hand, designers deal with foreseeable context variations by providing the required system evolutions at design-time. They statically define the logic of evolution to the foreseeable context by means of different system variants that have to be deployed and un-deployed at run-time in order to keep satisfied a fixed set of system requirements. The selection among the system variants is driven by the context requirements and the non-functional properties that characterize each variant. On the other hand, the context may change unpredictably thus causing the change of user needs that can be expressed as a variation to the requirement set to satisfy. The user may specify a new requirement as a consequence of the unforeseen context variation. Therefore the evolution logic should be revisited at run-time by updating the space of system variants provided by the designer at design time. We refer to the first case as *foreseen evolution* because the evolution logic is embedded inside the system models and the foreseeable contexts have been completely characterized at design time. The second case is called *unforeseen evolution* as the evolution logic and the context is not known at design time and the user is placed into the evolution feedback loop. Fig.1 shows how the context affects system evolutions. While in the foreseen evolution the system evolves to keep satisfied a fixed set of system requirements in different known contexts, the unforeseen evolution is driven by new requirements arising from unforeseeable contexts.

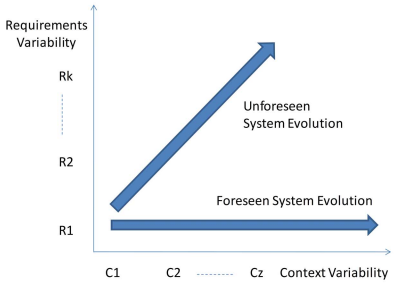

Figure 1: System Evolutions

Our approach to support context-aware adaptive applications is inspired by the Software Product Line Engineering (SPLE) perspective. Among the possible ways to model a system, SPLE is amenable to represent system variability and it supports a notion of consistency. In SPLE the basic unit of behavior is the so called feature that is the smallest part of a service that can be perceived by the user. The system variability is expressed through the space of the system variants. Each variant is obtained by putting together two or more features and it shows the feature interaction phenomenon if its features run correctly in isolation but they give rise to undesired behavior when jointly executed [26, 1, 6]. The work [10] has already shown common research questions between SPLE and adaptive system and the necessity to dynamically manage features at run-time. Most

recently, Dynamic Software Product lines (DSPL) have been presented as a new direction in SPL engineering field to deal with software capable of adapting to changing user needs and evolving context environment at different binding time [16, 29]. However in the DSPL field dealing with evolutions arising at runtime is still an open issue.
Our intention is to take advantage of the methodologies proposed in SPLE to support consistent evolutions for context-aware adaptive systems. We define a context-aware adaptive application in terms of sets of features each one implemented with a component and / or a service. We define a feature as composed by a context-independent requirement, a context-dependent requirement, and an implementation part. The notion of requirements we adopt is inspired by the taxonomy proposed by Glinz in [15]. A feature is a triple $f_i = (R_i, C_i, I_i)$ where each element is defined as follows:

- R_i is a conjunction of functional, performance and specific quality requirements (context-independent)
- C_i is a context-dependent constraint requirement
- I_i is the feature implementation expressed as Java code.

The definition above is inspired by [9] which has been in turn inspired by the Problem Frame approach defined in [24]. Differently from these approaches we refer to C as the context requirement instead of the domain assumption.
Each *system variant* is represented as a different combination of features expressed as $G_F = (R_F, C_F, I_F)$. At this level of description we do not explain how to combine features. We just suppose to have an abstract union operator among features which is defined in terms of union operators for context-independent requirements, context-dependent requirements and implementation components. In concrete instantiations of the framework these operators will take a precise form.
We represent the space of admissible system variants by using the *feature diagram* defined in the SPLE with a tree structure [33]. We adopt the absence of feature interaction as the notion of *consistent evolution* for a system variant. A certain variant G_F shows a feature interaction phenomenon if the features in F run correctly in isolation but they give rise to undesired behavior when they are jointly executed. The feature interaction for G_F is checked as: $I_F, C_F \vdash R_F$. This formula is automatically verified at design time to cope with foreseen evolutions and at run-time in order to support the unforeseen evolutions. We have defined the consistency check formula by means of three different problems: the satisfiability for the context requirement C, the satisfiability for the requirement R and the validation/verification problem of I against R. The context model is a key aspect in performing the system evolution; it has to be monitored and updated at each evolution step. It entails the set of entities based on which our context requirements are defined. Since in ubiquitous computing the unpredictability of the context is the main cause of system failures, we believe that the problem of checking the satisfiability for the context requirements plays a key role in supporting consistent evolutions. In addition we consider the three problems to be run-time problems since we support a notion of run-time unforeseen evolution. As a consequence computational issues become important to actually achieve the consistency check at run-time. Furthermore we assume that requirements artifacts are available.

4. RESULTS AND CONTRIBUTIONS

We have defined a general lifecycle process for context-aware adaptive systems in order to support the consistent evolutions based on two decision making mechanisms. The static decision making mechanism supports foreseen evolutions whereas the dynamic decision making mechanism supports unforeseen evolutions. As shown in our preliminary work [19] we have provided easy-to-use modeling techniques for software engineers along with preliminary ideas to enact run-time evolutions. We have presented an explicit model of context and we have shown how the system can evolve its behavior at run-time based on context changes and changing user needs. We have dealt with the problem of selecting and executing the most suitable system configuration in a given context and how to reconfigure the system to augment it with un-anticipated behavior arising from new requirements at run-time. We have formalized a notion of consistent evolution based on the context-dependent requirements.
In [22] we have designed and partially implemented a framework that supports our general lifecycle process for context-aware adaptive systems. We have defined a generic interface architecture that can be implemented by any evolution framework to support our kinds of evolution.
Our framework is generic and it is amenable to augment the system with new requirements arising at run-time. Since new requirements may interact with deployed requirements we have provided the support to keep those entities at run-time and check their correctness jointly. In [21] we have formalized three consistency checks to provide high-assurance guarantee to the evolution. In this paper we have also proposed a technique to check context requirements at run-time with respect to the variations of context resources. In [20] we have shown how to verify context-independent requirements with respect to implementation artifacts. We have defined a practical technique which exploits the Java Path Finder tool[1].
As far as static decision making mechanisms are concerned in [27], we have proposed a practical support for context-aware system reconfiguration based on changing user preferences. In order to select the best configuration to deploy we have defined a multi-criteria selection problem by means of an aggregative objective function that combines user benefit and cost. This function accumulates the perceived current user benefit and the probable future user benefit evaluated exploiting a state-based context prediction model. The reconfiguration cost measures the distance between two system variants in terms of features to deploy and un-deploy. We have validated this approach by simulating the reconfiguration process at large scale in order to capture the relevance of considering future contexts. The proposed selection algorithm performed satisfactory in the set of experiments that we carried out.
Currently, we are implementing an instance of our framework with specific technologies to define requirements specifications and implementation artifacts and technologies to carry out the consistency check. We will also extend the PLASTIC approach [4] to achieve system reconfigurations on Java implementation artifacts at run-time for our unforeseen evolution. My research will result in a new methodological approach in developing and evolving context-aware adaptive applications. I will show the applicability of the

[1]http://babelfish.arc.nasa.gov/hg/jpf/jpf-core

proposed lifecycle process by applying the target framework to large scale systems.

5. REFERENCES

[1] M. Alférez, A. Moreira, U. Kulesza, J. Araújo, R. Mateus, and V. Amaral. Detecting feature interactions in spl requirements analysis models. In *FOSD*, pages 117–123, 2009.

[2] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *SEAMS*, pages 27–47, 2009.

[3] A. F. Andrea and A. Savigni. A framework for requirements engineering for context-aware services. In *STRAW 01*, pages 200–1, 2001.

[4] M. Autili, P. D. Benedetto, and P. Inverardi. Context-aware adaptive services: The plastic approach. In *FASE*, pages 124–139, 2009.

[5] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *IJAHUC*, 2(4):263–277, 2007.

[6] J. Bisbal and B. H. C. Cheng. Resource-based approach to feature interaction in adaptive software. In *WOSS*, pages 23–27, 2004.

[7] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, 2009.

[8] J. Choi. Context-driven requirements analysis. In *ICCSA (3)*, pages 739–748, 2007.

[9] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a feature: A requirements engineering perspective. In *FASE*, pages 16–30, 2008.

[10] A. Classen, A. Hubaux, F. Sanen, E. Truyen, J. Vallejos, P. Costanza, W. De Meuter, P. Heymans, and W. Joosen. Modelling variability in self-adaptive systems: Towards a research agenda. In *Proc. of McGPLE at GPCE08*, pages 19–26, 2008.

[11] P. D. Costa, L. F. Pires, and M. van Sinderen. Architectural patterns for context-aware services platforms. In *IWUC*, pages 3–18, 2005.

[12] B. Desmet, J. Vallejos, P. Costanza, W. D. Meuter, and T. D'Hondt. Context-oriented domain analysis. In *CONTEXT*, pages 178–191, 2007.

[13] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[14] C. Ghezzi, P. Inverardi, and C. Montangero. Dynamically evolvable dependable software: From oxymoron to reality. In *Concurrency, Graphs and Models*, pages 330–353, 2008.

[15] M. Glinz. On non-functional requirements. In *RE*, pages 21–26, 2007.

[16] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, and T. Sintefict. Dynamic software product lines. *IEEE Computer*, 41(4):93–95, 2008.

[17] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[18] J. Hong, E. Suh, and S.-J. Kim. Context-aware systems: A literature review and classification. *Expert Syst. Appl.*, 36(4):8509–8522, 2009.

[19] P. Inverardi and M. Mori. Feature oriented evolutions for context-aware adaptive systems. In *EVOL/IWPSE*, pages 93–97, 2010.

[20] P. Inverardi and M. Mori. Model checking requirements at run-time in adaptive systems. In *ASAS at ESEC/FSE*, 2011.

[21] P. Inverardi and M. Mori. Requirements models at run-time to support consistent system evolutions. In *Technical Report, University of L'Aquila, Department of Computer Science*, 2011.

[22] P. Inverardi and M. Mori. A software lifecycle process to support consistent evolutions. In *Tec. Rep. Univ. of L'Aquila, Department of Computer Science*, 2011.

[23] P. Inverardi and M. Tivoli. The future of software: Adaptation and dependability. In *ISSSE*, pp 1–31, 2008.

[24] M. Jackson. *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[25] G. M. Kapitsaki, G. N. Prezerakos, N. D. Tselikas, and I. S. Venieris. Context-aware service engineering: A survey. *JSS*, 82(8), 2009.

[26] D. O. Keck and P. J. Kühn. The feature and service interaction problem in telecommunications systems. a survey. *IEEE TSE*, 24(10):779–796, 1998.

[27] M. Mori, F. Li, C. Dorn, P. Inverardi, and S. Dustdar. Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems. In *SEFM*, 2011.

[28] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion*, pages 899–910, 2008.

[29] C. A. Parra, X. Blanc, and L. Duchien. Context awareness for dynamic service-oriented product lines. In *SPLC*, pages 131–140, 2009.

[30] N. A. Qureshi and A. Perini. Requirements engineering for adaptive service based applications. In *RE*, pages 108–111, 2010.

[31] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009.

[32] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *RE*, pages 95–103, 2010.

[33] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[34] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. Awareness requirements for adaptive systems. In *SEAMS*, pages 60–69, 2011.

[35] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: a combined approach to self-management. In *SEAMS*, pages 1–8, 2008.

[36] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *RE*, pages 79–88, 2009.