IMT Institute for Advanced Studies, Lucca

Lucca, Italy

A Framework to Support Consistent Design and Evolution of Adaptive Systems

PhD Program in CSE

XXIV Cycle

By

Marco Mori

2012

The dissertation of Marco Mori is approved.

Program Coordinator: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Prof. Paola Inverardi, University of L'Aquila

Individual Evaluation Committee :

Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Prof. Antonia Bertolino, Consiglio Nazionale delle Ricerche, Pisa

The dissertation of Marco Mori has been reviewed by:

Prof. Anthony Cleve, University of Namur, Belgium

Prof. Klaus Pohl, Lero, Limerick, Ireland - University of Duisburg-Essen, Germany

IMT Institute for Advanced Studies, Lucca

2012

Contents

List of Figures **viii**

List of Tables **x**

Acknowledgments **xi**

Vita and Publications **xii**

Abstract **xv**

1 Introduction **1**

1.1 Problem statement 2
1.2 Research questions 4
1.3 Contributions 6
1.4 Structure 7

2 Background **9**

2.1 Adaptive Systems 9
2.2 Context-aware systems 11
2.3 Context Models 11
2.3.1 Context models approaches 12
2.3.2 Context modeling evaluation 13
2.4 Phases for building adaptive system 14
2.4.1 Requirement engineering for adaptive systems . . . 14
2.4.2 Design for adaptive system 16
2.4.3 High level design 17

2.4.4 Low level design 20
2.4.5 Context-aware middleware 22
2.5 Assurance 25
2.6 Framework for adaptive systems 27
2.7 Support to foreseen evolutions 29

3 Approach **32**
3.1 Taxonomy of the evolution 32
3.2 Software Product Line Engineering perspective 33
3.3 Case Studies 34
3.3.1 eHealth application 34
3.3.2 Mandelbrot fractal 36
3.4 Context model 36
3.5 Requirements taxonomy 38
3.5.1 Context requirements 39
3.6 Adaptive application 39
3.6.1 Feature 40
3.6.2 System variant 42
3.6.3 Examples 43
3.6.4 Feature diagram 44
3.7 Formalization of evolution and execution 45
3.7.1 Semantic rules 47
3.8 Evolution consistency 49
3.8.1 Requirements for requirements at run-time 50
3.8.2 Context analysis 51
3.8.3 Model checking context-independent requirement . 55

4 Software Process for Adaptive Systems **60**
4.1 Software Process 60
4.1.1 Example: mandelbrot fractal 63
4.2 System Evolution 66
4.2.1 Foreseen Evolution 66
4.2.2 Unforeseen Evolution 69

5 Static decision-making reconfiguration 74
5.1 Motivating scenario 75
5.2 Basic model 76
5.2.1 Operative context model 77
5.2.2 Variants 77
5.2.3 User Context Model 78
5.2.4 Transition cost 79
5.3 Rankings of the variants 80
5.4 Case Study 82
5.5 Experiment 86

6 Evolution framework 90
6.1 MAPE cyle 90
6.2 Framework interface architecture 92
6.3 Framework instance 92
6.3.1 Description 93
6.3.2 Framework primitives 95
6.4 Mandelbrot fractal application 96
6.4.1 Application without framework 97
6.4.2 Application with framework 99
6.4.3 Evaluation 103

7 Conclusion 105

A Variants examples 108

References 112

List of Figures

1 Event-Control-Action pattern structure 17
2 Context Source and Managers Hierarchy pattern structure 18
3 Action pattern structure 19
4 Unifying middleware archietecture 24

5 System Evolutions 33
6 E-Health architecture 35
7 Context model (a) eHealth application (b) Mandelbrot fractal application 38
8 Syntax of context requirements 39
9 Feature $f_{graphOx} = (R_{graphOx}, I_{graphOx}, C_{graphOx})$ 41
10 Feature $f_{genPro} = (R_{genPro}, I_{genPro}, C_{genPro})$ 42
11 Example: class diagram (G_1) 44
12 Feature diagram 46
13 System state-based model 47
14 New feature: graphical visualization (eHealth) 54
15 New feature: textual visualization (eHealth) 55
16 Example: graphical oxygenation 56
17 Example: graphical respiratory rate 57
18 Visualization interaction 58

19 Software process 61
20 Application features 64
21 Probabilistic evolution automata 67

22 Example: new feature 71
23 Refined feature diagram 72

24 User Preferences Example 75
25 Conceptual Model 76
26 Probabilistic automaton excerpt 79
27 User Context Automata 84
28 Normalized average user benefit (a) and normalized average reconfiguration cost (b) with $h = 1.0$, $h = 0.5$ and $h = 0.2$ depending on utility objectives weights α. 87
29 Pareto-optimal and sub-optimal configurations 88

30 MAPE cycle 91
31 Evolution framework architecture 93

List of Tables

1 System variants 65
2 Feature consistency table 66
3 Variant consistency table 66
4 Refined variant consistency table 72

5 System variants 83
6 Distance evaluation 84
7 Evaluation of the variants 86

Acknowledgements

Firstly I would like to thank my supervisor Prof. Paola Inverardi for having introduced me to the research world. I have been very fortunate to have a supervisor who constantly supports my research activity and provides me essential feedbacks for my daily work. I am very grateful to her for her pragmatic support to my research. I would also like to thank Prof. Marco Autili for having giving me valuable contributions to my research. I thank my supervisor and the University of L'Aquila for having economically supported my participation to conferences, workshops and summer schools which are essential for the research work.

I would like to thank everyone that helped in my PhD activities and everyone that supported me during this very intensive three years. I thank my family and my friends for giving me the encouragement. Finally, I sincerely thank Lucia who has given me the strength to face any problems.

Vita

September 25, 1984	Born in Chiaravalle, Ancona, Italy
2006	Bachelor of Science (B.Sc.) Degree in Computer Science Final mark: 110/110 with honors University of Camerino, Macerata, Italy
2008	Master of Science (M.S.) Degree in Computer Science Final mark: 110/110 with honors University of Camerino, Macerata, Italy
2008-2009	(1-year) Scholarship under the supervision of Prof. Flavio Corradini, Prof. Emanuela Merelli and Dr. Alessandro Olivi LULab Research Group, University of Camerino, Macerata, Italy - "Gruppo Loccioni", Angeli di Rosora, Italy
2010-2011	(6-month) Visiting Phd student at the "Distributed Systems Group, Information Systems Institute" at Vienna University of Technology in Austria under the supervision of Prof. Schahram Dustdar, Dr. Fei Li and Dr. Christoph Dorn

Publications

1. Paola Inverardi and Marco Mori, “Feature oriented evolutions for context-aware adaptive systems”, in Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pages 93-97, 2010.
2. Marco Mori, “A software lifecycle process for context-aware adaptive systems”, in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 412–415, 2011.
3. Paola Inverardi and Marco Mori, “Model checking requirements at run-time in adaptive systems”, in Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems (ASAS), pages 5–9, 2011.
4. Paola Inverardi and Marco Mori, “Requirements models at run-time to support consistent system evolutions”, in Proceedings of the 2nd International Workshop on Requirements@Run.Time (RE@RunTime), pages 1–8, 2011.
5. Marco Mori, Fei Li, Christoph Dorn, Paola Inverardi, and Schahram Dustdar. “Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems”, in Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM), pages 286–301, 2011.
6. P. Inverardi, M. Mori. “A Software Lifecycle Process to Support Consistent Evolutions”, 2nd book on Software Engineering for Self-Adaptive Systems, 2012.

Presentations

1. M. Mori, “Engineering Context-Aware Adaptive Systems” Poster Presentation at *University of Oulu*, Finland, 2010.
2. Paola Inverardi, M.Mori, “Feature oriented evolutions for context-aware adaptive systems” Seminar at *University of L'Aquila*, Italy, 2010.
3. M. Mori, “Engineering Context-Aware Adaptive Systems” Seminar at *Vienna University of Technology (TU)*, Distributed Systems Group - Information Systems Institute, Vienna, Austria, 2011.

Abstract

Nowadays software systems in the ubiquitous environment have to consider variability as their main characteristic. The ever-changing environment affects these systems and their ability of satisfying functional and non-functional requirements. It is challenging to create and to support the variability of such applications taking into account different variability dimensions. Traditional software processes are not suited for adaptive applications since they consider a fixed definition of context and a clear division between design-time and run-time activities. Attempts to manage variability are only focused at specific phases of the process while it is missing a comprehensive process to face variability at all phases thus supporting the complete creation and the evolution of adaptive applications.

The thesis aims to give a possible solution to these problems by defining a new software lifecycle process for building and evolving adaptive applications in a consistent manner. A system is represented following a feature engineering perspective which considers together requirements and code artifacts. We have identified which are the inconsistencies for evolving a system and we have discovered that in order to keep the correctness of the evolution it is necessary to consider system models ranging from the model space to the solution space. The proposed process encompasses two different kinds of evolution: design-time and run-time evolutions. Design-time evolutions are planned before running the system by means of a set of variants whose behavior consistently fit a set of predefined contexts. These alternatives are checked for inconsistencies at design time whereas reconfiguration decisions are taken at run-time based on the current context. Results shows that it is promising to consider predictive information for selecting the best reconfiguration especially in the presence of multiple competing objectives. Runtime evolutions are enacted by enhancing the system with new requirements that may be introduced by the user as a consequence of unpredicted context situations. In this case the consistency check for a new alternative behavior is performed directly at run-time.

The proposed process is supported by an evolution framework. The framework is defined in terms of a generic definition and one possible instantiation. The generic interface architecture describes the interfaces that should be provided to support the software process for adaptive systems. A possible instantiation with current technologies shows the feasibility of supporting the process.

Chapter 1

Introduction

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 (AdLMW09). In this scenario adaptive systems are able to modify their structure and / or behavior as a consequence of different variability dimensions (ST09):

Self-adaptive systems aim to adjust various artifacts or attributes in response to changes in the self and in the context of a software system. By self, we mean the whole body of the software, mostly implemented in several layers, while the context encompasses everything in the operating environment that affects the systems properties and its behavior.

Software engineers support this system's variability by defining the set of possible software alternatives at design time. Consequently, the system takes autonomously the actual reconfiguration decisions at run-time. Context drives adaptivity since it determines which reconfigurations are admissible and it helps to select the best reconfiguration possible. As a consequence it is necessary to explicitly model the context as proposed in the literature of context-aware systems (BDR07; KPTV09; HSK09).

Context is not completely known at design time thus making the process of designing and developing ubiquitous applications continuing at execution time (CdLG$^+$09; GIM08; IT08). At design time software engineers can only define software alternatives having in mind a partial representation of the context in which the system is going to operate. In addition resource-constraint devices limit the number of deployable alternatives. Thus the set of software alternatives provided at design time may have to be augmented in order to face new unforeseen environmental conditions. Whenever new context information becomes available at run-time, the user may specify new requirements that are unknown to the designer. As a consequence of a new requirement, the system should automatically adapt/evolve itself in order to satisfy the new user's expectations.

However, reconfigurations have to be completed maintaining the system in a consistent state in order to avoid incorrect system behaviors. To this end we propose a notion of high-assurance that is suited for adaptive systems:

high-assurance provides evidence that the system satisfies continuously its functional or non-functional requirements thus maintaining the user's expectations despite predictable and unpredictable context variations.

Context plays a key role in the process of providing high-assurance for adaptive systems. An effective high-assurance methodology should provide guarantees with respect to predictable and unpredictable context variations. Such a methodology should be performed at design time to deal with predictable context variations and it should be performed at run-time to support unpredictable context variations. At design time it should be possible to check the assurance for a predefined set of software alternatives whereas at run-time it should be possible to check unpredicted software alternatives that satisfy unexpected user needs.

1.1 Problem statement

The thesis focuses on the problem of creating adaptive applications and supporting their variability. In order to prevent the incorrect behaviors that may be caused by system variability, it is necessary to define a notion of consistency. By considering system models at different granularity levels it is possible to capture a wide spectrum of inconsistency notions. We consider system models ranging from the problem space level to the solution space level, thus we take into account three different phases namely requirements, design and implementation. At each phase, context has to be explicitly modeled in order to account for variability. In the literature there exist different attempts to manage variability at requirements, design and implementation level. These approaches provide practical techniques to develop adaptive systems and to support the activities of the specific phases. The main problem is the lack of a unified support which integrates all the phases together thus supporting a complete lifecycle for adaptive applications.

A consensus is emerging in the SE community that adaptive 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 (GIM08; IT08). A challenging research problem is to define a software lifecycle process to enable software engineers to design 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 and they should be 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 their evolution. Models should be exploited by an integrated support in order to automate, as much as possible, the process of developing and evolving adaptive applications. This would enable the software engineers to reuse a set of ”good practices” and tools for building and maintaining such applications (Ost87).

1.2 Research questions

The thesis concentrates in defining a new system model which represents adaptive applications at different levels and in defining a new methodology which entails the steps required to create such applications from requirement artifacts to the corresponding code artifacts.

The objective of this thesis is to define a software lifecycle process for adaptive systems. Through this process it should be possible to check the correctness for the set of possible software variants. It should be also possible to check the correctness for new variants that are dynamically created at run-time as a consequence of unpredicted context variations. The final goal of the thesis it to promote a set of good practices and to devise a practical support for developing and evolving adaptive applications.

Within the overall objective of the thesis we have identified five different research questions:

- RQ1: How to manage context-dependent system variability? Which abstractions for the system can better handle variability and how could they support an automatic decision-making procedure?

- RQ2: How to classify context-dependent evolutions?

- RQ3: How to represent requirements models and their evolutions at run-time?

  Evolutions of requirements should be performed consistently. At run-time a user may introduce a new requirement in the system, thus it should be possible to check the already implemented requirements along with the new one. As a consequence it is necessary to keep a representation of requirements at run-time and to enable their evolution.

- RQ4: How should the software lifecycle process deal with the uncertainty coming from the environment?

  Following the approach presented in (SBW^{+}10) there are different levels of uncertainty:

  - level one: it is possible to estimate possible outcomes about the future; there are variables with unknown values but there is almost no uncertainty about the change.
  - level two: there are exhaustive and mutually exclusive future scenarios that can be listed.
  - level three: there is a set of possible future scenarios, but they are not exhaustive.
  - level four: it is very difficult to identify future scenarios.

  In the literature of context-aware systems the firsts two levels of uncertainty have been addressed. Almost everything can be anticipated thus it is sufficient for a software engineer to define a set of predefined software alternatives. At run-time the system automatically adopts the behavior that better fits a certain context.

  In order to manage the two last levels of uncertainty it should be possible to enhance the behavior of the system at run-time. For example it should be possible to revise the space of the possible alternative behaviors to deal with new unforeseen contexts and system alternatives.

- RQ5: How to perform reconfigurations taking into account competing objectives?

  Evolutions should provide the maximum user benefit possible while they should be performed minimizing time and cost factors. There exist reconfiguration processes which consider only few of the following factors: user preferences, cost, context variability and predictive information. In order to achieve better performance for the reconfiguration process it seems to be valuable to consider all these factors together in one formal framework.

1.3 Contributions

The thesis provides the following contributions:

- a system model able to manage variability ranging from the problem space (requirements) to the solution space (code). This model is based on an explicit model of context which drives the evolution.

- two different notions of evolution: foreseen evolutions deal with context variations that can be anticipated while unforeseen evolutions deal with context variations that cannot be predicted at design time.

- two notions of consistent evolution that can be applied to foreseen and unforeseen evolutions. These notions exploit our system notation and they consider different abstractions of the system (requirements and code).

- a general lifecycle process for context-aware adaptive systems which supports consistent evolutions. This process support the creation of the adaptive application from the requirements definition to the code implementation.

- a mechanism to reconfigure the system in order to augment it consistently with new behaviors arising from new requirements at runtime.

- a mechanism to select and execute the most suitable reconfiguration in a given context. We have implemented a decision-making mechanism as a multi-objective optimization problem which considers current preferences, probable future preferences and a generic reconfiguration cost component.

- a framework that supports a general lifecycle process for adaptive systems in terms of a generic interface architecture and one of its possible implementations with current practice technologies.

1.4 Structure

The thesis is divided in the following chapters:

- Chapter 2 introduces the definitions of adaptive system and context. The Chapter surveys the state of the art for the approaches and the frameworks that manage variability at requirement, design and implementation phases. It also introduces the problem of providing assurance for adaptive systems.

- Chapter 3 describes how we model the system and its variability following the feature engineering perspective. We describe two different kinds of evolution and we formalize them through some basic semantic rules. We show how it is possible to check the correctness of the system at design time and at run-time based on two different mechanisms: a context analysis process and a model-checking process. We give a detailed description of two adaptive applications starting from two different case studies.

- Chapter 4 defines the software lifecycle amenable for adaptive systems. In this chapter we explain each single phase of the process by applying it to one of the case studies introduced in Chapter 3. We show how the process supports the two different types of evolution.

- Chapter 5 describes a mechanism for selecting the best possible reconfiguration decision by considering current and future contexts. In this chapter we introduce an ad-hoc case study to illustrate the mechanism and we present a formalization of a multi-objective optimization problem. Differently from most of the approaches found in the literature, our generic decision-making process considers multiple factors together. This makes our process configurable for many environments thus enabling the software engineer to tune his interest on specific objectives.

- Chapter 6 describes our implementation to support the software lifecycle process. This chapter gives a description of the interfaces architecture which defines a generic framework. Based on the generic framework we show one of its possible instantiations with current practice technologies.

Chapter 2

Background

2.1 Adaptive Systems

Adaptive software systems are a class of software systems that can modify their behavior at runtime due to changes to the requirements, to the environment in which the system is deployed or to the system itself.

The approach presented in (AdLMW09) defines a possible classification for the key modeling dimensions of self-adaptive systems. The authors propose a classification of the modeling dimensions for such systems based on four different groups, namely systems goals, causes of adaptation, mechanisms to achieve adaptation, effects of adaptation and causes of adaptation.

The system goal is associated with the lifetime of the systems or with the scenarios associated to the system. The goal can be associated to the adaptability aspects of the application, middleware or infrastructure. Among the different dimensions characterizing this group, the *flexibility* dimension can range over three different values: rigid, constrained and unconstrained. The *evolution* dimension identifies whether the goal of the system can change during the lifetime of the system. Thus evolutions may be static if the goal does not evolve while they could be dynamic if the goal can change at run-time. The *duration* dimension defines the persistency of the goal; indeed while some goals can be relaxed and they can be valid only for a limited period of time, other goals should always be achieved. Finally the *multiplicity* and *dependency* dimensions express respectively the number of goals to achieve and the relations among them. Relations among goals may be independent or dependent based on the fact that they may or may not affect each other. If goals are dependent, they may be complementary or conflicting. In this last case a trade-off analysis is needed in order to weigh among competing objectives.

The second group is related to the changes that are the causes of the adaptation. Changes can be be classified based on the place where they occur, type, frequency and whether they can be anticipated or not. The *source* dimension identifies if the change occurs in the external environment or internally to the system. The *type* dimension refers to functional, non-functional and technological upgrade. *Frequency* describes how often changes occur, they can be either rare or frequent. In case of frequent changes the responsiveness of the adaptation could be jeopardized. Finally changes can be predicted or not, thus there exist different degrees of *anticipation* which range from foreseen to unforeseen.

Within the group related to the mechanisms of adaptation, the *type* dimension captures whether the adaptation is related to the parameters of the systems or to its structure. The *autonomy* dimension defines the degree of outside intervention during adaptation namely autonomous or assisted. A mechanism of adaptation could either be centralized if it is performed by a centralized component or decentralized if it is performed by decentralized components. The adaptation mechanism can also involve the whole system or one of its parts. Depending on the application domain, this mechanism can last for a short, a medium or a long time period. The *timeliness* dimension describes whether the time period for performing adaptation can be guaranteed or not thus ranging from guaranteed to best effort. Finally, the *triggering* dimension defines which is the change that triggers the adaptation, either event-based or timely-based.

The last group is related to the effect of the adaptation. The *criticality* dimension establishes which is the impact of the adaptation in case of failure. Within this group, the *predictability* dimension captures the possibility to have a deterministic or non-deterministic behavior as result of the adaptation. The *overhead* dimension is useful to define the impact of the adaptation with respect to the quality of the system. Finally, the *resilience* dimension establishes the persistence of the adaptation ranging from resilient to vulnerable.

2.2 Context-aware systems

Context-aware systems have the ability of managing context information as first-class information. Many definitions of context are found in the literature. Among them the definition of context given in (Dey01) is widely accepted in the community of context-aware systems:

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including location, time, activities, and the preferences of each entity.

Once given a definition of context we refer to context-awareness as the ability of using context information that is the ability of extracting and interpreting context information and adapting system's functionality to the current context of use (BC04b).

2.3 Context Models

Many models to reason about the context have been already presented to be exploited from the requirement, design and implementation layers. Since the access to context information should be automated, context models have to represent context information in a machine-readable form. In the literature are found different approaches for modeling the context; many of them lack generality and they are tailored to a specific application domain.

2.3.1 Context models approaches

Strang and Popien (SLP04) have surveyed the main context modeling approaches adopted to manage context information:

- The most simple data structure for modeling contextual information is the *key-value* model approach. Key-value pairs are easy to manage and they enable the retrieval of context information as environment variables. The problem with key-value models is the lack of capabilities for structuring complex context data.

- A more complex context modeling approach is the *markup scheme* model. The main feature of this model is its hierarchical structure consisting of markup tags with attributes and content. Markup tags are recursively defined by means of other markup tags. These approaches to model context information are based on XML (Extensible Markup Language) and RDF (Resource Description Framework) languages. XML is the most well known markup language whereas RDF is a framework which enables the definition of resources in the form of subject-predicate-object expressions.

- *Graphical* context models are very intuitive models with a low level of formality. They can either be extension of UML (Unified Modeling Language) such as the ContextUML approach presented by Sheng et al. (SB05), or they can be extension of other graphical modeling languages.

- *Object-oriented* context models are mainly designed to exploit the main characteristics of the object-oriented approach such as encapsulation, reusability and inheritance. The contextual information is embedded as the state of an object whereas interfaces are provided to access the information and to modify the internal state.

- *Logic* based context models have a greater level of formality with respect to the previous models. Context is usually defined through facts, expressions and rules that can be processed either to keep a consistent set of context information, or to infer new context information. For instance, let us suppose to have two different rules: the first declares that the number of persons in a certain room is four, whereas the second asserts that the application running inside the room is Powerpoint. Starting from this information, processing activities can create a new rule asserting that inside the involved room the current social activity is a presentation activity.

- *Ontology* context models are promising to represent a description of context concepts and their relations. They have the highest expressivity and they allow the most complex reasoning processes (CFJ04).

2.3.2 Context modeling evaluation

Ubiquitous applications demand for context modeling approaches with specific characteristics. Strang and Popien (SLP04) have compared the main context modeling approaches based on a set of requirements for ubiquitous environments. They consider the following features: distributed composition, partial validation, richness and quality of information, incompleteness and ambiguity, level of formality and applicability to existing environments.

The first requirement they consider is the distributed composition of context information; since the ubiquitous paradigm belongs to the distributed computing field, there is a lack for a common entity being responsible to manage context information. In order to have a common reasoning over the context data, the system should allow the composition and administration of context models. The second requirement is the partial validation of context information. It consists of validating contextual information at instance level against a context model, such as via simple type checking or more complex full data content validation. This should be achieved even if there is no single place or point in time in which a common context information instance is kept by a single node. A further requirement refers to the quality of information that should enable the system to exploit context values gathered from different kinds of sensors; context models should be able to express quality and richness of context information that may change over time. Yet another requirement for a context modeling approach is the ability to deal with the incompleteness and ambiguity of context information. This requirement arises from the fact that it is difficult to characterize the entities in the ubiquitous environment; thus it is necessary to provide mechanisms to merge incomplete information in order to have a certain consistency of context information at instance level. Finally, formality of a context model approach is necessary to enable automatic reasoning whereas applicability to existing environments should guarantee that these approaches are applicable with existing infrastructures such as web services.

Based on all the above requirements, the conclusion of the authors is that the ontology based models is the approach which is better suited for ubiquitous environments. Moreover, the ontology based approach has some intrinsic characteristics such as simplicity, flexibility, extensibility, genericity and expressiveness (KMK$^+$03) which make it preferable for ubiquitous applications.

2.4 Phases for building adaptive system

In order to build an adaptive application we have considered the following set of phases: requirement, high level design, low-level design (i.e. implementation) and middleware (run-time support). In the literature can be found different approaches to define system models at different phases all exploiting a context definition to drive adaptivity.

2.4.1 Requirement engineering for adaptive systems

Adaptive systems have a set of high level goals that should be met regardless the conditions of the environment (CdLG$^+$09). Information belonging to the environment is not always complete thus the requirement engineering phase has to face the uncertainty belonging to the context. This leads to continuously changing requirements in response to the context variations. Therefore, it is not feasible to predict the set of the possible adaptations.

When we talk about context we mean something that can be monitored from the environment in the same way presented by Finkelstein and Savigni's work (AS01). They have studied the key problems associated with requirement engineering in the area of context-aware services. They propose a conceptual model to explain the relations among main concepts concerning the requirement phase. In their approach the environment is out of control of the system and it provides the surrounding in which a machine is supposed to operate. Monitoring the environment leads to extract the context, that is the elements that influence the system behavior at a certain point in time during the execution. The requirements are the way to achieve a goal within a certain context; whenever the change of the environment leads to a context change, the requirement could be no longer valid. In such a case a new requirement, if it exists, should be found in order to maintain the validity of the high level system objectives (goals). Finkelstein and Savigni's work (AS01) is just a first step toward the direction of requirement engineering for adaptive systems. It is not actually explained how to manage the runtime variations of context and requirements and how to define the software requirements.

In the literature different approaches address the problem of defining requirements and how to exploit them to support adaptivity. Hong et. al (HCS05) introduce a methodology for the elicitation of context-aware requirements and their matching with the possible variations of the system. Based on the presented meta-model the right adaptations for the system are selected during the execution phase according to the context and to the user preferences. This approach allows the definition of the adaptation but it does not explain how to structure the requirements and how to deal with them. Context Oriented Domain Analysis (CODA (DVC$^+$07)) is an approach to analyze and to structure software requirements for context-aware adaptive systems. The CODA authors point out that traditional models for requirement engineering are not useful within the domain of context-aware adaptive services, because classical models do not incorporate the context dimension. In CODA, requirements can be defined in terms of context unaware behavior and then refined by means of context-dependent adaptations at certain variation points. The main problem is that they do not take into account a well-defined context model that can be exploited to elicit the context-aware requirements. Moreover CODA lacks a computation level supporting the execution phase; just a decision table mechanism is proposed to support the adaptation. Choi (Cho07) introduces a requirement analysis process based on a definition of context and a definition of context-aware service. The author proposes some extensions of UML notations which exploit the proposed definitions. The problem in this approach is the lack of mapping the requirements from the development to the execution phase of the software lifecycle.

Different approaches (DVC$^+$07; Cho07) deal with the requirement elicitation problem for context-aware adaptive systems. They define requirement artifacts by means of first-class context entities but they do not support a formal specification phase. Other approaches (WSB$^+$09; SSLRM11) propose a formal representation of context-aware requirements by including a notion of uncertainty. Most of the approaches found in the literature only consider requirements specifications at design time while they do not propose a run-time support for eliciting and specifying requirements and for augmenting the system with new requirements.

We believe that in order to support the evolution of requirements it is necessary to perform a consistency checking process which validates the requirements specifications enhanced at run-time. To this end, managing requirement entities at run-time is essential for adaptive systems.

2.4.2 Design for adaptive system

The most classical software systems deal with context-aware adaptations using the context information in the same way they deal with any other kind of input data. Therefore simple "if-then" rules are employed and according to the actual value of the context variables, different actions can be executed. Even if this approach is easy to program, it lacks code maintainability, code extensibility and code re-usability. In this section we will investigate the main issues arising from high level and low level design for adaptive systems.

2.4.3 High level design

From the high level design point of view, architectural patterns describe particular recurring design problems and show concepts for their solutions. Costa et al. (CPvS05) have presented three main architectural patterns for context-aware service platforms: Event-Control-Action pattern, Context Source and Managers Hierarchy pattern, and the Action pattern.

The Event-Control-Action pattern decouples the context sensing activity from the reaction management. Through condition rules it is possible to manage the adaptation policies defined over the different behavioral descriptions. The class diagram for the Event-Control-Action pattern is depicted in Figure 1. The Context Processor unit shown in the

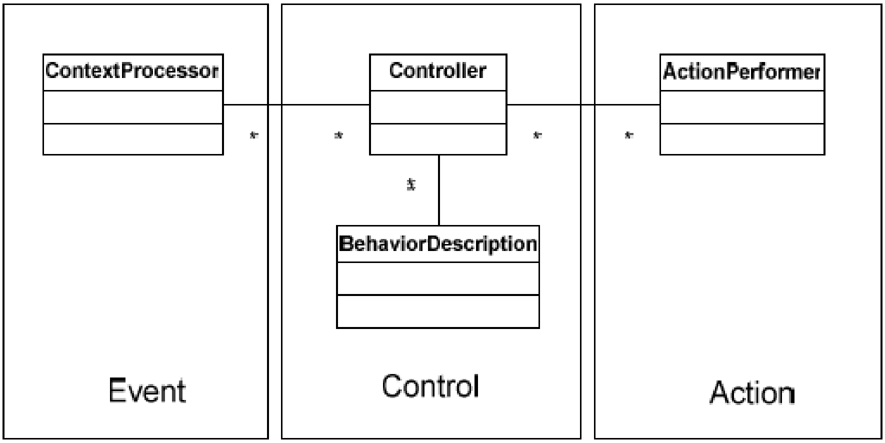


Figure 1: Event-Control-Action pattern structure

Figure is dependent from the context model; its goal is to query the context model and to contact the Control unit to notify the occurred context changes. The Control unit contains the alternative behavioral descriptions of the application upon which the adaptation rules are defined. The

Action Performer is activated based on these rules in order to trigger actions (for instance if the system is a service platform this means to invoke a service). The main benefits for this pattern is the distribution of sensing, controlling and performing activities that leads to a distribution of responsibility among different business parties. Moreover, this architectural design leads to extensibility and flexibility due to the possibility to add/delete Event or Action components.

The Context Source and Managers Hierarchy pattern provides a hierarchical structure for context processing components (Figure 2). The outcome of a context processing unit becomes the input for a higher level unit in the hierarchy. Each Context Manager inherits the features from the source in order to gather context information from various Context Source or other Context Manager. This pattern enables the distribution of

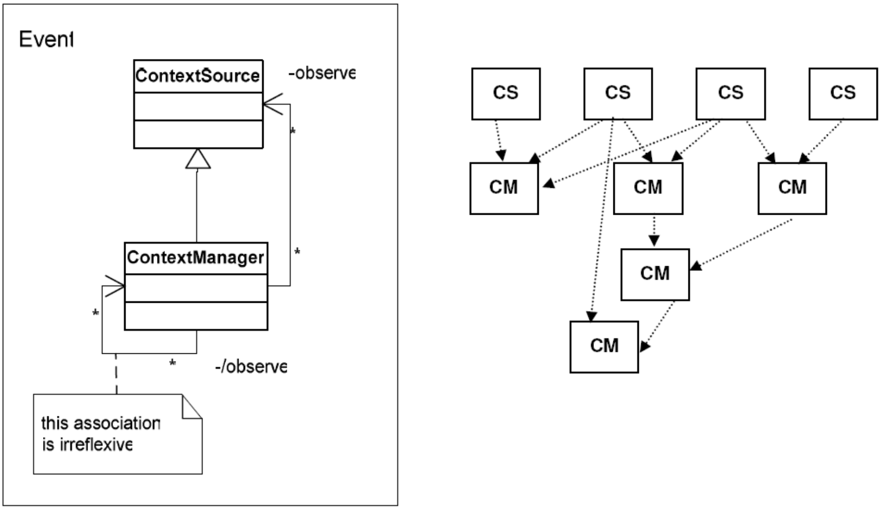
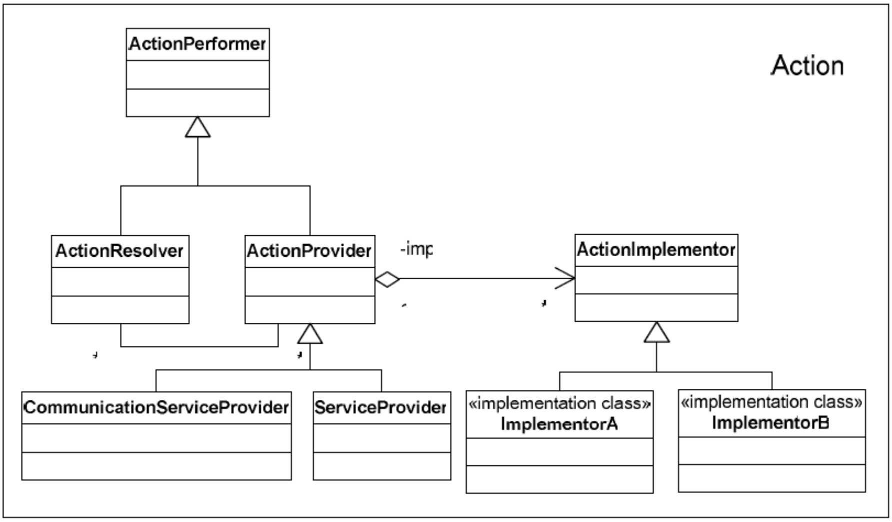


Figure 2: Context Source and Managers Hierarchy pattern structure

sensing, aggregating, inferring and predicting activities. Context information can be easily filtered along the path hierarchy in order to delete non-relevant information and to reduce their overhead.

The Action pattern enables the action triggering and it decouples the action purpose from the actual action implementation. As shown in Figure 3 the Action pattern supports an abstract action definition along with its implementation and it provides means to coordinate the composition of actions. This is useful in order to manage at run-time the selection among different implementations of actions.



Action
ActionPerformer
ActionResolver
ActionProvider
-imp
ActionImplementor
CommunicationServiceProvider
ServiceProvider
«implementation class» ImplementorA
«implementation class» ImplementorB

Figure 3: Action pattern structure

A more basic work dealing with high level design issues is presented by Winograd (Win01). The author presents the main architectural style to support the middleware layer. The three main coordination models are widget based model, blackboard model and network service model. The former is based upon an interface mechanism to directly access the context data. The blackboard approach follows the publish and subscribe pattern (CMM09) and is implemented via tuple spaces (CLZ98). The network service approach exploits discovery techniques to find context information using a client server approach.

2.4.4 Low level design

From the low level design point of view, suitable programming models are essential to manage the complexity and the effort for implementing adaptive applications. Most of applications do not make any use of programming toolkit or infrastructure dealing with related context-aware aspects. Nowadays, developers directly hard-wire the logic of adaptation to the context within the source code.

At implementation level Hirschfeld et al. (HCN08) 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.

The Chameleon framework (ABI10) provides a context-aware programming model to develop adaptable Java applications. The adaptation behaviors are implemented through different extensions to the Java programming model, that are adaptable methods and adaptable classes. The decisions among the different adaptations are taken during the deployment phase based on the actual context. The model characterizing the context enables to express the current status of the available resources. The Chameleon framework lacks a complex context model that enables reasoning upon the resource relations and hierarchy that can be exploited to make the context programming easier.

Two programming models based on more complex context models have been presented by Henricksen and Indulska (HI06). Their work is mainly based on the situation and preference abstractions. The former is a way to define conditions on the context in term of "fact abstraction" that represents an high level view of the context. The latter enables to manage the user preferences in order to set the user context-aware requirements. To each user preference is associated a rate used during the evaluation of the preferences to decide the more suitable adaptation in a certain context. The first programming model shown in (HI06) is a classical event-driven programming style called "triggering". Whenever a situation changing occurs an event is triggered with some lifetime condition. The second model presented within the same work is the ″branching″ model: it offers a novel and flexible means to insert context and preference decision points into the application logic flow. Through the branching model a set of preferences are evaluated at run-time to select which program branch to visit to better fulfill the user requirement in a certain context situation. A bunch of APIs are provided to the programmer in order to dynamically choose the right adaptation based on the context evaluation and user preferences.

The context-oriented programming model proposed in (KR03) is developed implementing an extension of Python. The context dependent behaviors are kept in a stub repository separated from the running code. The source code contains "open terms" that are the gaps to be filled at run-time. The procedure of "context filling" enables the selection of the appropriate stub from the repository in order to fill the gap of the source code. The filling procedure depends on the goal to achieve and on the context. A set of adaptation stubs are defined in order to reach the goal for any possible arising context. This approach requires an a priori global vision of the context. Moreover, it lacks suitable models to represent the context situations and the goals to achieve within the context. Context is modeled by simple XML tags checked during the adaptation phase. Context modeling is provided without the possibility to define relations upon the context elements.

Context information could be required in different parts of the software and its handling could be seen as a concern that spans across several software units. Aspect Oriented Programming (AOP) is a programming paradigm that handles cross-cutting concerns. It is implemented extending Java and C++ languages. It introduces jointpoints and pointcuts. The formers are the points inside the program where aspects can produce additional behaviors, whereas pointcuts define expressions to detect jointpoints and code fragments to be applied on jointpoints. Aspects enable the application of the same code fragments at different points. Tanter et al. (TGDB06) propose context-aware aspects in order to use the context to drive the use of aspects. The aspects are invoked based on the actual context information which is modeled as first class entities. Dynamic AOP extends the original notion of AOP by allowing weaving at load or run time. Dynamic AOP has been shown to be a very suitable mechanism for run-time adaptation of applications and services. In (VBAM09) the authors propose a system offering dynamic AOP in Java based on AspectJ. The approach proposes a system offering dynamic AOP in Java based on AspectJ. It supports a wide range of standard AspectJ constructs for dynamic cross-cutting, is portable, provides complete method coverage and is compatible with standard JVMs.

Mixin is a programming style where units of functionality are created in a class and then mixed in with other classes. Two models which exploit mixin style are the language Scala and the Fractal component model [1]. Mixins can be used to construct new classes by combining functionalities defined in other classes thus supporting run-time code variations. Nevertheless mixins approaches require programmers to explicitly and a priori specify what are the classes to be combined.

2.4.5 Context-aware middleware

The activity on context-aware computing has began during the second half of 1980s. Nowadays, many researchers have made efforts to design and implement network, user infrastructure and middleware to provide context-aware services to the users. A very in depth analysis at each of these levels was done and a lot of resulting applications have already been presented in different domains. The main architectural style guiding the diffusion of context-aware system is the middleware approach. Many such approaches have been presented and related in order to get a common vision and architecture.

Henricksen et al. (HIMB05) provide an overview of the state of the art for context-aware middleware. They compare five different middlewares in terms of requirements for traditional distributed systems such as scalability and tolerance to components failure. Moreover, they consider the issues related to context-aware applications development such

[1]http://www.scala-lang.org/, http://fractal.ow2.org/

as privacy preferences. They restrict themselves to looking at general systems which spans multiple layers of a proposed layered architecture. It consists of application components at the top layer, decision support tools, context repositories, context processing components, and context sensors and actuators at the bottom layer.

Kjær (Kjæ07) proposes a taxonomy for categorizing some relevant properties expressing the capabilities arising from the analyzed context-aware middlewares. Among them the first concept shown is the environment; some middlewares assume that devices can communicate without relying on external services whereas some other middlewares rely on a service providing infrastructure. Another important capability is the reflection mechanism, that is usually available through meta-data in order to reason about application, middleware and context information. Reification and absorption processes allow to keep updated the change in the metadata with the described related entity. Another important issue for middleware is the composition among entities in order to adapt to different contexts. For this reason even adaptation should be considered from the middleware point of view with three different categories:

- transparent: middleware reacts to changes without the application being aware of it
- profile: middleware adapts in order to satisfy the application requested profile
- rules: middleware reacts based on the rules explicitly defined by the user or application.

The authors also explain Quality capability that could refer either to the classical Quality of Service provided by the middleware (QoS) or to the Quality of Context information (QoC). The latter could be expressed in terms of parameters expressing the quality of the context elements with respect to the application. The QoC is an important issue because with an open-world scenario, context providers and context consumers should reach a kind of agreement over the quality of the information provided (BKS03). Finally the taxonomy of capabilities provided in (Kjæ07) could be useful either to select a middleware or to start the development of a new one.

An example of transparent middleware adaptation is depicted by Cheung et al. (CCYC06) based on a previous work proposed by Cao et al. (CXC$^+$05). Here a fuzzy based service adaptation model is presented. The policy of each service is selected based on a fitness function that assesses the proximity between each best context for each policy with respect to the actual context situation. The adaptation is therefore driven by the context and each user preference is considered.

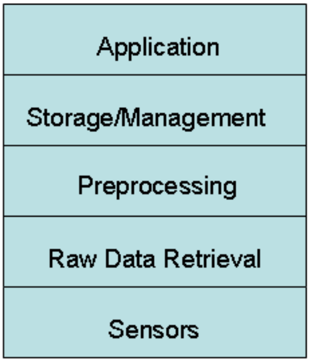


Figure 4: Unifying middleware archietecture

Baldauf, Dustdar, and Rosenberg (BDR07) proposed another approach to compare different context-aware middlewares. They defined a common architecture that can be considered suitable for each of the middleware analyzed through their paper (see Figure 4). The lower layer shown in the figure is composed of three kinds of sensors: physical, virtual and logical sensors. The physical sensors are hardware sources of data like microphones, thermometers, biosensors whereas the virtual ones gather context data from applications or services like emails, travel booking system, electronic calendar. Logical sensors can combine data coming from physical and virtual sensors in order to solve higher level tasks. The upper layers use the raw data derived from the sensors in order to expose interfaces. Those methods are exploited for aggregation and composition purposes to get context data at higher level. For instance an application could not be interested to know the exact position of a person but could find useful to know the name of the room. Moreover, if a certain number of people is situated at the room where a presentation is running, then other context information can be inferred. At this level all the sensing conflicts should be solved using time stamp and resolution information and even QoC reasoning could be exploited to reach this goal. The application level contains the reaction logic that can be implemented exploiting the structured context information gathered from the lower levels. The communication mechanism to get this data is the synchronous callback design pattern; indeed, an asynchronous communication would be difficult to maintain because of the rapid change of the context information.

2.5 Assurance

Techniques for high-assurance have to provide evidence that the system satisfies either functional or non-functional properties during operation. To this end, verification and validation techniques rely on descriptions of software models and properties. For traditional (non-adaptive) software systems it is enough either to verify or to validate the system at design time since system's goal and underlying requirements are fixed. For adaptive systems there could be a variation of the system's goal or requirements at run-time since there exist unpredictable contexts that cannot be anticipated at design time. Therefore, it is necessary to support a notion of assurance which considers system models ranging from the problem to the solution space. There exist inconsistencies that can be discovered by considering requirements artifacts whereas other inconsistencies can be discovered only by considering code artifacts. To this end it is necessary to maintain the code casually connected to requirements models.

Novel verification and validation techniques have to be defined to provide high-assurance for adaptive systems. One of the key challenges for these new techniques is the existence of algorithms that do not require high space/time complexity since they have to be performed at run-time.

In the literature different approaches that provide design time assurances are found. Alférez et al. (AMK$^+$09) propose a methodology to discover misbehaving requirements by exploiting system models at different abstraction levels. Here interactions are discovered only at the requirements level without considering the implemented system. Classen et al. (CHSL11) offer a model checking algorithm for validating the space of the system alternatives at design time. The algorithm only verifies properties on an abstract model neglecting the actual system implementation. The approaches (AMK$^+$09; CHSL11) only consider design time assurance activities on high level system abstractions. Adaptive systems should evolve while maintaining an high-assurance property at run-time. Cheng et al. (CdLG$^+$09) define the characteristics for a generic framework which aim to support high-assurance for adaptive systems. They claim that design time activities should be supplemented by agile run-time assurance activities that should be performed upon the identification of evolving requirements. Nevertheless, they provide only a theoretical framework which describes a set of guidelines and best practices that should be adopted. Filieri et al. (FGT11) describe a practical technique to perform run-time model checking on probabilistic system model. The authors show how to provide high-assurance at run-time on evolving models. Nevertheless, they do not take into account actual system code in their definition of assurance.

In the literature there is a lack of practical solutions to support assurance checks at run-time by considering both abstract system models and code artifacts. Models can play a key role for developing adaptive applications since they are able to support the consistent evolution required by context variations. Different models are required to achieve a consistent evolution:

- a model for representing the system and its variability;
- models to represent the context surrounding the system;
- requirement engineering models;

- models representing executable artifacts.

All mentioned models should be exploited and managed at run-time when the development of the system is still required (BBF09; SBW$^+$10).

2.6 Framework for adaptive systems

Several frameworks address the problem of achieving system evolution at different granularity levels. They exploits models at different granularity such as context-aware requirements models, context-aware architectural models and context-aware implementation models.

The Rainbow framework (GCH$^+$04) enables architecture-based self-adaptation for software systems. It proposes adaptation rules to specify how to reconfigure system components whenever certain situations arise. System components are reconfigured based on decision taken at design time whereas un-anticipated adaptations can not be achieved. The framework supports non-functional reconfigurations while it does not consider the consistency checking of the evolution. The context is not explicitly modeled but only simple variables are considered in the framework. The PLASTIC approach (ABI09) applies reconfiguration strategies at implementation level by exploiting an explicit definition of context. The approach supports non-functional reconfigurations to statically defined Java artifacts driven by context variations. The framework only deals with foreseen evolution while run-time evolution is not allowed.

A common aspect for the above mentioned frameworks (GCH$^+$04; ABI09) 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 un-anticipated user need variations arising at run-time.

Qureshi and Perini (QP10) have defined a framework for requirement engineering to distinguish activities at design time from activities at run-time. They have provided a mechanism to evolve the requirement specification at run-time driven by the user thus supporting a notion of run-time evolution. Nevertheless the proposed method does not consider a definition of consistency. The Javeleon framework (GJ09) as well as the JavAdaptor framework (PGS$^+$11) aim to support the run-time evolution by means of transparent dynamic updates of running Java applications. Developers can simply modify their applications at run-time and they can trigger an on-line update without stopping the running application. Javeleon and JavAdaptor do not support a definition of context for the evolution but the developers is directly in charge of injecting new behaviors in the application at run-time. In addition they do not provide a process for checking the consistency of the evolutions.

Kramer and Magee (KM07) have presented a three-layered conceptual model to support run-time and architectural reconfigurations of self-managing systems. They consider a Component layer, a Change management layer and a Goal management layer. The Goal layer identifies the plan to execute 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, non-functional and run-time evolutions there is no definition of context and a definition of consistency check for the composition of components is still missing.

The approaches presented so far do not provide a process to asses the consistency of foreseen and unforeseen evolutions. Ali et al. (ADG10) have proposed a goal-based framework to enact the evolution among system variants at requirement level. This approach provides a context analysis phase to discard variants which are inconsistent based on the context predicates. This phase checks the consistency for alternative behaviors based on a context-dependent goal model. The problem in this approach is that it can not support run-time evolutions to the goal model.

To the best of our knowledge the frameworks presented in the literature only apply reconfigurations at specific granularity levels, either at requirements models, or at architectural models or at implementation models. Most of the frameworks are not based on an explicit context model to support system evolution in all phases of the software lifecycle process. Only a few of them supports the evolution which is required at run-time whereas there is almost no support to check the consistency of the evolution. In order to provide high-assurance for context-aware adaptive applications it is necessary to support a definition of consistency as proposed by Zowghi and Gervasi (ZG02). They suggest that an effective support to consistency is based on system models at the different granularity levels, ranging from the problem space models to the solution space models. We claim that adaptive applications are not developed and evolved following a software process which considers all these models together thus making difficult to effectively support the consistency of the evolution.

2.7 Support to foreseen evolutions

Supporting system evolutions in face of unforeseen context variation is the most difficult problem which will be discussed in this thesis. Nevertheless it is still very interesting to solve the problem of achieving resilient reconfigurations in presence of foreseen context variations. Reconfigurations should meet quality requirements according to user preferences and they should be performed at a reasonable cost and in a timely manner. Adaptive systems need automatic (as much as possible) reconfigurations taking into account the execution environment and the user preferences. The first determines the admissibility for the reconfiguration whereas the second determines the goodness for the reconfiguration. These pieces of information are not fixed since they are strictly context-dependent thus making difficult to achieve reconfigurations that are resilient to these changes. These changes are not predictable, thus it is not in general possible to determine how the context may affect the variations to user preferences and to the availability of resources. To this end predictive approaches can be adopted in order to explicitly consider this uncertainty thus making possible to achieve reconfigurations decisions with incomplete information.

In the literature there are a number of decision-making mechanisms exploiting user preference and context information to support the adaptation. Sykes et al. (SHMK10) evaluate the utility of each system component primarily by the user preferences upon each non functional property. Then the overall utility degree for each system variant is obtained as the average of each component utility. The authors define the space of adaptation strategies without considering the environment condition explicitly. The PLASTIC approach (ABI09) considers how to exploit user preferences in performing service based adaptation. The approach performs a non-functional selection among the system variants that can be deployed in the current execution environment based on the required resources. The approach proposes a resource model to support the definition of eligible system variants. However no predictive information is included to drive their adaptation process. In the field of service discovery, Li et al. (LRT^{+}10) exploit a user preference model to support the service recommendation to the user. At run-time, services are checked with respect to their precondition and then they are ranked based on the user preferences upon their possible outcomes. The approach considers only a simple context model without considering future changes. Dorn and Dustdar (DD10a) observe the behavior of multiple users to adapt the available software capabilities (i.e. features) to the preferences of the whole group. Their approach, however, does not take into account context constraints, neither do they apply predictive knowledge on potential future context changes.

All the mentioned approaches neither consider predictive information about the context resources nor about the user preferences. Adaptations are performed only by exploiting information on the current context.

Cheng et al. (CPGS09) extend the Rainbow evolution framework (GCH^{+}04) in order to exploit the predictive availability of context resources to enable the adaptations. However they lack the notion of user preferences. Poladian et al. (PGS^{+}07; PSGS04) face the problem of selecting a sequence of system variants for a predefined sequence of fixed time slots, each of which is characterized by a prediction of resource availabilities. The sequence which best fits the fixed user preferences at each time slot is selected. Also a factor of cost is introduced in order to give an increased utility to components which are already running.

To the best of our knowledge there are no approaches that support system adaptation by considering run-time user preference changes, context changes and cost factors coherently in one formal framework. We claim that considering all these factors together promotes better performance of the evolution process.

Chapter 3

Approach

3.1 Taxonomy of the evolution

We have identified two different types of evolutions each one addressing a different dimension of context variability. On the 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 possibly 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 can be defined statically and the foreseeable contexts can be 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 takes part to the evolution feedback loop.

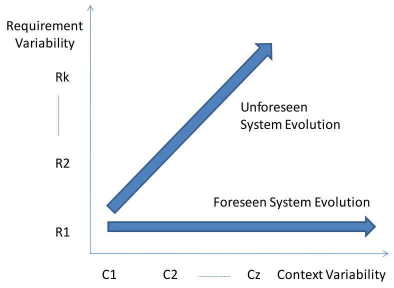


Figure 5: System Evolutions

Figure 5 shows how the context affects both system evolutions. In the foreseen evolution the system evolves to keep satisfied a fixed set of requirements by switching among different alternative behaviors provided at design time for different known contexts. In the unforeseen evolution the system evolves by switching to a new alternative behavior that includes a new functionality necessary to satisfy the emerging requirement. In our approach we only consider how to augment the system with new requirements at run-time whereas we do not take into account the deletion of requirements already implemented. Indeed, we believe that augmenting functionalities at run-time is the main challenge in supporting evolutions suitable for unforeseen contexts.

3.2 Software Product Line Engineering perspective

We represent the system following the Software Product Line Engineering (SPLE) perspective since it breaks the system complexity in feature components thus reducing the impact that any change may have on the system (KK98a). SPLE perspective provides a uniform abstraction to all the development approaches that consider a system made out of a combination of basic software entities, like for example the Component Off

The Shelf (COTS) approach or the service-oriented one. In addition the SPLE perspective already provides models to manage the system and to support consistent evolutions. Our intention is to take advantage of the methodologies proposed in SPLE to support consistent evolutions for context-aware adaptive systems.

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 (KK98b; AMK$^+$09; BC04a; PCBD10). We will build our notion of consistent evolution on the feature interaction phenomenon.

The work (CHS$^+$08a) has already shown common research questions between SPLE and adaptive system and the necessity to dynamically manage features at run-time. Most recently, Dynamic Software Product lines (DSPL) have been presented as a new direction in SPL engineering field to deal with software capable of adapting to changing user needs and evolving context environment at different binding time (HHP$^+$08; PBD09; PBCD11). However in the DSPL field dealing with evolutions arising at runtime is still an open issue.

3.3 Case Studies

In this section we propose two motivating scenarios in order to give some examples of our approach. We have developed two different adaptive systems. The first is an eHealth application to support the doctors' activities while the second is an application to visualize a Mandelbrot fractal.

3.3.1 eHealth application

Our application for the medical domain supports doctor's activity in showing the vital parameter of the patients. A remote monitoring system, situated at the patient's home, gathers oxygenation rate and heart rate data through two different probes. Probes sense patient's information and transmit them to the hospital through a home gateway. A server collects these information and make them available. Finally, doctors using their mobile or desktop devices visualize elaborated graphics or numerical data for oxygenation and heart rate parameters. Figure 6 describes the basic elements for the e-Health system.

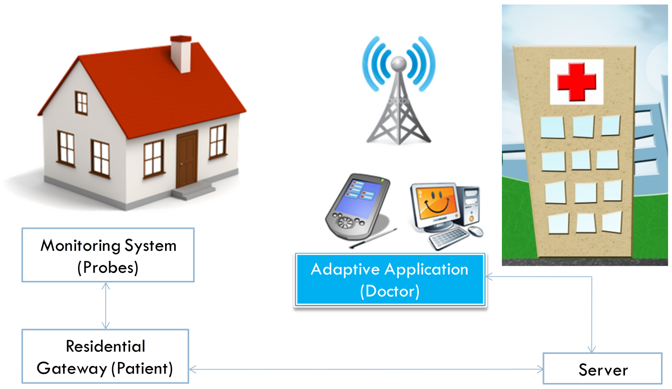

Monitoring System (Probes)
Adaptive Application (Doctor)
Residential Gateway (Patient)
Server

Figure 6: E-Health architecture

We focus our attention on the adaptive application that performs the visualization. The screen visualization should follow different alternatives each one suited for a certain context since it could require different hardware resources. For instance a first variant may visualize only one parameter as text and graph, a second variant may visualize both parameters as text, while a third variant may visualize graphically non real-time data memorized in the doctor's device. In addition, in order to view real-time data it is necessary to run the application through a device connected to the network. Whenever this is not possible the most recent memorized graph is shown to the doctor. We model this mobile application through subsets of features that are combined in order to have alternative variants.

While the system is running, it could be the case that a new unforeseen sensor for monitoring the respiratory rate is added to the remote monitoring system by the patient. Hence, the doctor may require a new monitoring activity to visualize the new patient data. In order to prevent the application from non-consistent behavior we should provide an automatic process to augment the system with the new feature. This process should support the unforeseen evolution that is required at run-time as a consequence of a new requirement identified by the user.

3.3.2 Mandelbrot fractal

This application elaborates a Mandelbrot fractal (Man82) that better fits the characteristics of the mobile device (CPU, memory, number of display colors,...). The application requirements consist in visualizing a fractal image to the user through the device screen. The higher level of context resources is available, the more beautiful will be the fractal image shown to the user. The Fractal context-aware adaptive system will be modeled through a set of features that represent the basic alternative behaviors to color, build and view the fractal image.

At run-time features may need to be activated or deactivated based on the context-resource availability changes. In addition because of environment unpredictability, the user may want to introduce new unforeseen behaviors as the system operates in an unforeseen context. For example whenever the software device characteristics makes the visualization of the Fractal image format impossible because the format in unknown, the user may guide the introduction of a new software plug-in to decode that specific format.

3.4 Context model

Our context model entails context entities as key-value pairs and it is defined using two perspectives: the *context model structure* and the *context model space*. The *context model structure* expresses context entites in term of types and categories. We adopt the taxonomy where each element belongs either to the system, the user or to the physical environment. In addition we consider the entity types enumerate, boolean and natural as in (ABIM08; ABI12). The *context model space* expresses the variability for the assignments of the context entities. Each context entities is identified through a tag $ContextId$ and it can assume one among its admissible values contained in $dom(ContextId)$. The context model space for the context entities $ContextId_1, ..., ContextIdId_n$ is defined as the Cartesian product:

$$S = \bigotimes dom(ContextId_i) \quad s.t. \quad i = 1, ..., n \tag{3.1}$$

Each valid assignment of entities $\overrightarrow{c} \in S$ will be considered as a different context state.

The software engineer defines the context at design time whereas an automatic process may update it at run-time. The model extension is caused either by new unforeseen resources that appear in the environment or by new requirements that may refer to new context entities.

Example (eHealth application)

For the eHealth application we have identified a set of user context entities, a set of physical context entities and a set of system context entities. $heartRateProbe$ and $oxygenationProbe$ are the user context entities which represent the two sensors to sense the medical information from the body of the patient. They are boolean entities which expresses if it is possible to retrieve the heart rate and oxygenation data. The system context entity mem expresses the level of free memory available for the adaptive application whereas $cRate$ identifies the CPU speed offered by the device. The entities $conn$ and $netB$ are physical context entities which define if a connection is available and the corresponding available bandwidth. Figure 7(a) depicts the context model for the eHealth application.

Example (Mandelbrot fractal application)

The context entities for the fractal application belong to the system and to the physical environment. The $conn$ physical context entity expresses the availability of an Internet connection. The system context entity sc identifies the number of colors for the device whereas mem and $cRate$ express the available memory and CPU speed respectively (Figure 7(b)).

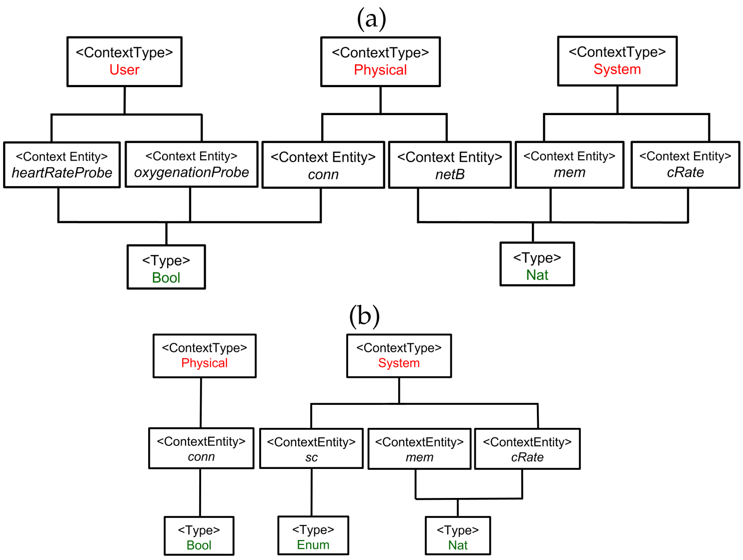


Figure 7: Context model (a) eHealth application (b) Mandelbrot fractal application

3.5 Requirements taxonomy

Our evolution framework is based on a requirements model that has to be manageable at run-time. For our requirements specifications we adopt the notion of requirements inspired by the taxonomy proposed by Glinz in (Gli07). The definition of requirements is based on the concept of *concern* that is a matter of interest in a system. Each kind of requirement is defined based on the correspondent kind of concern to which it pertains. Functional requirements pertain to functional concerns, performance requirements pertain to performance concerns and specific quality requirements pertain to quality concerns. In the same taxonomy, constraints are defined as requirements that restrict the solution space for meeting functional, performance and specific quality requirements. In our approach we consider as crucial the subset of constraint requirements which are expressed in terms of context entities. We will name these requirements with the term *context requirements*. They will be generated either by analyzing the resources required by the implementation or by extracting the context entities from functional, quality or performance requirements. For the first option, the implementation may be analyzed by means of different methodologies such as workbench analysis or by static analysis like proposed in various approaches, in particular (ABI09; ABI12). This approach is a possible solution to evaluate the consumption of resources caused by the code. For the second option, quality, functional and performance requirements have to be analyzed to extract the portion that refers to the context entities. These entities have become first class entities within requirements specifications as devised by different approaches to the elicitation of context-aware requirements, e.g. (DVC$^+$07; Cho07).

3.5.1 Context requirements

We define context requirements as predicates defined over the context entities which belong to the context model. These entities are beyond the system's control but they may influence the system execution. In the following we depict the grammar that we use to express context requirements starting from the context entities:

```
<C>::=<ContextEntity><rel−op><value> |<C><log−op><C>
<rel−op>::=≥ | ≤ | < | > | =
<log−op>::= AND|OR
<value>::=<natural>
```

Figure 8: Syntax of context requirements

Each expression generated by the grammar may be related to a single feature or to a system variant.

3.6 Adaptive application

We define a context-aware adaptive application in terms of sets of features each one implemented with a component and / or a service. Then we combine features in order to obtain different variants which express the variability of the application.

3.6.1 Feature

We define a feature as composed by a context-independent requirement, a context-dependent requirement, and an implementation part. 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.

The definition above is inspired by (CHS08b) which has been in turn inspired by the Problem Frame approach defined in (Jac00). Differently from these approaches we refer to C as the context requirement instead of the domain assumption. The system model we propose links requirements entities to the implementation artifacts; as a consequence evolutions of requirements may be easily mapped to the correspondent code artifacts.

Example (eHealth application)

In the e-Health application we define the feature to view the graphical oxygenation rate in terms of three components (Figure 9).

$R_{graphOx}$ entails two parts. The first part is a simple textual representation. The second part is defined in terms of context entities and operations in $I_{graphOx}$, it enables the traceability from $R_{graphOx}$ to $I_{graphOx}$ and from $R_{graphOx}$ to $C_{graphOx}$.

The context requirement $C_{graphOx}$ is derived from the feature implementation $I_{graphOx}$ and from the requirement $R_{graphOx}$. On the one hand, we may use the CHAMELEON framework (ABI12) to evaluate the consumption of memory and clock rate caused by $I_{graphOx}$. On the other hand by looking at the requirement $R_{graphOx}$ we extract the portion that

$R_{graphOx}$: If Oxygenation data are available Receive Oxygenation rate and View it on the graphical widget
If "oxygenationProbe" then (Each 10 times "getOximetryData" follows a "displayGraph")

$I_{graphOx}$:

```java
public class GraphOximetryViewer {
 XYDataset oximetryDataset = new XYSeriesCollection();
  ...
 public void viewGraphicalOximetry(Graph g){
   ...
  for(int  i  =  0;i<10;i++){
   XYDataItem dataOx = OximetryRetrieving.getOximetryData();
   dataVectOx.add(dataOx);
  }
  g.displayGraph(dataVectOx);
 } ... }
```

$C_{graphOx}$: $mem \geq 50\ Kb \wedge cRate \geq 1000\ Khz\ \wedge\ oxygenationProbe = true$

Figure 9: Feature $f_{graphOx} = (R_{graphOx}, I_{graphOx}, C_{graphOx})$

refers to each context entity. Since $R_{graphOx}$ refers to the context entity "oxygenationProbe", we insert it into the context requirements. Only if this variable is true, then oxygenation data are retrievable from the remote monitoring system.

Example (Mandelbrot fractal application)

Figure 10 shows a feature for the fractal application. This feature evaluates the fractal pixels and it shows them a pixel row at a time. For this feature we only describe the textual representation for the requirement component. This requirement entails a functional and a quality requirement; the functional requirement can be expressed as "Compute and visualize each fractal pixel", whereas the quality requirement can be expressed as "The image is visualized a pixel row at a time" which in terms of implementation can consist in assigning the value $Progressive$ to the quality property $DisplayModel$. Finally, the context requirement C_{genPro} only refers to the required amount of memory caused by I_{genPro}, which is the Java class implementing the feature.

R_{genPro} : Compute each fractal pixel and show it a pixel row at a time

I_{genPro} :
```
public class MandelCanvas extends Canvas{ ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas,1, height, false);
   repaint();
  } } ... }
```

C_{genPro} : $mem \geq 200$

Figure 10: Feature $f_{genPro} = (R_{genPro}, I_{genPro}, C_{genPro})$

3.6.2 System variant

A system variant represents a possible alternative behavior in terms of a set of functionalities that it can provide to the user. Software engineers have to assemble together one or more features in order to obtain a system variant. Starting from a set of feature F we define a *system variant* as the triple $G_F = (R_F, C_F, I_F)$. 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. Given two features $f_1 =<R_1, I_1, C_1,>$ and $f_2 =<R_2, I_2, C_2>$ their union is defined as: $f_1 \cup_f f_2 =< R_1 \cup_R R_2, I_1 \cup_I I_2, C_1 \cup_C C_2 >$. The software engineer combines the features at design time in order to obtain a set of alternative requirement specifications. At run-time the set of alternatives may be augmented considering an un-anticipated features that arise from a new user need.

In the following we show a possible example on how to create the context requirements and implementation components for a system variant starting from its features.

The union operator $\cup_C$ merges context requirements depending on the nature of resources. For example if we have two requirements demanding bandwidth for 20 kbps each one, their union will express a demand of bandwidth for 40 kbps. The approach presented in (ABI12; ABIM08) proposes a technique to create our context requirements.

For the implementation portion I the software engineer combines the code artifacts in order to have a single access point to the whole variant. Each variant is composed by a Java class for each single feature plus a Java class which is the entry point for the variant. This class entails the method $execute$ to trigger the execution of the variant.

3.6.3 Examples

Listing A.1 represents a possible system variant $G_{EHealth}$ for the eHealth application. This variant performs the visualization for the oxygenation rate through a graphic and a textual widget. As for a feature a variant is represented as a triple (R, I, C) where requirements R and context requirements C correspond to a Java implementation I. The context requirement expresses the combined request of resources as they appear in each feature. The implementation component contains a Java class for each feature and an additional class which represents the entry point for the variant. The class $OximetryRetrieving$ retrieves the oxygenation data from the remote monitoring system. The two classes $GraphOximetryViewer$ and $TextOximetryViewer$ exploit the retrieved information and perform the visualization trough a graphic and a textual widget. Finally class $VariantEHealth$ implements the logic for the whole variant.

Let us consider the fractal application and one of its possible variants $G_{Fractal} = \{f_{genPro}, f_{colB}\}$ implemented by the class diagram in Figure 11. In this example, each feature is implemented as a single Java class. The class $MandelCanvas$, which implements f_{genPro}, provides the interface $generateProgressiveFractal$ that generates and draws the fractal image progressively a row at a time exploiting the operation $drawFractalPixel$. The class $Colouring$, which implements f_{colB}, provides the interface $pixelColourAsBands$ and the operation $initColourAsBands$ in order to color the image with different bands of colors. The only class which does not correspond to any of the features is $VariantFractal$ which is the external interface to access the whole application variant. Listing A.2 shows the variant $G_{Fractal}$ in terms of its main components.

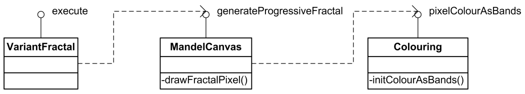


Figure 11: Example: class diagram (G_1)

3.6.4 Feature diagram

Evolving the system means switching from a combination of features to another combination of features. In order to support the evolution process, it is necessary to model the possible combinations of features each of which corresponds to a different system variant. The variability model that we have chosen is inspired by the *feature model* which has been first introduced in the Feature-Oriented Domain Analysis method (KCH^{+}90). Since then, feature modeling has been widely adopted by the SPL engineering community and a number of extensions have been proposed (SHTB07). In our approach we consider a possible abstract syntax for the feature model defined starting from nodes (features) and arcs between nodes:

- The root node of the model is the label which stands for the system.
- Each node expresses a feature which can be either optional or mandatory.
- Each edge between two nodes expresses a decomposition relation (consist-of) between the parent node and the child node. It enables the possibility to add behavior to the parent feature. We consider two decomposition relations: AND decomposition and XOR decomposition.

- ″Requires″ constraint is a directed relation between two features. If one feature is present in the variant the second has to be present as well.
- ″Mutex″ constraint enables the mutual exclusion between two features; therefore they cannot be in the system variant simultaneously.

Starting from the feature model (abstract syntax), the feature diagram (concrete syntax) is commonly expressed as a tree structure. We adopt a subset of the syntax presented in (CE00). In this diagram, features are represented in a tree-like format. Dark circles represent mandatory features, while white circles represent optional features. An inverted arc among multiple arcs expresses a XOR decomposition meaning that exactly one feature can be selected. Multiple arcs that start from a parent node express an AND decomposition.

Starting from the feature diagram, the set of possible system variants is obtained by combining the features in subsets compliant to the diagram. The diagram shown in Figure 12 concerns our adaptive Mandelbrot fractal and contains 8 features which give rise to 10 system variants. Each of them contains only one feature to generate the image and only one feature to color it. An admissible variant contains the features to download a predefined image from a remote server.

3.7 Formalization of evolution and execution

We propose a set of semantic rules in order to model how the evolution and execution steps can be performed by the adaptive application. At a certain point the application provides a set of features, i.e., a variant $G_i = (R_{G_i}, C_{G_i}, I_{G_i})$. Figure 13 depicts how the system I_{G_i} may change its state by performing either an execution or a evolution step. The execution step is represented with a vertical arrow whereas the evolution is represented with an horizontal arrow. In case of an evolution step the system is reconfigured by changing the set of features that are provided to the user; the application switches from one variant to another.

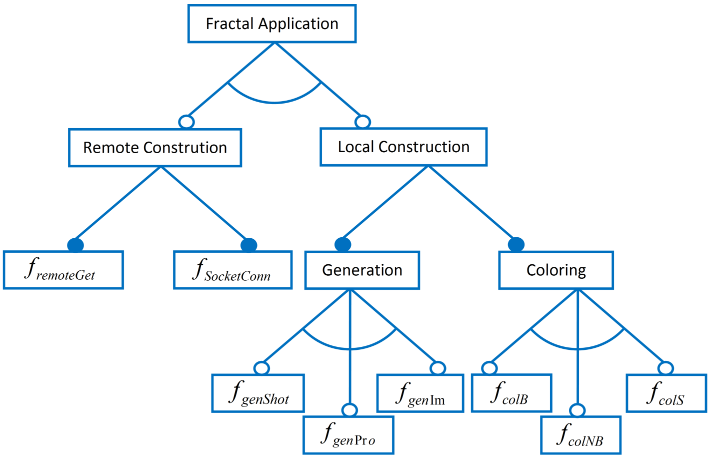


Figure 12: Feature diagram

We have identified two evolutions: in case of the foreseen evolution the application adopts a variant that has been statically checked whereas in case of an unforeseen evolution the application adopts a new unforeseen variant that has to be checked at run-time. Finally, in case of an execution step the system performs internal actions while maintaining the same configuration of features.

In order to take into account both standard and evolving behaviors we consider the state of the system to be composed by an internal and external component. The first is completely managed by the implementation, whereas the second is driven by the foreseen and unforeseen context variations. The state is defined as $\sigma = (\sigma_s, \sigma_c, \sigma_e)$ where:

- σ_s is the portion of the internal state managed by I which is not affecting any of the evolution scenarios.
- σ_c is the portion of the external state which addresses the foreseen evolution. It represents the current state for the context variables.
- σ_e is the portion of the external state which addresses the unforeseen evolution. It may contain either a new requirement arising

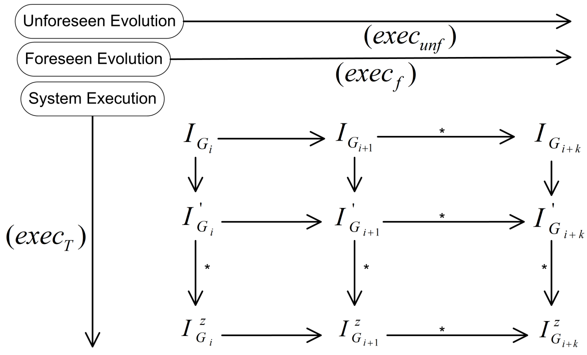


Figure 13: System state-based model

from the user $<R_{New}, +>$ or a requirement to delete $<R_{Del}, ->$.

We assume the existence of a monitor that is able to check the external portion of the state. Therefore, we simply define a function *monitor* that returns a pair $<Boolean, State>$ where the first element assumes value *true* only if the monitor can attest the variation of the state. The second element expresses the changed state.

We also assume the existence of the procedure *BestRanked* which evaluates the best variant that should be adopted by the application.

3.7.1 Semantic rules

Whenever an evolution is not required (either by the context or by the user) the system may perform an execution step changing the internal portion of the state (σ_s). A step performed following this rule consists of executing a code instruction which affects some local variables. In the following is depicted the rule describing an execution step:

$$\frac{monitor(\sigma_c) = <false, null> \quad monitor(\sigma_e) = <false, null>}{\langle I_{G_i}, < \sigma_s, \sigma_c, \sigma_e > \rangle \rightarrow_{exec_T} \langle I'_{G_i}, < \sigma'_s, \sigma_c, \sigma_e > \rangle}$$

In order to define how the system switches among variants we define two different rules: one for the foreseen evolution and one for the unforeseen evolution. In the following we depict the rule for the foreseen evolution:

$$\frac{\begin{array}{c} monitor(\sigma_c) = <true, \sigma_c\prime> \\ Space = \{G_r | r = 1, ..., n\} \quad BestRanked(\sigma_c', Space, Prefs) = G_j \end{array}}{\langle\ I_{G_i}, <\sigma_s, \sigma_c, \sigma_e>\ \rangle \rightarrow_{exec_f} \langle\ I_{G_j}, <\sigma_s, \sigma_c', \sigma_e>\ \rangle}$$

If the current assignment of context variables changes (σ_c'), we evaluate which is the best eligible variant through the function $BestRanked$. This function takes as input the space of possible variants, the current context values and the context-based user preferences. If this function returns a variant (G_j) different from the current one, the application switches from I_{G_i} to I_{G_j}. Following this rule, the system can only switch between alternatives which have been provided by the designer at design time.

We propose the unforeseen evolution in order to augment the application with new variant at run-time. The rule describing this evolution is enacted by the variation to the requirements set to satisfy. The user may specify a new requirement $<R_{new}, +>$ in the external portion of state σ_e. Thus it is necessary to check if a new variant has to be added to the application. In the following we show the rule for the unforeseen evolution:

$$\frac{\begin{array}{ll} monitor(\sigma_e) = <true, \sigma_e'> & <R_{New}, +> \in \sigma_e' \\ G_r = (R_{G_r}, C_{G_r}, I_{G_r}), r = 1, .., n & I_{G_r} \nvdash R_{New} \cup_R R_{G_i} \\ \multicolumn{2}{c}{Search(R_{New}) = f \quad G_j = G_i \cup_f f \quad Check(G_j) = true} \end{array}}{\langle\ I_{G_i}, <\sigma_s, \sigma_c, \sigma_e>\ \rangle \rightarrow_{exec_{unf}} \langle\ I_{G_j}, <\sigma_s, \sigma_c, \varnothing>\ \rangle}$$

Once the monitor has attested the variation to the requirements set to satisfy, the unforeseen evolution has to check if the variation to the requirements can be satisfied with the variants that have been provided at design time. Thus it is necessary to check if there exist at least a variant in the space of the known variants whose implementation I_{G_r} satisfies the requirements of the current variant R_{G_i} plus the new requirement R_{New}. Only if this is not the case, we proceed by evolving the application with a new variant. We query the search engine to retrieve the feature that implements the new requirement. We choose the first feature that does not give rise to the feature interaction phenomenon with the current variant G_i (see Section 3.8). Finally, the system evolves from the current variant towards the new one $G_j = G_i \cup_f f$.

If there exist already a variant that implements the new requirement R_{New} and the initial requirements R_{G_i}, then the application has to adopt this variant. We do not show the rule to perform this evolution since it is similar to the rule of the foreseen evolution except for the fact that it is triggered by a new requirement.

3.8 Evolution consistency

We adopt the absence of feature interaction as the notion of *consistent* evolution for a system variant. A certain variant is consistent if its features can be jointly executed without showing interactions. In our approach we take into consideration interactions that are not explicitly observable by the feature diagram, i.e. *implicit* feature interactions (PCBD10). Given a certain variant $G_F = (R_F, C_F, I_F)$ we formalize our notion of consistency following the approaches in (CHS08b; Jac00) as:

$$I_F, C_F \vdash R_F \tag{3.2}$$

In our approach this formula entails three different checks:

(i) $(C_F)[\overrightarrow{c}/\overrightarrow{x}]$: this formula checks the joint context requirement (predicate) C_F assigning the current context values $\overrightarrow{c}$ to the formal parameters $\overrightarrow{x}$, if the predicate is true the evolution can occur in the "adequate" context state;

(ii) $R_F\ is\ Satisfiable$: this formula checks if the joint context-independent requirement can be satisfied;

(iii) $I_F \vdash R_F$: this formula validates the joint implementation with respect to the joint requirements either by model checking or by a testing process.

These formulas are automatically verified at design time to support the foreseen evolutions and at run-time in order to support the unforeseen evolutions. At design time we check each variant to discover possible inconsistency. At run-time we check if it is possible to augment the current variant with an emerging feature which implements the emerging requirement.

Our goal is to prevent the system from behaving incorrectly. Each of the three problems addresses different aspects and it has different sources of errors. In addition, each of them differs in the needed algorithms and the required computation effort to perform the checks. These are relevant aspects since we are considering to perform the checks also at run-time.

Since adaptive systems for ubiquitous computing are mainly characterized by the serendipitously of the environment we believe that checking context requirement (problem(i)) plays a key role. Such systems are sensitive to the environment thus we should check first if a set of features can be executed together at a certain context state. This allows us to discard the variants that are already inconsistent in a certain context thus avoiding to check problem (ii) and (iii). Since problem (i) is only a necessary but not sufficient condition for the consistency, we should check if there exist other inconsistencies that may prevent the system from behaving incorrectly. Problem (ii) has been already treated in the literature of requirement engineering (not at run-time) in order to discover conflicts among canonical requirements. Finally problem (iii) captures the inconsistencies that emerges just at the implementation level.

3.8.1 Requirements for requirements at run-time

In this section we identify which are the characteristics that a requirement language should have in order to manage and to check the requirements specifications we propose.

In our approach requirements have to be managed at run-time. The set of system's functionalities is not always provided once and for all at design time; while the system is running, users may require new functionalities that should be included into the system through an on-line process. Hence, it should be possible to add or delete requirements R while the system is running. New functionalities, added at run-time, may provoke inconsistencies of different nature. We have discussed in Section 3.8 the three different problems that we consider to avoid inconsistent evolutions at run-time. To address these problems, the requirement portion R should be expressed in a language that supports the satisfiability checking. It should be possible to validate requirements R with respect to implementation I. Furthermore, the context requirements C should be expressed as predicates in quantitative terms in order to quantitatively limit the assignments of context entities.

To address all the above issues we propose a generic meta-layer to manage the requirements entities at run-time. This meta-layer is also independent by the adopted requirement and implementation languages. It defines which are the operations that have to be provided by any requirement language for our approach:

- Add / Delete a requirement R;
- Check requirement satisfiability for R;
- Validate R with respect to the correspondent implementation I;
- Check a quantitative requirement C in a context state;

The main challenge is to make available these operations on requirements specification at run-time. The most of the approaches found in the literature implements these operations only as design-time activities.

3.8.2 Context analysis

In this section we describe how we address the problem of checking context requirements (problem (i) in Section 3.8). First we give a simple illustration for checking context requirements belonging to a certain variant, then we explain how the context analysis can support a possible unforeseen evolution.

For the first example we take into account the fractal application. We assume that the variant $G_{fractal} = \{f_{getRem}, f_{sockConn}, f_{tiffViewer}\}$ is currently deployed at the user device. Each feature has a different context requirements, i.e.:

(i) $C_{tiffViewer} ::= cRate \geq 300 \wedge mem \geq 35$
(ii) $C_{getRem} ::= mem \geq 100$
(iii) $C_{sockConn} ::= conn = 1$

We assume that $G_{fractal}$ has to be checked at the context state $\overrightarrow{c} = (100, 300, 4096, 1)$. This state provides 100 Kb of memory, a CPU clock rate of 300 Mhz, a screen device with 4096 colors and an Internet connection. Although the predicates for each context requirement are true separately with the value in $\overrightarrow{c}$, the predicate that belongs to the whole variant is not true in the same context state because of the limited availability of free memory. Indeed, if we combine the request of memory coming from the context requirement (i) and (ii) we obtain a total request for $135Kb$ of memory that cannot be satisfied at the context state $\overrightarrow{c}$. Therefore it is not possible to execute the features $f_{getRem}, f_{sockConn}$ and $f_{tiffViewer}$ together at $\overrightarrow{c}$.

Example scenario

In this section we shown a possible scenario that exploits a context analysis phase in order to prevent incorrect evolutions for the eHealth application. Let us suppose that the variant $G = \{f_{textHeart}, f_{getHeartData}, f_{graphHeart}\}$ is currently deployed at the doctor device. The context requirement for this system alternative is:

$$C_G = cRate \geq 1100 \ \wedge \ mem \geq \ 70 \ \wedge \ conn = 1 \wedge \\ b \geq 20 \ \wedge \ heartRateProbe = true$$

The current context state is:

$$c_{curr} = (cRate(3000Khz), mem(100Kb), conn(1), \\ b(100Kbps), heartRateProbe(true))$$

At a certain point in time a new sensor to monitor the respiratory rate is added to the remote monitoring system as a new *UPnP* device ($respRateProbe$). When this happens the doctor is notified about the new gathered data. As a consequence the doctor requires to visualize the patient respiratory rate. This is expressed as a new requirement that has to be considered into the system:

$$R_{viewRespRate} : \textit{Receive and view the respiratory rate data} \tag{3.3}$$

A search phase takes place in order to discover which feature may implement $R_{viewRespRate}$. The searching phase responds with a two features that satisfy the following condition:

$$R_{New} \rightarrow R_{viewRespRate} \tag{3.4}$$

Two different features are proposed each one implementing a different visualization modality:

- $R_{graphRespRate}$: *if "respRateProbe" then (Each 10 times "getRespRData" follows a "displayGraph")*
- $R_{textRespRate}$: *if "respRateProbe" then ("getRespRData" follows a "displayText")*

Let us suppose that the doctor chooses the first alternative since he wants to see the respiratory trend as a graph. The application should be augmented with the feature shown in Figure 14.

The feature to visualize the graphical respiratory rate requires also another feature to get the data from the monitoring system ($f_{getRRate}$). Therefore we will consider the new context requirement that results from the union of the two features defined as:

$$C_{New} = C_{graphRespRate} \cup_C C_{getRRate} = cRate \geq 1000 \wedge$$
$$mem \geq 50 \wedge conn = 1 \wedge b \geq 20 \wedge respRateProbe = true$$

The new context requirement has to be valid in the current system context c_{curr} by itself and when it is jointly considered with the context con-

$R_{graphRespRate}$: **if** "respRateProbe" then (Each 10 times "getRespRData" follows a "displayGraph")

$I_{graphRespRate}$:
```
public class GraphRespRateViewer {
 XYDataset respRateDataset = new XYSeriesCollection();
  ...
 public void viewGraphicalRespRate(Graph g){
   ...
  for(int i = 0;i<10;i++){
   XYDataItem dataRespR = RespRateRetrieving.getRespRData();
   dataVectRespR.add(dataRespR);
  }
  g.displayGraph(dataVectRespR);
 }
  ...
}
```

$C_{graphRespRate}$: $mem \geq 50 \ \wedge \ cRate \geq 1000 \ \wedge \ respRateProbe = true$

Figure 14: New feature: graphical visualization (eHealth)

straints belonging to the current variant G:

$$C_G \cup_C C_{New} = mem \geq 120 \ \wedge \ cRate \geq 2100 \ \wedge \ conn = 1 \ \wedge$$
$$b \geq 40 \ \wedge \ respRateProbe = true \ \wedge \ heartRateProbe = true$$

The above formula is not valid at the current context state c_{curr} since it does not provide enough memory. Using his feature will lead to a new variant that is inconsistent in the current context. Hence, we consider the second feature $f_{textRespRate}$ to visualize a textual representation and we check the consistency in a similar way. Figure 15 shows the feature for the textual visualization.

Also $f_{textRespRate}$ requires another feature $f_{getRRate}$ to retrieve the data from the server. The joint context requirement obtained combining these two features is:

$$C_{New_2} = C_{textRespRate} \cup_C C_{getRRate} = cRate \geq 100 \ \wedge$$
$$mem \geq 20 \ \wedge \ conn = 1 \ \wedge \ b \geq 20 \ \wedge \ respRateProbe = true$$

In order to evaluate the consistency of the evolution, we check the new context requirement C_{New_2} together with the context requirement be-

$R_{textRespRate}$: if "respRateProbe" then ("getRespRData"
follows a "displayText")

$I_{textRespRate}$:
```
public class TextRespRateViewer {
  ...
 public void viewTextualRespRate(Text myTextViewer) {
  XYDataItem dataRespr = RespRateRetrieving.getRespRData();
  myTextViewer.displayText(dataRespR.getYValue());
 }
  ...
}
```

$C_{textRespRate} : mem \geq 20 \ \wedge\ cRate \geq 100 \ \wedge\ respRateProbe = true$

Figure 15: New feature: textual visualization (eHealth)

longing to G:

$$C_{G_3} \cup_C C_{New_2} = cRate \geq 1200 \ \wedge\ mem \geq 90 \ \wedge\ conn = 1 \ \wedge$$
$$b \geq 40 \ \wedge\ respRateProbe = true \ \wedge\ heartRateProbe = true$$

Since the joint constraint above is valid in c_{curr} we consider this second evolution to be consistent. Hence, the textual respiratory rate can be visualize by the doctor. As a consequence of the evolution, the context model is augmented with a new context entity $respRateProbe$; in addition a new system variant $\{f_{textHeart}, f_{graphHeart}, f_{getHeartData}, f_{textRespRate}, f_{getRRate}\}$ is added to the set of admissible variant as represented by the feature diagram. The context requirement for the new variant will be: $C_G \cup_C C_{New_2}$.

3.8.3 Model checking context-independent requirement

In this section we describe the technique we have adopted for checking context-independent requirements on code artifacts (problem (iii) in Section 3.8). We utilize an on-line verification technique which model-checks Java code through the Java Path Finder (JPF) tool (HP00). We adopt Linear Time Temporal Logic (LTL)(Pnu77) to specify the requirements of each feature.

We assume that the features are already instrumented with a procedure that checks the satisfaction for their requirements R. This procedure is invoked inside the feature implementation in order to generate exceptions only if the result of the check is negative. We then use JPF to prove the property satisfaction, i.e. if the exception is not thrown in any of the execution paths. Indeed, JPF is able to discover if there exists at least an execution path which leads to an unhandled exception.

$R_{graphOxygen} = [](GraphOxViewer.viewGraphOx(Graph) \rightarrow$
$(<> GraphOxViewer.outcome))$

$I_{graphOxygen}$:
```
public class GraphOxViewer{
 ...
 public void viewGraphOx(Graph g) throws Exception{
  for(int i = 0;i<10;i++){
   XYDataItem dataOx = OximetryRetr.getOximetryData();
   dataVectOx.add(dataOx);
  }
  g.displayGraph(dataVectOx);
  outcome = Checker.Check(g.currData, dataVectOx);
  if (!outcome){throw propertyViolation;}
 }
 ...
}
```

Figure 16: Example: graphical oxygenation

Figure 16 illustrates a feature to visualize a graphical trend for real-time data coming from a remote system. In this example $R_{graphOxygen}$ is the functional requirement for the feature implemented by $I_{graphOxygen}$; whenever the method $viewGraphOx$ is invoked, the variable *outcome* has to become true in a certain number of execution steps. In order to verify if $R_{graphOxygen}$ is true, we run the JPF core tool[1]. It checks if at least a path of execution generates an un-handled exception. This happens only if the function $Check$ can not attest that the graphical widget contains exactly the data currently written.

In our example, we assume that a certain variant G is executed on the doctor's device in order to display the data of oxygenation rate gathered by the remote monitoring system. Whenever an unforeseen context variation occurs, the user may request a new feature in the same man-

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

ner as presented for the scenario of Section 3.8.2. As for the previous scenario, the patient links a new probe to the remote monitoring system and the probe gets accepted to the system as a new UPnP device. The doctor gets a notification of such unforeseen context variation, and consequently specifies a new requirement (Eq. (3.3)). We suppose that there exists an available repository of feature code modules that responds with a set of features that satisfy the condition at Eq. (3.4). In the following we present the LTL requirement for the two retrieved features.

- $[]GraphRespRViewer.viewGraphRespR(Graph) \rightarrow$
  $<> GraphRespRViewer.outcome$

- $[]GraphRespRViewer.viewTextRespRate(Text) \rightarrow$
  $<> TextRespRViewer.outcome$

The first feature is able to view a graphical representation for the respiratory rate, whereas the second can only show a textual value for the same vital parameter. The user selects the first feature (Figure 17) since he prefers a graphical visualization.

$R_{graphRespRate} =$
$= [](GraphRespRViewer.viewGraphRespR(Graph) \rightarrow$
$(<> GraphRespRViewer.outcome))$

$I_{graphRespRate}$:
```
public class GraphRespRViewer  {
 boolean outcome=false;
 private static Exception propertyViolation;
  ...
 public void viewGraphRespR(Graph g) throws Exception{
   ...
  for(int i = 0;i<10;i++){
   XYDataItem dataRespR = RespRRetr.getRespRData();
   dataVectRespR.add(dataRespR);
  }
  g.displayGraph(dataVectRespR);
  outcome = Checker.Check(g.currData, dataVectRespR);
  if (!outcome){throw propertyViolation;}
 }
  ...
}
```

Figure 17: Example: graphical respiratory rate

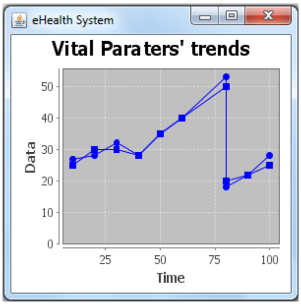


Figure 18: Visualization interaction

This feature is consistent by itself since its implementation $I_{graphRespRate}$ satisfies the requirement $R_{graphRespRate}$. In order to establish if it is possible to augment the system with such functionality, it is necessary to check if the augmented implementation $I_G \cup_I I_{graphRespRate}$ satisfies the augmented requirements specification $R_G \cup_R R_{graphRespRate}$. If the model checking result is negative, the evolution is prohibited. On the contrary, if the model checking result is positive, the evolution can take place since the current functionalities are also preserved. Listing A.3 shows the augmented requirements specification along with the augmented Java implementation for our example. By running the verification of R_{GNew} with Java Path Finder we discover that the requirements are not satisfied by the implementation I_{GNew}. The graphical visualization for the oxygenation parameter interacts with the graphical visualization for the respiratory rate. This problem arises because the methods $viewGraphOx$ and $viewGraphRespR$ access the shared common widget to plot the corresponding curve. Both methods assume that the graph is cleaned before they start their visualization. Hence, if we run only one feature at a time we can correctly show the curve for the data. Indeed, the requirements belonging to the two features are both verified by their Java implementations only if we consider them separately. On the contrary, if we want to execute both features together, the result is an interfering visualization (Figure 18) that we can discover through the verification process before switching to the new variant (Listing A.3). Indeed, the requirement R_{GNew} is false because after the visualization for the oxygenation rate, the method invocation $Check$ within $viewGraphRespR$ discovers that the graph does not contains the respiratory rate data exactly. The graph also contains the data of the oxygenation rate parameter. This generate the un-handled exception detected by JPF.

At this point the assurance process further proceeds by checking a bound number of retrieved features depending on the user (e.g. doctor) needs which are application dependent.

Chapter 4

Software Process for Adaptive Systems

The process we describe in this chapter is amenable for developing and evolving adaptive systems. We assume to have a set of basic behavioral elements as input to our process. These elements are implementation artifacts and requirements artifacts. We will explain the mechanisms to create the application and to enact its evolution from requirement to implementation level. We will assume that the evolution may be performed in a state in which it is allowed (e.g. quiescent state or weaker notions (KM90; VEBD07)).

4.1 Software Process

We have defined a software lifecycle process which follows the structure described in (AdRI^{+}11). Our software process implements four different activities, namely Explore, Integrate, Validate and Evolve as shown in Figure 19. The *exploration* phase exploits a feature library containing the code implementation and the correspondent requirements descriptions. The *integration* phase takes these features as input and produces the space of the system variants as a feature diagram. Each variant is checked through a *validation* phase which performs the context analysis

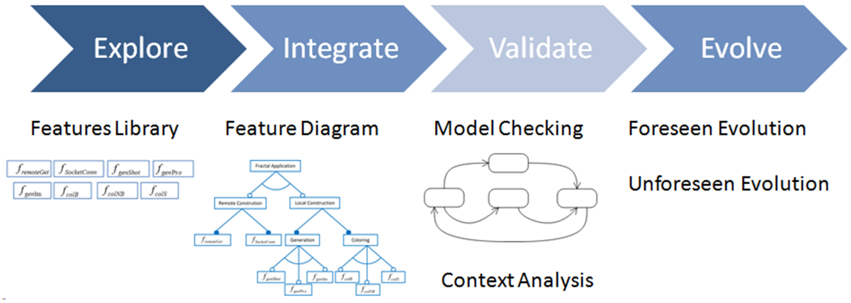


Figure 19: Software process

and model checking presented in sections 3.8.2 and 3.8.3, respectively. Finally, the *evolution* phase reconfigures the system by switching from the current variant to the new one.

The problem we face is the complexity for the software engineer to specify which are the context conditions under which each system variant is admissible. Given n features it could be required to set the context conditions for 2^n variants in the worst case. Our methodology makes the generation of the system variants automatic by exploiting the models provided in the SPLE as described in Section 3.2.

At the *exploration* phase the software engineer defines the set of features of interest. Starting from a standard software component it is possible to define a feature ($f = (R, I, C)$) by considering the requirements of the component and its code. The feature code I will be exactly the same as the code of the component. R will contain the requirements of the component that are not context-dependent. In general the requirements of the component will always contain requirements about the execution context, thus they will be added to C. Further context requirements, for example concerning resources consumption can be obtain through suitable static code analysis. For example in our environment we use the Chameleon framework (ABI09; ABI10) in order to extract the consumption of resources caused by the code I (e.g. memory and CPU clock rate). At the end of the exploration phase we have a set of features defined in terms of their basics elements, i.e. $A = \{f_1, ..., f_n\}$.

At *integration* phase the software engineer combines the features through the feature diagram definition. Architectural constraints may be defined here at the integration phase. Starting from the feature diagram an automatic process generates all the system variants:

$$G = \{G_1, G_2, ..., G_m\} \text{ s.t. } m \leq 2^{|A|} \tag{4.1}$$

We assume that the requirements belonging to each variant imply the system requirements. We further assume that each variant satisfies its requirements: $I_{G_i} \vdash R_{G_i} \; \forall i = 1, .., m$. An automatic process generates the context model structure and the context model space S considering the context entities exploited by the context requirements belonging to the created variants.

At *validation* phase we create the data structure to support the evolution. This phase takes place by means of two main steps. The first step consists in labeling each context state $\overrightarrow{c}$ in S with all the features that are consistent in $\overrightarrow{c}$ (context analysis and model checking in Section 3.8). The *feature consistency table* is built inserting value 1 each time a feature is consistent in the corresponding context state. The second step consists in building the *variant consistency table* by labeling each context state $\overrightarrow{c}$ in S with all the system variants that are consistent in that state. Finally, we aggregate the context states that make the same set of variants consistent. Nevertheless, we do not address the scalability problems arising from the number of context states and variants within the mentioned tables. Different approaches (BHRE07) have been presented to reason about the variants belonging to the feature diagram. Moreover the exponential growth of context states could be mitigated by clustering the states (DD10b).

The *evolution* phase reconfigures the system whenever either a foreseen or an unforeseen evolution is required. In the first case we query the variant consistency table to retrieve the space of the admissible variants. Among them we select the most suitable one based on the data structures provided at the validation phase. In the second case we have to re-iterate the first three phases of our software process in order to evolve the system. We query a remote feature library to retrieve the feature implementing the new requirement and we integrate the new feature with the current variant. Finally, we have to validate the new unforeseen variant before we can add it to the variant consistency table. The evolution processes are further discussed in Section 4.2.

4.1.1 Example: mandelbrot fractal

In this section we show how we design and develop the adaptive application to visualize a Mandelbrot fractal. To this end, the software engineer defines the set of features A in terms of requirements and code implementations:

$$A = \{f_{genShot}, f_{genPro}, f_{genImm}, f_{colB}, f_{colNB}, f_{colS}, f_{remGet}, f_{sockConn}\}$$

The set A contains the features to generate and color the fractal pixels and the features to download a standard fractal image from a remote server. The generation may be performed by visualizing a pixel at a time (f_{genImm}), a pixel row at a time (f_{genPro}) or the whole fractal image at the end of the drawing process ($f_{genShot}$). The pixel colors are defined following three different schemas: f_{colB} colors pixels as bands exploiting a limited number of tones; f_{colNB} colors pixels as bands exploiting a wide spectrum of tones while f_{colS} follows a smooth schema to color pixels exploiting a wide spectrum of tones. Finally, $f_{sockConn}$ connects the device to the Internet whereas f_{remGet} retrieves and views a standard fractal image from a remote server.

Figure 20 shows an excerpt of the features entailed in A. It is possible to define the context requirement of each feature by exploiting the Chameleon framework in order to obtain the consumption of resources, e.g. CPU and memory. Further, context requirements can be defined by extracting the requirement on the number of screen colors derived from the requirement of the component.

In order to design the fractal application the software engineer combines the features and produces the feature diagram as shown in Figure 12. The logic operators in the feature diagram guide the automatic generation of 10 system variants as shown in Table 1. The first nine variants

$\mathbf{f_{genPro}} = (\mathbf{R_{genPro}}, \mathbf{I_{genPro}}, \mathbf{C_{genPro}})$
$\mathbf{R_{genPro}}$: Compute each fractal pixel and show it a pixel row at a time
$\mathbf{I_{genPro}}$:

```
public class MandelCanvas extends Canvas{ ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
   }
   offsetX = x;
   image = Image.createRGBImage(column_ArrayCanvas,1, height, false);
   repaint();
  } } ... }
```

$\mathbf{C_{genPro}} : \mathbf{mem} \geq \mathbf{200}$

$\mathbf{f_{colS}} = (\mathbf{R_{colS}}, \mathbf{I_{colS}}, \mathbf{C_{colS}})$
$\mathbf{R_{colS}}$: Paint the fractal pixels as smoothly nice colored bands
$\mathbf{I_{colS}}$:

```
public class Colouring{...
 private int pixelColorSmoothly(boolean interno, int iterazioni,
 double dist){
  if (interno) return 0;
  iterazioni = iterazioni + 2;
  double mu_IterationsDistance = iterazioni−
  (Float11.log(Float11.log(dist))) / log2;
  int tmp= DBL_ToRGB(mu_IterationsDistance);
  return tmp;
 }
 private void initColorsSmoothly() {
  log2 = Float11.log(2.0);
 }...}
```

$\mathbf{C_{colS}} : \mathbf{cRate} \geq \mathbf{500} \wedge \mathbf{sc} \geq \mathbf{4096}$

$\mathbf{f_{remGet}} = (\mathbf{R_{remGet}}, \mathbf{I_{remGet}}, \mathbf{C_{remGet}})$
$\mathbf{R_{remGet}}$:Retrieve and view the fractal image from the server
$\mathbf{I_{remGet}}$:

```
public class RemoteViewer extends Canvas{...
  public void viewRemoteFractal(){
   this.image = getFractal(startTime*1000,maxExecutionTime);
   repaint();
  } ...}
```

$\mathbf{C_{remGet}} : \mathbf{mem} \geq \mathbf{100}$

Figure 20: Application features

are obtained combining the three different building mechanisms with three different coloring schemas. The last one simply gets an already defined fractal image from a remote server. Each variant is characterized by the context requirement and by the offered qualities. The $DisplayModel$ quality represents the modality of showing the fractal while $ColorModel$

quality expresses the coloring modalities.

Table 1: System variants

System Variant	**Context Requirement**	**Offered Quality**
$G_1 = \{f_{genShot}, f_{colB}\}$	$mem \geq 300 \wedge cRate \geq 100$	$DisplayModel = Shot$ $ColorModel = BandOfColors$
$G_2 = \{f_{genShot}, f_{colNB}\}$	$mem \geq 300 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = NiceBandOfColors$
$G_3 = \{f_{genShot}, f_{colS}\}$	$mem \geq 300 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Shot$ $ColorModel = SmoothyBandOfColors$
$G_4 = \{f_{genPro}, f_{colB}\}$	$mem \geq 200 \wedge cRate \geq 100$	$DisplayModel = Progressive$ $ColorModel = BandOfColors$
$G_5 = \{f_{genPro}, f_{colNB}\}$	$mem \geq 200 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = NiceBandOfColors$
$G_6 = \{f_{genPro}, f_{colS}\}$	$mem \geq 200 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Progressive$ $ColorModel = SmoothyBandOfColors$
$G_7 = \{f_{genImm}, f_{colB}\}$	$mem \geq 120 \wedge cRate \geq 100$	$DisplayModel = Immediate$ $ColorModel = BandOfColors$
$G_8 = \{f_{genImm}, f_{colNB}\}$	$mem \geq 120 \wedge cRate \geq 300 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = NiceBandOfColors$
$G_9 = \{f_{genImm}, f_{colS}\}$	$mem \geq 120 \wedge cRate \geq 500 \wedge$ $sc \geq 4096$	$DisplayModel = Immediate$ $ColorModel = SmoothyBandOfColors$
$G_{10} = \{f_{remGet}, f_{sockConn}\}$	$mem \geq 100 \wedge conn = 1$	$DisplayModel = Shot$ $ColorModel = BandOfColors$

After creating the variant, the integration phase generates the context model which contains the relevant resources for the fractal application as shown in Figure 7(a). It creates the context model space as $S = mem \times cRate \times sc \times conn$.

As far as the validation of the fractal application is concerned we only show the consistency based on context analysis. The validation phase creates the feature consistency table (Table 2) by checking the consistency for each feature at each context state in S. It evaluates the validity for the context requirements (predicates) of each feature by assigning all the possible context values. The table assigns value 1 if it is possible to deploy a feature in a certain context state and 0 otherwise. After defining the feature consistency table the context analysis phase creates the variant consistency table (Table 3) by considering the features included in each variant. This table contains value 1 only if all the features in a certain variant are jointly consistent at a certain context state. The process checks the validity of the joint predicate as shown in Section 3.8.2.

Table 2: Feature consistency table

$C(mem, cRate, sc, conn)/f_j$	$f_{genShot}$	f_{genPro}	f_{genImm}	f_{colB}	f_{colNB}	f_{colS}	f_{remGet}	$f_{sockConn}$
$C_0 = (100, 200, 256, 0)$	0	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	1	1	1	0	1	1
...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	1	1	1	0	0	1	1
...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1

Table 3: Variant consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1
...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1

4.2 System Evolution

Our development process supports the system evolution required by the context variations. In the following we show that while in the foreseen evolution, system and context models are queried to support the reconfigurations; in the case of unforeseen evolution the same models may have to be refined as a consequence of incoming user needs.

4.2.1 Foreseen Evolution

In the foreseen evolution we consider only variants that have already been proven consistent. A monitoring process notifies the context variations that invalidate the context requirement belonging to the running variants. Whenever such a new assignment of resources is discovered, the framework queries the variant consistency table to get the possibly new admissible variants. In order to perform the static decision-making process among consistent variants we take into consideration context and user preferences.

Since we want to make our mechanism resilient to future contexts we take into consideration which is the probable future evolution for each context state. We consider the predictions for the user centric information (user task, user mobility) and the predictions for the evolution laws of resources obtained as explained in (PGS$^+$07). Exploiting such information we build a probabilistic automaton according to the approaches in (MM07; BCM10). Each different state corresponds to a different context and each arc expresses the probability to move from a context to another (e.g. Figure 21). In the next section we show a possible decision-making

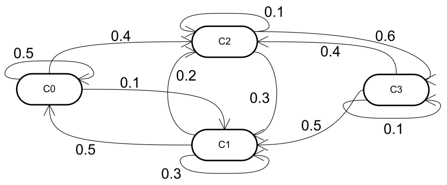


Figure 21: Probabilistic evolution automata

process which considers fixed user preferences and probable context evolutions.

Example: Mandelbrot fractal

Starting from the set of admissible variants we evaluate a fitness value in order to discover which of them is the most suitable with respect to future context variations and the fixed user preferences. Starting from the automaton in Figure 21 we evaluate the steady-state probability vector $\overrightarrow{p} = [0.2794\ 0.2794\ 0.2647\ 0.1765]$ which expresses how often the context belongs to a certain state. Then we evaluate the *context fitness vector* by multiplying the vector $\overrightarrow{p}$ with the matrix m representing the variant consistency table:

$$f = p \cdot m \tag{4.2}$$

This vector assigns a fitness value at each variant that depends on the number of states in which the system variant is admissible and on the relevance for the states as evaluated by the steady-state probability vector. This ranking mechanism considers only how often the context belongs to a certain state whereas it ignores which is the current state and its future transitions thus leading to globally optimum solutions. Parallel to f we also evaluate a *user fitness vector* t expressing how each variant is suitable with respect to the user preferences. We express preferences as weights over the quality attributes which characterizes the variants. Each weight w_q (from 0 to 1) indicates the interest for the user towards a certain quality q. We use a predefined utility function $u_q(G_i)$ to assign a value from 0 to 1 at each quality dimension q provided by each G_i. The software engineer defines the utility functions and the weights for each quality since they are strictly application dependent. The *user fitness vector* is evaluated as:

$$t(G_i) = \sum_{q \in Qualities} w_q \times u_q(G_i) \tag{4.3}$$

Our decision-making process will consider together the user fitness $t(G_i)$ and context fitness $f(G_i)$ to evaluate the overall fitness of each system variant G_i.

Let us consider the scenario as depicted in Table 3 and let us suppose that the variant $G_4 = \{f_{genPro}, f_{colB}\}$ is running at the context state $C_{43} = (350, 200, 4096, 1)$ whereas the user preferences assign higher weight to the $DisplayModel$ quality. In this simple example we assume the user preferences to be fixed. The system is producing a fractal image drawing a row at a time and coloring pixels as bands of colors. Let us now suppose that because of a new application started on the mobile device, the current memory availability changes and the monitoring detects a context variation. By looking at the new context state $C_{33} = (150, 400, 4096, 1)$ in Table 3 we obtain the set of admissible (consistent) variants. Among them we select the one with the highest overall fitness. Therefore the current fractal application is stopped and it is evolved towards the variant $G_7 = \{f_{genImm}, f_{colB}\}$ which represents the best trade-off between user and context fitness.

The weak point of this methodology is that it only considers fixed user preferences which are defined once and for all. By contrast user preferences are not fixed but they may change over the execution since the user could be involved in changing context. For instance, the user could either be involved in new tasks or run the application in different locations. As a consequence we should include the possible preferences variations within the automaton. In Chapter 5 we will present a more detailed mechanism to perform reconfigurations based on a probabilistic model which considers changing user preferences. We will present this mechanism along with a problem formalization and experimental results.

4.2.2 Unforeseen Evolution

Let us assume that during the execution phase the set of requirements the system needs to satisfy evolves because of changing user needs. For example the user has to deal with a new context situation that has not been foreseen by the software engineer at design time. Since a new behavior may have to be injected into the system it is necessary to modify at run-time the context-based decision table presented in the earlier sections. In addition also the models related to the system variability and context may have to be refined at run-time. Two different cases can arise: either a new requirement has to be added to the current variant or an already existing requirement has to be deleted from the current variant. We suppose that the requirement to add or to delete does not imply other requirements causing side effect phenomena to be managed. Thus, in order to evolve the application with a new requirement we augment the current selected variant with a new feature implementing the new requirement. This leads to a new variant that has not been anticipated at design time. Adding new requirements is more problematic than deleting requirements, thus we only discuss the first. Further, adding new behaviors seems to be appropriate for facing unforeseen situations.

In our approach we only evolve the current selected variant whereas we do not consider how to augment the whole space of variants with the new requirement. We neither discuss how the addition of a new requirement to a variant may affect the qualities attributes offered from the variant.

The user may press a specific button within the application interface in order to communicate to the framework the variation of his/her needs. Then the user may specify the new requirement R_{New}, for example in natural language. The unforeseen evolution phase has to upgrade the running variant with a new feature implementing the requirement R_{New}. We assume to have a search engine that given a requirement is able to return the set of features implementing it (exploration phase). Among them, we select the first feature $f_{New} = (R_{New}, I_{New}, C_{New})$ that is consistent with the current running system variant $G_F = (R_F, I_F, C_F)$ at the current context. The integration phase creates the new variant $G_F \cup_f f_{New}$ and the validation phase checks the consistency of the variant at the current context state. The variant is added to consistency table and since new resources may be required by the new feature it could be necessary to augment the context. Also the feature diagram is kept updated by adding the incoming feature. We recall that in our approach, the integration of a new feature to the feature diagram only leads to a new variant. We do not consider how to perform the integration of the new feature with all the possible variants since we only evolve the current variant.

Example: Mandelbrot fractal

Let us suppose that at the context state $C_{33} = (150, 400, 4096, 1)$, the foreseen evolution select a new variant $G_{10} = \{f_{remGet}, f_{sockConn}\}$ which visualizes a precomputed fractal image after it has been downloaded from a remote server. The retrieved image complies to the TIFF image format. Because of unforeseen characteristics of the mobile device, the user cannot visualize the retrieved image. The device cannot decode TIFF images and therefore the fractal application has to be upgraded. To this end, the user interacts with the framework to add a new requirement in the application. After accessing the upgrading wizard, the user specifies the new requirement in natural language:

$$R_{New} = \textit{The system shall visualize TIFF format images} \qquad (4.4)$$

This requirement has not been foreseen at design time but arises only at run-time when the unforeseen device characteristics (context) make the fractal visualization impossible. Thus after the evolution process we have to re-iterate the exploration, integration and validation phases at run-time in order to evolve the application with the feature (i.e. the software codec) to view TIFF format images. This will lead to a new variant with same features of the current variant plus the new feature.

The exploration phase queries the search engine in order to retrieve a feature which implements the new requirement, e.g. see Figure 22.

$I_{tiffViewer}$:
```
public class Viewer{ ...
 public RenderOp tiffViewer(Object stream){
  ParameterBlock params = new ParameterBlock();
  params.add(stream);
  TIFFDecodeParam decodeParam = new TIFFDecodeParam();
  RenderedOp image = JAI.create("tiff", params);
  return image;
}...}
```
$C_{tiffViewer} : cRate \geq 300 \wedge mem \geq 35$

Figure 22: Example: new feature

The integration phase augments the feature diagram with the new feature as shown in Figure 23. An optional feature $f_{tiffViewer}$ is added to the diagram thus leading to a new variant $G_{New} = \{f_{remGet}, f_{tiffViewer}, f_{sockConn}\}$.

For the validation phase we consider how the new context requirement affects the context requirements provided at design time. The new context requirement $C_{tiffViewer}$, that we consider for consistency, refers to the resources *cRate* and *mem* which have been already foreseen at design time; thus a context model extension is not required. To establish if the new variant $G_{New} = G_{10} \cup_f f_{tiffViewer}$ is consistent we evaluate the new context requirement jointly with the context requirement for G_{10}, i.e.

$$C_{New} = cRate \geq 300 \wedge mem \geq 135 \wedge conn = 1$$

This predicate is true at the context state C_{33} since this state provides

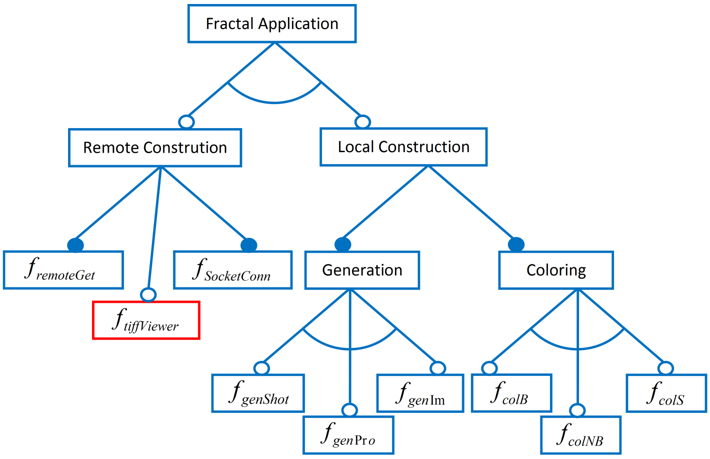


Figure 23: Refined feature diagram

enough memory, cpu speed and an Internet connection. Only if the new predicate is false it is necessary to restart the evolution process by the exploration phase in order to consider other features. Finally, if the variant is consistent (the new predicate is true with the current context values), the validation phase adds the new variant G_{New} to the consistency table as shown in Table 4 by checking the consistency property also for the other context states.

Table 4: Refined variant consistency table

$C(mem, cRate, sc, conn)/G_k$	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	G_{New}
$C_0 = (100, 200, 256, 0)$	0	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{33} = (150, 400, 4096, 1)$	0	0	0	0	0	0	1	1	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...
$C_{43} = (350, 200, 4096, 1)$	1	0	0	1	0	0	1	0	0	1	0
...	...	...	...	...	...	...	...	...	...	...	...
$C_{47} = (350, 600, 4096, 1)$	1	1	1	1	1	1	1	1	1	1	1

Even if it is not shown in the example, a new feature may also require new unforeseen context entities in its context requirements. Thus, it may be necessary to refine also the context model in order to consider the values for the new resources. As a consequence it would be also necessary to augment the consistency table with the new context states arising from the augmented context model space.

Chapter 5

Static decision-making reconfiguration

Beyond changes that affect the validity for context requirements there exist changes that affect user preferences. Reconfigurations should meet the desired quality requirements according to changing user preferences while they should be performed at reasonable costs. In this scenario, considering predictive information allows us to anticipate upcoming reconfiguration needs. When determining the most suitable variant, the challenge lies in finding a suitable trade-off between two objective functions: maximize the user benefit while minimizing reconfiguration cost. User benefit determines how good is a certain variant for the user according the quality offered by the variant and the user preferences over these qualities. Costs is defined as a function of the distance between the current and the target variant.

Pure user benefit decision mechanisms come with high costs due to frequent reconfigurations. In contrast, pure cost-driven adaptations neglect user preferences and always choose the current variant. They only change the current variant when it is absolutely necessary thus leaving the user unsatisfied. Our solution is to define the selection among different alternative variants as a multi-criteria optimization problem where the aggregative objective function combines user benefit and cost.

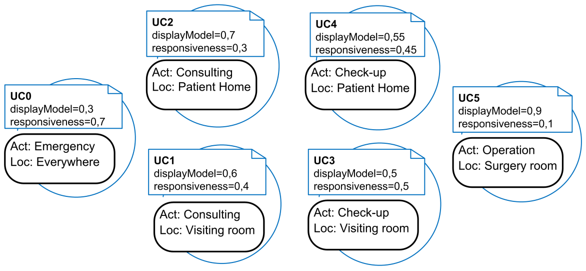


Figure 24: User Preferences Example

5.1 Motivating scenario

An e-Health application supports doctors' activities by providing the most relevant services to visualize per-patient case history. Patient information is available at three levels of granularity: (i) a complete case history that includes textual reports and medical images, (ii) a compact version with only the recent history of reports and images, and (iii) only a textual case history. In addition images are displayed either as black and white images, in low color (256 colors), or as fully colored images (4096 colors).

Doctors need to receive aggregated per-patient information to support their activities at different locations. These activities include patient consulting, check-up, and medical procedures such as operations. Moreover they may be involved in emergency situations. These activities are performed at different locations such as common visiting rooms, surgery rooms, patient home or outside the hospital when an emergency arises. The doctor is able to visualize per-patient information through an accessible device inside or outside the hospital. Devices differ in their hardware resources such as bandwidth availability ($netB$), number of screen colors (sc), CPU speed ($cRate$) and available memory (mem). Hardware has an impact on the available services: e.g., low bandwidth and 8 bit colors restrict the responsiveness to retrieve the patient's medical history and available image quality.

Activity and location influence the doctor's preference for displaying

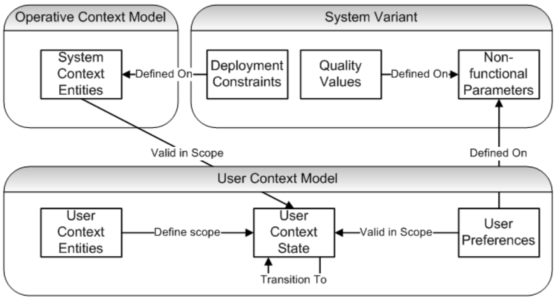


Figure 25: Conceptual Model

the case history and image quality, see Figure 24. The doctor might prefer a responsive system in case of an emergency activity. In another case, immediate retrieval of per-patient information is not as important as a detailed history for consulting activities. Upon context changes, the e-Health application needs reconfiguration based on the underlying hardware resources and the doctor's (context-dependent) preferences.

5.2 Basic model

System reconfigurations aims at satisfying two objectives: maximizing user benefit and minimizing cost which arises due to the reconfiguration. Figure 25 visualizes the basic elements to enact the static decision-making reconfiguration.

As introduced in Chapter 3 system variants can be expressed in terms of context requirements (i.e. *deployment constraints*) and *offered qualities*. We consider a set of deployment constraints in order to assess the admissibility of the variant. These constraints are evaluated against the current underlying context (resources) to establish whether the environment can support the execution of that particular variant. We also map a system variant to *non-functional properties* to represent its quality. This quality becomes a utility of the variant (i.e., user benefit) when matched with user preferences.

In order to consider how the context affects the evaluation of the system variants we propose two types of context models: the (operative) context model as defined in Section 3.4 and the user context model. On the one hand *deployment constraints* are conditions on the *operative context model*. On the other hand, user preferences, which are defined as weights over qualities, are not static but depend on user information such as user location and user activity. Thus here we also define a *user context model* which maps particular user preferences to specific user information. We exploit the operative context model to evaluate the set of admissible variants while we exploit the user context model to consider current and probable future user preferences (CT11).

For each user, we assume the availability of historical transitions between the various context states. We also assume the time required for system adaptation upon a state transition to be negligible compared to the frequency of user context changes. Finally, we will exploit a generic cost model in which the cost of deploying a feature is independent from the running variant.

5.2.1 Operative context model

The operative context model is defined in terms of a context model space and a context model structure as proposed in Section 3.4. Further we extend original context model definition with the definition of *operative context scope* which is a subset of the operative context space (Eq. 3.1): $os \in 2^S$, e.g. $os = (netB(100-200Kbps), mem(10-50MB), sc(10-20colors), cRate(100-150Mhz))$. A context scope entails a set of context states.

5.2.2 Variants

Each variant c is defined in terms of a context requirement and the fitness values for the non-functional properties.

An operative context scope os_c contains the set of states which make the context requirement (predicate) for the variant c true. Then, we evaluate if a variant c is admissible in a context scope os with the function f_c which is equal to 1 only if $os \subseteq os_c$ and 0 otherwise. In our problem we also exploit the function $Eligible(r)$ to evaluate which variants are admissible with the context values in r.

Non functional properties represent the qualities offered to the user. These variables $NFP = nfp_1, nfp_2, ..., nfp_s$ are normalized in the real range [0,1]. The vector fv_c contains the fitness values for the variant c.

5.2.3 User Context Model

User context entities characterize the user′s situation. As they are beyond the control of the application, they play a key role in the adaptation process. As mentioned in Section 5.1, the user′s preferences change when switching from one user context state to another. Note that our approach is independent from the actual user context entities and how they change as long as there is a mapping of the various observable user context states to user preferences.

We define a mapping between the user context state UC — as defined by a set of user context entities — and the associated user preferences. User preferences express the importance (i.e., weight) of the various non-functional properties in a given context state. Higher weights express higher importance applied in the mapping functions $w : NFP \rightarrow [0, 1]$. Furthermore, we introduce a probabilistic automaton to represent the changing user preferences as induced by the underlying transitions between context states.

This automaton is defined as $A = (UC, P, E)$ where:

- $UC = \{UC_0, ..., UC_t\}$ is the set of states expressing the space of the user preferences. Each state is represented as a different combination of weights upon the non-functional parameters: $UC_j = [w_1(nfp_1)...w_s(nfp_s)]\ j = 1, .., t$;
  at each state the weights are defined as: $\sum_{i=1}^{s} w_i(nfp_i) = 1$
- P is the set of transition probabilities
- $E : UC \times P \rightarrow UC$ is the probabilistic transition function

This probabilistic state-based model shows how the preferences reflect the changes of user context entities. Historical data collected during

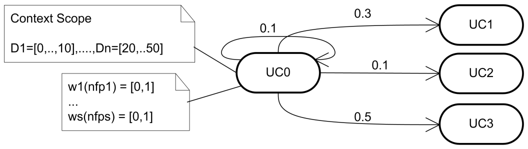


Figure 26: Probabilistic automaton excerpt

system execution allows us to determine the actual transition probabilities between user context states. We continuously sample user context data at fixed intervals of time so that the probability to have two or more preference changes (i.e., context changes) within one interval is negligibly low. This process, however, is beyond the scope of the thesis. Nevertheless techniques like (KSS06) show the possibility to get preferences from user context, whereas methodologies like (MKUM09) define how to build a probabilistic model and maintain it updated with current system execution.

We expect that the various user context states come with changes in the operative context space. For example, bandwidth will not be the same in every location. Consequently, we consider also if a particular system variant is admissible in the observed user context state, independent from user preferences. We define a mapping function to associate each state in UC with an operative context scope within the set OS ($UCR : UC \to OS$). This models the correspondence between the user preferences and the observed system context entities. Figure 26 provides an excerpt of a probabilistic automaton, detailing the mapping of user preferences and operative context scopes to a user context state.

5.2.4 Transition cost

An important factor to consider during the reconfiguration process is the penalty of switching from the source variant to the target variant. Since in our approach system variants are made by features, we characterize this penalty based on the distance between the two variants as $Dist_{y,z} = [NToDeploy\ NToUnDeploy]$ expressing the number of features to deploy and un-deploy switching from y to z. The vector $FCost = [CDeploying_f \; CUnDeploying_f]$ contains the same cost of deploying and un-deploying a feature. Based on the two vectors we define the transition cost of switching from y to z as:

$$TC(y, z) = (Dist_{y,z} \cdot FCost^T)/MaxCost \tag{5.1}$$

This cost is normalized to the maximum theoretical cost, which depends on the maximum number of features to deploy and un-deploy:

$$MaxCost = [MaxToDeploy \; MaxToUnDeploy] \cdot FCost^T \tag{5.2}$$

This simplified cost model is sufficient for our purpose since we do not address the problem of executing the actual system reconfiguration at the implementation level.

5.3 Rankings of the variants

Two events trigger the optimization problem and subsequent reconfiguration. Either the user moves into a new user context state characterized by a changing preference or the operative context cannot support the execution of the current system variant anymore. The best variant to select depends on the achievable user benefit and the associated costs for reconfiguring the system. A strategy that maximizes the user benefit after each transition possibly requires many system reconfigurations. On the other hand choosing a fixed variant which is always eligible throughout all states may result in possibly sub-optimal user benefit or may not exist at all. As a consequence we have to consider a trade-off analysis between two potentially conflicting criteria, i.e. user benefit and reconfiguration costs. In the following we formalize the user benefit, the reconfiguration cost, and describe their combination in a single utility function.

As shown in Eq.5.3 the component B_{curr} evaluates how well a certain variant c fits the current user context state. The user benefit at each state is the product of the corresponding user preferences vector with the quality attribute fv_c offered by the variant.

$$B_{curr} = UC_{curr} \cdot fv_c^T \tag{5.3}$$

A system variant that gives optimal user benefit for a certain state may be sub-optimal if we consider the probable future states. Therefore we introduce an equation component that evaluates the expected user benefit in the future as given by the probabilistic context transitions. The cost component B_F shown in Eq.5.4 computes the future benefit of a variant. We limit the calculation of future benefit to a single hop in the transition graph. Considering additional states (i.e., multiple hops) is expected to yield little additional benefit as each of the reachable states will have very small probability and thus hardly any impact.

$$B_F = \sum_{j=1}^{\#OutLink(UC_{curr})} p(UC_{curr}, UC_j) \cdot \left[UC_j \cdot fv_c^T\right] \cdot f_c(os_j) \tag{5.4}$$

B_F aggregates the user benefit for each subsequent user context state weighted according to the respective transition probability. A variant yields user benefit only if it is eligible in the corresponding operative context scope ($f_c(os_j) = 1$). Ultimately, the overall user benefit equation is obtained by combining the current and future user benefits as follows:

$$B_{Agg} = h \cdot B_{curr} + (1 - h) \cdot B_F \tag{5.5}$$

The horizon h regulates the importance of the current user benefit compared to the future user benefit. The horizon close to 1 expresses a preference for the current state, whereas for h close to 0 we deem the future more relevant. Thus for environments where the user is expected to rapidly switch between states, the horizon configuration parameter should be closer to 0 as he/she will leave the current state soon.

The reconfiguration cost TC represents the cost of switching from the current variant to the variant c (Eq. 5.1). The problem of selecting the best variant in a operative context state r, given a predefined user context model, is formalized as a max optimization problem combining the expressions defined in Eq. 5.3, 5.4, 5.1:

$$\max_{c \in Eligible(r)} \alpha \cdot [h \cdot B_{curr} + (1 - h) \cdot B_F] - (1 - \alpha) \cdot TC(c_{curr}, c) \tag{5.6}$$

The parameter α regulates the trade-off between user benefit and reconfiguration cost. Setting α closer to 1 makes the optimization more likely to meet the user benefit in spite of a high cost of reconfiguration. When setting this parameter closer to 0, we reduce the reconfiguration cost by selecting general purpose system variants that may be sub-optimal on the user benefit. The parameter h enables to tune the interest between the current user preferences and the probable future user preferences as explained above.

By introducing the variables α and h we make our optimization process customizable to various environments. The horizon h enables tuning to self-transitions in the context automata. After the creation of the automata, if the resulting self-transitions are very high but still we are interested in optimizing future preferences we need to decrease the value of h. On the other hand if we end up with low self-transitions but we want to better match current preferences we have to increase the value of h. In addition, by setting $h = 1$ we enable comparison to existing approaches that are future-unaware.

5.4 Case Study

Applying the feature engineering perspective the e-Health scenario yields the following alternative features to view the per-patient case history: $S = \{f_{viewAllIm}, f_{viewLastIm}, f_{viewAllRep}, f_{viewLastRep}, f_{viewSum}, f_{paintBW}, f_{paintCol}, f_{paintFCol}\}$.

Table 5 lists the 7 variants built from these proposed features. Variant c_1 provides only a textual representation of the patient's case history ($summary$). The next three variants display only the very recent case entries ($lastHistory$) by means of textual reports and medical images which may be colored following the three different modes (BW, $Fullycolored$, $Colored$). The last three variants display the complete case history ($completeHistory$) with a different coloring modality. The features combination in each variant determines the responsiveness level (ranging from Low to $High$). In the following we describe how the reconfiguration process takes place whenever the user switches context.

Table 5: System variants

Variant	**Deployment Constraint**	**Non-Functional Properties**
$c_1 = \{f_{viewSum}\}$	$netB(5kbps) \wedge mem(0,1MB)$	$displayModel = summary$ $responsiveness = high$
$c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$	$netB(20kbps) \wedge mem(2,5MB) \wedge$ $cRate(40Mhz)$	$displayModel = lastHistory$ $responsiveness = mediumHigh$
$c_3 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintCol}\}$	$netB(20kbps) \wedge mem(2,5MB) \wedge$ $cRate(50Mhz)$	$displayModel = lastHistory$ $responsiveness = mediumHigh$
$c_4 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintFCol}\}$	$netB(200kbps) \wedge mem(40MB) \wedge$ $cRate(10Mhz) \wedge sc(4096colors)$	$displayModel = lastHistory$ $responsiveness = mediumLow$
$c_5 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintBW}\}$	$netB(40kbps) \wedge mem(10MB) \wedge$ $cRate(40Mhz)$	$displayModel = completeHistory$ $responsiveness = medium$
$c_6 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintCol}\}$	$netB(40kbps) \wedge mem(10MB) \wedge$ $cRate(50Mhz) \wedge sc(256colors)$	$displayModel = completeHistory$ $responsiveness = medium$
$c_7 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintFCol}\}$	$netB(800kbps) \wedge mem(160MB) \wedge$ $cRate(100Mhz) \wedge sc(4096colors)$	$displayModel = completeHistory$ $responsiveness = low$

We potentially observe a change of the user preferences when the doctor moves to a different location or engages in a different task. If this is the case, we then have to evaluate which variant maximizes Eq. 5.6. We select the best variant starting from the following inputs: the set of eligible variants in the current operative context, the user context automata and the reconfiguration costs. In this case study, we obtain a user context automaton with the transitions probabilities shown in Figure 27 by analyzing historical user data detailing the movements and the doctor's working timetable. Each state is characterized by different weights for each quality attribute ($UC_0 = [0.3\ 0.7]$, $UC_1 = [0.6\ 0.4]$, $UC_2 = [0.7\ 0.3]$, $UC_3 = [0.5\ 0.5]$, $UC_4 = [0.45\ 0.55]$, $UC_5 = [0.9\ 0.1]$). The first component of each vector indicates how important the *displayModel* property is, while the second expresses the weight for *responsiveness*. In addition each user context state is associated to a different operative context scope $OS_0, .., OS_5$.

Suppose the doctor changes from an emergency activity to a check-up activity within the hospital visiting room. As a consequence the user context switches from UC_0 to UC_3 and the reconfiguration process commences. Note that the user is free to switch between context states which exhibit no corresponding transition in the automaton. Let us suppose that the running variant is c_2 and the operative context state is $r_{curr} = (netB(50Kbps), cRate(100Mhz), mem(20MB), sc(256colors))$. We check the deployment constraints for the variants in Table 5 against the state

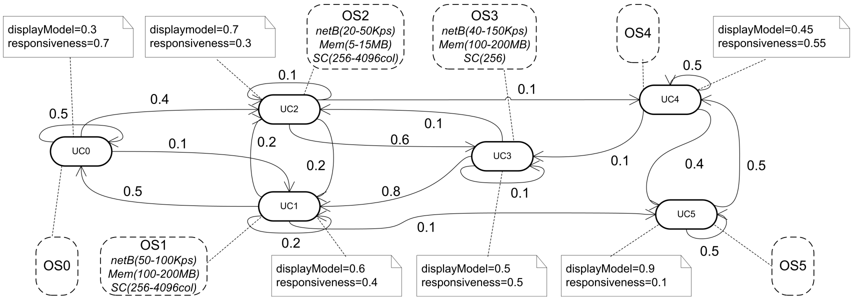


Figure 27: User Context Automata

r_{curr}. Thus we compute the set of eligible variants as $Eligible(r_{curr}) = \{c_1, c_2, c_3, c_5, c_6\}$. Each variant provides two non-functional properties $NFP = \{displayModel, responsiveness\}$. The first assumes one value among $summary$, $lastHistory$ and $completeHistory$ whereas the second assumes one value among low, $mediumLow$, $mediumHigh$, $medium$ and $high$. Starting from the qualities offered by each variant we evaluate the parallel fitness vectors by means of a possible normalization:

$$\begin{array}{lll}
c1: [summary\ high] & \Rightarrow & fv_{c1} = [0.1\ 0.8] \\
c2: [lastHistory\ mediumHigh] & \Rightarrow & fv_{c2} = [0.5\ 0.65] \\
c3: [lastHistory\ mediumHigh] & \Rightarrow & fv_{c3} = [0.5\ 0.65] \\
c5: [completeHistory\ medium] & \Rightarrow & fv_{c5} = [0.9\ 0.5] \\
c6: [completeHistory\ medium] & \Rightarrow & fv_{c6} = [0.9\ 0.5]
\end{array}$$

Table 6: Distance evaluation

Dist./Conf.	c_1	c_2	c_3	c_5	c_6
ToDeploy	1	0	1	2	3
ToUnDeploy	3	0	1	2	3

For purpose of demonstrating, we assume the cost of deploying and un-deploying any feature is $FCost = [2\ 1]$. The distance between each admissible variant and the current one ($c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$) in terms of features to deploy and un-deploy is given in Table 6. The normalized costs of switching from the current variant to each possible target one are: $TC(c_2, c_1) = 0.556$, $TC(c_2, c_2) =$

0, $TC(c_2, c_3) = 0.333$, $TC(c_2, c_5) = 0.667$, $TC(c_2, c_6) = 1$. The maximum theoretical cost we use for the normalization is evaluated as $MaxCost = [3\ 3] \cdot [2\ 1]^T = 9$ (Eq. 5.2). We solve the optimization problem at Eq.5.6 considering the new user context state UC_3, the set of eligible variants at the operative state r_{curr}, and the costs. For demonstrating our approach we set α to 0.9 to express that the user benefit is more important than costs. We also set the variable h to 0.3 to consider future user preferences more relevant than the current preferences.

Our proposed methodology enables selecting the variant which fits better the current preferences while considering the future user preferences. Future preferences are determined by the probable future task and location in which the doctor will be involved. In addition also the costs of switching variant are taken into account.

At the current user context state ($UC3$, the one where the user just arrived), the doctor is performing a check-up activity at the visiting room where the $responsiveness$ and $displayModel$ properties are equally ranked (Figure 24). By looking at the automata in Figure 27 we reason that with very high probability the doctor will thereafter switch to another state (UC_1). This probable subsequent state comes with different weights for $responsiveness$ and $displayModel$ (UC_1). As a consequence we anticipate this future transition by selecting a system variant which provides already better display modality now, even if it does not strictly meet the current user preferences. Nevertheless, in this example the top ranked variant maximizes also the current preferences.

Table 7 presents the overall utility value for the eligible variants obtained by combining the user benefit component (Eq.5.5) and the cost (Eq. 5.1). User benefit components do not need normalization since they are evaluated exploiting normalized user preferences and normalized quality vector. In this illustrative example the best variant is c_6 since it corresponds to the best trade-off between user benefit and costs with given h and α.

Table 7: Evaluation of the variants

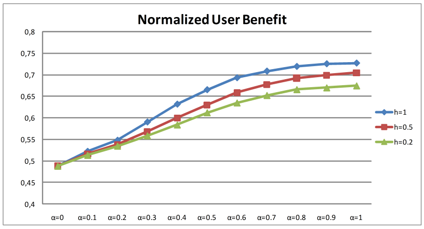
Variant	B_{curr}	B_F	Cost	**Overall utility**
$c_1 = \{f_{viewSum}\}$	0.45	0.38	0.556	0,305
$c_2 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintBW}\}$	0.575	0.56	0	0,508
$c_3 = \{f_{viewLastIm}, f_{viewLastRep}, f_{paintCol}\}$	0.575	0.56	0.333	0,475
$c_5 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintBW}\}$	0.7	0,74	0.667	**0,589**
$c_6 = \{f_{viewAllIm}, f_{viewAllRep}, f_{paintCol}\}$	0.7	0,74	1	0,555

5.5 Experiment

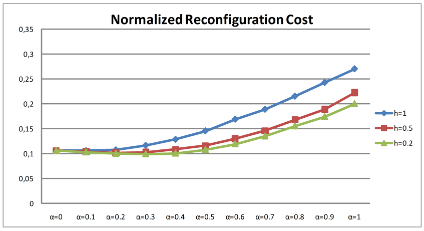
Besides a case study, we validate our approach by simulating context changes and the resulting reconfigurations for various parameter settings. The results demonstrate that a predictive approach considerably improves the reconfiguration process. The simulation process takes the user context automata, costs, and a set of system variants as input. During the simulation we measure two metrics: the achieved user benefit and the incurred reconfiguration costs.

We run the same experiment with different values for the parameters α and h to analyze the effect on the two metrics. For each experiment we construct a set of 200 paths of 100 hops generated according to the probabilities of a fixed user context automata (Sec. 5.4). We then generate a fixed number of alternative system variants. For each variant we define randomly the eligible context states and the values of non functional properties. Each experiment consists of iterating through the context automaton according to the 200 predefined paths. At each state, we select the variant that maximizes Eq. 5.6. For each chosen variant, we log the current user benefit and reconfiguration cost. Finally, we evaluate the averages of the two metrics over all paths within a single experiment configuration. Then we repeat the experiment with the same paths sequences but varying α and h values. We then compare the results for different combinations of α and h. Setting the horizon h to 1 we simulate a future unaware reconfiguration strategy. There was no difference in the resulting cost and benefit trends for cost vectors $FCost = [2\ 1]$ and $FCost = [10\ 1]$; thus we report only the results for the former.

Figure 28(a) and 28(b) show the normalized user benefit and reconfiguration across 33 experiment configurations. Figure 28(b) compares



(a)



(b)

Figure 28: Normalized average user benefit (a) and normalized average reconfiguration cost (b) with $h = 1.0$, $h = 0.5$ and $h = 0.2$ depending on utility objectives weights α.

reconfiguration cost with three different values of h. Here we observe higher reconfiguration cost if we consider only the current user context state ($h = 1$). On the other hand variants are likely to change less frequently whenever we consider future user preferences. This holds if we consider current and future preferences equally ($h = 0.5$) as well as if we give more relevance to the future state ($h = 0.2$). We can conclude that looking into the future lowers the cost. As shown in the Figure we can reduce the reconfiguration cost by regulating h independent from α. Since α represents the weight for the aggregated user benefit (Eq. 5.5), it increases the significance of user benefit over the cost when it is close to 1.

Although we can reduce the reconfiguration cost by exploiting future user preferences, we potentially lower user benefit at the same time. A system variant that optimizes both current and future preferences does not necessarily maximize current user benefit. Figure 28(a) presents the difference of user benefit considering the static and predictive approach with value of h. We get the best average user benefit if we consider only the current user context state ($h = 1$) while we get lower values if we consider the future user preferences ($h = 0.2$ and $h = 0.5$).

Figure 28(a) and 28(b) suggest that if we consider the current and future user preferences the relative decrease in user benefit is smaller than the reduction of reconfiguration cost. As shown in the figures there is

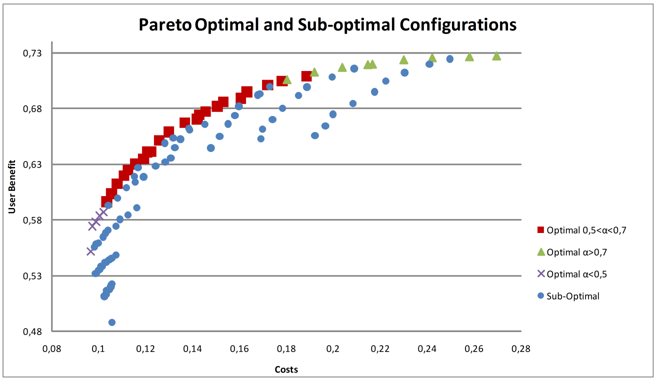


Figure 29: Pareto-optimal and sub-optimal configurations

potential user benefit without raising the cost. In addition, Figure 28(a) and 28(b) also suggest that within the user benefit component the parameter h regulates the benefit and cost objectives. In fact setting h closer to 1 increases the cost of reconfiguration in order to increment the benefit, whereas by setting h closer to 0 we partially alleviate the cost of reconfiguration by accepting lower user benefit configurations. The difference between α and h is that the horizon has a lower impact on the objectives compared to α.

Finally, we analyze the set of Pareto optimal configurations for $h = [0; 0.1; 0.2; \ldots 1]$ and $\alpha = [0; 0.1; 0.2; \ldots 1]$ for a total of 121 compared configurations. Pareto optimal points are roughly evenly distributed across h thus making it possible to select desirable values according to the specific application. We have discovered which range of α could be exploited to get most of the optimal points. In Figure 29, the Pareto optimal configurations are displayed following three different series; red squares stand for points in the range of $\alpha = [0.5; 0.7]$, crosses for optimal points for $\alpha = [0; 0.5]$ and triangles for $\alpha = [0.7; 1]$. Sub-optimal configurations are given in blue circles. As the configuration values are averaged over multiple transitions (as outlined above) also sub-optimal configurations close to the Pareto-optimal ones might me candidates. As shown in the figure we have noted that around 50% of optimal configurations lie in the range of $\alpha = [0.5; 0.7]$. We can thus conclude that too low α values put too much weight on costs and therefore waste a lot of potential to improve user benefit. Hence, our approach is able to realize considerable user benefit even in very cost-constrained environments. Pareto optimal configurations as shown in the figure help to decide how to set α while leaving to the designers the choice of h for specific ubiquitous applications.

The results demonstrate that predictive approaches ($h < 1$) allow the reduction of reconfiguration cost while providing an acceptable level of benefit to the user. We can conclude that our predictive approach is as good as non predictive approaches ($h = 1$) whenever we want to maximize the user benefit without focusing too much on cost. For cost-sensitive environments, a non-predictive approach fails to produce Pareto optimal points. Indeed, Pareto optimal points with $h = 1$ have high α values ($\alpha = [0.7; 1]$).

Chapter 6

Evolution framework

In this chapter we describe an evolution framework which supports the software lifecycle process for adaptive systems. This framework supports software designers and software developers in the activities of design, development and evolution of such systems. Since our focus is on the evolution mechanisms we describe how our framework implements the MAPE (Monitoring, Analyze, Plan, Execute) (BSG^{+}09) cycle in order to enact the evolution. Consecutively, we discuss two representations of our framework: a generic definition of its interfaces and a possible instantiation based on current practice technologies.

6.1 MAPE cyle

Our framework implements a MAPE cycle in order to supervise, execute and evolve an adaptive application. The monitoring phase activity collects information from the environment and from the user in order to establish if an evolution is required or not. On the one hand, foreseen context variations and user preferences variations may both enact foreseen evolutions. The firsts affect the admissibility for the system variants, whereas the seconds influence the fitness of the system variants. On the other hand, unforeseen context variations may force the user to introduce a new requirement into the running variant. The Analyze phase determines if the variant to adopt is consistent or not. In case of foreseen evolution we consider a pre-built data structure which records the consistency for each variant at each different context state. In case of unforeseen evolutions the analysis is performed at run-time by checking the consistency for the un-anticipated variant. This variant will contain the same set of features as the current variant plus a new feature that implements the new requirement specified by the user. After the Analyze phase the framework proceeds by planning the evolution. For the foreseen evolution, it performs a ranking procedure which establishes the most suitable variant based on context and user preferences. For the unforeseen evolution the framework simply puts forward the new variant which has been proven consistent at the Analyze step.

Finally the execution phase proceeds by switching from the current to the selected target variant. In case of an foreseen evolution the adaptive application invokes the entry point method for the target variant. In case of unforeseen evolution, the adaptive application incorporates a new code artifact and puts the new variant in execution by invoking its entry point method.

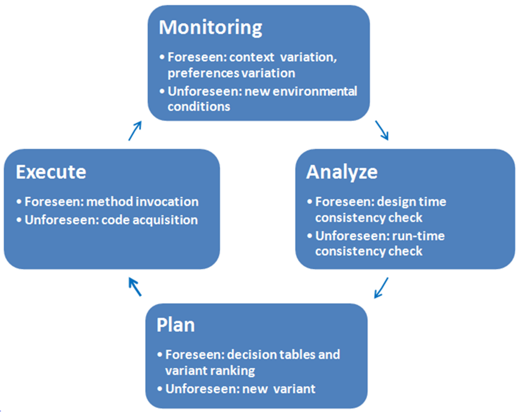

Monitoring
• Foreseen: context variation, preferences variation
• Unforeseen: new environmental conditions
Analyze
• Foreseen: design time consistency check
• Unforeseen: run-time consistency check
Plan
• Foreseen: decision tables and variant ranking
• Unforeseen: new variant
Execute
• Foreseen: method invocation
• Unforeseen: code acquisition

Figure 30: MAPE cycle

6.2 Framework interface architecture

In this Section we show a generic definition of our framework in terms of a set of interfaces and their relations. The objective of the interface architecture is to define the basis for creating instances of the generic framework. A concrete framework that supports the development and the evolution of adaptive applications should implement the interface architecture depicted in Figure 31. This architecture defines the operations whose implementations enact the MAPE cycle. Through the interfaces of the generic framework it is also possible to support the complete lifecycle process for adaptive applications.

The application, system variant and feature blocks represent the basic components. They enable the definition of the application along with its variability. The context manager component is able to monitor the resources and to manage their definitions and values by accessing to the context model component. It performs the monitoring phase and it triggers the required evolution phases. The decision-making component maintains the context-based tables and the probabilistic automaton in order to support the decision-making mechanisms; it also supports the consistency checking phase and the ranking process for the variants. A component for each kind of evolution is provided in the framework. As shown by the arrows, while the foreseen evolution accesses the decision-making component to select the most suitable variant, the unforeseen evolution interacts with the user who specifies variation to the requirements. Finally the execution component enacts the system reconfiguration for both evolutions.

6.3 Framework instance

In this section we describe a possible instantiation of the generic architecture with current practice technologies. We represent requirement R as Linear Time Temporal Logic expressions (Pnu77), whereas we represent the context requirements as predicates. We evaluate the context states in which a system variant is admissible by formalizing and solving a Con-

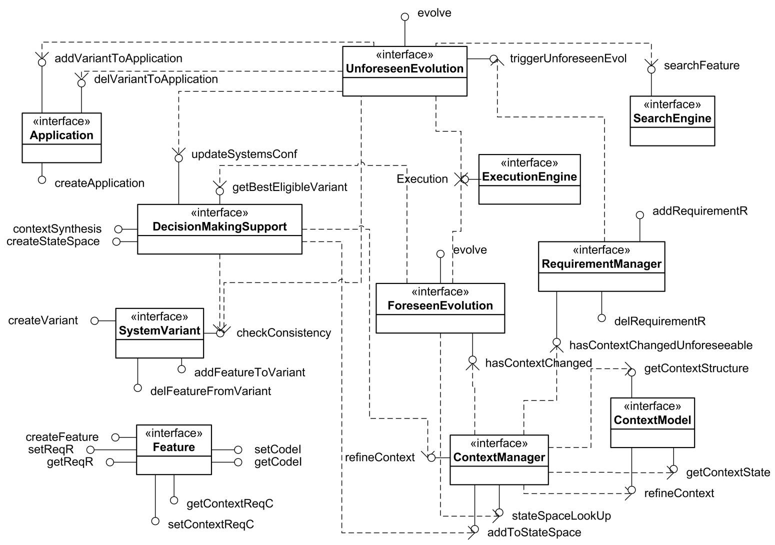


Figure 31: Evolution framework architecture

straint Satisfaction Problem (CSP) (MS98) by using the Java API available with the JaCoP tool[1]. We implement features in Java and we exploits the Java Path Finder tool in order to model check code artifacts with respect to their corresponding requirements R.

Our framework instance is implemented as a set of classes which materialize the interface architecture in Figure 31. Based on these classes we expose a set of web services which provide means to support the system variability in Section 6.3.2.

6.3.1 Description

First of all, a pool of classes supports the definition of the application along with its variants and features. The class implementing the interface $Feature$ contains a definition for a context-independent requirement, a context-dependent requirement and a code artifact. The class implementing the $SystemVariant$ interface supports the instantiation of a

[1]http://jacop.osolpro.com/

system variant starting from the set of its features. This class creates and maintains the components of the system R, I and C including the quality attributed that should be specified by the software engineer. Starting from the basic elements of each feature this class creates a context requirement as a union of clauses where each clause can express which is either the maximum, or the minimum or the exact value for a certain context entity. The class implementing the interface $Application$ manages the instance for the whole adaptive application.

The class implementing the $ContextManager$ interface maintains the set of possible context states. It contains the entities of the context model and their admissible assignments. The context entities are extracted by the context requirements entailed in each feature and consequently in each variant.

The class implementing the interface $DecisionMakingSupport$ is in charge of defining the space of the possible context states and in maintaining the data structures to support the variability. It exposes the method $CreateStateSpace$ to create the set of possible context states starting from the set of context entities and their assignments. The method evaluates the Cartesian product among the assignments for the context variables and it stores the set of states within the $ContextManager$ component. The method $contextSynthesis$ creates the table to support the evolution process. This method is in charge of checking the consistency for each variant. It exploits the space of context states maintained in the $ContextManager$ component and the instances of the adaptive application (features and variants instances). For each context requirement belonging to each variant, the method solves a constraint satisfaction problem to evaluate the set of context states that make the context requirement (predicate) true. Consequently we can easily obtain the list of admissible variants for each context state.

The class implementing the interface $ForeseenEvolution$ accesses the $DecisionMakingSupport$ component in order to evaluate which is the best variant to deploy among the ones that have been already proved consistent. The class implementing the interface $UnforeseenEvolution$ is able to build the new variant that implements a new requirement and to check its consistency without stopping the current execution.

6.3.2 Framework primitives

Starting from the classes of our framework instance we define a pool of primitives to instrument the framework and to support the evolution processes:

- InitializeFramework
- ForeseenEvolution
- UnforeseenEvolution

The developer can invoke these primitives within the application in order to provide it with adaptability. We describe how he can exploit these primitives for designing and developing an adaptive application.

The developer creates an adaptive application by defining a set of classes each representing a different feature. We assume that the developer is able to create the requirements and the context requirements belonging to each feature, thus she instantiates the feature objects by setting the triples (R, I, C). After defining each feature, the developer defines a class for each possible variant. Each class exposes a method $Execute$ which implements a different behavior by exploiting a different subset of feature classes. The variant also exposes the method $initVariant$ to record the set of features objects exploited by the variant. Finally, the set of system variants is given as input to the initialization of the framework.

The service $initializeFramework$ performs the operations that instrument the framework for supporting system variability. It takes as input the system variants of the adaptive application. For each variant, the service reads the set of its features (R, I, C) and it generates the requirements R and the context requirements C for the variant. Once the instances for the variants are create, the service calls the methods $createStateSpace$ and $contextSynthesis$ in order to generate the data structures that define the admissibility for each system variant at each possible context state.

In our instance of the framework the developer defines the adaptive application as a feedback loop which retrieves the current context and select the best variant. At each step of the loop the primitives for the evolution help to select the variant: either a new or an already known variant.

The $foreseenEvolution$ service provides a mechanism to retrieve a unique id for the best variant taking as input the current context values. It performs the selection of the variant based on the mechanism described in Chapter 5. To this end all the required input has to be provided to the framework at initialization phase (i.e. cost model and context-dependent user preferences).

The $unforeseenEvolution$ service searches for the code that implements a new requirement. It builds a new variant implementing the new requirement and it checks the consistency for the new variant. Finally, the service returns the definition for the new variant to the application. The adaptive application needs mechanisms to compile the new variant and to execute it. In section 6.4 we depict a possible mechanism to execute the new unforeseen variant.

6.4 Mandelbrot fractal application

In this section we show how to develop the adaptive fractal application by means of two different modalities: the first without the support of our framework, the second with the support of the framework. The goal of this section is to evaluate which is the benefit for the developer of either using or not using the primitives of the framework.

Since our intention is to assume that a monitor for the context already exists, we simulate a monitor by means of a set of context states with their time of validity as input to our adaptive applications. Applications will enact the possible required reconfigurations as a consequence of the context variations.

The developer has to create the set of software alternatives implemented each one as a different class (variant). Each variant exploits a different set of features which are implemented as Java classes. For the application developed with the support of the framework, the developer has to instantiate the feature objects with their requirement R and context requirement C. Further he has to record in each variant the features objects exploited by the variant. For the application created without the support of the framework, the developer manages directly the variability without specifying requirements and context requirements for each feature.

In both cases the application is implemented as a MIDlet[2] Java application for mobile devices.

6.4.1 Application without framework

In this section we represent a simple approach for developing an adaptive application that visualizes the Mandelbrot fractal. This mobile application visualizes the most appropriate fractal based on the current context values. Listing 6.1 shows the main class $MandelFractalMIDlet$ along with its entry point method $startApp$ which is executed as soon as the mobile application is started. This procedure instantiates the set of predefined variants each one performing the visualization of the fractal following a different modality for building and coloring the image; e.g. Listing 6.3 shows a possible variant which visualizes the fractal image at the end of the drawing process and it colors the pixels with a limited number of tones. After instantiating the variants the procedure $startApp$ iterates through a loop in order to read the current context values and to perform the consequent adaptation. It reads from the predefined sequence of context states which is the current assignment for the context resources and their validity time. Based on these values, the procedure checks through a series of "if-then" controls which variant has to be selected. Our procedure selects a variant by calling its method $Execute$. After that the validity time for the current context state expires, the procedure continues by looking at the next context values and by performing the next adaptation.

[2]http://www.oracle.com/technetwork/java/javame/javamobile/overview/getstarted/index.html

Listing 6.1: Mandelbrot fractal application (without framework)

```
public class MandelFractalMIDlet extends MIDlet {
 ContextEntities c;
 States stati = new States();

 protected void startApp(){
  int a = 0 ;
  int cycles;
  cycles =stati.getNumStates();
  Variant1 var1 = new Variant1();
  Variant2 var2 = new Variant2();
  Variant3 var3= new Variant3();
  Variant4 var4 = new Variant4();
  Variant5 var5 = new Variant5();
  Variant6 var6 = new Variant6();
  Variant7 var7 = new Variant7();
  Variant8 var8 = new Variant8();
  Variant9 var9 = new Variant9();
  Variant10 var10 = new Variant10();
  while (a<cycles){
   c = stati.getState();
   if ((c.getMemory()>=120)&&(c.getTcRate()>=500)&&(c.getNcolors()>=4096)){
    var9.Execute(this);
   }else if ((c.getMemory()>=120)&&(c.getTcRate()>=300)&&
            (c.getNcolors()>=4096)){
    var8.Execute(this);
   }else if ((c.getMemory()>=120)&&(c.getTcRate()>=100)){
    var7.Execute(this);
   }else if ((c.getMemory()>=200)&&(c.getTcRate()>=500)&&
            (c.getNcolors()>=4096)){
    var6.Execute(this);
   }else if ((c.getMemory()>=200)&&(c.getTcRate()>=300)&&
            (c.getNcolors()>=4096)){
    var5.Execute(this);
   }else if ((c.getMemory()>=200)&&(c.getTcRate()>=100)){
    var4.Execute(this);
   }else if ((c.getMemory()>=300)&&(c.getTcRate()>=500)&&
            (c.getNcolors()>=4096)){
    var3.Execute(this);
   }else if ((c.getMemory()>=300)&&(c.getTcRate()>=300)&&
            (c.getNcolors()>=4096)){
    var2.Execute(this);
   }else if ((c.getMemory()>=300)&&(c.getTcRate()>=100)){
    var1.Execute(this);
   }else if ((c.getMemory()>=100)&&(c.getNetwork()==1)){
    var10.Execute(this);
   }else{
    var1.Execute(this);
   }
   try{Thread.currentThread().sleep(c.getLife());}
   catch(Exception ie){}
   a++;
  }
}
```

6.4.2 Application with framework

The framework is accessible through three web services: $inizializeFramework$ initializes the framework to manage the variability for a certain application, $foreseenEvolution$ and $unforeseenEvolution$ perform the foreseen and unforeseen evolution.

Listing 6.2 shows how our framework can support the variability of the fractal application. The main class for the adaptive application instantiates each variant and performs the call to the web service which initializes the framework. The initialization service takes as input the system variants containing the feature objects. Starting from the information of each feature, the service produces the context requirements and the context-independent requirements for the whole set of variants. Based on these requirements the service generates the data structures to establish which context state can support the execution of each variant.

After initializing the framework, the application iterates through the set of context states and it performs the consequent adaptations. At each cycle of the loop we establish if it is necessary to perform either a foreseen evolution or an unforeseen evolution. Within each context state we have coded an integer variable in order to notify if an unforeseen evolution is required or not (either 1 or 0), i.e. the application has to be augmented with a new requirement. In our framework we assume that a new requirement can be identified if it exists, thus we do not take into consideration how to solve the problem of eliciting a new requirement at run-time.

If an unforeseen evolution is not required, the procedure calls the service $foreseenEvolution$ to obtain the most suitable variant to select in the current context state. This service accesses the data structures in order to know which variants are eligible with the current context values and it returns as result the most suitable variant. Consequently the application selects the variant by calling the method $Execute$. Differently from the application without the framework, this application does not have to check each single context values; the framework maintains the data structure to perform this check and the information to solve the static decision-making problem.

For the unforeseen evolution case we have statically defined a variant called $VariantNew$ that provides only the definition for the method $Execute$. At run-time we can augment the behavior of the application by modifying the implementation of this variant. Whenever an unforeseen evolution is required, the application performs a set of operations to augment its behavior with a new variant. First we call the service $unforeseenEvolution$ which searches for a new feature which implements the new requirement. If there is no variant that can satisfy the augmented set of requirements, the service creates a new variant that contains the same set of features of the current variant plus a new feature implementing the new requirement. The service checks if the new set of features (new variant) is free from interactions by evaluating context-dependent requirements in the current context and by model checking canonical requirements on code. Once the service has found such a new feature, it gives as result a new implementation for the class $VariantNew$ and the class implementing the new feature. This new variant has to be compiled and it has to be added on-line to the running application. Thus our procedure calls the method $ReplaceVariantUnforeseen$ to replaces the code for the empty variant $VariantNew$; the new definition of the variant implements the method $Execute$ which is empty in the first version of the class. After replacing the Java code, the procedure calls the method $CompileJavaFile$ in order to launch the commands to compile the Java code for the new variant and for the new feature. Once the compiling operation is completed our procedure forces the JVM (Java Virtual Machine) to reload the new compiled classes. The method $ReloadRunTimeSupport$ triggers the reloading phase by exploiting the Javeleon tool[3]. Finally the procedure enacts the execution for the new replaced variant.

After the selection of a variant the application interrupts its execution for the given amount of time which corresponds to the time of validity of the current context state.

[3]http://javeleon.org/

Listing 6.2: Mandelbrot fractal application (with framework)

```
public class SelfApplication extends MIDlet implements SelfApp{
 private long processingTime = 0;
 ContextEntities c;
 States stati = new States();
 private Service_PortType_Stub service;
 public static SelfApplication instance;
 ...
 protected void startApp(){
  int a = 0;
  int cycles = 0;
  int chooseVariant=0;
  Vector variantsInput = new Vector();
  service = new Service_PortType_Stub();
  service._setProperty(Service_PortType_Stub.SESSION_MAINTAIN_PROPERTY,
  new Boolean(true));
  cycles = stati.getNumStates();
  try {
   Variant1 var1 = new Variant1();
   Variant2 var2 = new Variant2();
   Variant3 var3= new Variant3();
   Variant4 var4 = new Variant4();
   Variant5 var5 = new Variant5();
   Variant6 var6 = new Variant6();
   Variant7 var7 = new Variant7();
   Variant8 var8 = new Variant8();
   Variant9 var9 = new Variant9();
   Variant10 var10 = new Variant10();
   VariantUnforeseen varNEW = new VariantUnforeseen();
   variantsInput.addElement(var1);
   ...
   variantsInput.addElement(var10);
   service.initializeFramework(variantsInput));
   while (a<cycles){
    c = stati.getState();
    if (c.getUnforeseen()==0){
     chooseVariant = service.foreseenEvolution(
   c.getMemory(),c.getTcRate(),c.getNcolors(),c.getNetwork());
     if (chooseVariant==1){
      var1.Execute(this);
     }else if (chooseVariant==2){
      var2.Execute(this);
     }else if (chooseVariant==3){
      var3.Execute(this);
     }else if (chooseVariant==4){
      var4.Execute(this);
     }else if (chooseVariant==5){
      var5.Execute(this);
     }else if (chooseVariant==6){
      var6.Execute(this);
     }else if (chooseVariant==7){
      var7.Execute(this);
     }else if (chooseVariant==8){
      var8.Execute(this);
     }else if (chooseVariant==9){
      var9.Execute(this);
     }else if (chooseVariant==10){
      var10.Execute(this);
     }
```

```
      }else{
        Result newCode = service.unforeseenEvolution("NewReq");
        ReplaceVariantUnforeseen(newCode);
        CompileJavaFiles(newCode);
        ReloadRunTimeSupport();
        varNEW.Execute(this);
      }
      Thread.currentThread().sleep(c.getLife());
      a++;
    }
  }catch (Exception e) {
    System.out.println("Error"+ e.getMessage());
 }
 ...
}
```

Listing 6.3: Variant definition

```
public class Variant1 implements VariantType{
  ArrayList features;
  MandelCanvasType mandelType=null;
  private long processingTime=0;
  String maxExecutionTime = "300000";
  Display currentDisplay;

  public void Execute(Object caller){
    processingTime = System.currentTimeMillis();
    mandelType = (MandelCanvasType) new MandelCanvasAsShot(
                  new ColouringAsBands());

    mandelType.setMaxExecutionTime(Long.parseLong(maxExecutionTime));
    mandelType.setStartTime(processingTime);

    currentDisplay = ((Display)Display.getDisplay((SelfApplication)caller));
    currentDisplay.setCurrent((Displayable)mandelType);
    mandelType.generateFractal();
  }
  public initVariant(ArrayList initFeatures){
    features = initFeatures;
  }
 ...
}
public class MandelCanvasAsShot extends Canvas implements MandelCanvasType{
  ...
  public void generateFractal() {
    ...
  }
  ...
}
public class ColouringAsBands implements ColouringType {
  ...
  public int pixelColor(boolean inner, int iteration, double dist) {
    ...
  }
  ...
}
```

6.4.3 Evaluation

In both application the developer defines the set of software alternative (variants) each one exploiting a set of classes (features). The first application does not exploit information about context requirements and context-independent requirements of each feature in defining the logic of adaptation. The developer establishes directly which variant to execute based on the context values. In the second application, the developer creates the feature objects entailing the information about the requirements specifications. Based on these information, the framework evaluates which variant to adopt at each context state. Thus, the developer does not directly implement the logic to select the best variant but he simply query the web service that is in charge of finding the most appropriate variant. The developer may also establish a ranking for the variant thus to solve conflicts in case of the presence of multiple eligible variants. To this end, in Chapter 5 we have shown how to consider dynamic user preferences to solve the decision-making problem.

The application supported by the framework is also able to perform reconfigurations that are not statically provided at run-time. Indeed, we have shown a possible mechanism on how to add a variant at run-time while preserving the consistency of the application. The definition of the variant is statically defined at design time whereas its implementation is completed at run-time with the support of the framework. We could have adopted the same mechanism for applying new variant at run-time even for the application that does not exploit the framework. Nevertheless, in this case the software engineer would have had to implement the procedure to check the consistency by hand.

We have shown that it is possible to design and develop an adaptive application with the support of the framework. The developer can exploit our support for creating the application and for checking its consistency while performing foreseen and unforeseen evolutions.

By adopting our framework, the developer can create the logic of evolution in a easier manner than without the framework. This logic is based on the context values that make a software alternative admissible or not.

Considering a set of n features, the developer without the framework may possibly have to define the context conditions for 2^n potential variants. For high-configurable systems with a big number of features, setting these conditions could be unfeasible by hand. The framework only asks the developer to set the conditions for the features, whereas it creates the requirements of the variants.

Starting from the same set of context states we have ran the two versions of the adaptive application. By observing the reconfigurations of the system with and without the framework we claim that the cost of adopting the framework is negligible.

Chapter 7

Conclusion

In this chapter we point out the main contributions of the thesis. We give the answers for the research question identified in Chapter 1.

- RQ1: How to manage context-dependent system variability? Which abstractions for the system can better handle variability and how could they support an automatic decision-making procedure?

  The thesis proposes a system notation based on the SPL engineering perspective. Our notation breaks the system complexity in single units of behavior (features) and it allows the definition of system variants which represent software alternatives suited for different contexts. We define the application as a set of variants each one characterized by a set of context-independent requirements, context-dependent requirements and implementation artifacts. Our approach defines the context based on key-value pairs that can assume integer values over finite domains. Through specific data structures that we call *decision tables* it is possible to map each variant to the contexts in which the variant can be adopted. At run-time our framework retrieves the current context values and it checks which system variants can be selected based on their context requirements. Among them the framework selects the system variant which optimizes cost and user benefit taking into account current and future information on context values and user preferences. To this end, the framework implements a decision-making process by exploiting a probabilistic model which describes probable future context variations.

- RQ2: How to classify context-dependent evolutions?

  We propose two different kinds of evolution namely foreseen and unforeseen evolution. In the literature most of approaches facing the evolution of adaptive systems only provide support for the foreseen evolution. It is challenging to provide solutions for achieving unforeseen evolutions since these evolutions aim to face the unpredictability of modern ubiquitous software systems. The case studies presented in the thesis show that it is valuable to provide the system with the capability of enhancing the behavior at run-time.

- RQ3: How to represent requirements models and their evolutions at run-time?

  Requirements specification entails two different portions. The first portion contains information that are independent from the context whereas the second portion contains information related to the context entities. On the one hand context-dependent requirement are represented as predicates over context entities. At design time they are checked in order to evaluate the scope of validity for alternative system variants whereas at run-time they are checked to assess the validity for an enhanced variant augmented with new requirements. On the other hand context-independent requirements are checked on the code artifact to discover inconsistencies at code level. At design time they are checked by means of a model checking procedure in order to assess which variants have a code implementation that does not satisfy its context-independent requirement. At run-time the framework performs the model-checking procedure for enhanced context-independent requirements with respect to enhanced code artifacts.

- RQ4: How the software lifecycle process should deal with the uncertainty coming from the environment?

  The thesis proposes a software lifecycle process for adaptive systems which faces foreseen and unforeseen context variations. The process supports the creation of adaptive applications along with its variability. It supports the creation of basic features, their composition in variants and the analysis of the variants. We have built a framework that implements portion of the process and supports the developer in creating adaptive applications. At design time the framework analyses a set of designed variants while at run-time it checks a new variant which implements a new unforeseen requirement. This variation of requirements is caused by unforeseen context variations that cannot be predicted at design time. The user is involved in the process of enhancing system behavior since she identifies new requirements for unpredicted contexts. The unforeseen evolution allows the software engineers to deal with future scenarios that are not identified at design time.

- RQ5: How to perform reconfigurations taking into account competing objectives?

  A generic framework which supports variability should be able to perform optimized reconfigurations which take into account multiple sources of information together. To this end we propose a process for selecting the most suitable reconfiguration by considering current and probable future context information. Results show that anticipating upcoming reconfiguration needs and considering multiple factors all together promote better performances for the reconfiguration process.

Appendix A

Variants examples

Listing A.1: Example: variant eHealth application

$R_{EHealth} : R_{graphOx} \cup_R R_{textOx} \cup_R R_{getOxData}$

$I_{EHealth}$:

```
public class VariantEHealth{
 static Graph myGraphViewer;
 static Text myTextViewer;
 public static void execute(){
  myGraphViewer =   new Graph();
  myTextViewer = new Text();
  GraphOximetryViewer graphOx = new GraphOximetryViewer();
  TextOximetryViewer textOx = new TextOximetryViewer();
  graphOx.viewGraphicalOximetry(myGraphViewer);
  textOx.viewTextualOximetry(myTextViewer);
 }
 ...
}

public class GraphOximetryViewer{
 XYDataset oximetryDataset = new XYSeriesCollection();
 ...
 public void viewGraphicalOximetry(Graph g){
  ...
  for(int i = 0;i<10;i++){
   XYDataItem dataOx = OximetryRetrieving.getOximetryData();
   dataVectOx.add(dataOx);
  }
  g.displayGraph(dataVectOx);
 }
 ...
}

public class TextOximetryViewer {
 ...
 public void viewTextualOximetry(Text myTextViewer) {
  XYDataItem dataOx = OximetryRetrieving.getOximetryData();
  myTextViewer.displayText(dataOx.getYValue());
 }
 ...
}

public class OximetryRetrieving{
 ...
 public static XYDataItem getOximetryData(){
  try {
   socket = (StreamConnection) Connector.open(connectionURL,
   Connector.READ_WRITE);
  }catch (Exception ex){
   System.out.println("Err.Open.Conn.To": +connectionURL);
   System.out.println(ex);
   }
  ...
  \* Get Oxygenation Data oxData*\
  ...
  DataOxymetryMeM.add(OxData);
  return oxData;
 }
 ...
}
```

$C_{EHealth} : mem \geq 70 \ \wedge \ cRate \geq 1100 \wedge$
$\quad oxygenationProbe = true \ \wedge \ conn = 1 \ \wedge \ b \geq 20$

Listing A.2: Example: variant Mandelbrot fractal application

$R_{Fractal} : R_{genPro} \cup_R R_{colB}$

$I_{Fractal}$

```
public class VariantFractal extends MIDlet {
 MandelCanvas mandelCanvas;
  ...
 public AppFractal(){
  mandelCanvas = new MandelCanvas();
  }
 protected void execute(){
  currentDisplay = Display.getDisplay(this);
  currentDisplay.setCurrent(mandelCanvas);
  mandelCanvas.generateProgressiveFractal();
  exitAction();
  }
  ...
}

public class MandelCanvas extends Canvas {
  ...
 public void generateProgressiveFractal(){
  int column_ArrayCanvas[] = new int[height];
  for (int x = 0; x < width; x++){
   for (int y = 0; y < height; y++){
    FractalPixel pixel_ArrayCanvas = drawFractalPixel(x, y);
    column_ArrayCanvas[y] = pixelColor(pixel_ArrayCanvas.isInsideFractal(),
    pixel_ArrayCanvas.getIterations(),pixel_ArrayCanvas.getDistance());
   }
```

```
    offsetX = x;
    image = Image.createRGBImage(column_ArrayCanvas, 1, height, false);
    repaint();
   }}
  ...
}

public class Colouring{
  ...
 private int pixelColourAsBands(boolean interno, int iterazioni,
 double dist){
  int tmp= (interno ? 0 : colors[iterazioni % paletteNumColors]);
  return tmp;
 }
 private void initColourAsBands(){
  int[] tmpColors_IterationsLimitedPalette = {-256, -16711681,-65281,-256,
  -4194304, -16728064, -16777024, -8323073, -32513, -128};
  colors = tmpColors_IterationsLimitedPalette;
  paletteNumColors = colors.length;
 }
 ...
}
```

$C_{Fractal} : mem \geq 200 \ \wedge \ cRate \geq 100$

Listing A.3: Example: system variant

$R_{GNew} = R_{graphOxygen} \cup_R R_{graphRespRate} \cup_R \ldots =$
$[]((GraphOxViewer.viewGraphOx(Graph) \rightarrow (<> GraphOxViewer.outcome))\wedge$
$(GraphRespRViewer.viewGraphRespR(Graph) \rightarrow (<> GraphRespRViewer.outcome)))$
$\cup_R \ldots$

$I_{GNew} = I_{graphOxygen} \cup_I I_{graphRespRate} \cup_I \ldots =$

```
public class VariantGNew{
 static Graph myGraphViewer;
 public static void Execute() throws Exception{
  myGraphViewer =   new Graph();
  GraphOxViewer graphOx =new GraphOxViewer();
  GraphRespRViewer graphRr = new GraphRespRViewer();
  graphOx.viewGraphOx(myGraphViewer);
  graphRr.viewGraphRespR(myGraphViewer);
 }
 ...
}

public class GraphOxViewer{
 boolean outcome=false;
 private static Exception propertyViolation;
 ...
 public void viewGraphOx(Graph g) throws Exception{
   ...
  for(int i = 0;i<10;i++){
   XYDataItem dataOx = OximetryRetr.getOximetryData();
   dataVectOx.add(dataOx);
  }
  g.displayGraph(dataVectOx);
  outcome = Checker.Check(g.currData, dataVectOx);
  if (!outcome){throw propertyViolation;}
 }
 ...
}

public class GraphRespRViewer {
 boolean outcome=false;
 private static Exception propertyViolation;
 ...
 public void viewGraphRespR(Graph g) throws Exception{
   ...
  for(int i = 0;i<10;i++){
   XYDataItem dataRespR = RespRRetr.getRespRData();
   dataVectRespR.add(dataRespR);
  }
  g.displayGraph(dataVectRespR);
  outcome = Checker.Check(g.currData, dataVectRespR);
  if (!outcome){throw propertyViolation;}
 }
 ...
}
```

References

[ABI09] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. Context-aware adaptive services: The plastic approach. In *FASE*, pages 124–139, 2009. 27, 30, 39, 61

[ABI10] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. A programming model for adaptable java applications. In *PPPJ*, pages 119–128, 2010. 20, 61

[ABI12] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. Hybrid approach for resource-based comparison of adaptable java applications. In *Journal of Science of Computer Programming (SCP) - Special issue of BElgian-NEtherlands software eVOLution seminar (BENEVOL) on Software Evolution, Adaptability and Maintenance*, 2012. 36, 39, 40, 43

[ABIM08] Marco Autili, Paolo Di Benedetto, Paola Inverardi, and Fabio Mancinelli. A resource-oriented static analysis approach to adaptable java applications. In *COMPSAC*, pages 1329–1334, 2008. 36, 43

[ADG10] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. A goal-based framework for contextual requirements modeling and analysis. *Requir. Eng.*, 15(4):439–458, 2010. 28

[AdLMW09] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling dimensions of self-adaptive software systems. In *SEAMS*, pages 27–47, 2009. 1, 9

[AdRI$^+$11] M. Autili, D. di Ruscio, P. Inverardi, P. Pelliccione, M. Tivoli, and V. Cortellessa. EAGLE: Engineering softwAre in the ubiquitous Globe by Leveraging uncErtainty. *new ideas track esec*, 2011. 60

[AMK$^+$09] Mauricio Alférez, Ana Moreira, Uirá Kulesza, João Araújo, Ricardo Mateus, and Vasco Amaral. Detecting feature interactions in spl requirements analysis models. In *FOSD*, pages 117–123, 2009. 26, 34

[AS01] Anthony Finkelstein Andrea and Andrea Savigni. A framework for requirements engineering for context-aware services. In *STRAW 01*, pages 200–1, 2001. 15

[BBF09] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *IEEE Computer*, 42(10):22–27, 2009. 27

[BC04a] Jesus Bisbal and Betty H. C. Cheng. Resource-based approach to feature interaction in adaptive software. In *WOSS*, pages 23–27, 2004. 34

[BC04b] H.E. Byun and K. Cheverst. Utilizing context history to provide dynamic adaptations. *Applied Artificial Intelligence*, 18(6), 2004. 11

[BCM10] Luca Berardinelli, Vittorio Cortellessa, and Antinisca Di Marco. Performance modeling and analysis of context-aware mobile software systems. In *FASE*, pages 353–367, 2010. 67

[BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *IJAHUC*, 2(4):263–277, 2007. 1, 24

[BHRE07] Gunnar Brataas, Svein O. Hallsteinsen, Romain Rouvoy, and Frank Eliassen. Scalability of decision models for dynamic product lines. In *SPLC (2)*, pages 23–32, 2007. 62

[BKS03] T. Buchholz, A. Küpper, and M. Schiffers. Quality of context: What it is and why we need it. In *Proceedings of the Workshop of the HP OpenView University Association*, 2003. 23

[BSG$^+$09] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009. 90

[CCYC06] Ronnie Cheung, Jiannong Cao, Gang Yao, and Alvin T. S. Chan. A fuzzy-based service adaptation middleware for context-aware computing. In *EUC*, pages 580–590, 2006. 24

[CdLG$^+$09] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, 2009. 2, 14, 26

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: Methods, Tools and Applications*. Addison-Wesley, 2000. 45

[CFJ04] H. Chen, T. Finin, and A. Joshi. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03):197–207, 2004. 13

[Cho07] Jongmyung Choi. Context-driven requirements analysis. In *ICCSA (3)*, pages 739–748, 2007. 16, 39

[CHS$^+$08a] 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. 34

[CHS08b] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *FASE*, pages 16–30, 2008. 40, 49

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ICSE*, pages 321–330, 2011. 26

[CLZ98] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination. In *Mobile Agents*, pages 237–248, 1998. 19

[CMM09] Gianpaolo Cugola, Alessandro Margara, and Matteo Migliavacca. Context-aware publish-subscribe: Model, implementation, and evaluation. In *ISCC*, pages 875–881, 2009. 19

[CPGS09] Shang-Wen Cheng, Vahe Poladian, David Garlan, and Bradley R. Schmerl. Improving architecture-based self-adaptation through resource prediction. In *SEAMS*, pages 71–88, 2009. 30

[CPvS05] Patricia Dockhorn Costa, Luís Ferreira Pires, and Marten van Sinderen. Architectural patterns for context-aware services platforms. In *IWUC*, pages 3–18, 2005. 17

[CT11] Paolo Ciaccia and Riccardo Torlone. Modeling the propagation of user preferences. In *ER*, pages 304–317, 2011. 77

[CXC$^+$05] Jiannong Cao, Na Xing, Alvin T. S. Chan, Yulin Feng, and Beihong Jin. Service adaptation using fuzzy theory in context-aware mobile computing middleware. In *RTCSA*, pages 496–501, 2005. 24

[DD10a] Christoph Dorn and Schahram Dustdar. Interaction-driven self-adaptation of service ensembles. In *CAiSE*, pages 393–408, 2010. 30

[DD10b] Christoph Dorn and Schahram Dustdar. Weighted fuzzy clustering for capability-driven service aggregation. In *SOCA*, pages 1–8, 2010. 62

[Dey01] Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001. 11

[DVC$^+$07] Brecht Desmet, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, and Theo D'Hondt. Context-oriented domain analysis. In *CONTEXT*, pages 178–191, 2007. 15, 16, 39

[FGT11] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *ICSE*, pages 341–350, 2011. 26

[GCH$^+$04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004. 27, 30

[GIM08] Carlo Ghezzi, Paola Inverardi, and Carlo Montangero. Dynamically evolvable dependable software: From oxymoron to reality. In *Concurrency, Graphs and Models*, pages 330–353, 2008. 2, 3

[GJ09] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic update of java applications - balancing change flexibility vs programming transparency. *Journal of Software Maintenance*, 21(2):81–112, 2009. 28

[Gli07] Martin Glinz. On non-functional requirements. In *RE*, pages 21–26, 2007. 38

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008. 20

[HCS05] Dan Hong, Dickson K. W. Chiu, and Vincent Y. Shen. Requirements elicitation for the design of context-aware applications in a ubiquitous environment. In *ICEC*, pages 590–596, 2005. 15

[HHP$^+$08] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, and T. Sintefict. Dynamic software product lines. *IEEE Computer*, 41(4):93–95, 2008. 34

[HI06] Karen Henricksen and Jadwiga Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64, 2006. 20

[HIMB05] Karen Henricksen, Jadwiga Indulska, Ted McFadden, and Sasitharan Balasubramaniam. Middleware for distributed context-aware systems. In *OTM Conferences (1)*, pages 846–863, 2005. 22

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *STTT*, 2(4):366–381, 2000. 55

[HSK09] Jongyi Hong, Euiho Suh, and Sung-Jin Kim. Context-aware systems: A literature review and classification. *Expert Syst. Appl.*, 36(4):8509–8522, 2009. 1

[IT08] Paola Inverardi and Massimo Tivoli. The future of software: Adaptation and dependability. In *ISSSE*, pp 1–31, 2008. 2, 3

[Jac00] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000. 40, 49

[KCH$^+$90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. *Technical report CMU/SEI-90-TR-21 SEI Carnegie Mellon University*, 1990. 44

[Kjæ07] Kristian Ellebæk Kjær. A survey of context-aware middleware. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 148–155, 2007. 23

[KK98a] Dirk O. Keck and Paul J. Kühn. The feature and service interaction problem in telecommunications systems. a survey. *IEEE TSE*, 24(10):779–796, 1998. 33

[KK98b] Dirk O. Keck and Paul J. Kühn. The feature and service interaction problem in telecommunications systems. a survey. *IEEE TSE*, 24(10):779–796, 1998. 34

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Software Eng.*, 16(11):1293–1306, 1990. 60

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE*, pages 259–268, Washington, DC, USA, 2007. 28

[KMK+03] P. Korpipää, J. Mantyjarvi, J. Kela, H. Keranen, and E.J. Malm. Managing context information in mobile devices. *IEEE pervasive computing*, 2(3):42–51, 2003. 14

[KPTV09] Georgia M. Kapitsaki, George N. Prezerakos, Nikolaos D. Tselikas, and Iakovos S. Venieris. Context-aware service engineering: A survey. *JSS*, 82(8), 2009. 1

[KR03] Roger Keays and Andry Rakotonirainy. Context-oriented 3. In *MobiDE*, pages 9–16, 2003. 21

[KSS06] Andreas Krause, Asim Smailagic, and Daniel P. Siewiorek. Context-aware mobile computing: Learning context-dependent personal preferences from a wearable sensor array. *IEEE Trans. Mob. Comput.*, 5(2):113–127, 2006. 79

[LRT+10] F. Li, K. Rasch, H.L. Truong, R. Ayani, and S. Dustdar. Proactive service discovery in pervasive environments. In *ICPS*, pages 126–133, 2010. 30

[Man82] B.B. Mandelbrot. *The fractal geometry of nature*. Freeman, 1982. 36

[MKUM09] Paulo Henrique M. Maia, Jeff Kramer, Sebastián Uchitel, and Nabor C. Mendonça. Towards accurate probabilistic models using state refinement. In *ESEC/FSE*, pages 281–284, 2009. 79

[MM07] Antinisca Di Marco and Cecilia Mascolo. Performance analysis and prediction of physically mobile systems. In *WOSP*, pages 129–132, 2007. 67

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: An introduction*. MIT Press, 1998. 93

[Ost87] L. Osterweil. Software processes are software too. In *ICSE*, pages 2–13, Los Alamitos, CA, USA, 1987. 3

[PBCD11] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Sci. Comput. Program.*, 76:1247–1260, 2011. 34

[PBD09] Carlos Andres Parra, Xavier Blanc, and Laurence Duchien. Context awareness for dynamic service-oriented product lines. In *SPLC*, pages 131–140, 2009. 34

[PCBD10] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-based composition of software architectures. In *Proceedings of the 4th European conference on Software Architecture*, ECSA'10, pages 230–245, 2010. 34, 49

[PGS$^+$07] Vahe Poladian, David Garlan, Mary Shaw, M. Satyanarayanan, Bradley Schmerl, and Joao Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *SASO*, pages 214–223, Washington, DC, USA, 2007. 30, 67

[PGS$^+$11] Mario Pukall, Alexander Grebhahn, Reimar Schröter, Christian Kästner, Walter Cazzola, and Sebastian Götz. Javadaptor: unrestricted dynamic software updates for java. In *ICSE*, pages 989–991, 2011. 28

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977. 55, 92

[PSGS04] Vahe Poladian, João Pedro Sousa, David Garlan, and Mary Shaw. Dynamic configuration of resource-aware services. In *ICSE*, pages 604–613, 2004. 30

[QP10] N.A. Qureshi and A. Perini. Requirements Engineering for Adaptive Service Based Applications. In *RE*, pages 108–111, 2010. 27

[SB05] Q.Z. Sheng and B. Benatallah. ContextUML: A UML-Based Modeling Language for Model-Driven Development of Context-Aware Web Services Development. In *International Conference on Mobile Business*, page 212, 2005. 12

[SBW$^+$10] Peter Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *RE*, pages 95–103, 2010. 4, 27

[SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *SAC*, pages 431–438, 2010. 30

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007. 44

[SLP04] T. Strang and C. Linnhoff-Popien. A context modeling survey. In *Advanced Context Modelling Reasoning and Management as part of UbiComp*, pages 1–8, 2004. 12, 13

[SSLRM11] Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *SEAMS*, pages 60–69, 2011. 16

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *TAAS*, 4(2), 2009. 1

[TGDB06] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In *Software Composition*, pages 227–242, 2006. 21

[VBAM09] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced runtime adaptation for java. In *GPCE*, pages 85–94, 2009. 22

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Software Eng.*, 33(12):856–868, 2007. 60

[Win01] T. Winograd. Architectures for context. *Human-Computer Interaction*, 16(2):401–419, 2001. 19

[WSB$^+$09] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *RE*, pages 79–88, 2009. 16

[ZG02] Didar Zowghi and Vincenzo Gervasi. The three cs of requirements: Consistency, completeness, and correctness. In *REFSQ*, 2002. 29