Software Adaptation in an Open Environment
A Software Architecture Perspective

Yu Zhou and Taolue Chen
# Contents

Foreword ...................................................... ix
Preface ....................................................... xi
Authors ....................................................... xv
List of Figures .............................................. xvii
List of Tables .............................................. xxi

SECTION I: BASICS AND FRAMEWORK 1

1 Introduction .............................................. 3
   1.1 Novel Computing Paradigms and Methodologies ........ 6
   1.2 What is Self-Adaptation? ............................... 16
   1.3 What is Context? ....................................... 19
   1.4 Challenges of Adaptation in an Open Environment ...... 20
       1.4.1 Characteristics of the open environment .......... 21
       1.4.2 Adaptation requirements .......................... 22
   1.5 Structure of the Book .................................. 24

2 Adaptation Framework ................................. 27
   2.1 Introduction and Background ......................... 28
       2.1.1 Architecture description languages .............. 29
       2.1.2 Software architectural views .................... 30
       2.1.3 Software architecture dynamics ................ 30
## Contents

2.1.4 Discussions .............................................. 32
2.2 Conceptual Adaptation Framework ............................. 32
  2.2.1 Architectural perspective .............................. 32
  2.2.2 Framework overview ................................... 35
2.3 Runtime Software Architecture ............................... 38
  2.3.1 Software architecture class model ....................... 38
  2.3.2 Reflective interaction .................................. 40
  2.3.3 Discussions ............................................. 41
2.4 Related Techniques for Self-Adaptation ....................... 42
2.5 Summary .................................................. 44

3 Context Modeling ............................................. 47
  3.1 Overview of Context Modeling .............................. 48
  3.2 Representative Ontology Models ........................... 50
  3.3 Ontology-Based Context Modeling ......................... 52
  3.4 Ontology Models for Architecture Domain .................. 54
    3.4.1 Architecture description ontology ..................... 55
    3.4.2 Architecture manipulation ontology .................... 56
    3.4.3 Architecture control ontology ........................ 58
    3.4.4 Discussion ............................................. 61
  3.5 Summary .................................................. 61

4 Implementation and Case Study ................................. 63
  4.1 Structural Overview ...................................... 64
    4.1.1 Adaptation support .................................... 65
    4.1.2 Component framework .................................. 66
    4.1.3 Context knowledge management ......................... 68
  4.2 MAC-ng Implementation ................................... 68
  4.3 Performance Analysis ..................................... 72
    4.3.1 Experimental setup .................................... 73
    4.3.2 Performance evaluation ................................ 73
  4.4 Case Study ............................................... 76
    4.4.1 Scenario statement .................................... 77
    4.4.2 Adaptation requirements ............................... 78
    4.4.3 Solutions based on MAC-ng ............................. 79
  4.5 Summary .................................................. 85
## SECTION II: ADAPTIVE MIGRATION, SERVICE DISCOVERY, AND INTERACTION  

### 5 Adaptive Component Migration  

5.1 Background  .................................................. 91  
5.2 Architectural Requirements  ......................... 93  
5.2.1 Application model  ........................................ 93  
5.2.2 Mobility management  ................................. 93  
5.2.3 Resource binding and service customization  .... 94  
5.2.4 Context awareness  ................................. 95  
5.3 Architectural Framework  ............................... 95  
5.3.1 Application management  ............................... 95  
5.3.1.1 Application architecture  ......................... 95  
5.3.1.2 Dynamic interaction  ................................. 97  
5.3.1.3 Coordination management  ......................... 97  
5.3.2 Resource description and agent reasoning mechanism 98  
5.4 Modeling by Attributed Graphs  ....................... 100  
5.5 Performance Analysis  ............................... 108  
5.6 Related Work  ........................................... 111  
5.7 Summary  .............................................. 113  

### 6 Service Discovery and Interaction Adaptation  

6.1 Introduction  ............................................ 116  
6.2 Approach  ................................................. 118  
6.2.1 Context model and system architecture  ....... 118  
6.2.2 Service discovery primitives and mapping  .. 120  
6.2.3 Interaction programming model  ................... 122  
6.2.4 Interceptor based multi-mode interaction  .... 126  
6.2.5 Adaptation  ........................................... 128  
6.3 Performance Analysis  ..................................... 130  
6.4 Related Work  ........................................... 131  
6.5 Summary  .............................................. 133  

## SECTION III: FORMAL MODELING AND ANALYSIS  

### 7 Adaptation Rules Conflict Detection  

7.1 Introduction  ............................................ 138  
7.2 Reconfiguration Modeling and Analysis  ........ 139  
7.2.1 Critical pair and dependency  ....................... 139  
7.2.2 Architectural reconfiguration analysis  ....... 141
Software is eating the world whilst the world is eating more and more software. The interplay between software and the world drives the evolution of software development methodologies. The way of constructing software is undergoing a fundamental paradigm shift. The execution environment of modern software is becoming more open, dynamic, and volatile. Such openness brings grand challenges to the adaptability of the inhabitant software systems. Since continuously delivering software satisfying users’ needs is always a timeless pursuit of software developers, self-adaptation attracts considerable attention from both industry and academia. There has been a lot of research conducted in this area, inspired by applications from a multitude of disciplines. This can be evidenced by a proliferation of new adaptation techniques and frameworks that have emerged in recent years.

Among others, software architecture related techniques represent an important subject. This is partly due to the increasing application of component based systems in the open environment, such as web services. Software architecture provides an adequate abstraction level, as well as an effective way to guide the adaptation.

This book is one of the first monographs to address software adaptation in an open environment from a software architecture perspective. The two authors are active researchers have great experience in this field and provide a comprehensive discussion of current adaptation frameworks in the light of software architecture. This includes service discovery and interaction adaptation, adaptive component migration, or context and ontological models. The chapter on formal modeling covers the essence of adaptation rules, conflict detection, or verification of dynamic evolution.

The book aims both at practitioners and researchers and conveys the foundations of software adaptation. It describes verification techniques and frameworks
and as such provides an excellent reference to this domain of dynamic software system evolution.

Zurich, December 12, 2016
Harald Gall
Preface

With the rapid development of computing and network technology, the operating environment of modern software systems is becoming increasingly open, dynamic, and uncontrollable. New computing paradigms, such as pervasive computing and cloud computing, are emerging. Among these paradigms, the common features of the environment exhibit an ever-growing trend of openness. This trend has a significant impact on software development and interaction. Moreover, a large number of systems are composed of distributed and autonomous components. The high reliance on software requires the system’s robustness, i.e., continual availability and satisfactory service quality. This requirement gives rise to the popularity of research on the self-adaptive software in such an open environment.

Traditional software adaptation approaches usually simplify the context and select some key variables from a specific application domain, and the adaptation logic is predefined, hard-wired, and mixed with business logic. Such an approach has the advantage of quick response. Nevertheless, it supports low reusability and adaptability in heterogeneous situations. Thus, in an open environment which is characterized by its dynamism and heterogeneity, software adaptation research faces many new challenges. The context is more complex than before. Diversity of platforms, variety of user preferences, difference of service discovery, and interaction protocols all possibly affect the system’s adaptive behaviors. Therefore, it requires elaborate inspection of the context features and explicit modeling techniques. By incorporating its semantic information, the context changes can be understood by the software applications, and the adaptive behavior is conducted accordingly. However, the richness and complexity of context information in an open environment will also bring about other problems. To name a few, multiple adaptation rules are possibly activated concurrently as these rules are not necessarily orthogonal. The problem is how to detect the potential conflict and dependency relations. Aside from the adaptation enabling techniques,
the problem is how to assure the dynamic evolution process is consistent with
the specification. In the heterogeneous protocol environments, the problem is
how to enable adaptive service discovery and interaction so as to provide con-
tinual service availability. In the case of mobility, the problem is how to support
adaptive component-level migration to improve users’ satisfaction and reduce
unnecessary overhead. Traditional solutions originated from a relatively closed
environment with limited applicability and flexibility.

For component based software systems running in an open environment, such
as the Internet, despite the variety of the underlying implementation details, they
share a common set of characteristics: high-level separation of computation and
coordination, loosely coupling and high autonomous entities, protocol-support
interactions, etc. These characteristics can be well captured by the concept of
software architecture which focuses on the abstract view of constituent entities,
their interactions, patterns of composition, and global constraints. In considera-
tion of the increasing openness and autonomy of these distributed components,
attention has gradually shifted from components’ internal details to the compo-
sition and coordination of these components. In these situations, software archi-
tecture offers an adequate level of granularity and becomes a crucial artifact for
the research on self-adaptation in software engineering. Having observed this,
this book attempts to address the aforementioned problems from a perspective of
software architecture and presents our recent research on the efforts of engineer-
ing self-adaptive software systems.

Constructing self-adaptive software is intriguing but not easy. It involves
knowledge and expertise from multiple disciplines. This book does not aim to
provide a comprehensive encyclopedia of building self-adaptive software. In-
stead, it focuses on the challenges raised by the open environment and is intended
to be used as a general introduction to the engineering of self-adaptive software
in such an environment.

Acknowledgements.
The authors would like to thank Professor Jian Lü (Nanjing University), Profes-
sor Harald Gall (University of Zurich), Professor Xiaoxing Ma (Nanjing Univer-
sity), Professor Luciano Baresi (Politecnico di Milano), Professor Carlo Ghezzi
(Politecnico di Milano), Professor Jiannong Cao (Hongkong Polytechnical Uni-
versity), Professor Zhiqiu Huang (Nanjing University of Aeronautics and Astro-
nautics), Professor Xianping Tao (Nanjing University), Dr. Andrea Mocci (Uni-
versity of Lugano), Professor David S. Rosenblum (National University of Sin-
gapore), Professor P. S. Thiagarajan (National University of Singapore), Profes-
sor Mingheng Ying (University of Technology Sydney), Professor Yuan Feng
(University of Technology Sydney), Professor Marta Kwiatkowska (University
of Oxford), Dr. Guoxin Su (National University of Singapore), and Dr. Ting-
ting Han (Birkbeck, University of London) for their collaboration, support, and
guidance of the work.
Yu Zhou is partially supported by the Natural Science Foundation of Jiangsu Province under grant No. BK20151476, the National Basic Research Program of China (973 Program) under grant No. 2014CB744903, the National High-Tech Research and Development Program of China (863 Program) under grant No. 2015AA015303, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities under grant No. NS2016093. Taolue Chen is partially supported by EPSRC grant (EP/P00430X/1), European CHIST-ERA project SUCCEED, ARC Discovery Project (DP160101652), Singapore MoE AcRF Tier 2 grant (MOE2015-T2-1-137), NSFC grant (No. 61662035), and an overseas grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2014A14).
Authors

Dr. Yu Zhou is an associate professor at the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA), China. He received a doctoral degree in computer science from the Nanjing University, China in 2009. From 2010 to 2011, he conducted postdoctoral research on software engineering at Politecnico di Milano, Italy. From 2015 to 2016, he visited SEAL lab at the University of Zurich on sabbatical, where he is also an adjunct researcher. He is currently a senior member of the China Computer Federation (CCF) and a member of the Technical Committee on System Software of CCF. He has broad interests in software engineering with a focus on software evolution and reliability analysis.

Dr. Taolue Chen received bachelor’s and master’s degrees from the Nanjing University, China, both in computer science. He was a junior researcher (OiO) at CWI and acquired a PhD degree from the Free University Amsterdam, The Netherlands. He is currently a senior lecturer at the Department of Computer Science, Middlesex University London, UK. Prior to this, he was a research assistant at the University of Oxford, UK, and a postdoctoral researcher at the University of Twente, The Netherlands. His research interests include formal verification and synthesis of stochastic systems, model checking, concurrency theory, process algebra, and computational complexity.
Chapter 1

Introduction

CONTENTS

1.1 Novel Computing Paradigms and Methodologies .......................... 6
1.2 What is Self-Adaptation? .......................................................... 16
1.3 What is Context? ................................................................. 19
1.4 Challenges of Adaptation in an Open Environment .................... 20
   1.4.1 Characteristics of the open environment ............................ 21
   1.4.2 Adaptation requirements ............................................. 22
1.5 Structure of the Book ............................................................ 24
It has long been a consensus in software engineering that software entity is constantly subject to pressure for change [30]. Numerous research efforts from various aspects have been devoted to addressing the issue. This can be observed from the perspective of the evolution of software process models and methodology. The waterfall process model first came to debut in the early 1970s. However, its assumption that each step could be done perfectly before moving to the next step is not true in many real projects, mostly due to the changes introduced from the requirements and environments. This fact had led to other iterative models such as incremental and spiral models, and the more recent agile models and DevOps models [19]. In accord with that, software methodology also evolves gradually from structure-oriented, then object-oriented, later component-oriented, to nowadays service-oriented. Each stage supports a more coarse-grained and loosely-coupled programming style than its predecessor. Correspondingly, the research focus shifts steadily from programming-in-the-small to programming-in-the-large [54].

These efforts have undoubtedly enhanced software robustness and productivity greatly. However, it is undeniable that the problem itself evolves as well. Remarkably, the operating environment of modern software is not as isolated or closed as before. In particular, with the rapid development of computing technology, the Internet, which was merely a portal of data and resources at its inception, has become the largest platform for software applications. As outlined in a vision document by the European Commission, Internet-of-Contents and Knowledge, Internet-of-Things, and Internet-of-Services are the three pillars of future Internet [138]. Different from traditional computing paradigms, the Internet is characterized by its scalability, openness, and heterogeneity. More concretely, it has the following salient features: decentralized distribution, highly autonomous network nodes, heterogeneous devices, unpredictable entity’s behavior, potential security threats, personalized usage, and co-existence of various network protocols [187]. Many factors introduced by this kind of open environment, such as different types of platforms, variable conditions of hardware, and heterogeneous networks and protocols, have a significant impact on the software artifacts running on top of the Internet. The underlying characteristics of such an environment pose strict requirements on the adaptability of the inhabitant software systems. The mutual interplay between the environment and the underlying application gives birth to new software paradigms, such as internetware, which largely denotes a class of systems characterized by flexibly evolvable, continually reactive, and multiple objectives oriented in the open environment [120, 187].

Adaptability is an important dimension of the software quality attributes. In IEEE standard glossary of software engineering terminology [27], adaptability is defined as “The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.” Clearly, since the very beginning, building robust software that can adapt in the presence of adverse conditions has always been one of the focuses...
of software engineering. Numerous research efforts from different dimensions have been invested to address this issue. In almost every stage of the software development life cycle, such as abstraction, specification, implementation, and maintenance, we can identify these efforts. Indeed, we have witnessed the evident progress of enhancing adaptability in software design methodology and programming language mechanisms, such as design patterns, polymorphism, dynamic binding, etc. In design patterns, one of the underpinning principles is open/closed principle which states that software modules should be open for extension, but closed for modification. Meanwhile, polymorphism and dynamic binding allow that the overridden method invocation can be decided at runtime.

However, the extant approaches largely originate from the traditional environment with the “closed environment” assumption. Changes are anticipated and hard-wired in the source code. It works well as long as that assumption holds. However, in an open environment, at design time, developers cannot anticipate and plan all possibilities before the code delivery. Therefore, in case of the occurrence of unanticipated events at runtime, the software has to be shut-down and maintained manually so as to assure it operates properly. Needless to say, software is expected to continue running in case of such events, since nowadays it is eating the world and plays more and more important roles in business areas, critical missions, and our everyday life. The cost of shutting down the applications and evolving them off-line is surprisingly high. For example, a technology market research firm, IHS Inc., published a survey on “the cost of server, application and network downtime” in January 2016. The result reveals that the cost of each downtime event ranges from one million dollars for a typical middle-sized company to more than 60 million for a large company. The total downtime cost for North American businesses can sum up to 700 billion dollars. This calls for the ability to be self-adaptive for modern software systems which remains a great challenge today. There is still a huge gap between fact and expectation. Throughout recent years, an ever-growing research effort is devoted to engineering such systems, both in academia and industry.

Traditional self-adaptive systems, such as robotics, avionics, and vehicle control systems, usually leverage some domain-specific algorithms or exception handling mechanisms to accommodate variabilities. As a result, this solution lacks generality and supports poor reusability. Supporting self-adaptation in a dynamic and open environment is far from a trivial task. Many efforts from multiple disciplines have contributed to this subject [158]. From the dimension of artificial intelligence and knowledge engineering, new dynamic planning or machine learning-based algorithms can be developed to guide the adaptation process. From the dimension of control theory, some parameter-tuning techniques based on specific control models have been proposed. From the dimension of  

---

In software engineering, many research areas are related, such as service computing, component technology, and software architecture [52].

Recently, the dynamic software architecture-based adaptation [75, 110] is gaining increased popularity due to its unique advantages. As software architecture is the global abstraction of the system and embeds the design decisions, it reflects the essential view of the application which is very useful to cope with the growing complexity of modern software. Secondly, nowadays, more and more software applications are developed through the composition from elemental autonomous components. The structure of such composition is not as static as before, and the elements can dynamically join and leave the integrated applications. Therefore, the composition and reconfiguration of the application structure can be naturally modeled by the software architecture dynamism. Moreover, decades of research on software architecture have resulted in a number of models and techniques which provide a useful knowledge base to analyze adaptation-related properties.

Given the above considerations, there is growing recognition that software architecture reconfiguration can be an effective way to enable software adaptation with the adequate generality and granularity. However, architectural reconfiguration is only the action part of adaptation. In an open environment such as the Internet, there are many other concerns to be taken into account during the adaptation process. This book focuses on the issues and attempts to give systematic solutions, instead of ad hoc ones, from a perspective of software architecture. The book inspects the characteristics of the open environment, and addresses the problems such as context modeling and reasoning, adaptive component-level migration, adaptive service discovery and interaction with heterogeneous protocols, conflicts/dependency detection for multiple adaptation rules, consistency verification, and adaptation decision-making process.

This chapter is intended to be a general introduction to the research on software self-adaptation. Some new computing paradigms and methodologies emerged in open environments are introduced as the background in Section 1.1. Then we proceed to explain some basic concepts of self-adaptation in Section 1.2 and Section 1.3 respectively. The problems of self-adaptation in an open environment are given in Section 1.4 followed by an overview of the book organization in Section 1.5.

1.1 Novel Computing Paradigms and Methodologies

In this section, we will review some representative computing paradigms and methodologies that have emerged recently and are relevant with adaptation in an open and dynamic environment.
Introduction

Service computing utilizes services as basic components for developing applications. Services are self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications. Services perform functions, which can be anything from simple requests to complicated business processes. As on the Internet, it is impossible to require all the communication parties use the same platform and language. Therefore, services allow organizations to exhibit their core competencies programmatically over the Internet (or intra-net) by using standard (XML-based) languages and protocols, and being implemented via a self-describing interface based on open standards [140].

To facilitate this process, service oriented architecture (SOA) is proposed. The basic SOA model has three elements, i.e., service provider, service registry, and service requestor. Service provider first needs to publish their service descriptions in the service registry. Service requestors first look up the service information at the registry center. After binding the corresponding information, the requestors will interact with the provider using some standard interfaces. This process is described in Figure 1.1.

In [100], Huhns and Singh summarize some key features of service oriented computing as below:

- Loose coupling. Tight transactional properties generally do not apply among components because conventional software architectures do not typically include transactional managers. Some high-level contractual relationships that specify component interactions to achieve system-level consistency should be considered.

- Implementation neutrality. The interface for each component matters most, because we cannot depend on the interacting components' imple-

Figure 1.1: Basic model of SOA
Software Adaptation in an Open Environment: A Software Architecture Perspective

Implementation details, which can be unique. In particular, a service-based approach cannot be specific to a set of programming languages, which cuts into the freedom of different implementers and rules out the inclusion of most legacy applications.

- Flexible configurability. An SOA system is configured late and flexibly, which means that different components are bound to each other late in the process. Thus, the configuration can change dynamically as needed without loss of correctness.

- Persistence. Services do not necessarily require a long lifetime. Because we are dealing with computations among autonomous heterogeneous parties in a dynamic environment, we must always be able to handle exceptions. The services must exist long enough to detect any relevant exceptions.

- Granularity. An SOA’s participants should be modeled and understood at a coarse granularity. Instead of modeling interactions at a detailed level, the high-level qualities that are visible for business contracts among the participants should be captured.

- Teams. Rather than framing computations centrally, we should think in terms of how autonomous parties, working on a team as business partners, realize those computations.

Because of the characteristics above, service-based systems allow more space for adaptation. Several techniques, such as orchestration and choreography are proposed to describe aspects of creating business processes from composite web services. Orchestration refers to an executable business process that can interact with both internal and external web services and the interactions occur at the message level. Choreography tracks the message sequences among multiple parties and sources, rather than a specific business process that a single party executes [142]. Related specification standards are proposed, such as BPEL [72], WSCI [8], to describe this process. These techniques focus on the weaving script rather than on the internal business-level programming logic. Due to the separation of concerns, the adaptation of such systems requires the rewriting or recomposing of these specifications.

Autonomic computing

In March, 2001, IBM announced the initiative of autonomic computing. The idea was inspired by human’s autonomic nervous system which can manage the body conditions without explicit intervention. Generally, the autonomous systems have the following characteristics [74]:

- To be autonomic, a system needs to “know itself” and consist of components that also possess a system identity.
• An autonomic system must configure and reconfigure itself under varying and unpredictable conditions.

• An autonomic system never settles for the status quo - it always looks for ways to optimize its workings.

• An autonomic system must perform something akin to healing - it must be able to recover from routine and extraordinary events that might cause some parts to malfunction.

• A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection.

• An autonomic computing system knows its environment and the context surrounding its activity and acts accordingly.

• An autonomic system cannot exist in a hermetic environment (and must adhere to open standards).

• Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed to meet a user’s information needs while keeping its complexity hidden.

In short, the above characteristics can be summarized into four fundamental categories, i.e., self-configuring, self-healing, self-optimizing, and self-protecting.

Self-configuring means that the autonomic systems can configure themselves automatically in accordance with high-level policies representing business-level objectives. For complex systems, manual configuration is time-consuming and error-prone, and thus self-configuration is desirable in these situations. Self-healing means that autonomic computing systems can detect, diagnose, and repair localized problems resulting from bugs or failures in software or hardware. A specific problem-diagnosis component is needed to analyze information from the context and to respond accordingly. Self-optimizing means that autonomic systems continually seek ways to improve their operation by identifying and seizing opportunities to make themselves more efficient in performance or cost. The components will proactively seek to upgrade their function by finding, verifying, and applying the latest updates. Self-protecting means that autonomic systems defend themselves as a whole against large-scale, correlated problems arising from malicious attacks or cascading failures that remain uncorrected by self-healing measures. It also means that the system can anticipate problems based on early reports from sensors and take steps to avoid or mitigate them [106].

IBM has suggested a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) reference control loop model for autonomic computing [101] which is depicted by Figure 1.2. The model is used to describe the autonomous component.
In this sense, the autonomous component is a special kind of intelligent agent situated in a context. The system is composed by a set of such autonomous entities. In terms of autonomy, each entity further consists of utility components to monitor the running environment, to analyze the change necessity, to plan the change actions, and to execute the changes. **Monitor** collects runtime information from physical/logical sensors and generates symptom reports to the **Analyzer** component which in turn generates a change request to the **Plan** component. Then the **Plan** component decides a change plan and effects the object software module accordingly through actuators. This cycle is an intelligent process based on a knowledge database.

To fully realize the vision of autonomic computing, grand challenges remain. Despite the difficulties, there are some research projects, such as ABLE toolkit [24] and autonomic toolkit [102]. Huhns et al. proposed a multi-agent based approach to enhance software’s adaptability in the presence of adverse situations [99]. These projects have demonstrated the feasibility of the basic ideas in autonomic computing.

**Grid computing**

Grid computing originates from the fast growth of the Internet and the availability of powerful computers, as a particular form of distributed computing. Different from traditional paradigms, it views the network of computers as a single, unified computing resource. The purpose of a grid computing project is usually to solve a single but complicated problem which usually involves huge computations. By the strategy of divide-and-conquer, the problem is cut into smaller tasks and located to member computers within the grid. To achieve the goal, grid
computing attempts to cluster or couple a wide variety of resources including
supercomputers, storage systems, data sources, and special classes of geograph-
ically distributed devices and use them as a single unified resource, thus forming
what is popularly known as a “computational grid” [11].

Due to the characteristics of the Internet, the resources for grid applications
are not limited to a single site or local network. Instead, it can bind resources
globally. In this light, grids enable users to solve large or new problems by utilizing
the available resources together. The characteristics of computational grids
are listed as below [11]:

- Heterogeneity. A grid involves a multiplicity of resources that are hetero-
genous in nature and might span numerous administrative domains across
wide geographical distances.

- Scalability. A grid might grow from a few resources to millions.

- Dynamism or adaptability. In a grid, a resource failure is the rule, not
the exception. With so many resources in a grid, the probability of some
resource failures is inevitably high.

Owing to the above characteristics, the grid computing paradigm is suitable
for those applications with extremely high demands for data manipulation and
calculation. The concept was initiated as a project to link super-computing sites.
Now, there are many applications benefiting from the grid infrastructure. Multi-
ple grid projects have been started. The most famous grid project is the World-
wide Large Hardron Collider (LHC) Computing Grid (WLCG) [163] which is
coordinated by CERN ². The interconnection among many associated national
and international grids across the world, such as European Grid and Open Sci-
ence Grid makes WLCG the world largest computing grid. Some other notable
cases are: SETI@Home [6], ACQUA@Home [104], and DAS [12].

Cloud computing

Cloud computing is the next natural step in the evolution of on-demand informa-
tion technology services and products [180]. Similar to grid computing, cloud
computing also attempts to abstract and virtualize resources on the Internet. But
different from grid computing, it is not oriented to huge computing tasks. Instead,
it is mainly designed for delivering IT services as computing utilities.

Under the umbrella of cloud computing, it can use a storage cloud to hold appli-
cation, business, and personal data. Or it can be the ability to use a handful of
web services to integrate photos, maps, and GPS information to create a mashup
in customer Web browsers [1]. It can also be the ability to use applications on the
Internet that store and protect data while providing a service. In [9], Armbrust et

²CERN is the acronym for European Organization for Nuclear Research situated in Switzerland.
al. believe that cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data-centers that provide those services. In this sense, the data-center hardware and software is what they call a cloud.

In [71], Foster et al. define cloud computing as a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet. A more widely accepted definition is given by the National Institute of Standards and Technology (NIST), i.e., cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [128]. Three service models co-exist in cloud computing, i.e., software as a service (SaaS), platform as a service (PaaS), and Infrastructure-as-a-service (IaaS). Briefly speaking, SaaS denotes the services provided to customers to use provider-owned applications through the network. PaaS denotes the services provided to customers to develop, run, and maintain their own applications through the network. IaaS denotes the whole infrastructure, such as virtual machines, including application servers, storage, and other fundamental computing resources are provided to customers.

Although the elements of cloud computing, such as virtualization, SaaS, and the Internet, are not novel concepts individually, their combination brings some new highlights. In [9], Armbrust et al. summarized the following three new aspects:

1. The illusion of infinite computing resources available on demand, thereby eliminating the need for clouding computing users to plan far ahead for provisioning;
2. The elimination of an up-front commitment by cloud users, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs;
3. The ability to pay for use of computing resources on a short-term basis as needed and release them as needed, thereby rewarding conservation by letting machines and storage go when they are no longer useful.

In the early ages of cloud computing, a few examples in practice have been designed to suggest likely directions. In [92], Hayes enumerates four representative categories.

1. Wordstar for the web. Google Docs is such a case in point. The set of programs includes a word processor, a spreadsheet, a form, and a presentation tool.
2. **Enterprise computing in the cloud.** Maybe the most famous example of this kind is the salesforce.com. Different from traditional standalone Customer Relationship Management (CRM) solutions, it offers a suite of programs over the Internet. The programs can be flexibly tailored according to the customers’ requirements.

3. **Cloudy infrastructure.** Amazon Elastic Computing Cloud (EC2) is a web service that provides resizable compute capacity in the cloud and allows to quickly scale capacity as the computing requirements change. The Chinese Alibaba corporation also built and is expanding such an infrastructure for e-business and other multiple professions.

4. **Cloud OS.** The eyeOS is of such a kind. It reproduces the familiar desktop metaphor—with icons for files, folders, and applications—all living in a browser window. With decades of development, cloud computing has achieved great success in commercial market. Nowadays, Amazon’s EC2, Google’s Compute Engine (GCE), and Microsoft’s Azure are typical examples of cloud computing.

Pervasive computing

In recent years, the integration of cyber space and its physical counterpart is becoming much closer. Smart devices with enhanced processing abilities and network-intensive environments enable a new computing paradigm — pervasive computing. Therefore, pervasive computing is characterized as the one saturated with computing and communicating capabilities, and integrated with users so that it becomes a “technology that disappears” [159].

Pervasive computing is a major evolutionary step from distributed systems and mobile computing. Therefore, it covers the traditional mobile communication techniques, micro computing device manufacturing techniques, and software methodologies. It also has its own characteristics. The evolution is mainly reflected in the following aspects [159].

- **Effective use of smart spaces.** The fusion of the physical space and the cyber space enables sensing and controlling of one world by the other. Software on a user’s computer may behave differently, depending on where the user is currently located. Smartness may also extend to individual objects, whether located in a smart space or not.

- **Invisibility.** If a pervasive computing environment continuously meets user expectations and rarely presents him with surprises, it allows him to interact almost at a subconscious level.

---

3https://aws.amazon.com/ec2/
4https://cloud.google.com/compute
5https://azure.microsoft.com/
• Localized scalability. With the growing complexity in the smart space, the intensity of interaction between a user’s personal computing space and his/her surroundings also increases. This has severe bandwidth, energy, and distraction implications for a wireless mobile user.

• Masking uneven conditioning. There will persist huge differences in the “smartness” of different environments. The variability between them should be reduced in order to improve users’ satisfaction. This also implies the human-centrism philosophy in pervasive computing environments.

In [157], Saha and Mukherjee provide a pervasive computing model which consists of pervasive devices, pervasive networking, pervasive middleware, and pervasive applications. Pervasive devices include traditional input devices, wireless mobile devices, smart devices, and sensors. With the tremendous growth of pervasive devices, their communications are becoming more and more important, and this fact leads to pervasive networking. Pervasive middleware will mediate interactions with the networking kernel on the user’s behalf and will keep the users immersed in the pervasive computing space [157]. Pervasive applications are the services that directly interact with users. Unlike traditional applications, they are more human-centric. This requirement will guide the middleware and networking issues to a large extent [157].

The vision of pervasive computing raises huge technological challenges. Many traditional mobile computing issues and new ones, such as scalability, heterogeneity, integration, and invisibility, are all tough to handle. To address these issues, there are many research projects in these fields. Some representative projects are Easy Living from Microsoft [161], Oxygen from MIT [155], Gaia from UIUC [153], Aura from CMU [78], and so forth.

In fact, the pervasive environment shares a lot of common characteristics with the open environment, and the research on adaptation in a pervasive environment can shed new light on that in an open environment.

Aspect oriented programming
Gregor Kiczales defines aspect oriented programming (AOP) as a new evolution in the line of technology of separation of concerns — technology that allows design and code to be structured to reflect the way developers want to think about the system [60]. AOP is based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. Concerns can range from high-level notions like security and quality of service to low-level notions such as caching and buffering. While the tendency in object-oriented programming is to find commonality among classes and push it up in the inheritance tree, AOP attempts to realize scattered concerns as first-class elements and eject them horizontally from the object structure [61]. As
AOP emphasizes more on the separation of concerns in software design, it supports cleaner organization of software modules and higher reusability. Some successful applications of AOP technology include JBoss application server, spring framework, and .Net framework, etc.

AOP can be basically classified into two categories, *i.e.*, static AOP and dynamic AOP, according to the weaving time and the style of concerns. In the early stages, some prototypes of AOP systems are mostly static, *e.g.*, AspectJ [107]. In static AOP, weaving the aspect is part of compiling process. For example, in AspectJ, the operation is at the byte-code level. After compiling, the result is the common java byte-code. The main advantage of this type is its simplicity to support the aspect weaving. However, the disadvantage is that it does not support the dynamic loading of new aspects. This problem leads to the introduction of dynamic AOP which supports more flexible aspect weaving. For example, Spring framework utilizes a specific proxy module to manage the loading and weaving of aspects.

The AOP based adaptation has two types. The first is to encapsulate the adaptation logic into the aspects. By identifying the adaptation concerns, AOP provides an effective means to modularize both application independent and application specific facets of adaptation. Examples can be found in [148]. The other kind of AOP based adaptation is the addition and removal of the aspect itself during execution. As aspects encapsulate specific concerns, functional or non-functional, dynamic loading and removal of aspects can cause the behavior adaptation of applications.

Discussion

We have described several novel computing paradigms and programming methodologies that are related to the software adaptation in an open environment. Technology advances and new requirements have given birth to the emergence of these novel computing paradigms and methodologies. Grid computing, cloud computing, and pervasive computing provide new computing environments. AOP provides a new programming methodology which supports higher reusability. Service oriented computing offers an insight on how to organize heterogeneous software and mask the underlying heterogeneity. The autonomic computing initiative recognizes the importance of self-management for future software and provides a basic research framework. In fact, because of the similarities in the background of these paradigms, the boundary between them is blurring nowadays. Many techniques overlap with each other.

For these computing paradigms, adaptation issues are also addressed. For example, in the grid environment, because the computer node may enter or exit the grid randomly, self-adaptation algorithm is necessary to cope with the uncertainty. Despite the similarities with the grid, cloud, and pervasive computing, an open environment has its own features. It requires systematic techniques and theory [187, 120] which include requirement definition, software architecture
design, analysis, adaptation techniques, and maintenance, etc. This book mainly address the adaptation concerns raised by the open environment. Considering the broad area adaptation covers, we particularly focus on context modeling, adaptation enabling techniques, adaptive component migration, connector-based service discovery and adaptation, and modeling/analysis aspects.

1.2 What is Self-Adaptation?

Self-adaptation is a compound. So first let us examine adaptation. In the Longman Dictionary of Contemporary English, adaptation has two meanings:

1. A film or television program that is based on a book or play;
2. The process of changing something to make it suitable for a new situation.

Evidently, our usage of adaptation fits the second explanation. Generally, it denotes a process that the entity can adapt itself according to the environment. As aforementioned, adaptation involves multiple software engineering disciplines. Many researchers have given their own definitions from different perspectives. For instance,

- A program can change its behaviors automatically according to the context [118].
- Being able to make last moment changes [3].
- Any automated and concerted set of actions aimed at modifying at runtime, the structure, behavior, and/or performance of a target software system, typically in response to the occurrence and recognition of some (adverse) conditions [177].

From the above definitions, we can observe that, except for the second one which is defined from the software process dimension, the other two share similar elements, i.e., contextual information and corresponding runtime reactions.

Self-adaptation emphasizes more on the autonomy. DARPA Broad Agency (BAA-98-12) provides a definition of self-adaptive software as follows:

Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

This definition is mainly given from an artificial intelligence perspective. In our opinion, self-adaptation includes the following key factors.

- First, it is an interaction between software and the environment. The software probes the changes from the environment and reacts accordingly.
Second, it is an intelligent behavior. Software can choose the embedded reaction strategies or load them dynamically based on the specific context.

Third, it has a realtime feature. Adaptation should be performed without causing system’s shutdown.

Fourth, it is a purposiveness process. It reflects the design aim, e.g., to provide continual availability or to improve users’ satisfaction.

In light of these considerations, we present our definition of self-adaptation as follows.

*Self-adaptation is a process through which software entity dynamically changes its behavior according to the context information in order to ensure or improve users’ satisfaction.*

There is a subtle difference between *adaptation* and another buzzword, *i.e.*, *online/dynamic/runtime evolution*. In [182], Wang et al. define online evolution as a specific kind of evolution that updates running programs without interrupting their execution. From this definition we can observe that online evolution mainly focuses on the techniques that can update a system dynamically which is the third factor listed above. But self-adaptation also covers other dimensions, such as, interaction with the context and the decision-making process.

Software adaptation can be inspected from different perspectives. Therefore, there are several classification criteria. In [151], Rohr et al. give a comprehensive classification schema. This tree-like schema examines software adaptation from five dimensions, *i.e.*, *origin, activation, system layer, operation, and controller distribution*.

- **Origin** is the location of the state change that triggers an adaptation cycle. The changes may take place in the external environment or in the systems themselves.

- **Activation** denotes the response types. There are three types, *i.e.*, reactive, predictive, and proactive. Reactive adaptation means that the system adapts only after that performance has degraded and is below a certain threshold. Predictive adaptation means that before the real drops of performance, some prediction algorithms can foresee the potential threats and respond beforehand. Proactive adaptation is mainly applied to improving system performance when it is still normal with no signs of potential drops. This type of activation is closely related to self-optimization.

- As computing systems are hierarchical, *system layer* denotes the corresponding adaptation layers. Generally, there is a hardware level, an operation system level, a middleware level, and an application level. Self-adaptation techniques can be applied to any of these layers.
Operation classification dimension uses an architecture description language viewpoint on the adaptation operation. Multiple component instances can belong to the same component type, and the implementation of a component instance can change without affecting the associated component type. Therefore, six types of adaptation operations are distinguished, i.e., data adaptation, intra-component behavior adaptation, component resource mapping adaptation, inter-component protocol adaptation, instance-level adaptation, and type-level adaptation.

Controller distribution denotes the localization of self-adaptation logic. Generally, there are three types, i.e., centralized, decentralized, and hybrid. The controller distribution style is centralized if the adaptation is controlled and executed from a central point in the system, while a decentralized system has no such central control module. Hybrid approaches combine the advantages of centralized and decentralized architectures by applying parts of the activities locally and other parts globally [151].

Rohr’s classification schema is based upon the different stages of adaptation process. Similarly, Subramanian gives a spectrum of adaptability phenomena [170] based upon the more fine-grained self-* facets, e.g., self-reparability, self-tuning, etc. Self-reparability is the ability of software systems to adapt at runtime in order to accommodate varying resources, system errors, and changing requirements. Self-configuring systems can configure themselves in the field rather than in the factory. Self-tuning can tune and reconfigure themselves. Autonomic systems are capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads put upon them. Extensibility is the case with which a system or component can be modified to increase its storage or functional capability. Tailorable systems can be adapted by the users to their particular situations and needs. Changeability is the ability of a software system to sustain an on-going flow of changes. Modifiability is the ability to make modifications to the software system. Evolvability enables easy evolution of software systems through enhancement to meet current needs. Flexibility has been defined as the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [170].

It should be emphasized that this spectrum is not static. New or similar self-* terms are under development, such as self-management [28], self-optimization [4], and self-healing [81]. Self-management systems denote those systems that not only implement the change internally but also initiate, select, and assess the change itself without the assistance of an external user [28]. Self-optimization refers to the automatic mechanism for recognizing significant changes and re-optimizing the system as a result of such changes [4]. Self-healing is defined as the property that enables a system to perceive that it is not operating correctly. Without (or with) human intervention, it can make the
necessary adjustments to restore itself to normalcy [81]. From the above introduction, we can observe that, in fact, there are no sharp distinctions between self-* concepts. Instead, there is only a subtle difference in the aspect of adaptation emphases.

### 1.3 What is Context?

*Context* is an intensively studied subject in recent years [13]. In the *Longman Dictionary of Contemporary English*, context has two meanings:

1. The situation, events, or information that are related to something and that help you to understand it;
2. The words that come just before and after a word or sentence that help you understand its meaning.

As the second explanation mainly deals with grammar, our usage corresponds to the first one. Many researchers have given their definitions on this subject. Brown *et al.* define context as a collection of locations, identities of the people around the user, the time of the day, the season, the temperature, etc. [31]. Ryan *et al.* define context as a synonym of environment, which includes the information such as the location, the time, the temperature, and the user identity [156].

In [55], Dey gives a widely accepted definition: *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 the user and application themselves.*

This definition is proposed from the pervasive computing perspective [159]. However the definition is too broad. We believe that for self-adaptive software, the context should be narrowed down to those factors that have potential influences on the software’s adaptive behaviors. Thus we give our definition of context as follows:

*Context is the relevant information that can potentially cause or constrain software’s adaptation.*

There is a need to differentiate this term with *environment*. These two concepts obviously have overlaps. On the one hand, environment definitely has potential influences on software’s adaptation, but not every piece of information from the environment makes sense in this process. On the other hand, environment information is not the only factor affecting adaptation, other factors also matter. As a concrete example, a web application responds slowly and requires adaptation to add another backup server. If the architectural style specifies that at most $N$ backups can be added, the current architecture information and the style constraints are both counted into context.
1.4 Challenges of Adaptation in an Open Environment

In an open environment, the research on self-adaptation software faces many new challenges due to the increasing complexity of the context. As the traditional methodology and theory for software adaptation is developed in a relatively closed, static, and controllable environment, some of them are not applicable. For instance, traditionally, software architecture is regarded to be static throughout the life cycle of the software system. However, this assumption does not hold under the new circumstances. Instead, in many cases, flexible adaptation of software architecture is needed to meet the requirements of the context changes [121].

As the new paradigm emerges and the environment becomes more and more open, the context is much more complex than ever before. An understanding of the interaction patterns between software systems and their environment is essential to construct self-adaptive software in such environments. Figure 1.3 illustrates an abstract view of the interaction. In this process, factors from the user, the interaction manner, and the environment all possibly affect the running system. These factors constitute the essence of context that causes software adaptation.

Having recognized the importance of context, the next issue is how to model it and provide a basis for application adaptation. An adequate and unified context model plays an important role in self-adaptive systems. Firstly, it can enhance the representation ability and facilitate the following reasoning and reacting process; secondly, a unified model can help avoid communication problems between different applications, which are usually caused by a lack of common understanding.

After getting the necessary information from the context, the system needs to adapt itself accordingly. This requires an adaptation enabling mechanism that can update the system dynamically. The process requires no interruption of the application. As in an open environment, systems are commonly composed by a collection of loosely coupled components, the adaptation mechanism is usually reflected in the coordination entity.
In [110, 137], Kramer et al. and Oreizy et al. stated the essential role that software architecture plays in self-adaptation. Respectively, they both proposed reference frameworks. However, the proposed frameworks stay at a very high level and share a lot in common. Nevertheless, they shed insight into the research on software adaptation. For example, they both use a control loop-like mechanism, and separate the adaptation logic from the business logic. However, the problem is that both of them are very abstract. In concrete scenarios, it is difficult to apply them directly. Moreover, both of them lack an explicit context model.

1.4.1 Characteristics of the open environment

Compared with traditional software adaptation research, in an open environment, adaptation faces new challenges. According to the interaction paradigm described in Figure 1.3, we will examine the characteristics and requirements from three perspectives.

1. The open environment has the following salient features.

   (a) Dynamics. This feature is reflected in the randomness of the computing node availability, existence, or the network conditions. The adaptation should handle the cases that the involved application components operate on these dynamic nodes and networks.

   (b) Openness. The new node may join or exit the existing network without notification. Thus, the size of the network may increase or decrease randomly. The adaptation should have the ability to incorporate or remove corresponding components.

   (c) Diversity of platforms and devices. With the development of technology, computers with high processing abilities are diversified into common consumer electronics which have different kinds of properties. Applications should adapt their behavior when running on different platforms. For example, some graphical display functions may have to be adjusted when the application migrates from a PC to a hand-held device due to their different display sizes.

   (d) Heterogeneity of protocols. The sub-network may have various different protocols. Standards may help to eliminate some heterogeneity, but the situation is there are usually several co-existing standards which still leads to the problem of heterogeneity. It requires that the coordination mechanism is flexibly adapted to support multi-mode protocol interactions.

2. The structure patterns of current software in such environments are different from those previously constructed.
Software Adaptation in an Open Environment: A Software Architecture Perspective

(a) Distribution. The components are usually scattered on the net, provided as a stand-alone service or an elemental service that can be integrated by third parties. These distributed components do not necessarily belong to the same organization. They only provide some public interfaces for communication. Thus, the integrators have limited control over these distributed components.

(b) Coarse-grained components. The component size is more coarse-grained and self-contained. They can be invoked directly through some standard protocols or composed by third parties. Service-oriented computing is emerging and offers a very promising business solution [100]. The adaptation of such systems is usually in the form of composition reconfiguration.

(c) Various interaction modes. As the integrator has limited control over the behavior of the components, the coordination mechanism becomes more important. Common interaction methods, such as RPC, and tuple space, implicitly know about the potential interaction entities. In multi-mode interaction context, it requires a more flexible coordination mechanism to adapt accordingly.

(d) Trust management issues. Usually, these services cannot guarantee the service quality and robustness. Even worse, some fraud services exist. To adapt with trust concern, this requires the deep analysis of the interaction confidence and the entity’s historical reputation.

3. With the growing intensity of computation and networking, computing is more human-oriented [155]. Factors from the interactions between users and computers also pose new challenges to software adaptability.

(a) Interaction profile. This usually includes personal preferences, operation habits, etc. Applications need to adapt according to the users’ special needs in order to improve their satisfaction.

(b) Users’ locations. Location is a concept closely related with mobility. This is an active research subject of context-aware computing [13]. To ensure continual availability, some applications need to migrate along with the users, such as editors or media players. This requires application migration. Complete migration costs are high. Therefore a light-weight, adaptive component-level migration is much more desirable.

1.4.2 Adaptation requirements

As mentioned before, the open environment poses new challenges to the self-adaptation of inhabitant software applications. We use a concrete example to
identify some key adaptation requirements. Consider the following scenario: a user is interacting with a data report application at his office. The application is composed of several components. The schedule alerts that, in several minutes, he is supposed to give a presentation about his work in a conference venue. When time is up, he has to leave the office, but he wants to continue working on the report on the way to the conference room. The sensor detects his movement and transfers the corresponding components of the application to his smart device. In this way, the user can continue editing. After arriving at the conference room, the sensor detects his arrival and restores the application component to the desktop computer in the new environment and establishes the connection with the original components. As the conference room is in a different network from the office, it is highly desirable to add a new security component on the fly. In the conference room, the user might need the printing service. However, the service protocol is not necessary the same as that in the old environment. For example, the printing service protocol in the office network is Jini and in the meeting room it might use the UPnP protocol.

The scenario reflects many characteristics of openness. For example, the fact that the application runs across networks is a kind of boundary openness. For the conference network, the user is new. He is eligible to use the network and hardware devices. This is a kind of usage openness. Different service protocols and devices co-exist and co-operate in the different networks. This is a type of protocol openness. In this scenario of open environment, we can identify some key self-adaptation problems.

1. In an open environment, the context is more complex and variant than before, how to understand and model the context information in order to facilitate the following adaptation process;

2. Because of the boundary openness and users’ mobility, how to adapt the application (migrate with the user) accordingly in order to provide continual services;

3. In an open environment, multiple service discovery and interaction protocols might co-exist, how to adaptively handle this heterogeneity and support the multi-mode interactions flexibly;

4. The richness of the context information makes multiple adaptation rules concurrently available possible, how to check the relationship between the adaptation rules in order to ensure that the rules themselves are not in conflict with each other;

5. In the adaptation cycle, how to assure the dynamic evolution process is consistent with the user’s expectation and how to model and analyze the decision-making process.
The above list gives some key self-adaptation problems, not only because these problems are essential in this scenario, but also because they are common and representative in an open environment. One hand, each problem is standalone and attracts many research efforts; on the other hand, these problems have underlying connections. To provide continual service during users’ mobility, application migration is desirable. During the migration process, it has to cope with the possible underlying heterogeneity. As these actions happen in an open environment, the context is much more complex. To ensure an effective adaptation, a careful study on the context model, adaption enactment mechanisms, the adaptation rule management, and related formal models is highly necessary.

1.5 Structure of the Book

Starting from the concrete scenario given above, in this book, we inspect the identified problems and attempt to develop more systematic solutions from a perspective of software architecture, including context modeling, adaptive service discovery and multi-mode interaction, adaptive component level migration, adaptation rule conflict detection, dynamic evolution verification, and Markov Decision Process (MDP) based decision-making process. The book is organized into three parts.

- The first part, which consists of four chapters, provides some basics and an overview of architecture based self-adaptation techniques. Chapter 2 presents the conceptual framework for self-adaptation in an open environment. Chapter 3 introduces the ontology-based context modeling techniques, and Chapter 4 presents a middleware platform to support the framework.

- The second part deals with specific issues of adaptive application migration and service interaction. It consists of two chapters, Chapter 5 and Chapter 6. Chapter 5 explains the solution for adaptive component-level application migration with users’ mobility in the open environment, while Chapter 6 presents the connector-based adaptation techniques for service discovery and multi-mode interaction in heterogeneous networks.

- The third part of the book covers the issues of formal modeling and analysis aspects of self-adaptation process. It consists of three chapters. Chapter 7 introduces the graph transformation-based adaptation modeling. On top of this, multiple architectural adaptation rules can be checked formally and the conflict/dependency relations can be detected. Chapter 8 introduces the techniques of modeling and verification of the enactment part of self-adaptation, i.e., dynamic evolution, so as to assure the adaptation is consistent with the specification. The formal modeling and analysis of decision-making process of self-adaptation is given in Chapter 9.
In the sequel, we would like to highlight some results which are achieved by the authors in the past 10 years via collaborations with other researchers. Some results were published in the form of research articles, but were somehow dispersed in the literature. Here, for the first time they are collected systematically in a single book.

1. An ontology-based context model is proposed. The necessity of explicit context modeling in an open environment is clarified. By introducing ontology into the architecture level and combining with our previous work on the dynamic software architecture, the semantic gap between external context models and the internal architectural specification is bridged. Therefore, the selection of adaptation strategies can refer to the architectural knowledge. Moreover, a basic conceptual framework for software adaptation is discussed. The framework can help to understand the related concepts, and techniques of adaptation in an open environment.

2. An adaptive component migration mechanism is proposed. With the loosely coupled application model and ontology-based context reasoning technique, the mechanism can support adaptive component-level migration. We also leverage attributed graph grammar to specify the changes of the architectural deployment view during the migration process. In this way, some properties, such as the deployment constraints, can be checked formally. Compared with previous work, the adaptive approach can reduce the network load and the response time.

3. A connector-based adaptation approach for heterogeneous service discovery and multi-mode interaction is proposed. In response to the heterogeneity characteristic of an open environment, the approach which is combined with the context-aware techniques can support service discovery and interaction in heterogeneous sub-networks. Preliminary experiments demonstrate the effectiveness of the approach.

4. A critical pair analysis based approach is proposed to detect the conflict and dependency relations of concurrent architectural adaptations. The architectural style is formally specified by the attributed graph grammar, and the reconfiguration adaptation is expressed by graph rewriting. Critical pair analysis can be used to statically check the relations between the rules. Four categories can be classified, \textit{i.e.}, parallel, dependent, asymmetric conflicting, and symmetric conflicting. With the graph grammar directed architecture development environment, the mechanism can help ensure that the software architecture conforms to the constraints of a specific architectural style in the process of composition and adaptation.

5. An integrated development environment, MAC-ng, is presented. MAC-ng is a middleware platform that supports architecture-based self-adaptive
software development in the open environment. It supports the ontology-based context modeling, reasoning, and the following architectural adaptation. The conflict detection module and the grammar directed editor are integrated with the platform.

6. A case study on the water management application is discussed. Water management usually involves a wide scope, diverse platforms, and complex context information. The thesis attempts to apply some of the proposed approaches to this domain. An experimental prototype is implemented. The experiments demonstrate the feasibility of these proposed approaches.

7. A behavioral modeling and verification approach for the dynamic evolution process is presented. Dynamic evolution is the underlying enabling mechanism of self-adaptation. To keep the evolution process consistent with the specification is a prerequisite of self-adaptation. Based on architecture characteristics in an open environment, we present a hierarchical timed automata based approach to model and verify the dynamic evolution process during adaptation.

8. An iterative decision-making scheme for self-adaptation is presented. The scheme infers both point and interval estimates for the undetermined transition probabilities in a Markov Decision Process (MDP) based on sampled data, and iteratively computes a confidently optimal scheduler from a given finite subset of schedulers. The most important feature of the scheme is the flexibility for adjusting the criterion of confident optimality and the sample size within the iteration.