Faculteit Wetenschappen

Departement Wiskunde & Informatica

Middleware-based adaptation evolution with reusable adaptation components

Jury:
Prof. Dr. Jan Broeckhove, chairman
Dr. Vincenzo De Florio, promoter
Prof. Dr. Chris Blondia, co-promoter
Prof. Dr. Francky Catthoor
Prof. George Papadopoulos
Porf. Serge Demeyer

Dissertation submitted to the graduate faculty of University of Antwerp in partial fulfilment of the requirements for the degree of DOCTOR of PHILOSOPHY

For

Ning Gui

Recommended for Acceptance
by the Department of Mathematics and Computer Science
September, 2012

Abstract

With recent development of mobile and pervasive computing, software applications are increasingly expected to dynamically adjust their behaviours according to the highly dynamic environments they are deployed in Applications must sense the environment changes and reacting upon those changes based on their contextual knowledge. For each new context, a new adaptation logic is required. This results in the high complexity of adaptive software development, especially for changing environments.

Multiple approaches have been proposed to facilitate the development of adaptive systems. However, such works mostly focused on providing support for particular set of systems and with pre-defined quality-of-service optimization goals. Therefore, they are costly to reuse in new systems and hard to adjust to other concerns. How to streamline the engineering of adaptation with multiple and evolving combinations of concerns remains a largely unexplored topic.

This work addresses this issue from two aspects: Firstly, a novel adaptive framework is proposed by extending the separation of concerns paradigm to adaptation logics design. System's global adaptation behaviour is contextually constructed with multiple reusable adaptation modules each of which embeds an adaptation strategy limited to one or more concerns. Rather than assuming these strategies to be orthogonal and thus not interfering with each other, this framework provides a systematic and customizable conflict detection policy and resolution mechanism. Secondly, a modular middleware architecture is designed to facilitate the incremental deployment of new and unforeseen adaptation modules. Software engineers are provided with the ability to add/remove/update adaptation modules during run-time. Development of adaptation modules is simplified by factoring out common adaptation mechanisms.

Our approach, embodied in an adaptation framework called Transformer, provides an engineering approach to monitor a target system and its environment, detect opportunities for improvements, select a course of adaptation strategies, resolve possible conflicts and apply changes to a software architecture. Our work was compared to the state of the art from both theoretical and practical viewpoints. Design evaluation and experiment results show that our system has significant advantage over traditional approaches in light of flexibility and reusability of the adaptation modules, with little complexity and performance overhead. Moreover, our framework was applied in a practical case study – autonomous robots control. Experience gained from this case justified that both the framework design and modular middleware-based implementation add significant value to developers in designing and incorporating new adaptation logics.

ACKNOWLEDGEMENTS

In the whole course of PhD experience, the most challenging part, I think, is the thesis writing period. I was not so well-prepared in the beginning to write such a big "book". Sweat and passion was spent and lessons were learned. After the course of misery and harvest, I finally made it. Now, I have a thesis truly belongs to my own.

Of course, a PhD thesis is never the result of a single person's effort. It would not have been possible without the support of many different people who help me from different perspectives. First of all, I would like to thank my advisor, Dr. Vincenzo De Florio, who provided his kindest support, understanding and encourage throughout my whole PhD research. I also want to express my gratitude to my co-promoter, Prof. dr. Chris Blonda, your consistent supports give the courage to live through those hardest time. The jury members of my PhD thesis committee also give me very important advices. Here, I will give me deepest thanks to you: Professors George A. Papadopoulos, Francky Catthoor, Jan Broeckhove and Serge Demeyer. Your support helps me go through the most difficult period in my life. I would give my special thanks to Prof. Tom Holvoet for your help and support.

Special thanks to my PATS friends Johan, Bart, Erwin, Nik, Michael, Kathleen, Nicolas etc. They also provided me with a great working environment and help me out of many personal problems. I enjoyed very much the time we spent together. Additionally, I would like to thank my friends, Jiwei, Xianglan, Ming for making the lunch breaks and other free times in the last few years so interesting and revitalizing.

My family is my most important of my thesis. My parents, Chengwang and Dongmei , give me their selfless-support and endless-trust. I would also like to give my thanks to my son –Wenfeng, who gives me endless happiness writing. Most importantly, I would like to give special thanks to my loving wife, Zhifeng. Your support and understanding help me go through the many sleepless nights and the difficulties in the thesis evaluation.

Ning Gui

Leuven, 2012

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Mobile devices, such as smart phones and iPad, are commonly used in people's daily life, as a consequence, software systems are increasingly operate in changing environments. Thus, more and more software systems are expected to dynamically self-adapt to accommodate for resource variability, changing user needs, and system faults or combinations of these factors. This thesis introduces those challenges in how to engineer self-adaptive software applications with existing mature solutions. We proposes a novel framework, which helps developers create contextual adaptation behaviour more efficiently, modular and with enhanced reusability.

1.1 **Thesis motivation**

Embedded devices, such as Set-top boxes, mobile phones, EReaders, already become an indispensable part of people's daily life. With the fast growing deployment of mobile and ubiquitous devices, there is a growing demand for autonomous applications that can dynamically adapt to their runtime environment with little or no human continuous guidance [153]. The fundamental source of this requirement is that human are surrounded with more and more computing devices. As pointed out by Paspallis [121]: "The ratio of computers (devices) -to- people is constantly increasing. While this ratio was originally well below one (mainframe era), it quickly reached the one-to-one value (personal computing era) and is now advancing to values well above one (mobile and ubiquitous computing era)." More and more user' attentions are required to interact with his/her surrounding devices. On the other hand, changing environments, typical in mobile and ubiquitous computing environments, normally require considerable human supervisions for software management to ensure software continue operation when changes happen.

These difficulties of managing today's computing systems lead to costly and time-consuming procedures and are referred as so called "a looming software complexity crisis". In 'The Vision of Autonomic Computing' [87], Kephart and Chess warn that the dream of interconnectivity of computing systems and devices could become the "nightmare of pervasive computing" in which humans are unable to anticipate, design and maintain the complexity of configuration and adaptation.

In this situation, only if computing systems became more autonomic, which is proposed by Kephart and Chess as "Autonomic Computing", could we deal with this growing complexity. This requirement means that software system needs to incorporate sophisticated context-aware, self-adaptive behaviours, which implies significant increase of the overall system design and implementation complexity. This complexity comes from three different aspects. First, when system operates in new environments, new adaptation logics are normally needed. Fast changing environments which is typical in mobile and ubiquitous computing, bring a great complexity in adaptation logic developments. Second, lack of methods to simplify the adaptation behaviour's development complexity: designing a new adaptation module is a very complex process; constraints from different aspects must be taken into consideration to ensure "correct" adaptation, such as application structure correctness, real-time, guaranteed lifetime, security, optimization etc. Third, for a changing environment, adaptation requirements cannot be always foreseen during the design time. In a typical ubiquitous computing environment, a number of mobile devices running adaptive applications may come and go in an unanticipated manner. How to deal with dynamicity also poses a major problem for adaptive software development. As pointed out by Oreizy, for a changing environment, not only the adaptive software needs to be changed to reflect new context requirements – the system adaptation behaviour itself should also evolve to reflect these changes[117]. Hillman and Warren [78] denote this process as "meta-adaptation."

The dynamic changing context proposed new challenges in adding context-specific and evolvable adaptation behaviours into existing self-adaptive applications as adaptation behaviours have to evolve with context changes. This thesis presents an adaptation framework that can construct system global adaptation behaviour during run-time with reusable and composable adaptation components. This framework, together with a support middleware implementation can effectively support the meta-adaptation in a modular, cost-effective and flexible way.

1.1.1 Self-adaptive software

There are several existing definitions for the self-adaptive software. One of the popular definition is given by Oreizy: "Self-adaptive software modifies its own behaviour in response to changes in its operating environment. [116]." Another well accepted definition was provided in a DARPA Broad Agency Announcement [95] on Self Adaptive Software (No. BAA-98-12) in 1997: " […] Self-adaptive software evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."

From those definitions we can find the basic form of self-adaptive software embodies a closed-loop mechanism, denoted as the "adaptation loop". Although many variants exists, they normally take a similar structure, which normally contains functions for sensing, planning and effecting. For instance, in the context of autonomic communication, Dobson et al. [48] defined an so called "autonomic control loop", including modules for collect, analyze, decide and act. Another famous reference model is defined in the context of autonomic computing, is called MAPE-K loop by Kephart and Chess [87]. This loop contains the Monitoring, Analyzing, Planning and Executing modules and an additional shared knowledge-base for sharing information among those four modules. This adaptation loop is also referred in self-adaptive software as "adaptation management" [116].

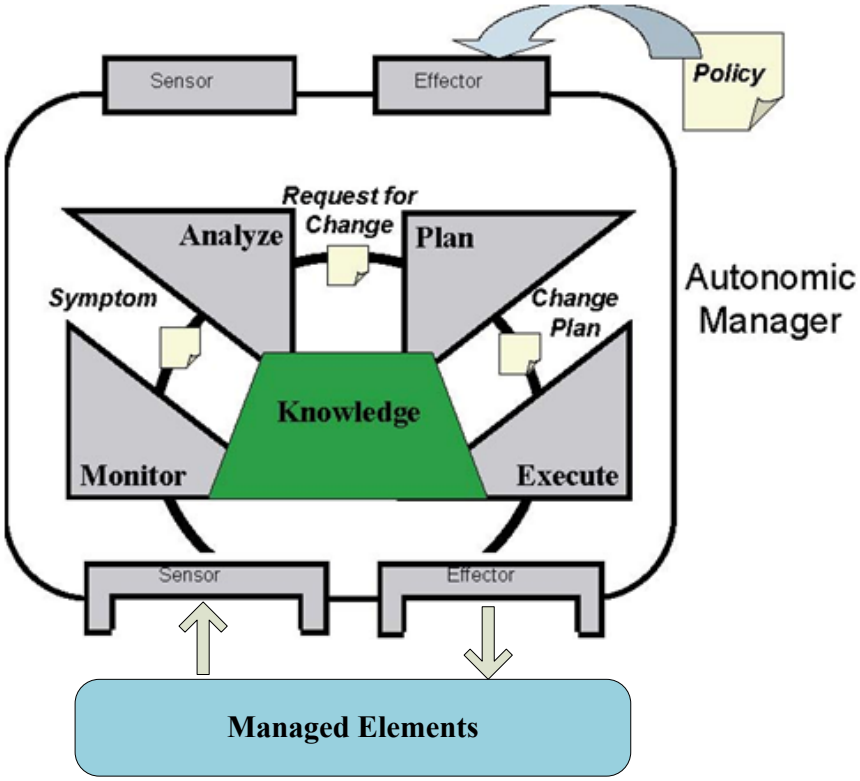


Figure 1-1. Autonomic adaptation manager – architectural model[87]

As Figure 1-1 shows, this closed-loop and the use of monitor and controller suggests the possible applications of control theory in self-adaptive software. Based on the control theory, an adaptation action can be calculated or derived from the difference between system's current behaviour with intended behaviour model. Such techniques have been successfully applied to many applications and proved their effectiveness. However, when an application is working in changing environments where the system intended behaviour models changes dynamically, as pointed out in reference [49, 110], it is less clear whether the techniques can work effectively. It also could not effectively requirements for real-time aspects or other hard constraints[103].

Partially due to this complexity in dealing with changing environments, the original vision of autonomic computing remains unfulfilled. Simon Dobson and his colleagues pointed out that "[…] yet the original vision of autonomic computing remains unfulfilled."

[49]. Many innovative adaptation solutions are devised to individual problems but how to combine those solutions into a larger adaptive software solution remains an unexplored area. Researchers must develop a comprehensive engineering approach to effectively build self-adaptive software by reusing existing solutions. Dobson later argued that: "More consideration must be given to integrating solutions and to choosing solutions from the range of possibilities".

We conclude that a comprehensive systems engineering approach with new conceptual models, development methods and related supporting tools are needed for the development of adaptation behaviours by reusing existing mature adaptation solutions. This approach is important for developing complex self-adaptive applications executing in changing and unexpected environments.

1.1.2 Introducing adaptation into applications

To reduce such software management costs, systems are required to dynamically adapt to accommodate changes. One of key design choice is to decide where an adaptation loop can be put into the application business logics.

One approach to introduce adaptation to application, shown in the left side of Figure 1-2, is called "Internal approaches" which intertwine application and the adaptation logic. Currently, mechanisms that support this type of adaptation are normally provided by certain programming language features, such as conditional expressions, exceptions and other specially designed features [55, 116]. Some languages, such as CLOS or Python, provide support inherently. While others, such as Open Java[142], r-Java[38], are extended from existing language with the introduction of new keywords and constructs. In this approach, key adaptation modules, such as sensors, effectors, and planners, are statically mixed with the application business logic. This high coupling leads to poor scalability and maintainability of developed applications. Furthermore, in this type of approaches, adaptation capability is provided to the targeted applications. Thus, it is not possible for the internal approach to provide system-level adaptation for multiple applications. As a result, self-adaptation in these systems is costly to build, difficult to modify, and limited in adaptation capability. Meanwhile, adaptation often needs global information about the system and correlating events happening within itself and/or from surrounding context.

Due to those limitations of internal approaches, many recent approaches choose the use of external adaptation loop to achieve various adaptation goals. In those approaches, adaptation models and mechanisms – sensing, reasoning and effecting etc. – are placed outside of managed applications. As shown in the right side of Figure 1-2, system adaptation control takes the responsibility of component management outside the system that is being controlled. Using this approach, a typical self-adaptive system normally consists of an adaptation engine and the targeted adaptable software. Compared to the internal approaches, external control mechanisms can provide a more effective engineering solution for self-adaptation as "they localize the concerns of problem detection and resolution in separable modules that can be analysed, modified, extended, and reused across different systems [61]." Due to its effectiveness, it is argued, such external control

mechanism is used extensively in many research projects. According to Mazeir's recent review [136], all of the surveyed 16 projects on self-adaptive systems use certain level of external approach, which supports separation of the adaptation mechanism from the application logic.

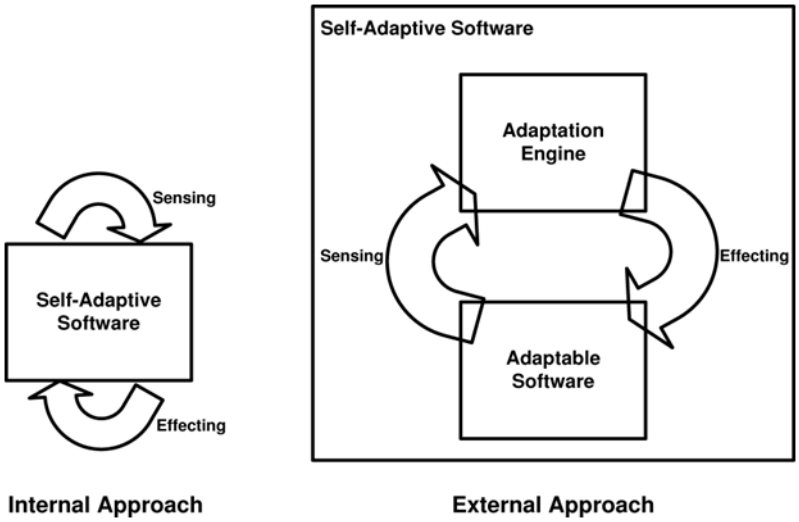


Figure 1-2. Internal and external approaches for building self-adaptive software[136]

Several researchers, such Oreizy and Garlan, have proposed to use architectural models [42, 60, 61, 116, 118, 155] to guide adaptation externally. In those approaches, the system is represented as a gross composition of components, their connections, and their properties. Adaptation is achieved by changing this component composition. As an abstract architectural model provides a global perspective of the system, it can explicitly represent system-level integrity constraints and properties that can be used across several applications. A well-accepted design principle in architecture-based management is to use component-based technology to develop management system and application structure [61, 93, 138]. Run-time application adaptation is achieved by run-time composition and reconfiguration of software components.

In this thesis, it is assumed that the developed adaptable applications comply with this external adaptation model and component-based technology is used to develop application structure. McKinley et al.[107, 108] pointed out that there are two major class approaches for adapting software from the form of adaptation actions: parameter-based and compositional adaptation. The parameter-based adaptation changes a component's parameter such as the encoding quality of a camera sensor. The compositional adaptation can perform more complex changes to the software system's architecture, for instance, adding, updating and/or removing components towards managed software systems.

1.2 **Engineering adaptation modules**

To avoid increasing management complexity, applications should be constructed and automatically adapted to their changing contexts. This concept is known as context-aware software adaptation. As an example, a video player program for a mobile phone can be optimized during the run-time by selecting computation components according to the target device's characteristics – for instance, CPU and screen size.

Several dynamic service composition/reconfiguration systems have already been proposed. In these solutions, customized adaptation strategies are used to deal with one domain-specific optimization goal, such as the self-healing in [138], or the performance oriented optimization in [61]. These systems exhibit very good results in their designed domain. However, these solutions are rather confined to a specific domain and cannot effectively deal with constraints from different aspects. As systems are bound to work in ever more complex environments with multiple co-existing requirements – for instance, providing performance optimization while providing self-healing behaviour – it becomes increasingly important that the system adaptation strategy take evolving and multiple adaptation goals into account. Sometimes, these adaptation goals themselves also contain other quality constraints. For instance, for the embedded systems, the self-healing algorithm might also need to be finished within certain time limits.

AOP based programming [89] provides ways to support simultaneously multiple adaptation strategies by injecting aspects offline [88] or during run-time [32]. This strategy can be oriented towards, for instance, security or performance optimization. However, aspect languages normally make assumption (or aim for) so-called "obliviousness," which assumes that different aspects are orthogonal and thus will not interfere with each other. In many situations, this assumption is not valid (or very hard to achieve) [65]. Moreover, existing AOP solutions lack the expressiveness to specify a number of interactions as well as possible conflictions between different aspects. Absence of this expressive power results in uncontrolled semantic interference that can endanger the integrity of component software, as existing contracts might breach because of conflicts between different aspects [28].

In existing solutions, systematic support towards adaptation with multiple domain-specific goals remains an unexplored topic. In order to solve this problem, three major challenges are identified here.

1.2.1 Framework support for run-time adaptation logic evolution

First challenge is how to provide framework-level support for run-time adaptation logic evolution. Similar to the self-adaptive software while should adapt according to the system changes, system adaptation module in itself must also able to be customized or changed during run-time to reflect current context concerns. For instance, in mobile environments, what is the "best" composition strategy of a media player program depends on user's current preference as well as mobile device's current state. However, in current

approaches, such as query-based component selection in declarative service [119] and component repairing algorithm for self-healing system[138, 155], predefined and immutable adaptation strategies are used. These static adaptation approaches could not effectively handle changing adaptation logics.

1.2.2 Enhance the reusability of adaptation logics

Second challenge is to be able to enhance the reusability of adaptation logics. Currently, adaptation modules are designed for specific environments with pre-defined environment assumptions. The resulting adaptation strategies are normally statically integrated in the underlying system implementation details. There lacks of clear separation between adaptation planning and actuation. At the same time, no clear adaptation interface definition make adaptation modules very hard to be reused in other environments. How to design a system that can effectively enhance the reusability of system adaptation models is one of our current major research topics.

1.2.3 Integration with multiple adaptation modules

Third challenge is the ability to integrate multiple adaptation modules, which provides its domain-specific optimization strategies. In order to make coherent and pointed context-specific adaptation, an effective adaptation framework not only takes different domain-specific optimization strategies and/or constraints into account, but also provides explicit support for conflict detection and resolution mechanisms. However, existing solutions provide no clear integration process for these different domain-specific optimization goals.

Certain approaches assume such conflicts do not exist. As an example, in AOP based solutions, different aspects are directly integrated without providing such integration support.

In other adaptation frameworks [61, 86, 138], adaptation integration processes are only implicitly discussed. For instance, in [138], the main concern is application self-healing while concerns for software structure maintenance are only implicitly considered and integrated.

Other approaches use utility functions to solve conflicting goals [40, 83]. However, as in different contexts, the utility values tagged to each context should also be altered to reflect new context concerns. A pure utility function based approach apparently could not provide this service.

In current approaches, there exists no systematic integration process between those possible conflicting concerns that able to be used across multiple contexts. How to resolve those complex relationships between different domain-specific goals remains unexplored in these approaches.

1.3 **Thesis statement & contributions**

Traditional approaches for developing self-adaptive software applications are facing difficulties in dealing with complex environments with multiple adaptation requirements. There is insufficient support for the developers to engineer complex adaptation behaviours in the self-adaptive applications. This thesis claims that systematic combination of adaptation logics modularity, appropriate adaptation evolution method and adaptation composition mechanism can make the adaptation module development cost-effective, efficient and easy of usage.

The main contribution of this thesis is as follows:

1) First, the design and implementation of an architecture-based adaptation framework and its supporting middleware is introduced. This framework provides the explicit integration support of multiple domain adaptation logics. The basic adaptation modules realize the roles of adaptation in specific domains and can be reused across multiple contexts.

2) Second, system adaptations behaviour is orchestrated by run-time composed adaptation modules according to context to date. Just like component are composited to build application, in our framework, multiple adaptation modules can be used to compose more complex global adaptation behaviours.

3) Third, since different domain-specific optimization behaviours have radically different properties of interest and a reconfiguration strategy, by explicitly defining a set of model fusion rules to resolve possible conflicts, our framework builds an architectural control model in an easy and verifiable way.

1.4 **Thesis innovations**

Compared to other existing context-aware dynamic adaptation framework [61, 66, 93], our approach is novel in that

1) Modular Adaptation logics. Our framework provides the design principles, as well as supporting reflective component model to structure adaptation logics. This component-based design enhances adaptation logic reusability and modularity;

2) Meta-adaptation via adaptation composition. In our approach, the adaptation evolution is achieved by run-time selecting and using multiple domain-specific adaptation strategies. This approaches can effectively support changing context with multiple adaptation concerns;

3) Explicit adaptation composition support. Our approach explicitly separates the adaptation integration process from other adaptation behaviours. This design makes the research of conflict detection & resolving a vital part of adaptive software researches.

4) Adaptation validation support. Our approach identify the necessarily in verifying the coherence of adaptation plans. An online verification algorithm is designed and integrated into system run-time to help the system reach stabilization in a limited number of steps, and the convergence criteria are analysed and proved.

The effectiveness of our architecture is demonstrated both from a qualitative and a quantitative point of view. Simulation results show the soundness of our implementation in terms of lines of code, adaptation capabilities and reusability of adaptation modules.

1.5 **Methodology and implementation**

This thesis illustrates our researches in providing adaptation behaviour development support from different perspectives. Firstly, in order to support more engineered way of adaptation development, a methodology is developed to facilitate the development of adaptation strategies by modularizing adaptation logics into reusable and composable components. Rather than limiting *Separation of Concerns* (SoC) only in application design, this methodology extends SoC from application development into adaptation logics development. This methodology includes a meta-model for expressing adaptation modules' contextual feasibility. This meta-model is used to facilitate the selection of adaptation modules. The methodology is also supported by a conflict resolution model for fusing multiple adaptation modules. Guided with this methodology, an adaptation framework, so called Transformer, is proposed. This framework provides supports for adaptation evolution, adaptation composition and adaptation validation.

In addition to the discussions of adaptation online construction methodology, this thesis introduces our work in the design and implementation of adaptive middleware which supports run-time adaptation behaviour construction. This middleware provides support for the deployment of domain-specific adaptation modules, which contain domain-specific adaptation logics. An adaptation behaviour component model is defined to enhance adaptation logics modularity and reusability. Rather than statically predefined as a monolithic module, system global adaptation behaviour is now contextually constructed during run-time. By compositing global architecture adaptation model during run-time, it is argued; this middleware simplifies adaptation module development and enhances the reusability. On the other hand, our approach allows hybrid approaches. It means existing adaptation algorithms designed and tested for a particular context, can be easily integrated into the system without major revision.

1.5.1 Adaptation composition – a new adaptation development methodology

In this thesis, in order to support adaptation evolution, an online adaptation construction methodology based on adaptation behaviour composition is proposed. This methodology is designed to support the development of system global adaptation behaviours in changing contexts, which becomes more and more command in the mobile and pervasive computing environments. In this methodology, a component-oriented approach is adopted to organize adaptation logics. The SoC design principle is used in two

different levels: first, applications are split into business logic and adaptation logics. This approach is already wildly used in many architectural –based adaptation approaches. Secondly, we extend SoC to the design of adaptation modules which makes adaptation modules can be used similar as normal components to construct system global adaptation behaviours.

System adaptation strategies are divided into adaptation building blocks representing particular optimization goals matching certain "situations". Each of those blocks assumes some contextual hypotheses are valid and provides one domain-specific adaptation solution. Just as components are used to compose applications, these domain-specific adaptation blocks can be used to construct more complex adaptation strategies and (possibly) be reused across multiple contexts.

Self-adaptation modules for multiple contexts are generated by an elaborate adaptation module composition model. This model provides context selection support and enables advanced functionality such as adaptation conflicts resolution. Finally, the methodology is supported by an online-verification algorithm to detect and resolve adaptation errors for the constructed adaptation model.

1.5.2 Modular and pluggable middleware

The second contribution of this thesis is a service-oriented middleware that is designed to support the proposed Transformer adaptation framework. This middleware facilitates the reuse and composition of adaptation modules for generating adaptive applications with contextual, multiple adaptation concerns. This middleware builds on top of the OSGi (formerly refers as *Open Service Gateway initiative*) framework [114], where the adaptation modules are defined and implemented as service components (terms used in declarative Service in OSGi v4.0). In this architecture, each adaptation module registers their adaptation service and capabilities in a central authority – the *adaptation modular manager*. According to current context-information from *context manage*r, the *adaptation modular manager* manages the lifecycles of these adaptation modules. By selectively activating the most appropriate set of adaptation modules and composing them into a context-specific adaptation module, adaptation evolution is explicitly supported. At the same time, this selective activation mechanism also optimizes system resource consumption. Our middleware also utilizes a modularized architecture which allows composition of different adaptation modules into a more complex global adaptation module which greatly enhances the reusability of adaptation modules across multiple contexts.

1.6 Thesis outline

This thesis is structured as follows:

Chapter 2 introduce the basic concepts that will be used in the following part of thesis. Key concepts such as the terms of context, software architecture, and adaptation are

defined with Quasi-formal format. Later a conceptual adaptation model with composable adaptation logic is defined for building self-adaptive applications with changing set of domain-specific adaptation concerns.

Chapter 3 provides the introduction of related work from two different perspectives. Firstly the researches on key enabling technologies for self-adaptive software development are surveyed. Then the key projects on middleware-based adaptation frameworks are studied. This chapter also surveys challenges and requirements identified in the literature, and examines specific solutions proposed by state-of-the-art approaches.

Chapter 4 illustrate design of the Transformer adaptation framework based on the foundations of the previous chapters. Key modules within this framework are introduced that enables the proposed adaptation composition model. Later in this chapter, the mathematic models of this adaptation process are specified. An on-line verification algorithm is proposed to avoid system entering infinite adaptation loops.

In Chapter 5, a supporting middleware architecture is introduced in supporting the Transformer adaptation framework. In this middleware, a service component model is defined to support the adaptation composition with meta-data and reflection support. This middleware allows adaptation component to be installed, updated and removed during run-time. This continues deployment support allows middleware to support future unforeseen adaptation behaviours.

Chapter 6 presents a practical use-case to which this middleware is applied. Two adaptations domain-specific modeller are designed for battery-based application construction and self-healing. This use case shows the cost-effectiveness in using our framework for building self-adaptive applications.

Both the framework design and the pluggable architecture are evaluated in chapter 7. This thesis is evaluated from both functional requirement and non-functional requirements identified in the chapter 1. Later that chapter, the design issues and possible limitations are outlined for discussion.

Finally, Chapter 8 makes final conclusions. This chapter summarizes this thesis's major contributions and proposes several directions for further improvements.

Chapter 2

Preliminaries

While Chapter 1 introduced the motivation, challenges, goals and the outline of this thesis, this chapter aims to provide background knowledge of the basic concepts related to software development in general, as well as on the application adaptation process for component composition in particular.

Firstly, this chapter provide introductions for the basic concepts that will be used in later part of this thesis. Quasi-formal definitions for context, adaptation, and system configuration evolutions are then introduced. Based on these definitions, the middleware-based solution is introduced as an enabling technology for building systematic support for adaptation across multiple contexts. Finally, a high-level conceptual model for designing such a middleware platform is presented. This conceptual model is used to analyse surveyed literatures' support for meta-adaptation. The chapter also provides a discussion on the challenges in realizing adaptation composition.

2.1 **Basic concepts and definitions**

This thesis is focused on the adaptation composition-based support for contextually constructing adaptation behaviours. Those adaptation behaviours will reconfigure application software architecture when system is facing changes. Before intensive discussion about this topic, some fundamental concepts and technologies that will be used later in this thesis are introduced. Some of them, such as component, connection, component composition graph and configuration evolution, are defined with quasi-formal definition.

2.1.1 Context

There are many different definitions for "context". One of the most accepted definition of context is from Dey in year 2001 [43, 44]: "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 the application themselves". A context instance is normally represented by several contextual metrics and their values.

2.1.1.1 Context expression

In practice, context can be represented by a vector, called a context vector with several orthogonal types. These types can be infinite (e.g., time) while others can have limited ranges (e.g., the switch state for a light can only "on" or "off"). As context is expressed as a vector, the context can be expressed as a multidimensional space. For instance, a typical context for mobile phone can be expressed as {{CPU_Usage, 0.5}, {Free_MEM, 20}, {Battery, 20}}.

Here, the format proposed in the literature [35] is used. In this approach, the context is denoted by Z while Z = R V, here R is the set of system resources or metrics used by the system and V is the set of values those resources or metrics may take. Then, at any time t, the context can be represented as a vector $r(t) = (r_t^1, r_t^2, \dots, r_t^{n-1}, r_t^n)$, where r_t^n represented the value of context type r^n at time t. As suggested in literature [121], instances of those vectors is denoted as "context instances".

When context vector changes, system context will migrate from one configuration to another configuration, this process is denoted as "context migration".

2.1.1.2 Context migration

In these definitions, the *context* is denoted as an n-dimensional space, and a context instance is a point in this apace. Context changing means at any time specified as t and t', ($t < t'$) context instance r_t migrates to another state $r_{t'}$, whiler$_t \neq r_{t'}$. Context migration is quite usual in current mobile and ubiquitous computing environments. In order to cope with these changes, an application configuration will also need certain changes. In the following section, the key elements of software architecture are defined for the compositional based adaptation, which will be used in our framework.

2.1.2 Software architecture & supporting concepts

The origin of the concept "software architecture" was first identified in the research work of Dijkstra's article on the "THE" operating system in 1968 [46]. Software architecture as a discipline was first proposed by Perry and Wolf [123]. Since the mid 1980s, the fundamental design principles of the software architecture have gradually been applied by software researchers. In 1996, Shaw and Garlan [137] published an introduction to many issues of software architecture. The above researchers agreed that the structure of a software system has significant impact on software performance and how to get the

structure right is very critical. In practices, the discipline of the software architecture highly relies on the idea of reducing software development complexity through the design principles: abstraction and separation of concerns.

However, until now, there is still no common agreement on what is the precise definition of the term "software architecture". One widely cited definition of this term is "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [27]. In some systems, these elements can be the physical components of the system. In many other cases, these elements are not physical, but instead, logical components.

In the software architecture domain, the term "architectural style" is defined to specific a set of software systems from the perspective of structural organization pattern. An instance of organization pattern includes the semantics of basic elements such as components, connectors etc., together with a set of constraints on how those elements can be composed. Other constraints on architectural structure, such as absence of cycles, could also be included in the style definition. In respect to self-adaptive software, while adaptation actions are performed, these constraints must be maintained.

An architecture design needs to be based on the principal considerations of overall functionality, and other non-functional requirements, such as performance, reliability, security and etc. Expected levels of functionality and performance are required during application design process. However, all too commonly, aspects of such non-functional requirements are ignored (or partially so) on architectural levels [147]. Emphasis is only put onto functionality. Depending on the deployment context, consequences of this can be fatal. Thus, to view all aspects as prioritized facets of a whole in any software architecture remains an important goal. This is especially true in the ubiquitous environments, in which system configuration must continually adapt with its external changing environments. Based on the researches of software architecture discipline, architecture-based adaptation is proposed to steer software system adaptation according to environmental changes.

2.1.2.1 Component

In 1968, Douglas McIlroy introduced the ideas that software should be componentized at the NATO conference on software engineering. In his address titled with *Mass Produced Software Components* [106], he denoted that software application can be built from prefabricated components. Clemens later defined component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition " [147].

Let U be a set of unique IDs in string form, O be a set of provided interfaces (which represent the functional contract that a component will provide), R be a set of required interfaces (which represent the functional dependences a component might have), P be a set of component properties, S be a set of component states, i.e. S = {Installed, Resolved,

Started, Stopped, Uninstalled}. Then, a component c is defined as a tuple of the form (u,o,r,p,s), where u ∈ U, r ∈ R, p ∈ P, s ∈ S, o ∈ O. Component is a basic unit for composing applications, as shown for instance in Figure 2-1.

2.1.2.2 Link

Let T be a set of components as defined in Definition 1. It defines Q ⊂ T x T, a relation such that for any two components c and d, c Q d ⇔ there exists a direct dependence of d from c. Such dependence could be e.g. a service invocation, or an event notification, or data flow dependence.

Let q be the type of the relationship Q and then a link is defined as a tuple in the form $\{v_i, v_j, q\}$ where $i, j \in \mathbb{Z}^+$; $q \in Q$; $i \neq j$; $v_i, v_j \in T$.

2.1.2.3 Component composition graph

These software elements – software components, and their relations – Links, build software applications. A Component Composition graph (CCG) is a structure taking the form of directed graph which dynamically represents the current system configuration. A directed graph CCG=(V，E) is used to express such the configuration variant. V is a nonempty set of Components as specified in Definition 1 and E is a non-empty set of links as specified in Definition 2. As shown in the upper part of Figure 2-2, the directed graph structure of all installed components and their links forms the configuration A – one configuration out of the set of all possible CCG.

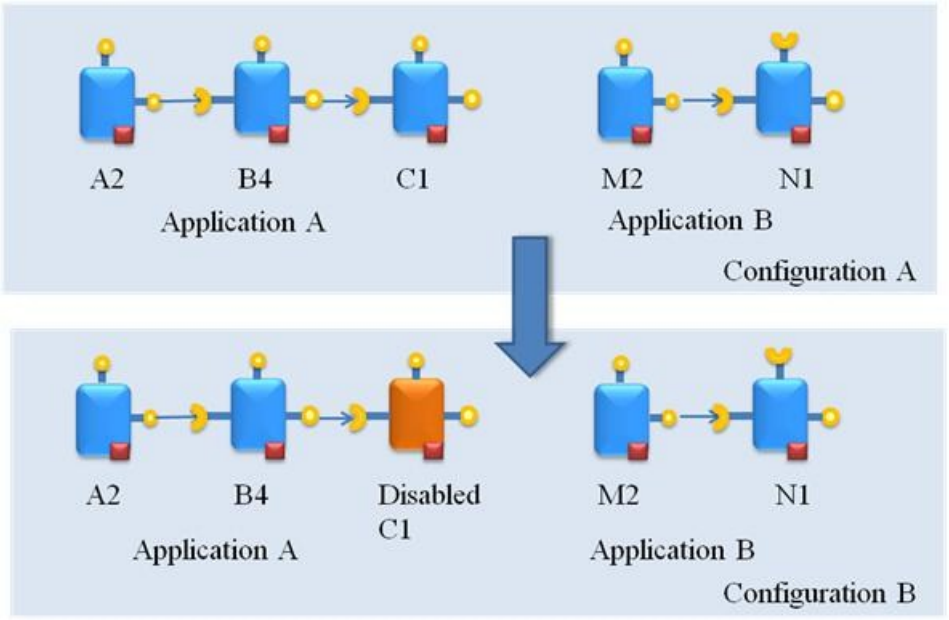


Figure 2-1. Dynamic evolution of CCG

The CCG is a reflective structure which will evolve when the system adaptation actions are executed. The adaptation actions let the system transit from the current configuration into new ones.

As components provide their services through well-defined interface, this merit allow the actual software configuration to be changed into other alternatives while keeps the

architecture of an application largely intact. These alternatives are called CCG variants. A basic principle of alternative CCG variants is that the new variant keeps the original functional goals and constraints, such as the functional dependences, while the non-functional properties can vary. From this perspective, the adaptation behaviours of self-adaptive systems can be seen as the migration to another CCG variant with new non-functional properties that are appropriate for the current contextual requirements, to optimize the system overall performance.

2.1.3 Architecture-based adaptation

In order to allow managed systems to self-adapt with minimal or no human interaction, the "loop of control" is wildly used in the self-adaptive software implementation. As shown notionally in Figure 2-2, this closed-loop control consists of three major parts: a) mechanisms on the system monitoring, b) deliberates on the problem observations, and c) controls the system and keeps it in a preferable state. They are placed outside of target adaptive software applications. This kind of structure is very similar to the feedback control system in control theory [98]. As it is already identified in last chapter, external approaches are normally used to implement such control-loop, as these approaches generally provide a more effective solution than internal mechanisms[107].

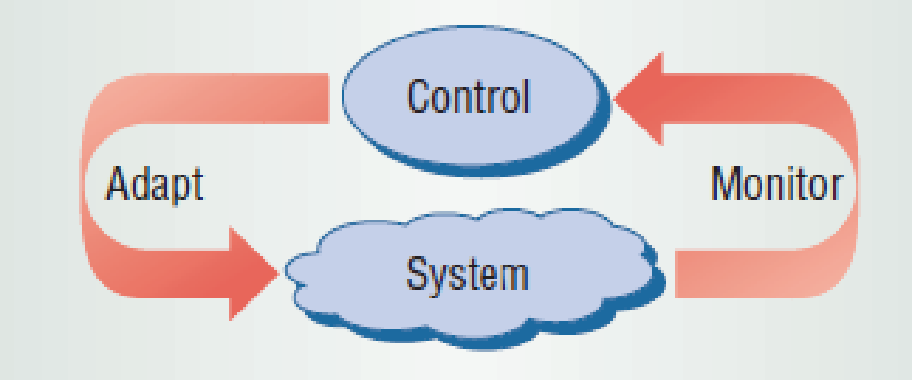


Figure 2-2. External closed-loop control

For the external adaptation loop, as it is designed separated from managed applications, how to effectively construct target adaptive software application and how to implement effective sensing and effecting mechanisms that can be reused across multiple applications becomes a major design challenges.

The development of software architecture, as it provides a powerful design and run-time abstraction, make it prominent enable technology for software adaptation. The architecture of a software system is represented as a abstract model in which "represent the system as a gross composition of components, their interconnections, and their properties of interest"[61]. With the help of software architecture abstraction, architecture-based self-adaptation approaches are proposed in which external adaptation approaches are adopted with the management applications' software architecture as adaptation target. As the abstract software architecture model can provide an accurate global perspective of

targeting software and can explicitly express system integrity constraints to the adaptation programmers, these features greatly facilitate the adaptation implementation process.

Due to these advantages, a major stream of researches advocates the use of so called "architecture-based adaptation" and applied it into several self-adaptive software domains [60, 61, 116, 155]. In the remaining part of thesis, the discussions of software adaptation are focused on the architecture-based adaptation approaches. However, architecture adaptation often requires a model of the system's execution environment [91]. This model is often referred to as context. In the following sections, basic concepts used in this thesis will be introduced and mathematical definitions are provided when possible.

2.1.4 Compositional Adaptation

Changes on a component's properties or the composition towards an application's architecture will make software adapt to different configuration. However, adaptation is inherently a dynamic process. From the CCG definitions we can see that there are many acceptable choices in terms of functional dependences. However, we normally will choose a particular one according to certain design choices or system adaptation goals. Those choices make applications dynamically migration among the space of feasible variants.

In respect to software, adaptation means software adjusts its configuration, behaviour in response to the changes of environment and the user preference. The advance of mobile computing and pervasive computing makes software adaptation becomes mandatory and more and more researches had been done in this filed.

In order to make adaptation, systems are designed to be reconfigurable, e.g. change to another CCG variant, i.e. a configuration with different combinations of property settings and component compositions. The adaptation actions are taken to maximize the utility of this software system in the current context instance.

2.1.4.1 Adaptation operation

An operation is one of the basic actions to change the state of certain basic element such as the state of components or the links between components which results in the alternation of current CCG. Operations may prune a sub-set of the component tree, change state of graph nodes, change state of one link, disable or reconfigure a link. In Figure 2-2, one adaptation operation is performed to disable component instance C. The set of operations that can be taken towards CCG highly depends on system current CCG configuration, denoted as $\mathcal{A}(\mathrm{t} = \mathrm{now})$.

The set of adaptation operations that can be performed is highly dependent on the underlying system support. For the architecture-based adaptation frameworks, there are two basic sets of software adaptation actions: parameter-based adaptation and compositional adaptation [107]. The parameter -based adaptation refers to actions that change a parameter (for instance, the compressing quality of a video processing component). The compositional adaptation normally refers to more significant changes,

which possibly includes adding/ updating/replacing/removing component instances within the system.

In this thesis, both adaptation actions are supported by the managed applications. For the parameter-based adaptation, it is implemented by using a predefined management interface that allows external programs to access and control certain parameters at run-time. For the compositional adaptations, an application is defined as a service composition architecture instead of hard-coupled component map. This design allows application configuration can be changed by using alternative component realizations. In order to achieve these two types of adaptation, a declarative and reflective service component model is introduced.

These operations result in modifications of a CCG. We call this process as configuration evolution.

2.1.4.2 Configuration evolution

During the run-time, when an adaptation operation is performed on a software system, system CCG will migrate from its current configuration variant into a new variant. We call this process as Configuration evolution.

Let SC = $\{cf_0 \dots cf_{i,} \dots cf_j \dots, cf_m\}$, m $\in \mathbb{Z}^+$ be a set of continuous configurations where cf_i is the current configuration and cf_m is the targeted configuration. Also, let $\mathcal{A}$ = $\{fx_0, .. fx_i, \dots, fx_n\}$ n $\in \mathbb{Z}^+$ be the set of operations that can be applied to CF; then, the configuration evolution CE is a sequence of interleaved configurations and operations such that

$$CE: \quad cf_i \xrightarrow{fx_k} cf_{i+1} \qquad fx_k \in \mathcal{A} \quad i, j \in \mathbb{Z}^+ \qquad (1)$$

The set of system configurations SC and the one-step transition links between different configuration constitute the Configuration Transition Graph (CTG), denoted as CTG = (SC,CE). SC is assumed to be nonempty. Figure 2 shows a possible configuration evolution process. In order to reduce the complexity of analysis, we assume a particular software system can only have a finite set of alternative system configurations.

In many self-adaptive solutions, these CCG variants are pre-defined by self-adaptive software developers during application design time. Examples can be found in many approaches based on the internal adaptation loop. However, in order meet the changing requirements of highly dynamic environments, it is more flexible to let the variants to be formed dynamically at run-time. In the domain of architecture-based adaptation, variants can be calculated by examining the both the software functional requirements and non-functional requirements. The satisfaction of functional requirements can be easily checked by matching the provided and required service contracts of each enabled components[75], for instance, the java interface-based matching in [119] or port-based matching in [143]. With the condition of satisfying functional dependencies, it is possible to form variants that by using different compositions of components. However, in any given context, due to its context-specific requirements, the actual configures are only a small set of functional satisfied variants.

In this thesis, our primary focus is on adaptation towards component-based applications. As our goal is to design and implement an architecture-based adaptation framework adapting multiple component-based applications. This system also assumes to operate in a changing environment with multiple optimization goals coexists. To the best of our knowledge, how to simultaneously support multiple adaptation goals remains an open question.

2.1.5 Adaptation strategy

According to system current context, adaptation operations are executed to let software system architecture enter into more appropriate state, which we denoted as context-aware adaptation strategy. This concept illustrate to the behaviour of software system to sense the environment and autonomously react to those changes in order to accommodate those changes.

This strategy can be seen as the Sense-Analyze-Plan-Act architecture used in autonomic computing [87] and is extensively studied in Artificial Intelligence. The state of the art includes different approaches for enabling autonomic adaptation decisions, such as Finite State Machine based approach [81] and utility function based approach [121]. In this thesis, *FSM –based adaptation* is used as it fits well with multiple domain-specific adaptation models. *FSM* model facilitate the fusion process of multiple modellers, which will be discussed in Chapter 4.

An adaptation strategy can be expressed as a sextuple (C, A, S, s0, δ, ω), where:

C is the input context information (a finite, non empty set of string-value pairs).

A is the output adaptation operations (a finite, non-empty set of adaptation actions).

S is the states of the CCG variants (a finite, non-empty set of states).

s0 is the initial CCG configuration, an element of S.

δ is the CCG variant selection function: $\delta : S \times C \rightarrow S$.

ω is the output which contains a set of adaptation actions.

The state of the system will change over time, partially due to the action choice of the decision made by adaptation modellers. In each state, $s \in S$, there are a number of actions, $a \in A$, from which the decision maker may choose. The destination state $s' \in S$ is determined according to the transition function δ. Of course, for certain mission-critical system, the adaptation function might contain more constraints, such as the response time of the δ transition function.

2.2 **Basic design methodology**

In order to deal with the adaptation problem outside single application scope, architecture-based adaptation frameworks are proposed in [61, 116, 117] to handle the cross system adaptation. Rather than scatter the adaptation logics in different applications

and represent them as low-level binary code, architecture-based adaptation uses external models and mechanisms in a closed-loop control fashion to achieve various goals by monitoring and adapting system behaviour across application domains. A well-accepted design principle in architecture-based management consists in using a component-based technology to develop management system and application structure.

In this thesis, architecture-based adaptation is used to guide the design of adaptation framework, in which component oriented development is used to build applications, SoC is used for the design of both application logics and adaptation logics and meta-adaptation layer is proposed for supporting changing adaptation requirements.

2.2.1 Component oriented development

Component oriented development is a branch of software engineering that stresses on the use of SoC in the design of software application. This practice normally decomposes target software application into separated entities so that the data and functions are inside each entity. With this separation, those entities are only semantically related.

These entities are represented as components. Component has several different definitions. Here, we used the wildly cited Szyperski's definition[147]: software component is: " a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition".

Component oriented development provides many good features in application design and implementation. It enhances the reusability of software by formally specify the functional role of a given software entity. Interfaces are used to define a service contract which specifies what services how the rest of the system can utilize the provided service. Components can only communicate with each other only through their interfaces. This interface-based design makes components can be independently developed. Multiple vendors can only implement different components with the same interface contract.

2.2.2 Separation of concerns

In the computer science domain, Separation of Concerns (SoC) is normally regarded as the designing process that spate a computer program into distinct features with little or no overlapping functions. This term was probably invented by Dijkstra in the paper "On the role of scientific thought"[47] in 1974. Later, it became an accepted idea through the inclusion of the term in [127]. SoC is intensively used in this thesis as a key principle for designing and implementing adaptive applications with multiple and changing adaptation concerns. In this thesis, two levels of SoC are used: separation of adaptation from business logics and separation of domain-specific adaptation.

2.2.2.1 Separation of adaptation from business logics

In order to separate the adaptation mechanism from application logic, two major sets of approaches can be identified.

One approach – application based adaptation, keeps this separation only at design time. In this approach, the whole set of sensors, effectors as well as adaptation logic is integrated together with an adaptive application's business logic. These adaptation modules are mixed with the application code. The deployed execution entity intertwines application and its adaptation logic. In the implementation perspective, this approach requires special programming language to support such adaptation. It is best fit for handling local adaptations. However, due to the limitation of application-based adaptation, it lacks the capability to provide global context knowledge of the current systems.

Architecture-based adaptation uses an external adaptation engine, which takes the responsibility to manage adaptation processes by changing the configuration of managed applications. A typical architecture-based adaptation system contains of an external adaptation engine and other system level sensing and actuation mechanisms that are reused across multiple application domains. This external engine contains adaptation logic and is normally implemented with certain middleware platform [93] or with the help of reusing certain a policy engine [139, 156]. Here, the discussion is limited to the localized situation. In complex and/or distributed environments, it is nature to have multiple such adaptation system with those key elements. Then, how to compose those elements into a coherent architecture and how to support interoperability between those elements becomes an essential problem.

2.2.2.2 Separation of domain-specific adaptation

In the most researches in the architecture-based adaptation, the separation of concern paradigm is limited to the process of separating adaptation logics from business logic. The adaptation module is treated as single entity and focuses on providing supports for particular set of systems and predefined quality-of-service optimization goals, for instance, self-healing, security or QoS. However, in order to make coherent and accurate adaptation, an adaptation framework needs to take adaptation concerns from different aspects into consideration, such as, for instance, component repairing algorithm for self-healing system [138] need to respect two different concerns – application structure maintenance and self-healing adaptations. Due to its complexity, this one-for-all solution makes adaptation makes system adaptation module very complex and hard to be developed for new system or for a new context.

In order to simplify the development of adaptation, the SoC paradigm is extended into adaptation modules design. We believe that a better solution, it is argued, is to divide the adaptation strategies into adaptation building blocks representing particular optimization goals matching certain "situations". Each of those blocks assumes some contextual hypotheses are valid and provides one domain-specific adaptation solution. For instance, one adaptation module designed specifically for application structure maintenance, another module is designed for self-healing mechanisms.

Just as components are used to compose applications, these domain-specific adaptation blocks can be used to construct more complex adaptation strategies and (possibly) reused across multiple contexts. By contextually selectively reusing these adaptation modules, system's global adaptation behaviour can be built according to changing contextual

requirements. This means, with the different contexts, system's adaptation behaviour can evolve into new set of adaptation concerns. In order to systematically support this adaptation evolution, a new meta-adaptation layer is needed to control the process, which is denoted as Meta-adaptation Layer.

2.2.3 Meta-adaptation

As we can see from Figure 2-3, the lower level of the diagram represents the typical adaptive software system. The adaptation module monitors system changes and come out adaptation plans and controls the targeted system. However, as during the whole lifecycle of adaptive software system, this system normally needs to have different adaptation behaviours or concerns in temporal changes. In order to deal with adaptation evolution, a new layer – meta-adaptation layer is introduced. This layer monitors the changes of contexts, such as user's preference, plans changes or changes of adaptation configuration and reuse available adaptation modules and composite them into a consistent adaptation plan.

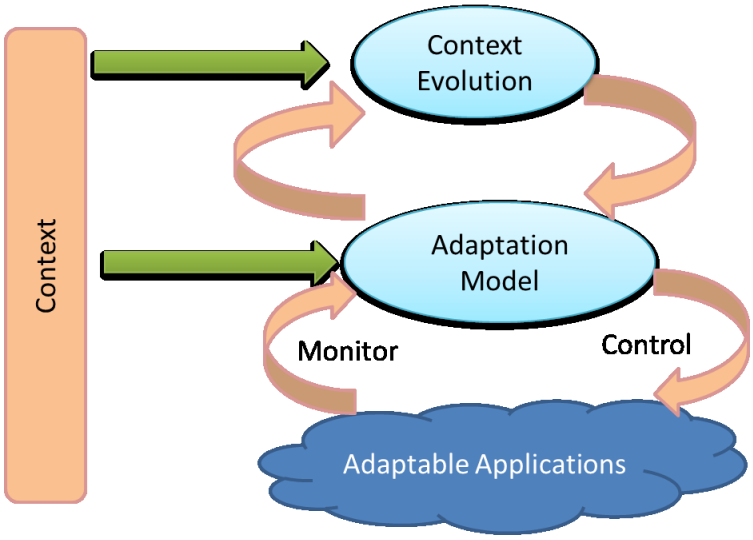


Figure 2-3. Meta-adaptation for self-adaptive software

The other role of this layer is to ensure the coherence and correctness of generated adaptation plans. Although mechanisms for runtime adaptation are supported in the level of operating system layer or middleware layer, they normally do not they do not ensure the correctness, consistency or other desired constraints during the course of run-time adaptation evolution. Without other insurance provided by this layer, the risks that might occur by runtime adaptation evolution might outweigh those benefits associated with the enhanced reusability of adaptation module or avoidance in shutting down, manually reconfiguring and restarting a system.

2.3 Adaptation with composable adaptation modules

In order to better support self-adaptive software with changing and multiple concerns, we defined a conceptual model for supporting adaptation with composable adaptation

modules. In Chapter 4, the Transformer adaptation framework is designed according to this conceptual model.

As described in Sect. 2.2.3, in this thesis, applications are designed and implemented by a set of functional components. The availability of components is dynamic that means components can be added, updated and removed. This dynamicity will greatly change the availability of application configuration variants. As discussed in previous chapter, rather than only treat application as composition of component, our middleware-based solution is to make adaptation strategies run-time composite-able. Several key requirements are identified to implement such composition.

2.3.1 Conceptual model for adaptation composition

In this section, a conceptual structural is defined to support the run-time composition of adaptation logics. According to the requirements listed in the previous section, this structure contains five kinds of modules shown in Figure 2-4.

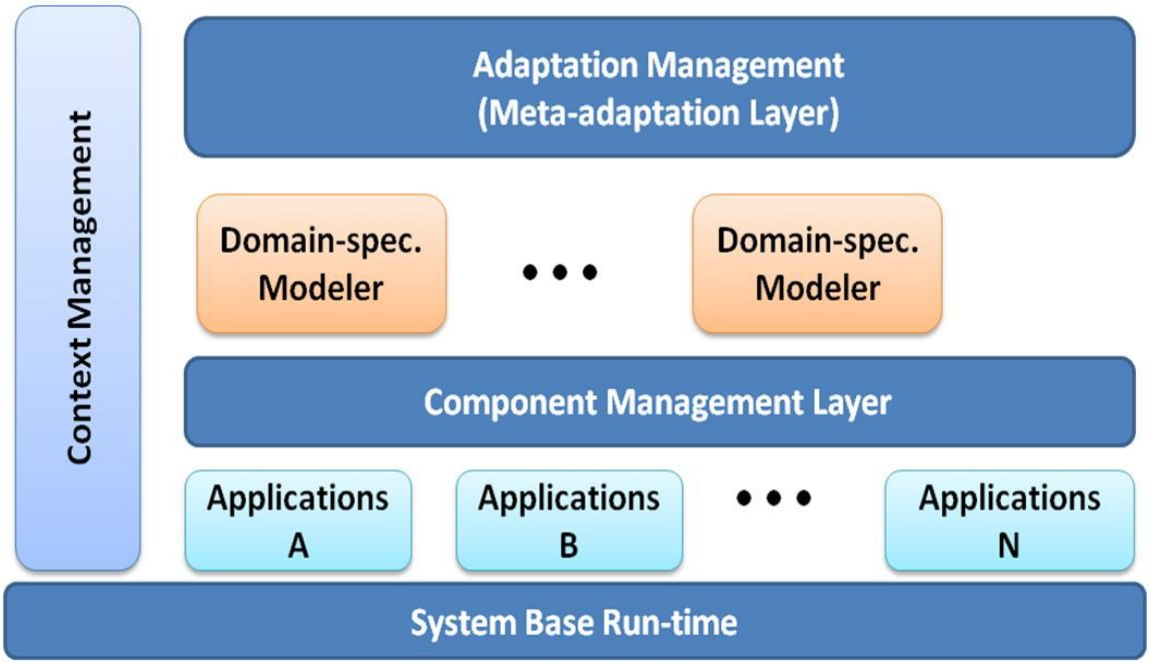


Figure 2-4. Conceptual model for middleware with multiple adaptation concerns

First, the system basic runtime is designed to provide basic support for installed components within the system. It provides services for component installation, uninstallation, start or stop etc. It also works as the central coordination for the whole adaptation process. In this conceptual model, rather than treating component only as design-time artefacts, in this thesis, each component is treated as run-time deployable entities. As each component might have its specific feature, it is important to allow the system basic runtime to inspect and control the installed components. Chapter 5 provides the detailed introduction of declarative & reflective component model in which meta-data and management interface are used together.

In software adaptation, system evolution can be driven by different viewpoints, for instance, security and performance. These concerns can be expressed as a set of optimization strategies and constraints and implemented as a system adaptation modeller. To express this domain-specific optimization goal, we use the concept of *Domain-Specific Modeller* (DSM). Like other adaptation modeller, each DSM will receive notifications from external world and computes adaptation plans. As shown from Figure 2-4, multiple DSM can be simultaneously installed and possible used in the system. Finally, the adaptation models are responsible for reasoning based on context information and for selecting the most appropriate component composition for each of deployed applications from its own individual merits. Furthermore, just like the off-the-shelf business components are composition to build applications, the composition of multiple DSM to form global adaptation behaviour is formally supported. Of course, here the focus is for the adaptation for the target applications. If additional QoS are required for the DSM's performance, an additional layer might need. For instance, a scheduling layer might be needed for DSM with real-time constraints. Detailed discussion on this topic is out of the scope of this thesis.

The component management layer is provided here to separate the adaptation modules from the underlying component implementation details. It monitors the states of system-installed components, which are used construct applications. At the same time, it executes the adaptation actions sent from adaptation modules. In order to mask the implementation diversity from upper level adaptation modules, this layer also should provide a translation service to translate adaptation actions from adaptation modules into those actions supported by underlying component model.

As discussed in our previous section, context information plays a key role in making effective and accurate adaptation. In order to retrieve, manage, reason and expose such context information, the context management module is designed. Thus, this layer is responsible for making the context knowledge available to the adaptive applications and other middleware modules (such as DSM) and other middleware module (e.g. Modeller manage layer). There are many researches on this filed, for instance, the Music project [134], to study more effective support for context-aware applications. As this thesis's goal is to support adaptation composition, this module will not be discussed in details.

As can be seen from the conceptual structure, in this thesis, three major designs are used. Firstly, the component-based approaches for the middleware design as well as application development, as component orientation can effectively facilitate adaptation (in terms of architecture recomposition). Secondly, in order to provide coherent adaptation mechanisms that can be reused across different component implementation, the management layer is designed to provide this isolation. Thirdly, reflective middleware is used in designing and implementing the underlying component model. This enables system run-time to get information about managed components through meta-data and management interface. Finally, as can be seen from the conceptual structure, in this thesis, this conceptual model enables not only the separation of self-adaptation concerns from application business logics, but also the adaptation decomposition and composition process. By extending this principle

into adaptation modules design, it makes adaptation modules much more concise and much easier in implementation.

2.3.2 Design principles of DSM

One of the challenges in supporting adaptation composition is to enhance the reusability and composability of adaptation modules. From the design perspective, most adaptation frameworks [37, 50, 52, 75, 138] provide no clear design principles to facilitate the composition of DSMs, which results in non-composable and un-reusable DSMs. From the implementation viewpoint, their adaptation logics and actuation mechanisms are normally mixed together. This coupling makes it very hard, or even not possible, for adaptation modules to be composed with each other.

In order to achieve composable and reusable DSMs, the following design principles are identified from our research and development experience.

1. Orthogonality: This principle is meant to lower the chance of interference of concerns and reduce the complexity this interference may bring to the later conflict detection and resolution process. This is quite similar to the orthogonality design principle of AOP. However, due to the existence of the adaptation fusion process, in Transformer, this feature is not mandatory required – though highly preferable.
2. Clear separation from the adaptation actuation process: This principle aims at enhancing the reusability of DSMs. We addressed this principle by forbidding a DSM to perform any adaptation action inside its reasoning logic. From this perspective, the DSMs are side-effect free – they can only expose their adaptation decisions to the Model Fusion through their pre-defined interface and have no direct impact towards system configurations. All adaptation plans will be sent to Model Fusion rather than being directly executed by DSMs. This design allows conflicts between different DSMs to be easily captured and resolved.
3. Restriction on adaptation actions: An DSM can only use adaptation actions with well-defined semantics. Relationships among all supported adaptation actions should also be clearly defined. This requirement is to promote composability and to simplify the conflict detection and resolution performed in the adaptation fusion process.

A DSM is designed to be individually developable and deployable and can be used together with other DSMs. From this aspect, it is quite similar to aspects in AOP [88, 89]. However, in the AOP-based solutions, aspects may be attached virtually to any position of the original source code and adaptation logics mix with adaptation actions. In comparison, our work clearly defines what a DSM can do (that is, adaptation interface and action sets) as well as when it can do it (that is, only during adaptation process). Therefore conflicts identification and resolution can be largely simplified.

2.4 **Conclusion**

In this chapter, the basic elements of this thesis have been introduced. The introduction of software architecture for self-adaptive applications provides a base on which the software adaptation strategies can be performed. This section also formally defines the key concepts used in this thesis, such as component, links, context and adaptation. After the introduction of those basic concepts, this section proposes a middleware-based solution in which multiple adaptation concerns support is supported. This chapter also provide a conceptual structure to integrate multiple domain-specific adaptation strategies by extending *Separation of Concerns* from just for application development process to the development process for adaptation logics.

However, it is desired to be pointed out, although this middleware and its corresponding conceptual model contains all key modules proposed in the MAPE-K model – it covers component management, context management, adaptation management, etc. – the main focuses of this thesis are the adaptation management, especially adaptation composition and conflict resolution process. These processes include adaptation modular installation, context-aware selecting, adaptation behaviour composition, and adaptation actuation and adaptation conflict resolution. This thesis proposes an adaptation framework for developing adaptive software and a middleware architecture to support those reusable adaptation modules.

Chapter 3

Related Work

The design of an adaptation framework that supports the building of adaptation behaviours with reusable adaptation components must naturally leverage off of the work that preceded it. In this chapter, several supporting disciplines as well as previous researches in architecture-based adaptation frameworks, especially their middleware design, will be discussed

This chapter will firstly focus on adaptive software related technologies; software architecture is firstly introduced as a supporting discipline, such as languages in describing target software architecture and techniques in building software architecture model. Then, control theory is introduced to build self- adaptive software decision logics. Finally, the AI based technologies are introduced for building more flexible adaptation behaviour in semi- or un-structured environments. Then, this chapter also discusses literatures and projects focusing on middleware-based adaptation frameworks.

In this chapter, two major aspects of state-of-the-art are discussed:

a) Supporting disciplines.

b) Selected approaches in middleware-based adaptation frameworks.

The former aspect illustrates different research disciplines that are wildly used to support adaptive software systems design and construction, while the latter aspect (section 3.2) presents a general picture of the current middleware-based approaches in supporting architecture-based adaptation. At the end of this section, comparisons are made and key challenges are identified for providing an engineering way of on-line adaptation construction via reusable adaptation components.

3.1 **Supporting disciplines**

Many different research disciplines are utilized in developing adaptive software systems. One of the common agreements on adaptive software development is that it is inherently interdisciplinary [136]. Actual combinations of employed disciplines highly rely on the design methodology used by self-adaptive software developer in implementing their specific adaptive software. In the following section, three major disciplines – namely software engineering & architecture, control theory and artificial intelligence (AI) – are discussed with the focus of their applications in self-adaptive software domain. Of course, other disciplines, such as distributed computing, optimization techniques etc. also play important roles in the self-adaptive system design. Due to space limitations, this thesis only focuses on the following three major supporting disciplines.

3.1.1 Software architecture

With the work of numerous researchers on software architecture, software architecture has been used as a fundamental reference model to express and reason about a given software system.

In their landmark paper – "Foundations for the study of software architecture", Perry and Wolf define the term of "software architecture" and establish this concept as a new discipline [123]. Their work forms a basis for software architecture by drawing analogies from building architectural styles. By intensively analysing many typical cases, Bass, Clements and Kazman investigated the supporting techniques for software architecture design and implementation [26, 27] . Here we briefly discuss four major aspects— architecture model & description languages, quality attribute component-based software engineering, service computing and service-oriented architecture.

Software Architecture **models and languages**, Architectural Description Languages (ADL) are designed to help software modelling and management. Many ADLs have been developed for various modelling purposes with different domain-specific features, including C2[109], xADL[112], UniCon[18], Acme[61] etc. Those approaches view a system as comprising components and interconnections, and aim at separating structural description of components from component behaviour. Papadopoulos et al. [120] concluded the fundamental concepts in the architectural models and languages and made detailed classification on existing approaches. Bradbury et al. [33] pointed out that those ADL are supported by different formalisms such as graph theory, process algebras etc.

The C2 ADL language and associated tool suites support the description and analyses of event-based, hierarchical publish-subscribe systems [109]. xADL describes systems as interfaces and connections, and supports direct translation to logic sentences for correctness analysis [112]. In reference [61], the Acme ADL is used to define the architecture of adaptable software and invariants for the architecture. Violations can be automatically detected when those invariants were no hold. XML variants of ADLs exist,

which are meant to support interchange as well as style-generic architecture descriptions, including xArch [21], xAcme [20] and xADL [41].

As the current research on the adaptation is shifting into a runtime world, Oreizy et al. point out that as the software architecture hold a global view of software constructions as well as constraints, it can be also useful in designing adaptation mechanisms during run-time [116, 117]. In order to include the quality attributes (e.g., security, performance...) in the architecture model, Klein et al. introduce the notion of Attribute-Based Architecture Styles (ABAS) [92] which includes a quality-attribute specific model (e.g., performance) to reason about an architecture design and interacting components' behaviours during run-time.

Quality attributes: In order to make effective adaptation, systematic approaches for realizing and measuring quality become one of the key research disciplines of self-adaptive software.

These quality attributes can be classified as two major classes – functional requirements and non-functional requirements, such as security or Quality of service. Most of the quality attributes can be classified as non-functional. In most cases, this is the major reason that triggers system to change in order to fulfil these requirements. Several researchers have used non-functional requirement models, especially the goal models [97, 144], in building self-adaptive software.

Formal methods permit to model software systems as well as analyze such models. Formal methods are used for self-adaptive software validation and verification, thus to ensure its correct functionality. It is also used to understand adaptation behaviour [96]. New formal models approaches, such as Model-Integrated Computing [146], are developed for the modelling of self-adaptive software.

Component-Based Software Engineering **(**CBSE): CBSE is used to facilitate the development of self-adaptive software. In one aspect, component model simplifies the process to design and implement an adaptive application. At the same time, it can be also used to construct the adaptation framework itself, making it customizable by using reusable components. In this thesis, components are used in both aspects – both for constructing application business itself and for the system adaptation engines [107].

Another related area, *Aspect-Oriented Programming (AOP)* [89], especially the dynamic AOP, can be used in introduction adaptation behaviours to existing application. This technique can encapsulate adaptation concerns in the form of aspects and weaves those adaptation behaviours through off-line or on-line weaving[32]. For the off-line weaving, one of the most popular package is AspectJ [88]. The on-line weaving approaches include AspectWerkz [32] and JBoss aspect [10]. As aspect can introduce new application behaviours at the source code level, this technique can build fine-grained adaptation actions at a level lower than components [65, 80]. AOP can also be used for building sensors and adaptation actuators as in AAOP platform [80, 81].

Service computing **and** ***Service-Oriented Architecture*** (SOA): SOA can also support realizing self-adaptive software as its loosely coupled service-oriented structure facilitates

the run-time compositional adaptation. Due to their flexibility for composition and run-time orchestration, it has been intensively used for implementing service-oriented software systems with run-time adaptation capabilities [58]. One important work in the domain of web service is done by Verma [151], so call Autonomic Web Processes (AWP). AWP supports self-adaptation properties for web service processes. For instance, it can track the health of components, integrates self-diagnosis into application. However, the high overhead and complexity of web service technology make it not appropriate for resource-constraint environments.

In the framework of SOA for embedded environments, Hall and Cervantes [37, 74, 75] propose extensions to the OSGi component model (in OSGi term – Bundle) to support so-called *Service Component Model*. This component model is capable to make compositional adaptation and support component dynamicity. Due to its lightweight and extensibility, this model is adopted in our framework. Additional extensions are provided to support semantics for context knowledge descriptions.

3.1.2 Control theory

Control theory is a branch of science that deals with the behaviour of dynamical systems. Control theory-based systems form a sense-plan-act control loop by repeatedly sense their environment, make plans and act on the actuators. This control loop is quite similar to the adaptation loop in self-adaptive software. Closed-loop control typically involves a controllable process or "target system" (e.g., a heat furnace) that usually includes the environment, a controlled variable or "measured output" (e.g., the outflow of a heating), and a load or "control input" (e.g., the temperate of such a heat furnace). The concepts of adaptation and feedback have been used in control theory for many years. This "control loop" has been utilized in designing system in a lot of domains, including adaptive software.

Due to those similarities, many theories and research results of control theory have been mapped to the software engineering domain[110]. Tools developed in control theory have been used to build adaptive software. Concepts in control theory are also used to evaluate self-adaptive feature, such as stability, and sensitivity. The control theory is often used as a software plant's model. In this model, adaptation is achieved by the target plant' parameter adjustments (parameter-based adaptation), e.g. Litoiu et al. [99] designed a so-called hierarchical Layered Queue Model (LQM) based on control theory. Adaptation is made by tuning the target's parameters. Later, control theory is also used in the compositional adaptation domain. For instance, in reference [29], the ACCORD component framework was extended with control theory to achieve self-managing goals by adding, updating and removing components.

In order to deal with problems with unpredictably disturbances, *adaptive control* was introduced by Karl Åström in his book "Adaptive Control" [24]. Compared to conventional feedback-based control, adaptive control is more flexible in dealing with disturbance. However, it also has limitations. One of the key limitations is that its environment identification and decision logic is designed and implemented when the

controller was developed. As a consequence, the adaptation behaviour of the controller remains fixed for the whole lifetime of the self-adaptive software[110]. Due to this limitation, this approach can only work effectively in controlling the system whose model is within a predefined class of models (the parametric uncertainties of the model). When the target software system characteristics have even slightly different characteristics than the presumed model, the results can be catastrophic. Some approaches, such as Catthoor et al[154] actively involved the hybrid approaches, which combines the design time adaptation (predefined model) with run-time adaptation logics for real-time applications. However, their approaches mainly limits to the energy-optimization domain.

Due to the complexity in modelling all possible contexts, control theory in itself cannot effectively solve the problem identified in Chapter 1. In order to deal with that problem, Artificial intelligence was introduced to generate new rules or adjust adaptation behaviours.

3.1.3 Artificial intelligence

AI has been intensively used in building self-adaptive software. AI can help in tracing history data and identifying abnormal conditions or violations of system constraints [121], which can be used for the change detection process. However, as it is powerful in planning, reasoning and learning, it is mainly used in the planning process as specified in the IBM MAPE-K model [87].

Within many AI techniques, *AI planning* is intensively used to realize self-adaptive software. In those approaches, rather than simply executing pre-specified algorithms, a software system plans and may re-plan its actions according to system design goals. One example of using AI-planning to Autonomic Applications can be found in [141] by Srivastava et al. which is built based on ABLE agent building kit [31]. The authors later pointed out that this planning technique can work effectively together with self-healing adaptation behaviours[140]. As pointed out by Maes [104], current systems face problems on assumption that situation descriptions are exclusive so there is never a conflict of action which is rather unlikely in the changing environments. She then proposed a goal-based model for action selection when conflicts happen. Her design of action composition and identification of conflicts has been intensively used in our thesis.

Machine Learning *is* the area that plays important roles in self-adaptive software. This discipline focuses on how to effectively analyse sensed data from environments and how to learn the best strategy to react to the environment changes. In order to deal with unprecedented changes, these algorithms generally use the environment properties (current or together with history) and knowledge gained from history to generate new settings for adaptation algorithms or even new algorithms themselves. Many on-line learning algorithms, such as genetic algorithms or reinforcement learning (RL), have been used for adaptation learning. In K-Component project, Dowling et al. [50] shows that RL can have good performance in decentralized collaborative self-adaptive software. Tesuro [149] also pointed out that RL normally has better performance and require less domain knowledge compared to traditional methods.

However, although RL works fine in restricted environments, it is not always feasible to learn adaptation knowledge online within complex and fast changing environments as RL requires a period of training to gain knowledge of target systems. Certain techniques are needed to restrict the problem domain to allow fast convergence.

Decision theory is an area of study concerned with problems on how an ideal decision-maker (should) make decisions and the quality of made decision. This theory can help in realizing the adaptation deciding process, from either classical and/or qualitative forms. Numerous techniques for choice under uncertainty have been developed, including the isomorphic forms of decision tree and decision table [124].

One of the areas within decision theory is *utility theory,* which has been intensively used in self-adaptive software projects [23, 61, 63]. The term *utility* is used to specify "the useful level" of a given action, choice etc. Utility values can be given either with certainty or with certain probability. In the Rainbow adaptation framework, Cheng et al. [40] demonstrate how to use utility functions to resolve possible conflicts – situations that several adaptation actions proposed by different adaptation strategies – to optimize network resource usage. In MADAM project, utility function is used for selecting optimized software architectures by scoring alternative component instances. The software configuration that maximized the end-user utility calculated by a predefined utility function will be selected. As fix utility functions have limitations in dealing with changing environments, Kakousis et al. [83] demonstrate how user feedback can be used to optimizing the utility function-based self-adaptive behaviour.

Although utility functions are extensively used in solving possible conflicts by simplifying adaptation actions conflicts in terms of simple utility functions calculation, lack of semantics on the context greatly limits the practical usage of these utility-based approaches.

3.1.4 Summary

As already been pointed out, self-adaptive software is an inherently interdisciplinary approach, which is especially true for adaptation in changing environments. Software architecture models provide ways for software model description, control theory makes effective adaptation and artificial intelligence is used for environments in which the targeted adaptable software system could not be fully modelled.

In this thesis, a middleware-based adaptation framework is designed and implemented by using techniques from these three disciplines. In order to capture the current research trends and to identify their limitations on adaptation in changing environments, selected literatures in middleware-based approaches are studied.

3.2 **Related middleware-based self-adaptation approaches**

There is a substantial volume of literature existed on the research domain of self-adaptive software systems. In order to limit our survey scope, this section only

examines existing middleware-based adaptation frameworks. The reason in selecting this domain is because middleware, as pointed out by McKinley et al. [15, 47], "provides a natural place to locate many types of adaptive behaviour", with its nature position between lower level OS and applications in higher level. Most architecture-based adaptations are implemented via middleware technology. These frameworks, to a certain degree, represent the state-of-the-art in the development of middleware-supported adaptive software applications. In order to capture an accurate global image of the self-adaptive software research domain, in this section, projects from both academic and industrial sectors are selected and analysed.

The major goal of the section' discussion is to identify the pro & cons of the reviewed projects. In order to provide a comprehensive view of those projects, they are analysed from different perspectives, with the focus of the support for adaptation in the dynamic changing environments

3.2.1 Quo

Quality Objects (QuO) [101, 102] is a framework that was designed to provide quality of service (QoS) in network-centric distributed applications. QuO adds QoS supports to existing middleware platform, such as CORBA and Java RMI. It is designed for creating adaptive applications within an unpredictable environment with strict resource constraints. The Quality Description Languages (QDL) is designed to specify various QoS requirements and constraints for system resources. In order to support system adaptation, data measuring mechanism, QoS controlling mechanism and logics for changing of QoS level, are also supported.

However, Quo only provides aspect level adaptation by intercepting messages between CORBA objects. This intrusive solution introduces considerable overhead when the communications between objects are frequent. Furthermore, its adaptation decision process is compiled off-line and very hard to be altered during run-time. These limitations make it not feasible for dealing with changing adaptation requirements.

3.2.2 Architecture-based runtime software evolution

Nowadays, self-adaptive software is more and more used in a changing environment. Thus, not only the targeted applications but also the adaptation behaviour itself, needs to evolve with environment changes. Oreizy et al. proposed [116, 117] an infrastructure that supports two different adaptation processes: system evolution and system adaptation. The former one is the typical sense-reason-effect adaptation loop (the cycle of detecting changing circumstances and planning and deploying responsive modifications) and the later one deals with the changes of adaptation logic (keeps the adaptation logics consistent with the changing requirements over time).

In the cited papers, a toolset – Archstudio – was proposed to provide: 1) explicit Architectural Model, 2) runtime change description, 3) runtime change verification, 4) reusable architectural infrastructure. In order to keep the coherence between the

architectural model (meta-model) and underlying implementation layer, the Architecture Evolution Manager (AEM) is proposed.

This approach identifies the necessity of providing system evolution and proposes a systematic approach in solving this challenge. However, this approaches failed to answer the key questions on how to generate new adaptation logics for the new environments and how to reuse adaptation logics across multiple contexts.

3.2.3 JMX

JMX [145] is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (e.g. printers) and service oriented networks, in which the lower layer is composed of components, called MBeans, representing the Java objects to manage. Starting with the J2SE platform version 5, JMX technology is natively shipped with the Java SE platform. This means that any Java-based solutions can use this technology without the need to reprogram the application. As this approach intensively used the reflection techniques provided by Java language, limited changes are required in introducing adaptation to existing legacy systems. This feature makes it an appropriate solution for migrating legacy systems into self-adaptive systems.

As JMX is mainly designed for application managing and monitoring, it does not have a systematic support for adaptation, let alone adaptation evolutions. It provides a limited component model so an application can only be constructed via its own property methods. Due to these characteristics, JMX is mainly used as an important enabling technology for Java-based application monitoring rather than for actual adaptations.

3.2.4 Gravity

In order to deal with component dynamicity, in the Gravity project, Cervantes and Hall [36, 37, 74, 75] proposed a service-oriented component based framework. This framework can maintain software architecture even when component availabilities change during run-time. This framework is built on top of SOA technologies.

The key part of the Gravity framework is the Service Binder module. This module automatically maintains and controls the relationship (the term “binding” is used in their paper) between components during run-time. Components’ dynamicity is explicitly supported and their relationships are built based on the meta-data attached to their implementations. Their approach later became the foundation of the declarative service in OSGi v4.0 specification [119].

In this thesis, a service component model was provided which enables run-time compositional adaptation support. This dynamicity support becomes one of the enabling features for our platform for compositional adaptation. However, it provides no parameter-based adaptation. For the adaptation logic integration, this framework uses a fixed set of adaptation policies. Its adaptation planning process is hardwired with adaptation actuation modules. This static adaptation policy and the lack of adaptation policy evolution support limit its usage in changing environments.

3.2.5 K-Component

In K-Component project [50], a meta-model is proposed based on Adaptation Contract Description Language (ACDL). This meta-model is designed for realizing dynamic software architecture based on specific reflective software techniques. ACDL simples the adaptation behaviour development by separating the specification of a system's self-adaptive behaviour (with the help of ACDL) from the system components' business behaviour.

The other focus of this project is to coordinate the adaptive behaviours of individual adaptation components so as to realize system global optimization goals. One of the key problems in realizing model is to establish consensus in dynamic and decentralized environment. In [51], Dowling and Cahill proposed a so called "collaborative reinforcement learning" to introduce learning capabilities to involving components. Groups of components learn and collectively adapt their behaviours so as to maintain system-wide constraints and preferable system properties in a dynamic context.

This project demonstrated that reinforcement learning can effectively cope with changes under comparable stable system optimization goals and execution environments. However, due to the slow learning process, such pure AI based solutions normally cannot effectively deal with fast changing environments.

3.2.6 Rainbow

Garlan etc. [61] propose a general architecture-based self-adaptation framework – Rainbow. The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. The use of external adaptation mechanisms allows the explicit specification of adaptation strategies for multiple system concerns and domains.

Rainbow is designed to be reusable. It aims at providing an engineering approach that can be reused to build different domain-specific self-adaptive software. In order to support this design goal, a framework of mechanisms as provided, such as monitoring functions (target system and the environment changes), knowledge maintenance (maintains architecture model coherence), analysis (detect changes that have to be handled or opportunities for improvements), planning (select a course of action) and actuation (enact changes). In order to support different implementation requirements, this framework is designed with well-defined customization points, which allow software engineers focusing only on their specific adaptation concerns. This systematic design facilitates the process in customizing Rainbow to particular systems. An adaptation description language – Stitch – is designed to represent routine human adaptation knowledge by supporting a core set of adaptation concepts.

Rainbow provided a systematic approach towards adaptation description, supported adaptation with multiple concerns and proposed utility-based resolving policies [40]. However, this approach uses the Acme ADL to predefine all the elements in the targeted system and provides no design of underlying component model which is crucial for

compositional adaptation. The adaptation logics it used could not be altered during run-time. Although it identified the problems in resolving conflicts adaptation behaviours by using a utility-function based solution, however, the static nature of the adaptation behaviour itself makes it not appropriate for changing environments.

3.2.7 SmartFrog

As pointed in [107], architecture adaptation have two major types of adaptation: parameter-based and compositional adaptation. SmartFrog [64] is a framework supports the former one. It provides the adaptation management for configuration-driven systems. In those systems, adaptive software is defined as a collection of software components. Each component has certain properties, such as quality level, frequency etc. Those properties can be changed during run-time through the predefined interfaces that all managed software components should implement.

SmartFrog provides three major contributions[135]:

- An ADL language is defined for the software configuration definition. Expressive notations are defined for describing system configuration description. This language also provides system modelling capabilities.
- A secure, distributed runtime system. This run-time is designed for software component deployment support as well as the online management for running software systems.
- A component library. A set of reusable components based on SmartFrog component model is provided with a wide range of services and functionality in supporting self-adaptive software developments.

The framework focused on providing component configuration, deployment and management supports. However, it lacks of support in providing adaptation logics towards adaptive software. Moreover, the description of component is static and could not support non-functional properties and constraints.

3.2.8 Architecture-based management – self-healing

Sicard et al. [138] identify novel requirements on reflective component models for architecture-based management systems. A meta-construct layer is designed for meta-data checkpoint and replication. Interfaces and processes for self-repairing are defined, such as lifecycle management, setter/getter interfaces as well as the meta-data based configuration. A faulty component can be repaired by restoring its state and all the meta-data information outside of the component instance.

However, their approach does not have clear definition and separation between adaptation plan and the underlying system sensors and actuators. Such hard-wired adaptation architecture makes it very hard to reuse their framework across different contexts. Besides, it provides no support for adaptation for other concern. Thus, no conflicts detection and resolution mechanisms are designed in this solution.

3.2.9 CoSMoS

In the domain of contextual compositional adaptation, Fujii and Suda [57] designed a adaptation framework called the semantics-based context-aware dynamic service composition framework (CoSMoS). In this framework, an application is composed with the composition of selected distributed components. The employed components are selected based on semantics of components and users' contextual requirements. The CoSMoS framework has three major parts: 1) a service component model with semantics 2) a semantic matching engine for component selection and 3) a component executing run-time support. Based on the proposed framework, a distributed middleware, called CoRE, is designed. Furthermore, a middleware, called CoRE, is designed to support CoSMoS on distributed computing environments. In this middleware, algorithms for semantics similarity and learning based algorithms are discussed in supporting context-specific workflow synthesizing for the application construction.

In this approach, no functional meta-model is provided for application construction. Due to lack of system architecture model, it is not able to verify whether a structural adaptation comply with an application's structural model. For the adaptation in changing environments or user preference, a decision tree based algorithm is proposed. However, this solution needs considerable users' feedbacks to build a new decision tree whenever there is a considerable context change [59].

3.2.10 Adaptable aspect-oriented programming – AAOP

This AAOP platform [80, 81] presents adaptable aspect-based approach to introduce adaptation capabilities to existing applications. The concept of adaptable aspect-oriented programming (AAOP) is proposed in which a set of "aspects" components can be selected and weaved during run-time to adjust an application with the logics specified in its adaptation strategy.

In the AAOP-based adaptation system, the applications are augmented with adaptation aspects contextually selected by the application execution runtime. Those adaptation aspects include aspects for sensors, effectors, and goals. As those aspect components can be dynamically weaved into application, application adaptation behaviours can be changed during run-time to meet different contextual requirements. In the reference [80], special focus is putted on the adaptive sensing and actuation by using dynamic AOP based approach.

However, in this approach, the adaptation capabilities are limited to one run-time selected adaptation goal. AAOP assumes adaptive software has no multiple co-existing adaptation concerns. At any given time, only one strategy will be used. The other limitation of this approach is that there is no support for compositional adaptation. As a consequence, it provides no support on adaptation composition, conflicts detection and resolution.

3.2.11 Mobility and adaptation enabling middleware

The *Mobility and adaptation enabling middleware* (MADAM) [55, 62], is a supporting middleware that build in the FAMOUS [76] (the *Framework for adaptive mobile and ubiquitous services*) research project. Similar to other projects, MADAM also adopts an architecture-based approach – the system's architectural model is explicitly represented and maintained at runtime. The so called computational reflection is used to reason about and control adaptation by generic middleware adaptation components. In this approach, adaptation decisions are enabled by scoring alternative architecture instances. The one that maximized the end-user utility, as calculated by a predefined utility function, will be selected [22, 23].

While this approach provides simplistic context model and management mechanisms, it does show some interesting and novel aspects. For instance, it provides support for dynamic behaviour and evolution. However, how to effectively build adaptation behaviour for the new context remains unexplored in this approach.

3.2.12 CARISMA

Another important approach for adaptation within changing environments was proposed by Capra et al. in [34, 35], so called *Context-aware reflective middleware system for mobile applications* (CARISMA). In this approaches, both reflection and meta-data is used to contextually build adaptive software systems.

In CARISMA, applications can instruct the middleware how to react to context changes through meta-data. These meta-data contains adaptation policy that its hosting middleware can employ when certain context occurs. These profiles can be parsed and used by the middleware. By utilizing those adaptation provided together with application's meta-data, middleware can realize applications adaptation behaviours without the need to implementing hard-coded adaptation logic for the application. Contextual adaptation can be also supported as those adaptation policies can be selected according to the requirements of the current context. In order to provide an extensive solution, the abstract syntax for the application profile definition was designed with XML Schemas.

However, meta-data based solution can only provide static information. In order to give external developer the accurate image of the system run-time status, this solution defines the reflection mechanism to access to the applications' meta-data and change them according to context requirements.

As the adaptation policies are chosen during run-time, there are possible conflicts existed within the selected policy sets. In order to deal with this problem, CARISMA proposed microeconomic-inspired conflict detection and resolving mechanism. Those conflicts are determined through a sealed-bidding mechanism [35] which is based on utility functions. However, its utility-based solution has limitation. When context changes, the utility values associated to the adaptation polices should also change to reflect new contextual concerns shift. Moreover, CARISMA does not deal with many of the requirements that are related to the dynamicity of adaptation behaviours. From the

implementation point of view, this sealed-binding mechanism introduces considerable implementation complexity and overhead, as each policy need to provide such bidding mechanisms. This system is compared hard to put into practical usage.

3.2.13 MUSIC

The European MUSIC [134] project targets on the design and implementation of an open platform for the development of innovative mobile and context-aware applications. It provides a new design methodology for self-adapting applications, which allows self-adapting behaviour to be incrementally designed and deployed.

In this project, an advance modelling language was developed for the specification of context dependencies. A software development framework was developed that facilitates the development of self-adapting, reconfigurable software. Its modular and run-time pluggable middleware architecture enables itself to adapt to highly variable user preferences and dynamic context requirements, while a high level of (re)usability can be achieved[121, 122].

Although MUSIC provides an adaptive software framework that enables adaptation modules to be separated from adaptable software, its major contribution focused on how to provide a context-enabled framework as well as development tools for context collection and reasoning. The project provides little support on adaptation with multiple concerns. No clear mechanism is designed on how to detect conflicts and solve them.

3.3 Limitations of the surveyed approaches

In the last two sections, we presented an overview of the related work to show how the state-of-the-art partially serves our thesis objective.

State-of-the-art researches provide the language, model, and analysis to represent and reason about a system's software architecture. Advances in component technologies provide a concrete basis for compositional adaptation. The existing trend in separating adaptation logics from adaptation engines greatly simplified the process to alter adaptation logics for target adaptable systems. Control theory provides the systematic support for control-loop based adaptation logic. AI technologies are used to deal with complex and semi- or un-structured environments and provide more flexible and intelligent adaptation behaviour. As self-adaptive software is intrinsically interdisciplinary, all these disciplines might need to create an effective and flexible adaptation framework.

In Section 3.2, middleware-based projects in context-aware adaptation systems and architecture-based adaptation was examined. Table 3-1 shows the comparison in different aspects. As we can see from this table, most adaptation projects adopt external control loops. In terms of separation of concerns, their work separates adaptation logic from application business logic. However, in most these approaches, there is no clear separation between different adaptation concerns. No systematic way is provided to

identify possible conflicts between different adaptation logics by either allowing only one concern at a time[81] or assuming there is no adaptation conflict [138]. Other approaches, such as Rainbow and CARISMA [35], identified the conflict problem when adapting with multiple concerns, however, the utility-function based approaches could not be effectively used in the changing environments as for different contexts, the utility values for each adaptation action also require considerable changes.

Current approaches present a number of limitations and unresolved issues, which we address in this thesis. Especially, traditional adaptive techniques – e.g., exception-handling mechanisms and network time-outs [102] – can only work effective in a certain application domain as it only has localized knowledge of system states, our approach uses an architecture-based approach is used to retrieve and maintain global perspective. While many architecture-based adaptation approaches treated adaptation modules as a stand-alone entity and mingled multiple adaptation concerns together, our approaches extend separation of concerns paradigm to adaptation module design. Rather than treating adaptation modules as pre-defined static modules as in most existing approaches, in our approach, adaptation modules are dynamically composed during the run-time. Although some of latest approaches [40, 134] identified the problem of adaptation with multiple concerns, to date, utility function-based solutions are provided. This solution has significant limitations on conflict detection and resolution as utility functions highly depend on current context.

The introduction of adaptation composition to develop adaptive systems that are capable to deal with multiple contexts leaves some important questions to address. Firstly, how can an adaptation module be decomposed into smaller adaptation construction entities? How can those construction entities be identified and selected according to the current context? How does one make decisions and reason about the adaptation actions according to the multiple selected adaptation entities? How can a system effectively and correctly identify the conflicts, as different adaptation entities might generate conflicting adaptation actions? Last but not the least, there is also the overall challenge of how to facilitate the engineering problem for designing and implementing self-adaptive software.

In this thesis, we focus on the following core challenges to achieve self-adaptation with multiple concerns:

1. How to modulate adaptation logics by using component;
2. How to provide framework supports for adaptation composition;
3. How to provide an engineering approach to support the proposed adaptation composition framework in a light-weighted, cost-effective and reusable way.

Table 3-1. Projects comparisons. Taxonomy Facets. -: not supported, ?: unknown, L: Layer, E/I: External/Internal, S/D DM: Static/Dynamic Decision-Making, O/C: Open/Close, S/G: Specific/Generic, L/C So: Legacy system/Custom solution, MB/F: Model-Based/-Free, SoC: Separation of concerns, AC: Adaptation composition, I/S DP: Incremental/Static Deployment, M/S:Multiple/Singular concerns

=	Adaptation target		Approach realization						Adaptation concerns		
	L	Object	S/D DM	E/ I	O/C	S/G	L/C So	MB/F	SoC	AC	I/S DP
JMX	application	comp.	-	E	-	g	legacy	free	business	-	-
SmartFrog	application	comp.	-	E	-	g	custom	mb	confi.	-	s
Self-healing	applicantion	arch.	Static	E	open	s	custom	mb	adap.	-	s
Rainbow	applicantion	arch.	dynamic	E	close	g	semi-c	mb	adap.	uti.	m
Gravity	applicantion	arch	Static	E	close	g	custom	mb	dynamicity	-	s
Quo	network & application	aspects	Static	E	close	s	custom	mb	adap.	-	-
K-Component	application	comp	dynamic	E	close	g	custom	mb	adap.	-	-
Runtime Software Evolution	application &middleware	arch.	dynamic	E	close	g	custom	mb	adap.	uti.	m
CoSMoS	application	comp.	dynamic	E	close	s	custom	mb	adap.	-	s
AAOP	application	aspects	dynamic	E	close	g	legacy	free	adap.	-	s
MADAM	application	comp.	dynamic	E	close	g	custom	mb	adap.	-	s
CARISMA	application	comp.	dynamic	E	close	g	custom	free	adap.	bidding	m
Music	application	comp.	dynamic	E	close	g	custom	mb	adap.	uti.	s

3.4 **Conclusions**

This chapter presented a general introduction of the state-of-the-art from different perspectives. Firstly, the basic technologies for self-adaptive software were introduced – ranging from software architecture and ADL languages, control theory to artificial intelligence. Secondly, a selected set of middleware based self-adaptive project are studied and their strength and limitation are identified. Then, the major challenges in designing a self-adaptation system are single out. Those challenges are used as a guideline throughout the following chapters: from framework design (Chapter 4), middleware implementation (Chapter 5) to the case studies (Chapter 6).

As can be seen from Table 3-1, existing researches have done considerable work for developing self-adaptive application from methodology, framework as well as tools. Those approaches, by adopting an architecture-based adaptation model, can effectively decouple the development of adaptation logic from the application logics. Furthermore, their approaches provides middleware support for basic adaptation-related modules, such as sensor and effectors, thus developers can concentrate on the development of adaptation logics. This design can greatly simplify the design, construction and maintenance of the adaptation module.

However, existing solutions satisfy only a subset of the detected requirements and most of them exhibit shortcomings in dealing with changing adaptation concerns in dynamic environments. They failed to answer how to effectively reuse existing adaptation solutions in other contexts. In addition, they also do not provide systematic solutions on how to combine several adaptation logics with explicit conflict resolution. These challenges are identified and solved in this thesis by the Transformer adaptation framework and a supporting middleware. Chapter 5 illustrate the key design principles of adaptation framework and proposed the online resolution & validation mechanisms. The middleware architecture is introduced in Chapter 5 with detailed description of the employed component model and key implementation of several adaptation modules. In Chapter 6, this adaptation framework and this middleware solution are applied to a practical application for autonomous NXT robot control. Chapter 7 evaluates the proposed solution by checking whether those identified requirements are satisfied. Chapter 8 concludes the major contributions of this thesis and points out possible directions for future improvements.

Chapter 4

Transformer Adaptation Framework

This thesis aims to provide a cost-effective engineering approach to enable incremental self-adaptation and offer adaptation evolution capabilities. In Chapter 3, we argued how the related state-of-art approaches these objectives. Researches on language, model and framework were introduced and analysed. In order to accomplish these objectives enumerated in Section 3.3, in this chapter, a novel adaptation framework – Transformer, is proposed. This framework is designed for adaptation evolution – an important feature for adaptive application executing in changing contexts. This framework also provides explicit support for adaptation with multiple quality objectives. In order to achieve these features, this framework provides the ability to explicitly represent adaptation concepts as reusable entities, the mechanism to automatically decide the best combination of adaptation modules, and an automatic online integration mechanism that saves engineers time and effort in adaptation behaviour developments.

This novel adaptation framework is able to create adaptive applications that are capable to adapt in multiple contexts in a flexible and cost-efficient manner. This is achieved by (1) separating adaptation logics from application business logics so as to decouple the dependence between those two key parts; (2) applying the Separation of Concerns paradigm to adaptation logics design. This approach allows adaptation logics to be expressed as separate components with specific concern(s); (3) implementing domain-specific adaptation logics as individually deployable and compose-able modules – DSM. These specialized adaptation modules can be later reused to compose a global adaptation modeller during run-time and automatically unified into global adaptation

process. As each DSM might relate to a certain application domain, which means it might only work in certain environments, a context matching mechanism is also proposed to select the appropriate candidate DSM for composition.

This approach has the following advantages: First, it extends the notion of software reusability – from application business logics design into adaptation logic domain, by designing adaptation logics in terms of reusable components. Secondly, it provides a systematic context-aware modeller selection mechanism, which enables system run-time to select the modellers best-matching the current context. Finally, this framework also provides an online-verification algorithm, which can effectively detect possible adaptation loops. This algorithm also provides a tentative solution to avoid/break unlimited loops.

This chapter starts with a motivation scenario is described and analysed to better demonstrate the problems on adaptation in changing environments. After this motivation scenario, a context-aware application construction methodology is proposed to better support compositional adaptation – by reconstructing applications according to the changing context and with multiple adaptation concerns. Then, we provide an overview of Transformer – our architecture-based adaptation framework for adaptation composition and evolution. We demonstrate that how to effectively fuse multiple adaptation modules and make them working together. This is followed by the definitions of the architectural processing modules and discussions on the adaptation correctness. With the framework design principles described in this chapter, in Chapter 5, a reflective and modular middleware is designed and implemented as a supporting middleware for the Transformer framework. The performance of this middleware implementation is evaluated in both qualitative and quantitative aspects. Finally, Chapter 6 evaluates the proposed solution by applying it to a practical case (adaptive control platform for the NXT robot) and shows how our approach enables software developers to develop adaptive software application in a fast and cost-effective way.

4.1 Motivation example

In this section, we will describe a family of case studies to motivate the need for multi-context adaptation. Let us suppose that user John has a set-top box he uses to watch and record TV programs from a cable TV provider. John likes to experience his TV at top quality, but occasionally wants to keep a compressed copy of the programs he likes the most. The set-top box is expected to adapt its behaviour so as to maximize the overall Quality of Experience (QoE) as perceived by John.

This very simple scenario already calls for a complex treatment, which is more easily described by a few examples:

Pure TV watching: John is watching the news; his current top interest is TV QoE. The current context requirement is to optimize the TV application. The basic strategy is to always use components with best video quality to construct TV application and allocate enough resources for TV application. The adaptation module for TV application will be selected.

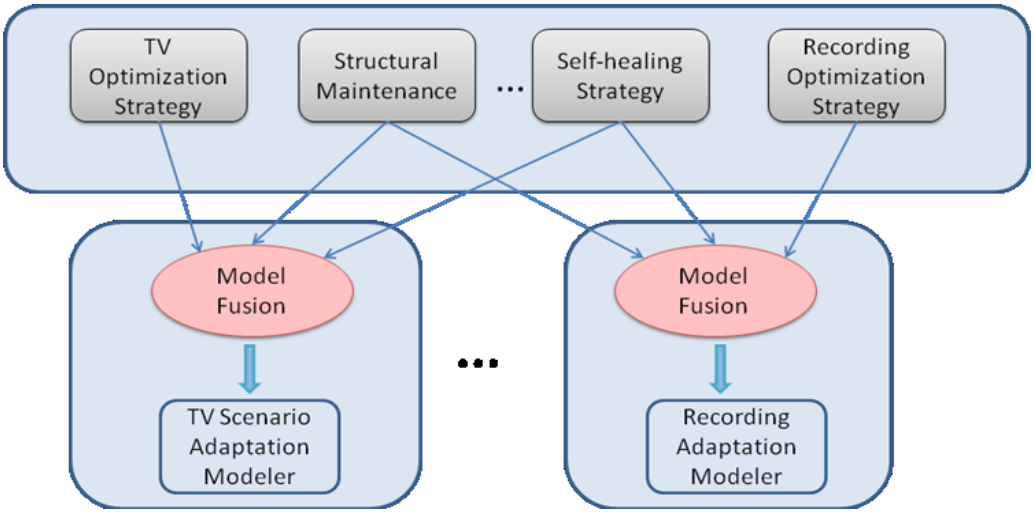


Figure 4-1. Domain-specific adaptation fusion. In a different context, a different set of candidate strategies can be selected and composed into a custom-generated adaptation strategy matching the current environments. Here, for TV watching adaptation scenario, three domain-specific adaptation strategies can be selected: TV optimization strategy, Structural maintenance strategy and Self-healing strategy. When the system context changes to Recording context, another set of basic adaptation strategies can be selected.

TV watching & recording: John's favourite film begins. His main objective is high quality recording. To this aim he deems as temporarily acceptable to have a low quality TV experience as long as the recorded copy is flawless and with the highest possible fidelity. (Major optimization for Recording QoE and medium or low settings for TV application)

Recording: John has an important meeting soon, so he can not watch his favourite film; however, he wants to continue the recording process even after he shuts the TV application down. In this case, the only optimization concern is for Recording QoE.

Self-healing: In all the above cases, John would like to enact a self-healing strategy so as to deal with possible application crashes.

What above are just three cases out of many possible situations that build up John's perceived quality of experience. For each new scenario, a new adaptation strategy is needed. However, as this adaptation strategies need to respect constraints from different aspects, building a new adaptation strategy implementation normally is a very complex task and requires a lot of expertise. For instance, in the Recording scenario, an adaptation strategy needs to handle constraints from at least three different aspects. That is: (1) Application structural maintenance for building application structure; (2) Optimization strategy for recording application; and (3) Self-healing strategy to tolerate possible application crashes. Likewise, for TV watching scenario, adaptation strategy also needs to take care of constraints from multiple aspects, however, with the specific concern of optimization for TV application.

It is clear that building a monolithic adaptation solution to deal with all the possible situations is not feasible. This solution cannot cover all and unexpected situations. As we can see from this example, though very different in their concerns, the above optimization strategies also have many points in common. We believe that a better solution is to divide

the adaptation strategies into adaptation building blocks representing particular optimization goals matching certain "situations". Each of those blocks assumes that some contextual hypotheses are valid and provides one domain-specific adaptation solution. The main target in this thesis is that, just as components are used to compose applications, these domain-specific adaptation blocks can be used to construct more complex adaptation strategies and (possibly) reused across multiple contexts. For instance, a self-healing module could match all three above scenarios. In what follows we describe the new application construction methodology that supports run-time application construction with respect to constraints from multiple perspectives.

4.2 **Application composition with multiple contextual concerns**

The funding principle of component-based development is that: applications are built by composing (i.e., assembling) reusable building blocks called *components*. From the definition of the component, we can see there is a clearly difference between component development process and the application assembly process. However, in the commonly used component definition[106, 147], no explicit description on when a component shall be composited to form an application is provided.

Traditional approaches treat components only as design time artefacts. Intensive studies have been carried out on designing languages to specify properties of a component, e.g. OMG IDL [4] and the Architecture Analysis and Design Language (AADL) [54]. Those languages are used to design and verify the application construction plan. Model-driven design tools are used to parse such descriptions and automatically generate auxiliary glue code. Later, those glue codes, together with component implementations, are assembled into a static entity and deployed with little capability of further changes.

The current trend on component model research is rather towards making components evolve at run-time [34, 61, 138, 148]; in these approaches, software adaptation is achieved by allowing the application structure as well as the adaptation behaviour to evolve in response to changes in the execution environments. This thesis proposes a new application composition methodology – both context-specific and compositional. This section also identifies the key concerns on the system designs in supporting such composition methodology.

4.2.1 The application construction methodology

In order to deal more effectively with run-time component composition, we propose a new methodology to explicitly incorporate domain-specific adaptation knowledge into the software composition & adaptation process. The new application composition flow, depicted in Figure 4-2, represents a procedure, which incorporates the functional design information with contextual concerns in compositing run-time software architecture. Depending on the employed design languages and corresponding tools, the compliance with the functional interface is enforced during the design process. However, after traditional off-line application construction process, an application's software architecture

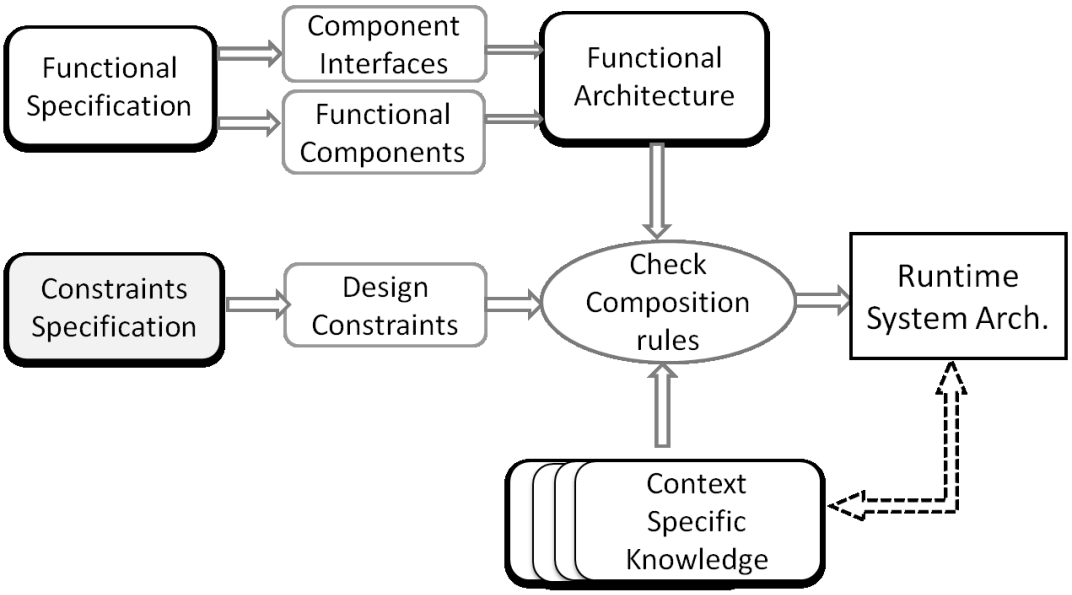


Figure 4-2. Context-specific application construction flow

knowledge is lost during the compiling process. Lacks of this knowledge can greatly hinder system adaptation capabilities. Without this design time knowledge, it is not possible to guarantee the design time constraints during adaptation. In our framework, rather than using this off-line construction process, an application is constructed during run-time. The design time information of each individual component is kept during run-time and explicit methods are provided to describe and expose these information.

As an application is constructed during run-time, in order to achieve correct and pointed adaptation, a set of constraints must be maintained. Among many aspects, three major factors influencing the application adaptation process are identified. 1) The functional dependence constraints must be satisfied 2) a component's non-functional constraints must be guaranteed: this information includes, for instance, requirements for CPU speed, screen size or that some property values are within a certain range. 3) domain-specific knowledge, which specifies the domain related information and adaptation strategy should also respected during adaptation process. This knowledge can be concerns for e.g. QoS, security and/or battery optimization. It normally includes one or more concerns and those concerns will change according to the changing context – which includes user's preference, system resource status and other factors. As described in the motivation scenarios, these concerns can be implemented as individual modules and used to guide the application adaptation process by composing a global adaptation modeller. Normally, in different contexts, different set of that domain-specific knowledge should be used.

The dashed arrow between the run-time architecture and domain-specific knowledge blocks means that managed applications are continuously restructured and evolved according to current context requirements and adaptation strategy. The combined knowledge enables automatic run-time verification for constraints from various aspects – functional dependence, non-functional constraints and domain-specific considerations. This allows the system to change the software structure according to its hosting environment and without violating constraints from these three aspects.

4.2.2 System design concerns

The main goal of this development methodology is to provide a guideline to integrate external context-specific adaptation capability, which includes adaptation strategies, context monitoring capabilities and adaptation actuator competence, into application (re) configuration process.

Under the guideline of this new application construction methodology, an adaptation framework is designed to provide support requirements identified from the motivation scenario and the new application construction methodology. The design of our Transformer adaptation framework is primarily focused on providing supports for the following requirements:

- Enabling adaptation modules reuse via separation of concerns (e.g. , separating the concerns of developing the self-healing adaptation strategy from video recording optimization strategies);
- Allowing contextual selection a set of applicable adaptation modules from plural available adaptation modules, as for different contexts, different (set of) adaptation behaviours might be desired;
- Providing adaptation plan fusion capabilities to detect and resolve possible adverse or conflicting adaptation actions from different adaptation modules;
- Providing mechanisms to identify the correctness of the fused adaptation model, as the global adaptation model incorporate adaptation concerns from multiple domains. It might generate incorrect or unexpected adaptation behaviours.

The first requirement is to support component reusability. Firstly, it allows the developers to reuse an appropriate off-the-shelf component from certain well-known component repositories to realize an application's business logics. Furthermore, in this thesis, focus is put on how to support the non-business components – adaptation modules to be reused in different contexts. For example, in mobile computing environments, mobility often suggests the adaptation strategies might need to change, for instance, while a user is on a train or he/she stays in his/her office. Optimization strategies can changes considerably in those two different locations. Contextual selection of adaptation module helps system to deal with the context evolution.

Adaptation module reusability can greatly reduce the burden of adaptive software programmer. To ensure system performance adaptation actions and strike a balance from requirements from different aspects, it is important for the system to provide explicit adaptation behaviour integration supports. This approach can also directly reuse many existing works in design-time adaptation or hybrid adaptation. As they can be modularized into adaptation modules, thus these research results can be directly reused and easily integrated into the system.

Systematic support for the run-time composition of the components involved should also be provided to simplify this adaptation evolution process. Finally, as these adaptation

modules can be run-time loaded into system without thoughtful integrity tests, it is very important to detect and deal with the situations when the installed adaptation modules give invalid (conflicting) tactics and strategies.

4.3 System architecture model

In this section, the key elements of the Transformer framework are introduced. Several key modules are designed corresponding to the application construction methodology described in Section 4.2.1. This adaptation framework is an enhancement of our previous work [70, 71] for context specific adaptation, in which only two modellers (the context-specific modeller and functional dependence modeller) are used to deal with system functional constraints and context-specific optimization strategies. And this framework is coherent with our conceptual architecture identified in Section 2.3.1

4.3.1 Design constraints for self-adaptation

We now briefly consider the design of a generic, self-adaptation framework that addresses the challenges discussed in Section 2.4.1. To fit for adaptation purpose, the run-time adaptation framework should be designed to monitor the target system dynamically without affecting the target system's normal operation. System global state should be tracked in a centralized way to collect overall system knowledge. This framework should provide supports for closed-loop based control and this closed-loop control should able to be tailored during run-time according to the system current environment and the user preference. This framework should also support composition and trade-off between different adaptation behaviours. These considerations lead to the following design goals for our framework.

- Run-time customable monitoring and adaptation support
- Global system model construction and maintenance
- Explicit separation between adaptation reasoning and actuation
- A modular design which supports incremental adaptation design

According to these design considerations, Transformer – an adaptation framework supports adaptation behaviour evolution, is designed and will be introduced in the following sections.

4.3.2 Architecture-based adaptation framework

Figure 4-3 shows the architecture of our Transformer adaptation framework. As can be clearly seen from that picture, our approach makes use of an extended control loop, consisting of six basic modules – *Event Monitor*, *Adaptation Actuator, System basic run-time, Structural Modeller*, *DSM* and *DSM Manager*. Here, the *DSM* and *DSM Manager* correspond respectively to the modeller defined in Section 4.5.1 and modeller selection process introduced in Section 4.5.2 .

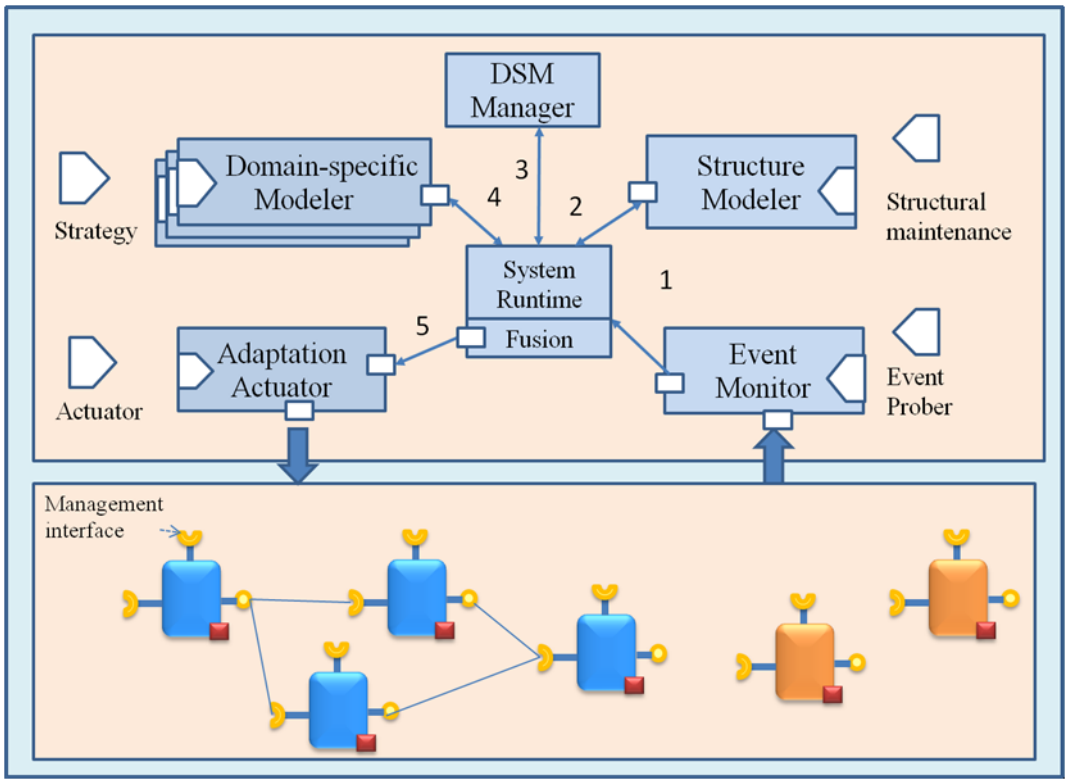


Figure 4-3. Transformer Adaptation framework

The Event Monitoring module observes and measures various system states. It sends notifications to trigger a new round of the adaptation process. Possible sources of adaptation may include a new component being installed or the CPU utility reaching a status that may have a significant effect on the existing system configuration. It could also be a simple Timer that triggers periodically. The Adaptation Actuator carries out the actual system modification. Its actual action set is tightly related to the component implementation. From our developing experience, in order to achieve effective architecture-based adaptation, the basic set of actions should include component lifecycle control, attribute configuration, and component reference manipulation. Adaptation actions might also trigger another adaptation process and create step-wise adaptation until the system reaches its new preferred state. The above two modules use a predefined component management interface to manage the installed component instances and form the Management Layer. The other three modules – DSM, Structural Modeller and DSM Manager, constitute what we call the Modelling Layer. This layer builds the system's global adaptation model according to the changing context. System run-time here takes responsibility to control the installation/de-installation of components, execution of component, and management of the references between components, and works as a mediator between the management and the Modelling layers.

As discussed in Section 4.2.1, building a software system adaptation model requires handling constraints from different aspects – this includes handling design-time knowledge such as interfaces or constraints as well as other domain-specific optimization aspects such as security, user's preference, performance etc. We identify that the software system adaptation needs to provide support for two main aspects – application architecture

management and domain-specific adaptation knowledge. Such aspects are managed by two specific types of adaptation modules – Structural Modeller and DSM.

The Structural Modeller – practically a custom DSM, handles functional dependence between components. Compared to other DSM, its major function is to manage software architecture. By checking composition compatibility, it decides whether a component is "structure-satisfied". An application can only be formed when all its required business components are structure-satisfied, no matter which context it is in. This characteristic gives it the uppermost priority in all the software adaptation aspects. At the same time, it is rather stable; in particular, it is invariant to context. That is why, in the software design, Structural Modeller is separated from the modeller selection process. It is always included in the adaptation process no matter what the external context is. As shown in Figure 4-3, for a system run-time, only one instance of this modeller will be installed. Separating the software structure maintenance from other adaptation logics can greatly reduce the development complexity for adaptation logic, as this design relieves the adaptation programmer the burden to manage component dependences, which becomes rather complex in the compositional adaptation process. Moreover, this design can also avoid the single point of failure problem. Event when the DSM Manager fails to perform correctly, the *Structure Modeller* can still keep software architecture coherent. Details will be discussed in Section 7.4.1.

In contrast, the DSM takes care of those adaptations, which will change according to each specific context. As our objective is to support adaptation with multiple concerns and run-time adaptation behaviour evolution to cope with dynamically changing environments, more than one DSM can be installed simultaneously. With many candidates DSM installed, a selection process is needed to determine the subset of the DSM that are appropriate for the current adaptation, see Section 4.5.2 for details. According to the current system context expressed as the current value of various system metrics, DSM Manager selects the set of modellers matching the current context.

Together with the Structural Modeller, selected DSM collectively builds a global adaptation modeller. In order to solve possible conflicts between multiple adaptation plans computed by different modellers, the Model Fusion module is designed.

In order to effectively support *Inversion of Control* – accepts and executes adaptation from external entities, the underlying component model needs to be redesigned to support external component management. At the same time, it should also be reflective – its design-time knowledge as well as its run-time status should be available to external adaptation modules, for instance, its provided/required interfaces and non-functional requirements, e.g. CPU speed > 500 MHz or Memory > 20 MB. Only with this information, can an external adaptation module make effective adaptation. In order to satisfy these requirements, a declarative and reflective component model is introduced (see Section 5.2.1).

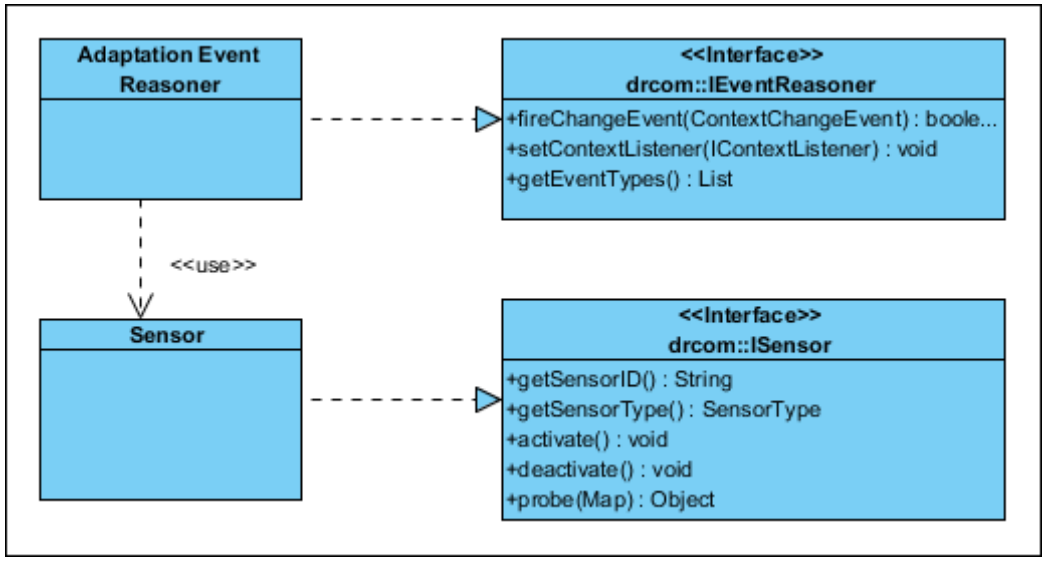


Figure 4-4. Sensor, Event Reasoner and their relationships

4.4 **Component management layer**

In order to get monitored data out of target system for building system architecture model for adaptation reasoning, and execute adaptation plans and push those changes back to the managed system in a uniform way, we need a layer that provides standard mechanisms for state monitoring and plan actuation for different software applications. This layer is denoted as *Component Management Layer*. This layer hides the implementation difference of underlying systems and provides a bridge between abstract system control model and practical system implementation. This layer is built on our prior work on general management interface design [68].

4.4.1 Adaptation event reasoning mechanism

An adaptation Event Reasoning probes (directly or indirectly) and gauges system states then uses its own logics to analyse the monitored state and decides whether to sent notifications. As we can see from Figure 4-4, the process of monitoring & reasoning data in the Transformer framework has a hierarchical nature. The source of the adaptation event notification can be constructed with two major types of modules – *Sensors* and *Adaptation Event Monitor*.

4.4.2 Sensor

Sensor is a basic entity that exposes monitored values to external users which can be application modules or adaptation modules. These sampling devices are equipped with a predefined interface that allows uniform and predictable access methods.

The interface of the sensor is extremely simple, the major method is:

Object probe (Map params) – this method returns a sampling value for the sensor with certain device-dependent and optional parameters *params*.

In order to minimize the resource consumption, it is argued; only one sensor for a specific monitored object is needed. The objects can be monitored basic system metrics such as current CPU utilization, the current user location, or application-specific values, such as the memory usage of one specific-application or application-specific performance metrics, etc.

Depending on implementation, a sensor may either probe the monitored value on demand or return a cached value for rarely changing data to get better performance. It can be implemented that its target value will only be retrieved when the probe methods are called (reactively) or autonomously select when to retrieve the target object information (proactively). Normally, a *sensor* only provides raw monitoring data. The role of collect, aggregate and abstract these raw data, which more related to system architecture model and adaptation model, is implemented by *Adaptation Event Reasoner.*

4.4.3 Adaptation event reasoner

For adaptive system, adaptation event reasoning support is one of the key requirements for system design. Different with the sensor module, which only monitors certain system basic metric, *Adaptation Event Reasoner* contains adaptation logics on how to accumulate, evaluate and send change events to the adaptation framework.

Rather than the sensors, which are designed for general resource monitoring, the Adaptation Event Reasoner is associated with the adaptation conditions in the system adaptation model. When a constraint violation is detected, it notifies the *system runtime* to trigger adaptation. As different types of sensors are developed into the target system to collect system state information and measure quality attributes, different *Adaptation Event Monitors* are needed. For instance, it is normal to have requirements for certain sensor value while different value users might need the data in a different way. For instance, one monitor user is interested in the current CPU utilization value while another one demand the value of average CPU utilization over the last 5 minutes. For different DSM, different and customized *Adaptation Event Reasoners* are needed to collect and interpret system properties.

The *Adaptation Event Monitor* should also emit system changes events to the system run-time by evaluating certain conditions, which reside in the evaluation logics in it. These change events will be processed by the system run-time and redirected to interested DSM. Depending on the generated adaptation plans, one event can trigger a round of system adaptation process until system reaches stable state.

4.4.4 Adaptation Actuator

In Transformer, the Actuators carry out change operations on the target software. Those actuators, implemented as individual components, are corresponding to the adaptation operators introduced in Chapter 2. According to the definition, the function of an actuator

could range from a simple component property setting, to a component update, to a complex adaptation action that can change a whole workflow (e.g. KX Worklet [150]). Similar to the sensors and the *Adaptation Event Reasoner, Adaptation Actuators* depend on system underlying structure and adaptation capabilities.

For instance, in our previous work, we build our adaptation framework based on our DRCom model which supports a general management interface [69, 71]. This interface allows two major adaptation actions to be supported – parameter-based adaptation and compositional adaptation. Thus the adaptation actions that can be taken are: components' lifecycle management – start, stop, pause, stop – as well as properties manipulations via the setProperty(…) method, for example, using this method to change a computation task's priority, period, etc.

As discussed in the introduction, modern software systems, such as Microsoft Dynamic Systems Initiatives[6], are increasingly support sensing and effecting capabilities.

In order to reduce the implementation complexity in implementing a new actuator, in this framework, a complex actuator can be orchestrated by basic actuators. In the following sections, we will discuss these two types of actuators – the basic actuator and the application custom actuator.

4.4.4.1 Basic actuator

A *Basic Actuator* represents a basic command provided by the underlying component model as well as system run-time. Normally, this type of actuators will be directly mapped to system-level adaptation actions. For example, an architectural operator to stop a component can be translated to a system-level operation that call the component's predefined "stop" method. Other examples of operators include: starting component, setProperty and (un) binding links between two depended components.

As these basic adaptation actuators highly depend on the underlying component model, in order to separate system adaptation modules from underlying implementation details, a general interface is required for different actuators to allow they are accessed in a general way. At the same time, these basic actuators are natively provided as basic middleware service. By providing many basic services ready available this framework, this design can greatly simplified the adaptation modules developments.

However, each specific DSM might need more complex adaptation actions. Hence, in the Transformer adaptation framework, user custom actuators with more complex and more powerful adaptation logics are supported by composing these basic components.

4.4.4.2 Application custom actuator

As distinguished from the *Basic Actuators*, *Application Custom Actuator* provides a more complex and featured adaptation enhancement. It packages basic adaptation actions into larger units of change, by orchestrating these actions in a temporal and logic orders. A user custom actuator should haves following characteristics:

- It specifies a sequence of basic adaptation operation
- It is guarded with a set of conditions that determine its applicability;
- It defines a set of effects that should be observed after sequence completion;
- It should define ontology on its relationship with other operations.

The *Application Custom Actuator* offers an abstract primitive for adaptation that has separate concerns from the supported operator to modify system elements. The user custom actuator provides DSM developer additional atomic adaptation units by allowing multiple adaptation operations to be composited into a more sophisticated, but one atomic step of adaptation. For instance, an atomic adaptation action – repair (decoder) for self-healing, can be finished in one step. Another key requirement for the User Custom Actuator is that – each actuator must provide semantic information on the relationship between installed actuators. This information is used by model fusion module to detect and resolve possible conflicts (see Section 4.5.3).

4.5 **Adaptation layer**

Once a problem is detected, a mechanism is needed to decide appropriate adaptation strategies to guide the adaptation. The other modules – *DSM*, *Structural Modeller*, *DSM Manager* and *Modeller Fusion*, constitute what we call the Modelling Layer. This layer builds the system's global adaptation model according to the changing context. System run-time here takes responsibility to control the installation/de-installation of components, execution of component, and management of the references between components, and works as a mediator between the management and the Modelling layers.

4.5.1 Domain-specific adaptation modeller

As already described in Section 2.3.1, in software adaptation, system evolution can be driven by different viewpoints, for instance, security and performance. These concerns can be expressed as a set of optimization strategies and constraints and implemented as a system adaptation modeller. To express this domain-specific optimization goal, we use the concept of Domain-Specific Modeller (DSM). Normally, each DSM normally has its specific application environment. Only within their environment one modeller can effectively conclude the right adaptation actions. We call it Domain-Specific Modeller with Context-constraints (DSM). In the following discussion, we will use these two terms interchangeably. DSM is represented as an adaptation function $\mathcal{F}$ with a list of Context Matching Constraints (CMC). CMC contains a DSM's possible range of acceptable resource values. For instance, A CMC constraints such as { (CPU_speed > 70MHz), (Preference = "TV Quality")} shows that this DSM best fits for a context where the CPU clock is faster than 70MHz and the current preference of the user goes to optimizing the quality of the TV experience.

$$\mathcal{F}: \quad \mathrm{SC} \xrightarrow{\mathrm{z}} \wp(\mathcal{A}_{\mathrm{sc}}) \quad \text{for} \quad z_{\,current} \in \mathrm{CMC} \qquad (1)$$

Once the current context $z_{current}$ is disclosed, a matching rate of $z_{current}$ with respect to CMC can be calculated. Such matching rate expresses how close the current context matches the acceptable resource values. To what degree the constraints match the system's current environment will greatly affect the accuracy of the adaptation actions that a DSM might take.

A DSM computes its domain adaptation plan according to system current configuration sc ∈ SC, SC being the set of all possible system configurations. Its adaptation module (expressed as function $\mathcal{F}$) computes a set of adaptation actions $\wp(\mathcal{A}_{\mathrm{sc}})$ according to current context z_{current}. Z = $\wp$ (Resource×Value) represents the set of all possible execution contexts and $\mathcal{A}_{\mathrm{sc}}$ is the set of all possible actions eligible in system current configuration sc. $\wp(\mathcal{A}_{\mathrm{sc}})$ is a subset of to $\mathcal{A}_{\mathrm{sc}}$. Here, $\mathcal{A}_{\mathrm{sc}}$ highly depends on the underlying component implementation. Only those actions supported by the underlying component model can be successfully executed. For instance, in our previous work in declarative real-time component model [69], a management interface was designed which permitted to start, stop, and change properties of components. In such case, three types of adaptation are supported {enable, disable, setProperty}. As we already discussed in Section 2.1.4.2, for a software system with limited number of components and limited configurations for each components, the number of possible actions is limited.

Whenever there is a significant change requiring the system to adapt, each modeller $\mathrm{M_i}$ involved in the adaptation process will compute $\mathcal{A}_{\mathrm{i}}$ as the adaptation action set – that will be taken by that modeller. To mean this we shall write

$$\mathcal{A}_{\mathrm{modeller_i}} = \mathcal{F}_{\mathrm{modeller_i}}(\mathrm{sc})$$

4.5.2 Domain-specific modeller selection

When there are multiple (n) DSM modellers installed in the system, in general, each of them will have their own specific modelling functions and application scopes. Only within its application scopes, one modeller can effectively conclude the right adaptation actions. One familiar example is notebook's power management – the modeller optimized for battery normally will not exhibit satisfactory QoE for the user while the notebook is plugged in. In each particular context, only the part of the installed modellers that match that environment should be used for the adaptation process.

As a DSM participating in the adaptation process is equipped with CMC to describe their context preference, it is possible to design a dynamic selection scheme to choose the right set of modellers to be used in the ensuing adaptation process. We call this process as context-specific modeller selection, and denote it as function $\mathcal{S}$

$$\mathcal{S}: \quad \mathcal{M} \xrightarrow{\mathrm{Z}} \wp(\mathcal{M}) \qquad \wp(\mathcal{M}) \subseteq \mathcal{M} \qquad (2)$$

Function (2) expresses that, given $\mathcal{M}$ − the set of all installed modellers − the semantic function $\mathcal{S}$, by checking a similarity degree with current context, computes the set of n enabled adaptation modellers $\wp(\mathcal{M})$ from the set of all modellers. In Section

5.3.3.2, the algorithm for context matching is introduced. After the system chooses the right set of modellers, all selected modellers will be inquired to compute their adaptation plans individually according to their domain-specific optimization goals. Such plans then must be combined into a global, conflict-free adaptation plan. This is what we call Model Fusion.

4.5.3 Model fusion

By using multiple DSM to compute their own domain-specific adaptation plans, global adaptation is derived by taking all these different adaptation plans into account. As we can see, although different modellers only deal with certain aspects of system constraints, their managed entities are the same, that is, the set of all installed components. Thus, their decisions on system adaptation can conflict with each other. One simple example can be e.g. one modeller choosing to stop one component that another modeller prefers to keep enabled. As such component cannot be in the enabled and disabled state at the same time, conflicts arise.

In order to achieve a coherent architectural adaptation, a process is needed to identify and resolve possible conflicts between adaptation plans. In our framework, a Model Fusion module is designed to provide explicit conflict resolving support. After collecting different adaptation plans from different selected modellers, it computes the conflict-free solution set $\wp(\mathcal{A})$, that is, the set of conflict-free actions that all modellers involved in the execution of the service component have agreed upon.

$$\mathcal{H}: \ \{\mathcal{A}_{\mathrm{modeller}_1} \dots \mathcal{A}_{\mathrm{modeller}_n}\} \xrightarrow{\mathrm{Z},\ \wp(\mathrm{M})} \wp(\mathcal{A})$$

$\mathcal{H}$ is a function that maps multiple modellers' computed action sets into one conflict-free action set. A common adaptation action set is computed based on all involved modeller's characteristics, adaptation actions properties and system current context $\mathrm{z}_{\mathrm{current}}$. By taking all these factors into account, the model fusion module comes out with a final agreed adaptation policy $\mathcal{A}^*$.

$$\mathcal{A}^* = \ \mathcal{H}\left(\mathcal{A}_{\mathrm{modeller}_1}, \dots \mathcal{A}_{\mathrm{modeller}_n}\right)_{[\mathrm{mlist},\ \mathrm{z}_{\mathrm{current}}]}$$

As we can see, in order to design an effective model fusion functions, many factors are needed to be taken into account, such as selected DSM, their characteristics and the relationship between adaptation actions. In Section 5.3.5, a set of fusion policies is provided as a proof of concept in merging the adaptation plans from multiple DSM.

4.6 System adaptation route

In the model fusion process, different modellers' adaptation plans are merged into one conflict-free adaptation plan. A new global adaptation model is generated through this process. However, as this global model is formed during run-time, it is possible to produce

an ill-formed adaptation model – e.g. create unlimited adaptation loops. In this thesis, an on-line adaptation loop detection & limitation algorithm is proposed to detect and break possible adaptation loop. In order to explain this algorithm, the system adaptation behaviour – the step-wise adaptation process is firstly introduced.

4.6.1 Step-wise adaptation

As many adaptation actions have temporal or logical dependences, these actions must be executed in different steps. To capture and deal with this, the adaptation process is managed step-wise. One typical example is the initialization sequence of component-based applications – for instance, in Figure 4-5, component N and component N' cannot be enabled until component M finishes its initialization process because of their functional dependence on component M.

A graph of interacting components composes a system configuration variant (denoted assc ϵ SC), such as states A, B, and C in Figure 4-5. Adaptation operations result in modifications of this graph. This process is called configuration evolution. This figure shows examples of such evolutions: action "disable N" makes sc move from state A to state B. Figure 4-5 also shows the state transitions among 3 states – an excerpt of CTG with only 3 components. For simplicity, we assume the CTG is a finite graph which has a limited number of vertices (states) and each has only a limited number of transition links.

In each step of adaptation, the fused modeller makes certain adaptations to let the system transit from current configuration $\mathcal{v}_{i-1}$ to one of its adjacent configuration state $\mathcal{v}_i$. These adaptation actions will raise events that will trigger next round of adaptation. A detailed example for the event-driven adaptation process can be found in Section 5.5.2. If system arrives at its preferred state configuration $\mathcal{v}_{pref}$, the adaptation process will stop: the system will stay in that configuration state until another context change triggers a new round of adaptation. Without loss of generality, we denote the migration route starting state with α and the ending state with $\mathcal{v}_{pref}$. We refer to a path P by the natural sequence of its vertices, writing, say,

$$P = \alpha x_1 x_2 \dots x_i \dots x_n \mathcal{v}_{pref} \quad \alpha, x_i, \mathcal{v}_{pref} \in SC$$

For the sake of simplicity we assume that an adaptation process takes less time with respect to contexts change intervals. So, during a round of adaptation, the $z_{current}$ will not change. Of course, this is no real limitation, for if a new context change occurs while we are in the middle of an adaptation, e.g. in any intermediate state x_m, then we just consider $\alpha = x_m$ as the new starting state and some $\mathcal{v}'_{pref}$ as the new preferred state for the new context..

As different modellers contain different adaptation policies and user preferences, each of them might deduce a totally different preferred configuration and evolution path. This great variety makes it very hard to verify the correctness and effectiveness of the fused modeller. In this paper, we make a preliminary step towards this goal – guaranteeing the absence of infinite loops in any possible adaptation modeller.

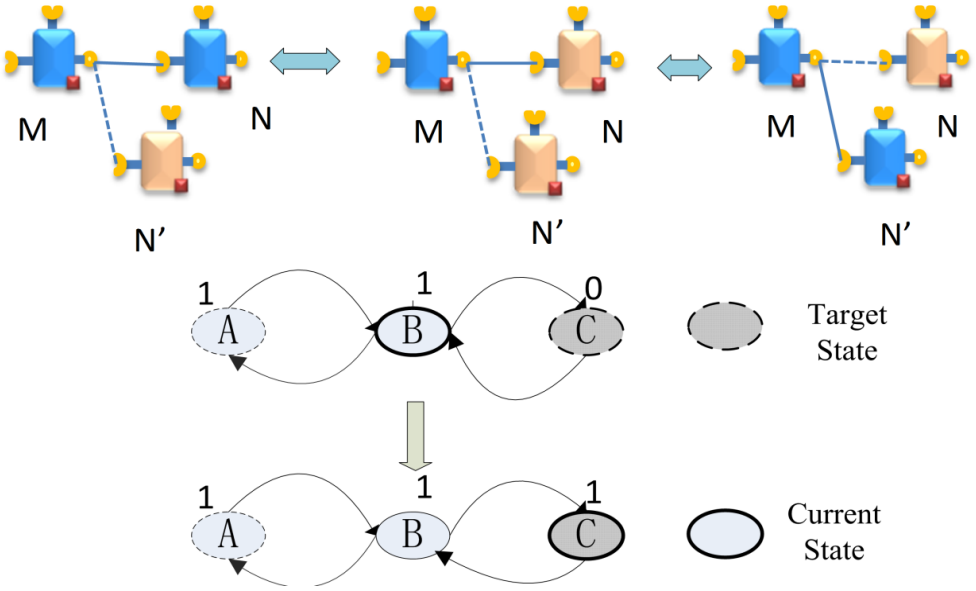


Figure 4-5. Adaptation loops for system configuration migration. The picture shows a fraction of the CTG with three states {A, B, C}. The current state is B and state C is the target state. System global adaptor might create adaptation loops such as B⇒A⇒B⇒A … and could not find the way towards C. The visiting count for each state shows that the visiting count of both state A and B is equal to 1 while visiting count for adjacent state C is 0 – thus smaller than A's. Although adaptation modeller chooses state B as target state, our algorithm will force the system to migrate to state C instead as it has the least visiting count. Thus the preferred state C is reached

4.6.2 Online loop detection & prevention

For this requirement, one natural solution is to use shortest path algorithms such as Dijkstra [46] algorithm to find an acyclic "best" path with maximum transition gain. However, as a global adaptation model is formed by fusing multiple modellers' adaptation plans, it normally could not be expressed as simple utility functions. This complexity makes it very hard to pre-compute each adaptation's transition gain for different models. Furthermore, for many merged global models, their preferred state v_pref can not be known until system reaches that state. These constraints make path algorithms inappropriate to solve this problem.

4.6.3 Online verification algorithm

Our goal is to design an online verification mechanism that can guarantee that the system reaches its preferred state no matter how the adaptation modeller was constructed. At the same time, this mechanism should preferably have minimal impact towards fused modeller's adaptation plan. In other words, this mechanism should only interfere when an error occurs. Following two guidelines, an online verification algorithm has been designed. Its effectiveness can be proved when the following two assumptions hold:

Assumption 1: Preferred state exists

$$\forall z_{context} \in Z \qquad \exists v_{pref} \in V$$

Assumption 1 says that for any possible context$z_{context}$, the fused global modeller will have at least one preferred state. So, this modeller will stop at a fixed configuration point for any particular context.

Assumption 2: Finite State Transition graph CTG is connected

In what follows, we assume that from any initial system configuration in CTG, it is possible, with a finite number of steps, to reach any other system configuration. In other words, this means that for any two system configurations (vertices in CTG), there exists a path between them. We can say that CTG is connected. This is a natural assumption because if the CTG were not connected, some target preference state could not be reached no matter how the modeller is designed.

Algorithm 4-1: Online Verification

Requires: system can record visiting count of each configuration

Ensure: system reaches its preferred state in the state graph within a limited amount of steps.

1. $v_{next} = \mathcal{F}(v_{current})$;
2. If $(v_{next} != v_{current})$ //if the computed configuration is not equal to current one
3. For (all adjacent vertices)
4. find minimal visiting count Min(V_{adj}) in point $v_{current}$ adjacent vertex set V_{adj} with minimal visiting count $V_{(adj,Min)}$
5. End for
6. If Count(v_{next})> Min(V_{adj})
7. //random select one from the adjacent configurations with $\mathrm{Min}(V_{adj})$ visiting count
8. $v_{next} = \mathrm{Random}(v_{(adj,Min(Vadj))})$;
9. End if
10. Count(v_{next}) + +; // Perform adaptation move to v_{next}

11.End if

12.else // find preferred state

13. for (int i=0, i<v(G); i++) // Clear all visiting counts for each vertex
14. Count(v_i) = 0; // in the state graph
15. Stop // stop adaptation process
16. end else

The algorithm is listed in Algorithm 4-1. In this algorithm, the system will record the visiting count for each configuration. The basic design guideline is to use the modeller's adaptation strategy unless a loop is identified. When this exception in the adaptation logics is caught, in our algorithm the system selects the vertex with minimal visiting count instead of the one prescribed by the adaptation strategy. The adaptation process will stop when the system reaches the preferred state. The preferred state is identified if targeted configuration is the same as current configuration, which means no adaptation action is needed.

Figure 4-5 shows an example for this on-line verification process. It shows that with the help of this online verification algorithm, the system can arrive at its preferred state C even

when the global adaptation modeller creates infinite loops. We also prove that, no matter what kind of global adaptation modellers are built, as long as the two above assumptions are satisfied, Alg. 1 can guarantee that the system reaches its preferred state. The proof can be found the next section. Of course this is only one possible verification strategy for which we can prove the correctness. Other approaches, such as proactive problem detection mechanism can be used to solve this problem [103].

4.6.4 Convergence criteria

In this section, we will prove that under the adaptation policy specified in the last section, system will migrate to the preferred configuration in limited steps.

Proposition 1: For any adaptation policy, under rule 1, from any starting state α, the system can reach its preferred state β with a limited number of steps.

Prove: As state graph G is connected, there is at least an acyclic route (from α to β) with limited length. Without loss of generality, we denote this path P as vertex sequence P $=\alpha x_1 x_2 \dots x_{n-1} x_n \beta$. Two adjacent vertices along the path are directly connected, e.g. verge $\{x_i, x_{i+1}\}$ and verge $\{x_n, \beta\}$. We denote a state's out degree as $d_{out}(v_i)$. We firstly prove the following two lemmata.

Lemma 1: Any vertex v_i along the path from α to β could only be visited a limited number times.From Rule 1, if state β is arrived, system will stop adaptation, so β could only be visited once. Its precursor vertex x_n (its precursory vertex means they have outgoing verge to state β) could not be visited more than its out degree $d_{out}(x_n)$. Otherwise, it will contradict the rule 1 which always selects the vertex with minimal visiting count. Similarly, as x_n can only be visited $d_{out}(x_n)$ times, x_{n-1}, adjacent to x_n, it could not be visited more than $d_{out}(x_{n-1}) * d_{out}(x_n)$ times. So we can prove that the visiting count $v(\alpha)$ of α

$$v(\alpha) < \prod_{i=1}^{i=n} d_{out}(x_i)$$

As the route is with a limited number of vertices – less than or equal the order of Graph G, and each node has limited out degree d_{out} – at most v (G)

with Length (P) $\leq v(G)$ and $d_{out} \leq v(G)$

We conclude that all the nodes along the route can only be visited a limited number of times.

Lemma 2: Any vertex inside graph G could only be visited limited times before the system reaches preferred state β.

As our graph is a connected graph, for any vertex, without loss of generality, $v_i \in V$, there is at least one path P_{v_i} from v_i to β. As Lemma 1, vertex v_i can only be visited limited amount of times.

We can prove the Lemma by contradiction. Suppose we can find a route R that has unlimited length before visiting β. As graph G has limited vertices, so there is at least one vertex which will be visited an unlimited amount of times, which contradicts Lemma 2 - no

vertex inside G can be visited with unlimited times. So, no route of unlimited length exists. **Proposition 1 is proved.**

4.6.5 Discussions

In this section we described our preliminary approach towards loop detection and prevention. Algorithm 4-2 particular tells us that that it is possible to carry out an adaptation procedure without unlimited loops in a limited number of steps, though we did not characterize yet the complexity of this process. Our future work will be to try to derive this algorithm's computation complexity. Another important aspect is the fact that, in our current implementation, breaking the loop implies choosing a different next state than the one expected in the original adaptation plan. Even though we hook back into the original plan immediately, the impacts towards the system in taking an unforeseen next state transition are largely unknown. In practice when breaking the loops we envision the addition of domain specific constraints to select an appropriate next action to break the loop.

4.7 **Conclusions**

This chapter described a development approach for the creation of adaptive applications that fit with changing contexts which is typical in mobile and pervasive computing environments.

Unlike the conventional approaches, which merge an application's non-functional considerations together with its business logic in a static and customized way, in this thesis, a modular, incremental approach, guided by separation of concerns, is proposed to build application adaptation behaviour. Firstly, this methodology separates the design concerns of self-adaptive behaviour from the task of application business logic design and implementation. Then, the adaptation logics are divided into multiple domain-specific adaptation concerns. An adaptation framework, which supports such adaptation behaviour composition, is designed and semantics of each module are described and defined. In order to check the correctness and convergence of a run-time fused global adaptation behaviour, an online verification algorithm is proposed to make sure the adaptive steps will not create infinite control loops. The convergence of this algorithm is formally proved in this chapter.

In the following chapter, a supporting pluggable and modular middleware architecture will be introduced to provide a middleware-based software architecture to support the design goals in the Transformer framework while strike a balance between the complexity and flexibility.

Chapter 5

A Reflective and Modular Middleware Architecture for Run-time Adaptation Composition

This chapter presents a supporting middleware architecture for Transformer adaptation framework. This middleware firstly support an adaptation component model which allows individual DSM to be modulated as components and then dynamically composed. A meta-adaptation layer is provided to form the system global adaptation behaviour via composing DSM components. In addition to the key functions designed in the Transformer adaptation framework, the proposed middleware architecture also allows DSM dynamicity. It mean that DSM providers (such as a self-healing adaptor) as well as their affiliate Event monitor(s) and Actuator(s) they can be added or removed during run-time, without requiring that the targeted applications (such as the TV and Recording applications) are restarted. This feature is favourable for the adaptation in the fast changing environments, for which users normally prefer to make adaptation while keeping their application running. Moreover, by building system run-time on top of a service component oriented architecture [72], we achieve the adaptation modules reuse at binary code level.

In order to optimize system resource consumption (e.g., CPU usage, memory usage, battery drain, etc), this middleware is equipped with capabilities to intelligently and autonomously employ and dis-employ DSM plug-ins according to system's current context. An extensible meta-data format is designed for DSM. This meta-data contains both functional related information (such as required Event Reasoners/Actuators) and other non-functional requirements (such as CPU, bandwidth requirements, etc) of a DSM. By

parsing this meta-data, the middleware will determine whether this DSM can be used. Furthermore, as the Transformer framework targets at run-time adaptation, this middleware must be run-time adaptable. This means the configuration of adaptive software can be revised during runt-time.

5.1 **Introduction**

As it was discussed in the previous chapters, our solution focuses on adaptation in the face of changing environments, which is typical in mobile and ubiquitous computing. While the importance of such applications is becoming increasingly apparent, the complexity of their executing environments also introduces a number of challenges for the software architecture design.

In this regard, a comprehensive software platform is desired to provide support for both software developers and users, from development time to run-time. Requirements for such a middleware have been discussed in Section 1.1, and these requirements were used as design guidelines for our middleware design. They are later used for evaluating the conformance between this achieved middleware and the predefined goals.

One of the most important requirements of our middleware architecture is to provide support for adding, removing, selecting, accessing and resolving adaptation modules, as it was discussed in Section 1.1. These functional requirements have lead to the formal definition of each process and the discussion of the component resolving process and DSM activation mechanisms. Beside these aspects, many design issues related to the software implementation will be discussed in this chapter.

This chapter introduces the software system design to implement the Transformer adaptation framework proposed in the last chapter. Several key modules are designed corresponding to the process described in Chapter 4, with detailed software system design. However, in order to strike a balance in implementation complexity while keeping system flexibility, some revisions are necessary. Software system design will be discussed in detail in this chapter. This software architecture is an enhancement of our work [70] in context-aware adaptation, in which only two modellers (the context-specific modeller and functional dependence modeller) are used to deal with system functional constraints and system optimization strategies.

5.1.1 Design consideration

As discussed in Section 2 and in the previous section, the middleware design consideration is summarized as follows:

Modularity: As our platform target with mobile and ubiquitous computing, those executing environments might have significant difference in both resource capabilities and/or executing requirements. Thus, a modular architecture is preferred to ensure that this system can be easily tailored to the changing environments. This component-oriented architecture enables that only those features explicitly needed by the current working

environment are used. Configurations of the middleware as well as target software applications can be composed according to the changing constraints. Furthermore, this custom-tailored architecture also means that resource footprint can be optimized as only those features needed are activated. It is especially important for mobile and embedded environments, in which normally only limited resources and capabilities are available.

***Dynamicity*:** This middleware architecture is designed to support dynamic behaviour of the adaptation modules, where components can be installed, uninstalled, activated and deactivated without the need to restart the whole system. This continuous deployment support is desirable for several reasons. Firstly, it facilitates the reconfiguration of adaptation modules with run-time available components. Instead of hard-coding the adaptation logics inside the application itself or within the middleware, this architecture allows other commercial off-the-shelf or future more complex adaptation strategies to be plugged into the system dynamically as the adaptation plug-ins. These plug-ins can be reused across multiple contexts in binary form. Secondly, such architecture allows for dynamic enabling and disabling adaptation modules as needed (which also holds true for monitors and actuators). Compared to those solutions with only deploy time customization capability, such dynamicity support can achieve far more flexibility.

Lightweight: This requirement comes from the need for both development and deployment aspects in achieve high resource efficiency. Lightweight design enables easy development of adaptation modules and fast deployment on targeted devices especially resource limited ones. In order to strike a balance between the completeness and complexity, it is preferred to have a flexible way to provide incremental deployment – capabilities to add/remove features as needed.

Global adaptation: One characteristic of architecture-based adaptation is the global-level architectural model construction and maintenance. This means that the middleware platform should keep tracks of system changes and maintain a global image of system's current configuration rather than the local view based approaches in application-based adaptation. This middleware should be able to monitor the changes of each installed component's lifecycle and retrieve the current configuration of the installed components during run-time. From a development perspective, this requirement actually means that either all the component control actions must be done through middleware exposed access interfaces, or that these actions are at least detectable by the system runtime. On the contrary, system would lose track of the precise knowledge of the current configuration.

According to those design consideration, several design choices are made. This includes the DRCom component model to enhance modularity and dynamicity, the DSM manager to achieve global adaptation and the customizable architecture to achieve low resource usage. Firstly, the DRCom component model is introduced.

5.2 **DRCom component model**

One of the key design principles of our architecture is to enable DSM independent development and deployment, which means the concerns of developing one

domain-specific adaptation logics are separated from the concern of other DSM, as well as from Structural Model Maintenance. This principle leads to the design of our pluggable architecture in which the DSM are designed and implemented as plug-ins. Such plug-ins can be independent deployed and activated. We refer to these components as DSM plug-ins. Adaptive applications – the targeted manageable elements – are only loosely coupled with the DSM plug-ins. Besides this layer of separation of concerns, DSM is designed to represent only one domain-specific optimization strategy, ideally mutually orthogonal with each other.

To support these two layers of separation of concerns (business logics vs. adaptation logics, and DSM vs. DSM), a middleware-based system is defined and implemented. This system works as a context-aware DSM management centre: it collects, stores, processes, fuses available DSM and orchestrates adaptation actions on the applications it manages. This approach enables developers of DSM to design their own DSM plug-ins and facilitate the reusability of existing ones. Each DSM can be run-time enabled and activated by the middleware according to the changing context. Such dynamicity requirements lead to a comprehensive run-time adaptable component model, which is referred to as Declarative & Reflective Component Model (DRCom). This component model is implemented based on the OSGi component framework.

5.2.1 Declarative & reflective component model

In this section, we describe our component model – DRCom, for the system basic composition unit. As components might have totally different business logics and possibly quite different application domains, in order to enable their execution container – system runtime – to reason about, and possibly alter, their behaviour, this component model is designed to have a general management interface. Thus, selected details of an installed component's instance can be retried without compromising its portability.

For a component, two different types of introspections are identified – structural reflection and behavioural reflection. Structural reflection addresses issues related to class implementation, component required/provided interfaces, interconnections, and data types. This information is comparably stable and normally will not change with a component's execution process. Conversely, behavioural reflection focuses on the application's run-time semantics. For instance, a component's properties or settings can be changed, by internal mechanisms or external ones, during run-time.

In order to achieve efficient structural reflection while providing reflective run-time behaviour support, a hybrid approach is adopted in our implementation. In this hybrid approach, meta-data is used to describe component static structure information as well as non-functional requirements. A XML-based schema for this meta-data is defined to described component structure related information (see Section 5.2.1.2), just like using the CORBA Interface Definition Language [4] to describe CORBA objects. For the run-time reflective support, a component's run-time status is exposed and managed by a management interface, which is a mandatory management interface for every DRCom component instance (see Section 5.2.1). In addition to the run-time status exposition, this

interface also provides the functionality to access the component's meta-data, which includes its contextual requirements. From the development perspective, these DRCom components are defined as OSGi bundles and can largely reuse many OSGi native services.

5.2.1.1 Extended component header

In order to enable the system run-time to check the XML documents for the component structure-definition, it is important to let the system run-time know where it can find the meta-data. Our component model is based on OSGi native component mode – bundles in OSGi terms –in which each bundle has a mandatory manifest file located in META-INF/MANIFEST.MF. This file contains metadata about the bundle itself, which contains Manifest-Version (defined in the JDK specification), Bundle-ManifestVersion (defined in the OSGi specification), Bundle-Name (defined in the OSGi specification), etc. One example o such manifest file is shown in Figure 5-1. Detailed introduction on these formats can be found in the OSGi specification [114]. It basically is a simple name-value pair based property file and can be extended to add user-custom properties. As an example, Diaz Redondo et al. use this manifest file to add semantic information shared between the MHP and the OSGi platform [128].

However, as the manifest only supports simple name-value pairs, it is not appropriate to describe more complex structural information such as a component's provided/required interface. In order to describe such complex information, certain extensions are needed. In our DRCom component model, a new entry – DRCom, is added into the standard manifest file. As shown in Figure 5-1, the entry declares the current component is a DRCom component and the DRCom component description can be found in the DR-component.xml file in the DR-INF directory located under the JAR root directory. When a component is installed, our DRCom run-time will try to check the DRCom entry. If it exists, the xml file designated in this entry will be read and parsed.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SmartCameraRTBundle
Bundle-SymbolicName: SmartCameraRTBundle
Bundle-Version: 1.0.0
DRCom: DR-INF/DR-component.xml
Import-Package: org.osgi.framework;version="1.3.0
……
```

Figure 5-1. Sample manifest file for DRCom

5.2.1.2 Structural reflection

In DRCom implementation, a XML-based description file contains a component's context requirements and functional contracts. In order to satisfy the lightweight requirement identified in Section 5.1 while maximizing system usability across multiple systems, instead of supporting multiple programming languages such as Corba IDL [4], we decided to limit our approach to the Java language implementation. This choice

eliminates the need to translate a general component description language into a language-specific component implementation.

In a basic DRCom component, its description normally contains the following elements: the component name, description, status (enabled, disabled), and types of initialization (immediate, lazy start…). This description can also contain a set of component specific properties, which can be used to configure and identify the component instance from those of other providers. The programmer can locate certain component instance with the help of component's properties. The component should also specify a set of functional interface that represent a component functional contract – including required interface and provided interface, so as to interoperate with other components.

Figure 5-2 shows a fragment of meta-data file, which describes a smart camera that can return regions of interests (subsets from a frame image data) on demand. Such application is being used in the framework of IST project ARFLEX [3], on Adaptive Robots in FLEXible manufacturing environments [67] .

Functional meta-data

The **component** element has the following attributes:

• *name*: The *name* attribute of a component is a simple string. It must be globally unique because it is used as a reference.

• enabled: Controls whether the component is enabled when the bundle is started. The default value is true. If enabled is set to false, the component is disabled until the "activate" method is called.

• immediate – Controls whether component configurations must be immediately activated after becoming satisfied or whether activation should be delayed.

The implementation element is required and defines the name of the component implementation class. It has therefore only a single attribute: class. It's a Java fully qualified name of the implementation class. The component instances will be created by system run-time by referring to this attribute.

The **service** and **reference** elements define the communication methods by means of which inter-components data are shared. The component may have 0 or more services or references.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component xmlns:drcr="http://win.ua.ac.be/~ninggui/drcr/v1.0"
immediate="true" name="ua.mw.robot.pilot.PilotImpl">
   <implementation class="ua.mw.robot.pilot.impl.PilotImpl "/>
  <service>
     <provide interface="ua.mw.robot.pilot.Pilot"/>
     <provide
interface="ua.mw.communication.commandprotocol.Commandable"/>
  </service>
  <reference bind="setSessionManager" cardinality="1..1"
interface="ua.mw.communication.commandprotocol.SessionManager"
name="SessionManager" policy="static" unbind="unsetSessionManager"/>
  <context-specific language= "UA.PATS.Language.SRDF">
   <![CDATA[
   <Priority> 2 </Priority>
   <ExecutionTime>8000</ExecutionTime>
   <Period> 12000</Period>
   <Deadline>12000</Deadline>
]]>
  </context-specific>
</drcr:component>
```

Figure 5-2. Sample DRCom description

Service: specifies the provided service interface for the client component. It consists of a service element, which will be used to expose the provided interface into the service registry.

• *interface*- the java interface based service type. This interface should be implemented by the class referred by the implementation element.

Reference: specifies the required service interfaces provided by the other DRCom instances. It consists of a service element, which will be used to expose provided interface into service registry. A reference element has the following attributes:

• name – The name of the service reference. This name is local to the component and is used by the component to locate the service corresponding to this name.

• interface – Fully qualified name of the class name(java interface based). It is used by other components to access this component exposed service. The reference of its residing component should be able to be cast into this service interface

• cardinality – Specifies if the reference is optional and if the component implementation supports a single bound service or multiple bound services.

• (un) bind – The name of a method in the component implementation class that is used to inject/unject dependence into/from the component configuration.

As references of a component are managed by the system, one natural requirement is to let the system take control of reference management. In its meta-data, a component specifies “bind” and “unbind” methods that the system can access to inject and unlink the references for specific managed components. In Figure 5-2, two methods – “getStreamService” and “ungetStreamService” – are defined for required interface “StreamService”.

Basically, our component meta-data format follows Declarative Service component model and our component provides backward compatibility with this component model. However, this static XML description is only capable to describe structural related information. In order to support non-functional description, we extended the declarative component model with customizable non-functional description capabilities.

A context-specific element is introduced to describe non-functional requirements. This element has only one attribute – language, which designates which language is employed in describing the resource prerequisites. In this element, users can provide their own non-functional requirements with their customized languages.

Simple Resource Description Language

In order to have more description capabilities to express its non-functional requirements, our system also supports user-defined context description languages.

In our previous work [71], a simple resource description language called “Simple Task Description Format” (STDF) was introduced as a proof of concept for describing DRCom real-time task’s characteristics. Figure 5-2 shows a sample component’s description – a TV decoder implemented as a real-time task. The STDF shows that it is a real-time task with period 33300μs, execution time 8000μs, and deadline 12000μs.

5.2.1.3 Behavioural reflection

In order to achieve a coherent way to control basic component behaviour, each compatible DRCom component is required to implement the component management interface. When this component is enabled, this interface, together with the component’s properties, will be automatically registered as management service in the OSGi service registry by the system run-time.

Upon registration, each component’s management service can be discovered dynamically by using the native OSGi service tracking mechanism. As each DRCom component provides a general manipulation interface, the component management layer can be designed in a much-simplified way. Of course, this interface increases the component implementation complexity. In order to limit the implementation complexity while keeping the flexibility, this interface is designed with only minimal functions and is kept as simple as possible. The current management interface is shown in Listing 5-1 with exceptions definition omitted.

By searching the service registry, users can locate the individual component’s exposed management service and use this service to control its behaviour or get state information.

```
public interface IManagement {
        public String getID();
        public IPluginMetadata getMetadata();
        public int getstatus();
        public void setProperty(String,string);
        public list getproperties();
        public string getProperty (String);
        public void activate();
        public void deactivate();
        }
```

Listing 5-1. IManagement interface for DRCom

DSM can use this interface to get the latest property values and adjust these parameters by using getProperty and setProperty. The method getMetadata returns the meta-data attached with this component, including functional description and non-funtional requirements described in the last subsection. Here, the *activate* and *deactivate* methods are used to control the component's lifecycle during run-time (see Section 5.2.3).

5.2.2 Run-time pluggable DSM

In our pluggable and modular middleware architecture, each DSM is implemented based on the DRCom component model. This means that the DSM will need to implements management interface and provide meta-data for structural reflection. However, DSM also needs to express its context requirements, and its functional interface is needed to be clearly defined to facilitate the model fusion process. A revised Sponsor-selector pattern is also introduced to support the adaptation modules composition.

5.2.2.1 Context matching conditions

In our implementation a meta-data based approach is adopted to describe context constraints. Several works have been carried out in describing resource policy and constraints, such as Web Service Policy framework (WS-policy) [19], and eXtensible Access Control Markup Language [8] . However, such tools are designed mainly for web-services and are too cumbersome for our system. Here, a simple constraints description format similar to the approach in [59] is designed and implemented in our prototype. Rather than treating each sub-condition equally as in [59], we identify that different constraints normally have different impacts towards DSM. A new argument – the impact factor – is introduced to express this difference.

CMC with Impact Factor: Each DSM's context constraints are written in the following format:

Context Factor, *Type*, *Operator*, *Value*, and *Impact Factor*;

separated by commas. Context Factor denotes the name of context factors, for instance, Battery_ Percentage denotes how much battery is required, Time preference denotes when it is appropriate to use this DSM, etc. Here, Type specifies the value type of context values on how to map the string-based value into typed values. Currently, basic java types, such as string, integer, double, float, Boolean, are supported. Operator here is used to specify the compare operator; here three operations are supported: ">" denotes bigger than, "<" denotes less than and "==" means equal. The Impact Factor is a normalized weight that expresses the influence of these context sub-conditions. If Impact Factor of sub-condition i is expressed as IF^i , the total number of sub-conditions is N, then

$$\sum_{i=1}^{N} IF^i = 1$$

The following example describes a hypothetical TV optimization DSM. Its CMC contains three context constraints: user's preference, battery percentage and time.

#Context requirements

User_Preference, string, == , TV_performance, 0.5;
Battery_percent, double, > , 50.5 , 0.3 ;
Time , hour , > , 21, 0.2

This example shows this DSM's has three sub-conditions. Among these sub-conditions: "*User_preference*" has the biggest impact (0.5) while battery status has medium impact (0.3) and Time of day has the minimum impact (0.2). These constraints with be examined during run-time by DSM Manager (see Section 5.3.3.2).

5.2.2.2 DSM functional interface

As different DSM might have totally different adaptation strategies and considerations, in order to effectively reuse DSM across different domains, they must implement the same adaptation interface to allow interoperability. Each DSM calculates its adaptation plan and exposes its plan to adaptation modules through the following interface:

```
public interface IDSMResolver {
  String getDSMID();
  IDSMMetadata getDSMMetadata();
  AdaptationPlan  resolveAdaptationPlan(SystemContext sc,
  ContextChangeEvent cev);
}
```

Listing 5-2. IDSMResolver Interface

In this interface, the getDSMMetadata method is to get this DSM's meta-data. It helps DSM Manager to get the meta-data related to the DSM which includes its functional dependence, requirements for Event Monitor as well as Context Matching Conditions. The resolveAdaptationPlan method is the key method for the DSM to generate adaptation plan. It has two major parameters: the SystemContext allows DSM to get the current snapshot of system status, for instance, the list of current enabled components, the list of disabled components, etc. This information is necessary for DSM to make correct and

accurate adaptation plans. The second parameter – ContextChangeEvent – denotes the just happened change. This parameter informs the DSM with updated change information. The returned class AdaptationPlan keeps the DSM resolver's calculated strategy. In current prototype, two different lists are supported in the AdaptationPlan class. They are enabledComponentList and changePropertyList. The enabledComponentList contains the list of all resolved enabled components. All installed components not in this list are tagged with "disabled". changePropertyList returns a list of Properties and their new intended value. System run-time inquires each enabled DSM to get adaptation plans and sends results to the *Model Fusion* module.

As can be seen from the interface design, compared to the *Structural Modeller*, DSM can make more fine-grained adaptation actions (setProperty) as *Structural modeller* can only perform lifecycle management (see Section 5.3.2). It is worth remarking here how, in our middleware design, no adaptation actions will be performed inside either DSM modeller or *Structural modeller*. Compared to other approaches, such as the Self-Healing approach [39] [138], which mix the adaptation actions with the reasoning logics, our approach makes clear separation between adaptation planning and effecting. This design helps the system more easily identify and resolve possible conflicts.

In our system, multiple DSM with different domain-specific adaptation knowledge can be installed simultaneously. All DSM implement the same IDSMResolver interface with tagged CMCs describing their application domains. By switching the set of used *DSM Modeller*s**,** system architecture model as well as adaptation behaviour can be easily altered, which could be beneficial in matching different environmental conditions. DSM plug-ins are characterized by additional dependence towards their executing environment, including the DSM functional dependence as well as context matching dependence. This extra-dependence calls for more refined lifecycle state management for DSM plug-ins with respect to native OSGi component mode. Except for the native lifecycle states, these extra-lifecycle states will be decided by the DSM manager.

5.2.3 DSM plug-in lifecycle

As discussed in Chapter 4, a DSM can only be used when its targeted context matches system current context. That means its lifecycle must be managed by middleware rather than the DSM plug-ins themselves. In order to support this feature, the OSGi native component model is extended. DRCom provides support for designing and deploying software components, with two levels of component model abstraction. The fist level of abstraction is the Java package-based dependence support (packaged in so called bundles) defined in the OSGi core specification. This abstraction is directly reused in the DRCom component model as each DRCom is implemented as an OSGi bundle. In the higher level of lifecycle management, DSM's other dependence, for instance, its context matching conditions, functional interfaces dependence requirements. These levels of abstraction are built on top of our DRCom component model. Firstly, the following section introduces the native OSGi bundle lifecycle management.

5.2.3.1 OSGi native lifecycle management

OSGi provides a native component management lifecycle management. As suggested in its while paper: “it allows for run-time installation, updating, resolution, activation and deactivation of components without the need to restart the system”. Figure 5-3 shows the OSGi component lifecycle (in the first layer of component model abstraction). As we can see from the above-mentioned figure, even in the simplest of models, an OSGi component contains six states and multiple transitions. The lifecycle management provides dynamic supports that are not necessary issued or controlled by an application. For instance, in the native component model, the OSGi run-time provides the support of Java library dependency check to ensure that the dynamic available components’ all required java library are already installed. In the OSGi specification, such lifecycle operations are fully protected with the OSGi security subsystem.

As we can see from the Figure 5-3, many of the transitions between the states are performed automatically by the system run-time. For instance, whenever a bundle is successfully installed into the system, the OSGi run-time will then check its functional dependence to see whether all bundle required java classes are available. When this is true, this bundle’s state will be automatically set as *Resolved*. This state indicates that the bundle is ready to be started.

This automatic dependence management greatly simplifies the complexity of bundle management. The Bundle dependence management is one of the key reasons, it is argued, which contribute to OSGi’s popularity.

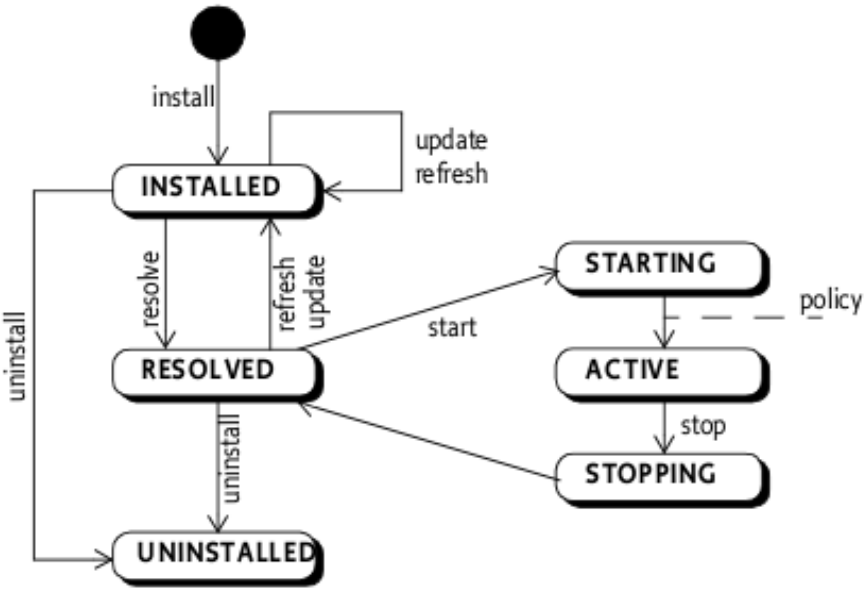


Figure 5-3. Life cycle of OSGi bundle, from OSGi specification[114]

5.2.3.2 DSM Lifecycle management

While OSGi lifecycle management is appropriate to resolving the static library based dependence between bundles, it is not sufficient for the contextual lifecycle management.

The source of this problem is that the deployed DSM plug-ins have more complex dependences towards other modules as well as their executing environments – such as context-specific dependence towards its working environments, rather than the package-based bundle dependence. Those dependences cannot be explicitly expressed in terms of Java library dependence, as they change dynamically according to the changing context. This changing context can be triggered by many factors, such as changing system resources, user's preference or a new component being installed.

In order to express the additional dependences, the DSM plug-in is extended from native OSGi component sub-state. Three DSM specific sub-states a – C_Deployed, C_Satisfied and C_Active – are introduced. These sub-states extend the Active state of OSGi native component model and are directly controlled by the DSM Manager (See Section 5.3.3).

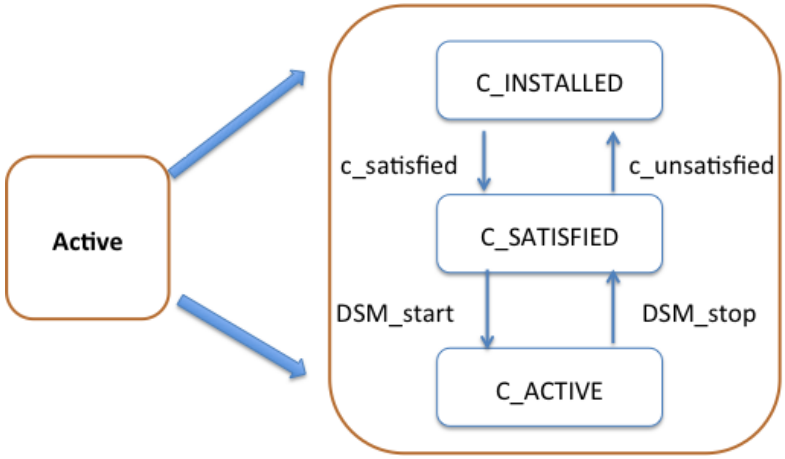


Figure 5-4. Extended OSGi component lifecycle(inspired from Paspallis thesis[121])

As defined in the OSGi specification for native bundle lifecycle management, when a bundle state is in ACTIVE, all java package-based dependences of this bundle are fulfilled. However, this does not include *context* dependencies, which might or might not be satisfied. As described in Section 5.2.2.1, a normalized context matching degree formula is used to measure the matching degree of one DSM with specific context environments. DSM manager monitors and calculates these constraints for each installed DSM plugins by evaluating their attached meta-data. When the calculated CMD is bigger than 0.5, we say that this DSM is context satisfied. Then, the state of this DSM is set to C_SATISFIED by DSM Manager. After this, the activate method will be called to activate this DSM (denoted as DSM-start in the picture) and the DSM will be set to the C_ACTIVE state. As changes of system context are dynamic, when a DSM's context matching degree fails to satisfy the matching threshold, a DSM component can transit back from C_ACTIVE to C_INSTALLED state by calling the *deactivate* method defined in the component management interface. When a bundle is stopped, this component will transit to C_ INSTALLED state. After this process, its lifecycle management will be handed over to OSGi framework and continue its normal transition flow.

The full specification of a DSMResolver interface does not contain lifecycle management methods. That is because IManagement – the DRCom component basic interface – already defined such lifecycle management methods. The IDSMResolver interface only focuses on DSM function-related methods.

It is worth to point out that there are many OSGi-based approaches that provided similar but different lifecycle managements. Examples include but not limited to: the Gravity project [36], declarative service[119], Context Sensor/Actuator in Music project[121], OSGi + MHP project [128, 129] and our previous work on real-time OSGi etc. That is because these approaches are extended from OSGi native component model (Bundle), and the only state that can be extended is the "ACTVIE" state, when the OSGi bundle is extended.

5.3 System key modules

In our system, different modules are designed to deal with requirements from different system aspects. Here we present several key modules that are used to achieve multi-DSM based modelling and adaptation.

5.3.1 System basic run-time

System Runtime Environment provides the basic middleware services to enable the proposed multi-domain context-aware dynamic service composition. Here, System basic run-time is the module which implements basic management functionalities: (1) install, discover, execute and uninstall components, as well as manage the component instance registry; (2) manage the service registry and monitor the service changes to provide run-time service component support; (3) parse the meta-data attached with installed component; (4) provide support to manage/inject the references between components. By invoking specific methods declared in components meta-data, the system can effectively enforce the bindings between the components; (5) on-line verification scheme is implemented here to break possible adaptation loops. Theoretically, all of a component's configurations state – including lifecycle state changes as well as property configuration changes – need to be tracked. Due to the implementation complexity as well as the high overhead to track all these configurations changes, in our current implementation, only component's basic lifecycle states are tracked for adaptation loop detection. Due to performance consideration, the on-line verification mechanism on default is disabled.

5.3.2 Structural modeller

As the application is constructed, configured and reconstructed during system run-time, how to derive the functional and structural dependency among components becomes one of the key problems in run-time component composition. The Structural Modeller consists of several processes, the most important of which is functional dependence compatibility check – which mark a component as "structure-satisfied" or "structure-unsatisfied".

"Structure-satisfied" are those components which either have no functional dependency or their dependencies are already provided by other activated plug-ins. A component can only be activated when it is "structure-satisfied". This guarantees that a component can only be initialized when all its functional dependences are satisfied. In our current prototype, two different component models are supported: one is the declarative real-time component model [12], the other one is the enhanced Declarative Service component model proposed in OSGi Version 4 [119].

The Structural Modeller provides the following interface:

```
public interface StructuralModelerResolver {
    List<ComponentConfiguration> resolveSatisfied
                                    (List ManagedComponentConfigurations); }
```

Listing 5-3: Structural Model Functional Interface

The above interface takes a list containing all managed component configurations. Each time the *Structural Modeller* is inquired, it will check whether all the required interfaces from one managed component have their corresponding service provided. After this process, it returns a list of all the components that satisfied the functional constraints. All these components will be tagged by Structural Modeller as enabled. Then, the disabled component configurations will be given by the complement set of the satisfied component list with respect to all ManagedComponentConfigurations. As we can see from this interface, two adaptation actions – enable and disable, can be performed by Structural Modeller.

The *Structural Modeller* manages component functional dependence by changing component lifecycle state. It is one of the key core modules existed in many run-time adaptation systems, such as Servicebinder [75] or Perimorph [86]. However, unlike these approaches, *Structural Modeller* here is explicitly separated from the other Domain-specific adaptation strategies. The detailed introduction for this module is listed in Appendix A.

5.3.3 DSM manager

The *DSM manager* is designed to support the dynamicity of the DSM for adding, removing, selecting, accessing and resolving DSM modules. It can intelligently and autonomously activate and deactivate DSM plug-ins according to system's current context. Also, by monitoring metadata from DSM —which encode their required Sensor/Actuator types, the middleware activates corresponding Sensors/Actuators as needed to optimize system resource consumption (e.g., CPU usage, memory usage, battery drain, etc.). When to activate a DSM depends on the similarity of the DSM's application domain with system current context, this process is denoted as CMC calculation.

5.3.3.1 DSM manager business logics

Figure 5-5 shows one example of business logic of *DSM Manager*. This *DSM Manager* depends on four major functional interfaces. One interface is the IDSMRepository which is used to cache parsed DSM meta-objects to enhance performance. It also needs a context

matching algorithm to match all installed DSM against current context. In this thesis, the normalized context matching conditions algorithm is used. The current context information is retrieved via the *Context Manager*. It provides simple context information – in current version, this is in the simple format of name-value pairs. In the following section, the normalized context matching rate (NCMD) calculation and the DSM resolving process will be discussed in details.

5.3.3.2 CMC calculation

DSM Manager locates "best-fitting" domain modellers for current context. System context knowledge comes from three different sources: through system basic metrics, through a context-aware discovery [44], or through inference [66]. In our current implementation, DSM Manager gets system context knowledge by periodically reading a list of current context factors, such as CPU, memory, disk space etc., from the *Context Manager*. As this thesis focus on providing adaptation management, it does not make detailed design on how the context information is generated. It assumes these context knowledge are ready available from the *Context Manager*, no matter whether they are basic system metrics or semantic information reasoned from the ontology knowledge.

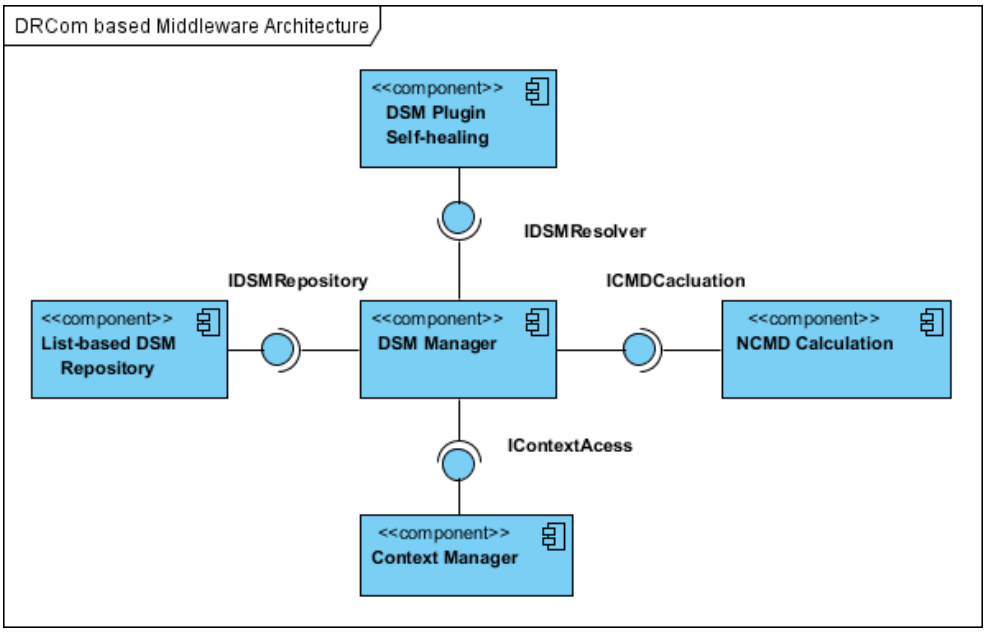


Figure 5-5. DRCom based DSM Manager

In our current implementation, the selection process is based on calculation results of the normalized context matching degree (NCMD). NCMD represents how similar the current context is to the context constraints specified in the modeller's meta-data. A NCMD is calculated by summing up all context matching impact degrees, which in turn are computed by multiplying condition values with their corresponding impact factor. Since CMC consists of multiple sub-conditions, NCMD is calculated as the ratio of the sub-conditions in CMC.

For a DSM with N sub-conditions, for any sub-condition with context resource cr^i, operator for evaluation $operator^i$ and the value of required context resource value $value^i$ and its impact factor ip^i. *eval* is the function that returns 1 if the value of

resource cr in the execution context e is among the value the values specified by the operator $operator^i$ to the targeted value value^i. If else, it returns 0. The context matching degree can be calculated via following formula:

$$\text{CMC}= \sum_{i=1}^{N} \text{eval}\left(\left(\text{cr}^i, \text{operator}^i, \text{value}^i\right), \text{e}\right) * \text{ip}^i$$

For the following CMC constraints with three sub-conditions:

User Preference, string, == , TV_performance, 0.5;
Battery, double, > , 50.5 , 0.3 ;
Time , hour , > , 21, 0.2

If system's current context is "Battery = 61%", "time = 7:00," "optimization=TV performance", the NCMD calculated is 1*0.3 + 1 *0.5+ 0*0.2 = 0.8.

The NCMD is a rather simple matching mechanism. For the use case described in the motivational example, it works well. Of course, it is by no means the only rating strategy that our framework allows. Other rating algorithm, such as semantic graph matching scheme [94, 111] can be used onto the DSM selecting process. They can be used to calculate the semantic similarity between a DSM context requirements and current context. At the same time, other existing context-aware technologies, e.g., Context toolkit [7] [44], or SOCAM [66], can be utilized to develop more advanced context retrieval systems by making use of semantic reasoning logic to infer additional context knowledge. For example, if the moving sensors report no movement for a period of time and the door sensor reports that the sleep room is closed, it can derive a new context "the user is sleeping" from current context knowledge [66]. Obviously such enhanced context knowledge can increase the accuracy on how the DSM is selected. Detailed discussion on this is out of the scope of this thesis.

Benefited from our service-oriented architecture, third party developers can install their own implementation of CMD calculation unit by simply implementing the ICMDCalulation interface to return a DSM's matching degree. As a consequence, the actual implementation of matching algorithm can be easily altered.

When the context-matching rate is calculated, the *DSM Manager* will use this information to choose those contextually applicable DSM and activates those DSM. Besides, those DSM that are no longer applicable should also be deactivated. This process is one of the key functions of the *DSM Manager* – referred as DSM resolution mechanism.

5.3.3.3 Resolution mechanism

As discussed in the previous chapter, for different contexts and/or different system configuration, not all the install DSMs can be activated. Only those DSM that satisfy all the constraints can be used. The resolution mechanism is designed to check whether a component is "resolved" or "unresolved".

Resolved DSM plug-ins must satisfy two different constraints, functional dependence and CMC matching degree. The functional resolution is achieved by *Structural modeller* and this requirement will be checked again any DRCom components installed in the system before they are ready to be activated. For the functional dependence, resolved components are those plug-ins which either have no functional dependence such as simple *Sensor* and *Basic Actuator, or* their dependencies are already satisfied by other, already activated plug-ins (or services provided by other non-DRCom components such as native OSGi bundles).

Algorithm 5-1. The algorithm used by the DSM selecting mechanism (pseudo-code)

Basic data-structures

[all DSM plug-ins] - set containing all installed DSM plug-ins

[DSMSelected] - subset of the [all DSM plug-ins]; contains only resolved & selected plug-ins

Triggered by Changes to the [Context Values] set or set of [all DSM plug-ins]

Algorithm

```
1.    # ensure that DSM plug-ins with NCMD above the threshold are included in
      [DSMSelected]
2.          for all p in [all plug-ins] do
3.            if calculateNCMD(p) >= Minimal_Matching_Degree and ({p} ⊆ ([all plug-ins] -
      [DSMSelected]))
4.              [DSMSelected] ⇐ [DSMSelected] ∪ {p}
5.    # ensure that DSM plug-ins with NCMD below the threshold are removed from
      [DSMSelected]
6.          else if calucuateNCMD(p) < Minimal_Matching_Degree
7.            [DSMSelected] ⇐ [DSMSelected] - {p}
8.         end if
9.        end for
```

When a DSM plug-in is classified by the DSM manager as "resolved, it then can be activated and provides adaptation service to the middleware. From the task description, it is easy to define an algorithm to provide the context resolution mechanism which is depicted in Algorithm 5-1.

In Algorithm 5-1, two data-structures are defined. Firstly, the set [all DSM plug-ins] contains all the installed DSM plug-ins. When an event is raised and necessary adaptation action is necessary, this algorithm will be executed. Events can be changes from event reasoning modules as well as changes in the set of [all DSM plug-ins], for instance, adding/removing a DSM modules. The other data structure – [DSMSelected], is a subset of the [all DSM plug-ins] set. When a DSM is contained in this set, it will be selected by the DSM manager and will be used in the adaptation process.

As can be seen from Algorithm 5-1, this algorithm has two major sections: Firstly, as system context will always change, it must ensure that all context-matching DSM will be included in the [DSMSelected] set. This check is achieved by checking all unselected DSM's current context matching degree with respect to the system current context condition. If the un-selected DSM is found to be NCND-satisfied, it will become *selected*

and be added to the [DSMSelected] set. The second section is to ensure that all newly unsatisfied DSM will be removed from the [DSMSelected]. If it is found not to be NCND-satisfied, it will be deleted from the [DSMSelected] set. The complexity of this algorithm is O(N), N being the number of installed DSM. Of course, each DSM might different degree of complexity which will not be discussed here.

A basic threshold, Minimal_Matching_Degree, is arbitrary defined to select the context matching DSM – in current prototype 0.5 is assigned. If ratings of all candidate DSM are below that threshold, only the *Structural Modeller* will be used to guide adaptation. In this case, only application architecture will be maintained.

The DSM manager periodically subscribes to context monitoring services registered in the service registry, provided by context data sensors and event monitors, and it also has a functional requirement with 0 to N IDSMResolver services (in declarative service term, with cardinality 0..n). Accessing to the context management service allows the DSM management system aware of the changes of dynamic context. The functional requirement for resolvers makes it keeping tracks of available DSM modules. With this information, it is thus able to intelligently activate and deactivate the DSM plug-ins that meets the current environment. The DSM with matching degree higher than the threshold will be marked as selected by DSM Manager.

5.3.4 Conflicts detection & resolution

As described in Section 4.5.3, system global adaptation model is built by collectively using the Structural modeller and the run-time selected DSM. As these modellers capture different features of a system, they may construct adaptation plans that conflict with each other. Thus, the process of combining these models and resolving possible conflicts becomes vital in order to guarantee system correctness. Rather than using the utility function based conflict resolution solution which ignores the semantics of each adaptation actions, in our approach, an actuator model is build. This model can express the semantic relationships between different actuators. This semantic approach builds a foundation, it is argued, for more accurate adaptation resolution. Of course, the approach described here is reactive rather than proactive because we assume, which is rather likely for the composed adaptation module, it is very hard to build an efficient proactive resolution mechanism.

5.3.4.1 Actuator model interface

In order to detect possible conflicts between adaptation plans, it is important to allow system to express those relationships between actuators. The IActuatorModel interface is defined in Listing 5-4 to identify those relationships during run-time.

In this interface, several methods are provided. The getAvailableActuators returns all possible actuators that the ActuatorModel supports. While the method – getAvailableRelationships returns all possible relationships in set. haveRelationship is used to identify whether two actions has relationship or not. getRelationships method returns all

the relationship between two different actuators. The method – getRelated, returns all related actuators of a actuator designate.

```
public interface IActuatorModel
{
   public String getDesp();
   public Set getAvailableActuators();
   public Set getAvailableRelationships();
   public boolean haveRelationship( final IActuator source,
                          final IActuator target,
                          final IRelationship relationship);
   public Set getRelationships (final IActuator source,
                                final IActuator target);
   public Set getRelated(final IActuator actuator);
   }
```

Listing 5-4. Interface definition of IActuatorModel

5.3.4.2 Hardcoded approach for actuators and their relationships

A set of fusion rules are needed to resolve the conflicts between adaptation plans. These rules highly depend on the actions that can be taken by different modellers – which are also one of the main reasons we put hard restrictions on the adaptation actions each model can perform. In our prototype, as only three actions are supported:

Basic Adaptation Actuators:

- Enable
  This Actuator will invoke the targeted component‘s activate method and set component state to “enabled”.
- Disable
  This Actuator will invoke the targeted component‘s deactivate method and set component state to “disabled”.
- SetProperty
  This Actuator will invoke the targeted component‘s setProperty method by using its management interface and set component property to new values

Relationship among three basic Actuators

- System cannot set a component both in “enabled” and “disabled” state
- SetProperty will not change a component lifecycle state. That is, for instance, this action cannot change component from enable to disable
- A component’s property cannot be set to different values at the same time

According to these relationships, a set of conflict detection rules are built. In current prototype, the relationships between different Actuators are hardcoded as we only have three major types of actuators. As our middleware supports user-defined Actuators, in order to support more flexible relationship expressions, we are currently planning to use

ontology-based description languages such as Ontology Web Language [152] to describe and automatically reason the relationships between available actuators.

5.3.4.3 Conflicts detection

According to the relationships described in the last subsection, conflicts can easily be identified by checking whether for instance one of such relationships is violated:

- Structure Modeller → disable , DSM → enable
- DSM → enable, another DSM → disable
- Different DSM set component x, property y with different values

When one of these rules is violated, a conflict will be detected. System run-time will log this event and send this event to the *reflective adaptation service* (described in Section 5.3.4.5).

5.3.4.4 Conflicts resolution

While a conflict is detected, the conflict resolution modules will be called to come out a conflict-free adaptation plan from multiple adaptation plans. This conflict resolution process needs to take multiple factors into account: the DSM types, the adaptation actions characteristics of each adaptation plans and current context.

The following list of rules specifies our current preliminary solution towards conflict resolution:

1. A component can only be enabled when it satisfies all modellers' constraints and is marked as "enabled".
2. A component will be disabled if it is marked as "disabled" by any of DSM or by the Structural Modeller.
3. If a component is disabled, change property actions will be dropped.
4. If two or more DSM plug-ins tries to set the same property of one component, only the value from DSM with highest NCMD will be used.
5. System always firstly performs enable actions, then disable actions, and changes to properties are performed only after these lifecycle management actions.

The just defined rule set aims at resolving the conflicts while maintaining software structure. Most of its rules are natural requirements, e.g. a component can only be enabled when it satisfies all modellers' constraints. However, some rules, such as the order of adaptation, are defined arbitrarily. Although this set works fine in our scenarios and simulations, it is clear that the more DSM and the more intertwined their concerns, the more difficult it is to merge those concerns effectively and correctly. This should not come as a surprise, as it is a situation common to other approaches where multiple concerns need to be resolved into one coherent strategy or application [53, 104].

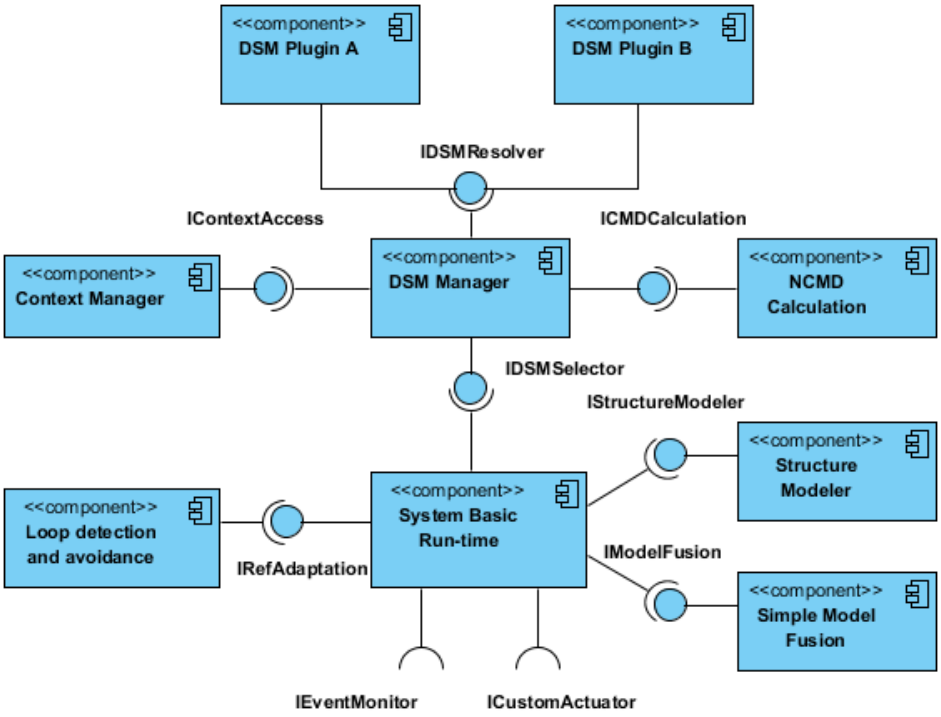


Figure 5-6. Service-component based middleware architecture

As an example, in AOP-based programming [89], the issue of detecting and resolving conflicts among aspects is now becoming a hot research topic. Some of these new researches, such as W.J. Lagaisse's approach of Semantic interference [28] can be largely reused in our system. Compared to AOP based approach in which aspects may be attached virtually to any position of the original source code [133] and each aspects can implement their customized actions, our work clearly defines what DSM can do (that is, adaptation interface and action sets), and when they can do it (that is, only during adaptation process). Therefore we believe that conflicts could be identified and resolved in a much more easy and cost-effective way. One of our ongoing works is to add more descriptive power to DSM and use semantic interference logic to build more accurate and flexible model fusion rule sets.

5.3.4.5 Reflective adaptation service

In the model fusion module, a set of rules are provided to explicitly fuse multiple modeller solutions into one global adaptation modeller; however, there is no guarantee that the fused modeller will perform as intended. Although our system provides a basic on-line verification algorithm, its major function is to detect adaptation loops and bring the system out of infinite adaptation loops. How to evaluate the effectiveness and efficiency of the fused global modeller is still one of the major issues in our system design.

As the effectiveness and efficiency can only be evaluated according to system current context optimization goal, it is very unlikely to have a general policy to deal with so diverse context environments. Rather than providing solution for specific contexts, our software system provides services to exposing a snapshot of the system internal state. For instance, the list of enabled components, currently used DSM, and the adaptation actions

proposed by each modeller as well as a trace of the adaptation actions being executed can be exposed and e.g. logged for future analyses. Third-party analysis programs can then be used to validate adaptation decisions taken by the various modellers. The system will also send asynchronous events to the external listeners when adaptation actions are taken. Programmers can selectively listen to those events that might be important to them and get adaptation actions accordingly.

This information can help system users to validate various modellers including *DSM Manager*. For instance, a Ping-Pong effect among several DSM in a short time might indicate an ill-designed *DSM Manager*. Formal models and specifications such as [85] can also be plugged into the system to understand different behaviour or ensure correct functionality of selected DSM and fusion rules.

5.4 **Middleware architecture**

The middleware architecture is designed with service component model, constructed by two central sub-components – the *Basic Run-time* and *DSM manager*.

These two components are supported by multiple secondary ones, including the DSM repository and CMD calculation, the Actuator Model, the Model Fusion. The internal structure of the service-component based middleware is illustrated in Figure 5-6.

5.4.1 Middleware core functionality

The DSM manager and System basic Run-time is the two main components that provides the key functionality of this middleware. Figure 5-6 shows the key structure and interface of each component in the middleware:

- The IDSMResolver interface is the service interface that all the DSM Plugin must implement. Its major goal is to facilitate the extension of adaptation modules. For this reason, it provides access to DSM meta-data as well as DSM functional methods to access its calculated adaptation plan.
- The IDSMRepository is the service that provides the service for caching and storing DSM. It also provides searching service for installed DSM. The DSM is represented as meta-objects which are serialized for store and query.
- The ICMDComputation is the services that to calculate an installed DSM's context matching degrees with respect to current context. It accepts DSM's attached context matching conditions and returns a double value in [0,1].
- The IActuatorModel is a service that provides the relationships between different actuators. It is used to specify whether two adaptation actions are compatible or not during run-time. In current prototype, the default implementation is realized using hard-coded logics; more complex, ontology based implementations are also possible.

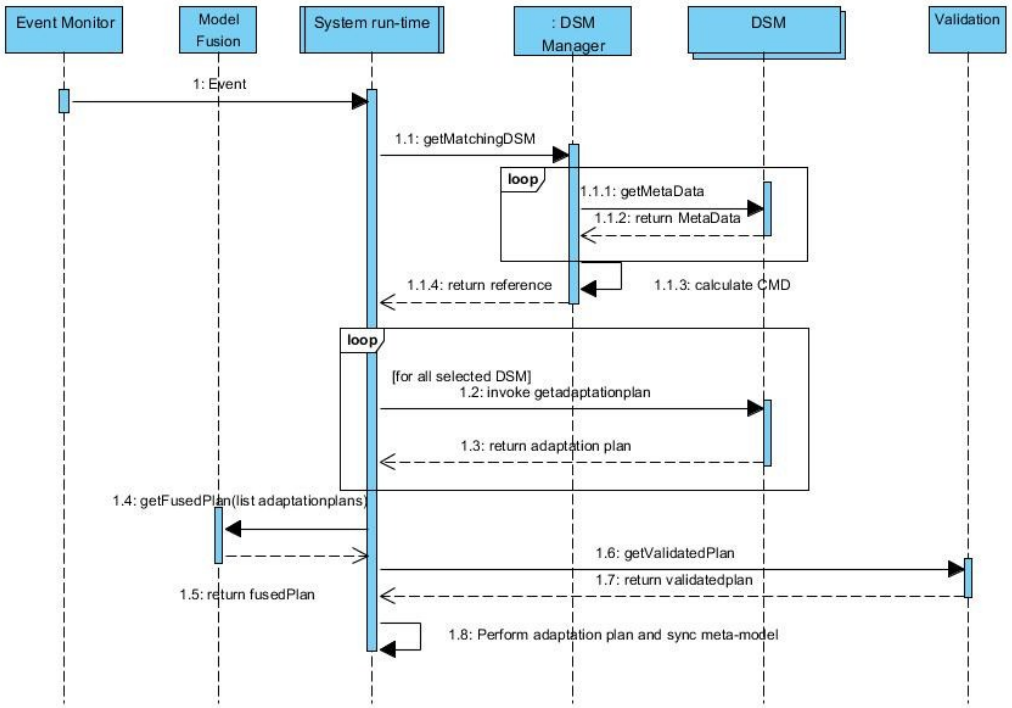


Figure 5-7 Sequence diagram for DSM selection

- The IModelFusion is a service that provides the conflict-free adaptation plan from a set of adaptation plans. In current version, a pre-defined set of rules are employed to generate adaptation plan. One of our possible future research directions, focusing on user interaction intensive environments, would be to use decision tree techniques to automatically refine rules after user's feedback.

5.4.2 Sponsor-selector pattern

As several DSM modellers may co-exist in a specific time, only those which matches current context will be selected. DSM Manager and DSM are implemented through a revised Sponsor-Selector pattern [7], as the DSM Manager selects the best contextual match DSM from a set of candidates that changes dynamically. By separating three kinds of responsibilities: 1) knowing when a modeller is useful, 2) selecting among different modellers, and 3) generating an adaptation plan, our software platform, this system integrates different modellers and various knowledge into the system during the run-time, in an extensive way while being transparent to system managed applications.

As described in Figure 5-7, the selection process works as follows:

The system run-time reaches a point at which certain adaptation needs to be taken and is notified by the *Event Reasoner*. It then asks the DSM manager for a set of contextual matching DSM. The DSM manager checks the meta-data from all registered domain-specific modellers (sponsors). It firstly checks whether the change event is among corresponding DSM interested events list. If yes, it rates these modellers for applicability in the current context. The modeller(s) which is appropriate for current context will be selected and its/their reference(s) will be returned by DSM manager.

In this process, only the meta-data is checked for event and context matching. This design enables a DSM to be lazy initialized to reduce system resource consumption – initialized only when it is required, while still expose most of its characteristics and requirements. Then, system run-time begins working with this modeller for domain-specific adaptation plan.

5.4.3 Architectural reconfigurability

In addition to these core functions provided by DSM manager and system basic run-time, this middleware architecture is also designed to be extendable what concerns the actuator model and context matching calculations. Alternative realization can be provided to optimize the functional implementations. For instance, we can design an Ontology-based Actuator model or use a database backed DSM repository. As those modules are designed under service component model – they can be easily installed and exchanged during run-time. Figure 5-8 illustrates two possible variants.

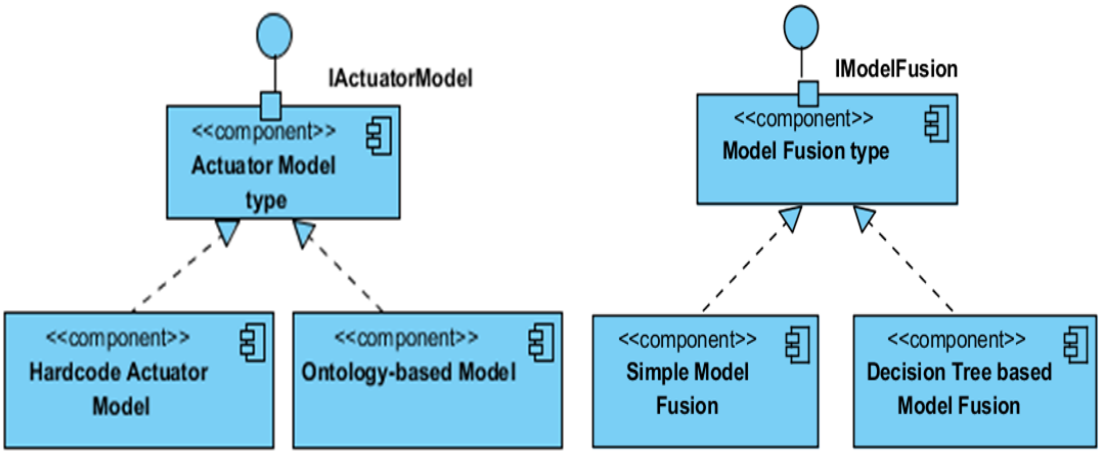


Figure 5-8. System variability of the modular middleware

In the left part of Figure 5-8, two variants of the IActuatorModel service are provided by Hardcoded Actuator Model and Ontology-based Model. In the right part of that picture, two implementations of IModelFusion service are demonstrated with simple model fusion as well as Decision tree based fusion rules. By selecting an appropriate service provider for those services, the middleware realization can better match the requirements of the deployment environments. This approach also facilitates the evolution of middleware structure to provide better implementation to the system.

For each component showed in Figure 5-8, Structure Modeller together with system run-time are used to automatically manage the dependence between the installed component instances. However, although our system allows the run-time adding/replacing of these service providers, it is not intended to do so. Our middleware only provides the DSM dynamicity solution, thus allowing the addition and removal of

DSM at run-time. Other modules, such as model fusion services, are not intended to be replaced during

Algorithm 5-2. Use best quality components (pseudo-code)

Requires: Installed components' CPU usage information

Ensure: always use the component with highest TV Quality and Allocate enough CPU time to the TV application

```
Create a new empty adaptation plan adapt_plan
for all cmp in SatisfiedComponents do
    if cmp is for TV application and  (not visited)
      Find all component (similarset )  with same functional contract
        Set visited=true for (similarset)
        Select one with highest quality attributes cp_quality_high
    remove all components except cp_quality_high from enabled component list(ecl)
    end if
end for
//check for performance violation
If Decoder. getProperty(overrun)  increase > Max_threshhold
for all cmp in SatisfiedComponents do
    if component not from TV application
        cmp.getProperty(CPUconsumption) from cmp attributes
        record the cmp with highest CPU usage.
    end if
end for
remove cmp from enabled component list(ecl)
end if
set ecl back to adapt_plan
return adapt_plan
```

run-time as special cares are needed to make sure system's correctness during this run-time replacement of these components which is out of the scope of this paper.

5.5 **Case studies**

In this section, we introduce how our middleware architecture supports DSM plug-ins and show how in some cases it is possible to resolve conflicting adaptation behaviours. Due to vast possible scenarios, we shall not go through all possible cases. Three different modellers – Structural Modeller, DSM for TV optimization (DSM TV) as well as DSM for Self-healing – will be used to guide software system composition as well as adaptation. Firstly, we will discuss our DSM for TV and DSM for self-healing.

5.5.1 Adaptation strategy

As we discussed in the motivational example in Chapter 4, applications, such as a TV application and a Recording application, will interfere with each other while competing for

system resources – in this case, the TV and Recording applications could not run simultaneously with maximal quality due to lack of system resources.

5.5.1.1 DSM TV

In order to maximize user's TV watching experience, two main policies are implemented in this modeller – one strategy is to always use the component with higher quality to build TV application; another one is to always allocate enough resources to the TV application by disabling other components. As we can see, the adaptation strategy is implemented as Algorithm 5-2. The first part of Algorithm 5-2 is to select components with highest quality to build TV application. When DSM TV finds that *Decoder's* execution task overruns increasing rate exceed certain threshold, this DSM will disable the most CPU-intensive components (save the TV applications) until TV decoding task overruns stop to increase.

5.5.1.2 DSM Self-healing

Here, we introduce a DSM for Self-healing. The management interface of DRCom is used to get all property values to date. This can also help to check whether a component works correctly. If it raised exceptions, we shall deduce that component might have problems. Then, the repairing process begins:

As in our current prototype, only three types of actions are supported (enable, disable and change property), these is no actuator that can directly repair a faulty component instance. However, this action can be built up by compositing these three actions in the following 3 steps:

1. Detect whether there is an error; if yes, add faulty component (denote as faulty) to *faultyComponentList*. At the same time, remove faulty component from *SatisfiedComponentList*. This step will disable the faulty component and remove its resources including its registered service interface in service registry.

2. After a component is disabled, the component's provided service is disabled. This event will trigger another round of adaptation. If DSM's *faultyComponent* list is not empty, add *faultyComponent* to the *enabledComponent* List. After this step, Component_A should be reinitialized and the component A will be repaired.

3. After component A is initialized and registers its provided services, this event will trigger another round of adaptation. DSM Self-healing will set the properties of repaired components to the newly initialized component instance.

As we can see, the adaptation process is built in a logic order and takes 3 steps to repair a faulty component. The main reason of this design is due to the limited adaptation actions that our system supports. Other implementation, for instance, performing adaptation inside the DSM's reasoning logic, might work more efficiently. However, such an approach would violate our design principles to separate the actuator from resolving process. Bypassing the fusion process might result in unchecked adaptation possibly leading to unexpected consequences. Another solution is to provide more complex actuators, for instance, turning "repairing component" into an atomic adaptation action. However, in our

experience, the richer and the more complex the basic adaptation action set, the more difficult will be the formulation of effective fusion rules.

5.5.2 Step-wise application construction

As we already described in Section 4.6.1, adaptation process is built step-wise. Here, we will demonstrate how TV application is constructed with multiple modellers co-leading the step-wise adaptation process. As we can see from Figure 5-9, two decoder components with different quality metrics are available. This ambiguity cannot be solved by Structural Modeller alone as it does not only depend on structural constraints.

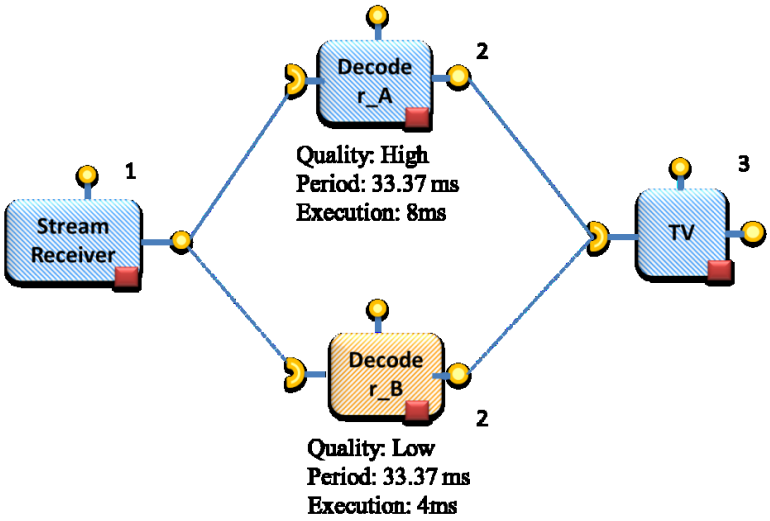


Figure 5-9. TV application Construction

There are three modellers involved in TV application process. However, as DSM Self-Healing will only join the adaptation plan when an error is detected, for simplicity of discussion here we only discuss the modellers that will affect the construction process – *Structural Modeller* and *DSM TV*.

1. *Stream Receiver* will be enabled as it is "structure-satisfied", and DSM TV will return the same result. The enabled component set will then be {Stream Receiver}. Its service interface will be registered automatically by our system run-time.

2. After the service of Stream Receiver is registered, Decoder_A and Decoder_B will be assessed as structure-satisfied by Structural Modeller. The enabled component set of structural modeller then becomes {Stream_Receiver, Decoder_A , Decoder_B}. According to TV optimization rule, only Decoder_A will be enabled as it provides higher quality. So the fused enabled component list is { Stream_Receiver, Decoder_A}. As Stream Receiver is already enabled, only Decoder_A will be enabled.

3. *TV Rendering* component will then be enabled, after the service interface of *Decoder_A* gets registered. TV application will then be successfully constructed.

5.5.3 TV optimization adaptation

When these two applications run simultaneously, the Event Monitor will start noticing that decoder module's overruns are increasing. Then it will notify the Modelling Layer to initiate an adaptation round. In this case, the Structural Modeller will not find any structural violation as both applications are functionally well constructed. Thus the satisfied functional model is sent to the DSM modellers.

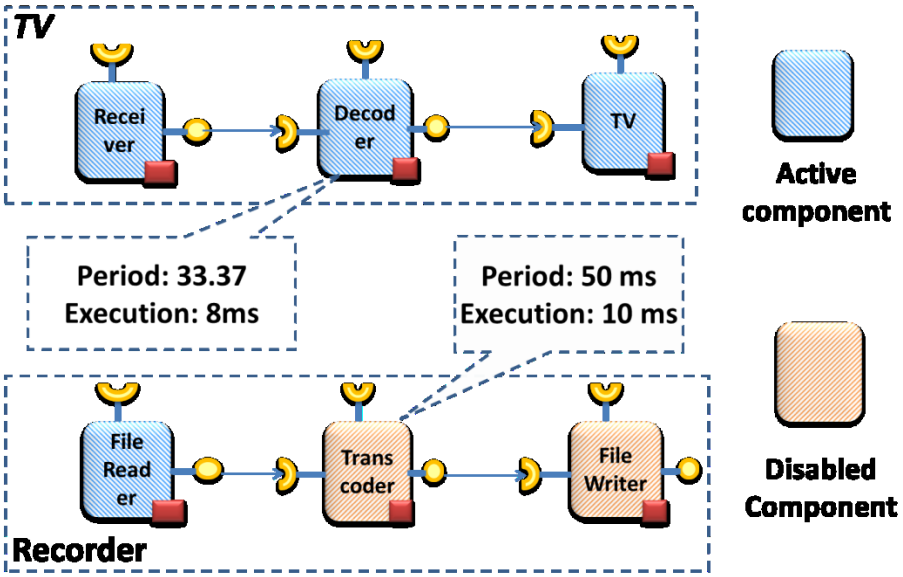


Figure 5-10. TV Performance adaptation

After this adaptation process, other actions may also be triggered as a consequence of these adaptation actions. Here, after Transcoder is disabled, the File writer will be disabled by Structural Modeller, as it functionally depends on the Transcoder components. After these state changes, the reference between enabled components will also be updated by Structural Modeller as described in Section 3.3.1. The component state after adaptation is shown in Figure 5-10, in which different colours correspond to different component states.

5.6 Simulation and comparison

In this section, we validate our framework design and implementation both from a qualitative and a quantitative point of view including such concerns as implementation complexity, adaptation flexibility, overhead introduced by our framework, etc.

5.6.1 System implementation complexity

We use the OSGi framework as our implementation platform1. OSGi [114] technology serves as the platform for universal middleware ranging from embedded devices to server

[1] Source code can be download at https://sourceforge.net/projects/s-transformer/

Table 5-1. Lines of code for Architecture-based adaptation

		Lines of code	Binary size (byte)
Monitoring	Reflections of code	242	3453
	Monitoring	354	7407
Parsing	Model classes	1329	2353
	Parser classes	1950	46680
Structural Modeller	Functional constraints	543	35230
	Reference management	249	5382
Adaptation executor	Dispose management	459	11782
	Instance management	369	8795
	Meta-function Invoking	280	6714
Model Fusion	Merge two sets of adaptation plan	345	9640
DSM Manager	Normalized CMD	360	7620
DSM	CPU-based admin.	95	1993
Auxiliary code		700+	

environments. There are several free OSGi implementation exist, such as Equinox [7], Apache Felix[2] and Knopflerfish [12]. In this thesis, the Equinox, a popular, free, open source OSGi Platform developed by the Eclipse organization, is used as our basic lightweight implementation for local applications management to mimic the performance

development platform. In its current state, our implementation focuses on providing a impact. Our framework has been generally extended to distributed environments by using R-OSGi [132] (Remote OSGi). In current tentative approaches, sensors and actuators can be deployed into different networked computing nodes [125]. Other approaches, e.g. web-service based such as Axis2 [1], can also be used.

Currently two component models are supported. One is our declarative real-time component model proposed in our previous works [69]. This component model was originally designed for the construction of dynamically configurable & reflective real-time systems. The other model is the enhanced version declarative Service component model (adding STDF description and management interface support). The figures listed in Table 5-1 regard the implementation of the extended declarative service component model. As these two component models have very similar structures, (only the service registration and Structural model need some revision), implementation complexity is largely the same.

Corresponding to the system modules discussed in Section 4.3.2, this system is implemented via 6 key modules. The lines of code of each implemented module are shown

in Table 5-1. Our framework also provides such mechanisms as deployment support and version control by simply reusing OSGi's system service, which leads to a lean and quite concise implementation. One of the key modules that are not mentioned in Section 4.3.2 is the Meta-data Parsing module. This module parses the meta-data and stores it in the form of meta-data objects. A simple component meta-data language is defined to describe component characteristics. Clearly the implementation complexity of *DSM adaptation modules* is highly implementation specific, thus the lines of code listed here refer to the simple implementations with component admission algorithm based on CPU utilization.

5.6.2 Adaptation to different contexts

In the traditional approach towards application-based adaptation, in order to achieve adaptation matching new context requirements, developers normally need to reprogram the whole adaptation architecture. This includes, to name but a few, modules for detection, modules for component management, adaptation logic as well as the execution modules. Changing system adaptation logic in current approaches actually means that almost the whole part of system run-time needs to be redesigned and redeployed. This strong coupling between adaptation logic and system run-time makes it is very hard and sometimes impossible to change the system adaptation behaviour during run-time.

Table 5-2. Reusability for Three different approaches

Context Change	Approaches	Binary Reusable module	Code for redeployment
T`V → Recording	Standalone approach	None	Structural + Healing + TV
	ACCADA	Structural Modeller	Healing + TV
	Transformer	Structural Modeller + Self Healing	Transcoding Optimization
With Self-Healing → Without Self-Healing	Standalone approach	None	Structural + Healing + TV
	ACCADA	Structural Modeller	TV
	Transformer	Structural Modeller + Self Healing	0 , just remove DSM for Self-Healing

Simplified Implementation: In this thesis, the adaptation logics are run-time formed by compositing several DSM's adaptation plans. Each DSM only deals with domain-specific adaptation knowledge. Compared to the traditional "one-modeller-for-all" solutions such as the one in [25, 35, 138], this feature allows the DSM to be implemented in a much simpler way.

Without the burden to implement software maintenance tasks, a DSM adaptor can be implemented very concisely. For instance, one adaptation for TV optimization adaptation (described in Section 5.5.1.1) can be implemented in less than 120 lines of codes. On the

other hand, an ad-hoc approach wound need re-implementing a new version of a basic component management run-time (in our case, about 2000 lines). Thus, programmers can focus on one domain-specific adaptation logic rather than having to take care of those low level details.

Reusability: Three different approaches are compared in terms of reusability: (1) the stand-alone approach, combining system run-time with all adaptation strategies, (2) the one separating a *Structural Modeller* and *Context Modeller* of our previous approach (ACCADA) [70] and (3) the Transformer approach reported in this thesis.

In (1), adaptation modules are tightly coupled with system run-time, thus it is impossible to reuse the adaptation module to other environments. In (2) (our ACCADA framework), *Structural Modeller* is separated from other adaptation modules and can be reused across multiple contexts while the contextual adaptation logics cannot be reused. In contrast, the Transformer framework in (3) provides the highest reusability towards those adaptation logics. Each DSM can be possibly reused to compose more complex adaptation logics, just as multiple components are used in composing applications.

Table 5-3. Application-based vs. Architecture-based Adaptation

	(1): Application adaptation	(2): ACCADA Framework	(3) Transformer
Adaptation logic	Prefixed	Changes during runtime	Run-time Composited
Context knowledge Integration	Static/Internal	Flexible/Architecture	Flexible and run-time composite
Implementation Complexity	High	Low	Low
Multi-context support	NA or static	Yes	Yes and flexible
Context-specific Adaptor implementation	Complex	Each model for different context	Composite
Separation of design concern	Mixed	Yes/limited	Yes
Level of Adaptation	Application specific	Across several applications	Across several applications
Reuse adaptation across multiple contexts	NA	NA	Yes

In the Transformer framework, each DSM can be independently deployed and (possibly) reused across multiple contexts. For instance, the DSM for Self-Healing can be used in both TV and Recording contexts. During context changes, only the adaptation strategy related to that domain should be altered and all the remaining modules can be kept largely unchanged. For instance, we can simply add self-healing feature to the system by

adding/removing the corresponding DSM to/from the registry. All these adaptations happen during the run-time. This makes the system deal more effectively with context transitions. Table 5-3 provides more an exhaustive comparison between application specific adaptation, ACCADA and our current Transformer framework.

5.6.3 Architecture performance

To evaluate the performance of the system adaptation, we instrumented a test to measure the time for fetching, parsing, reference management, and configuring. We focused on the time for installing a single component as we vary the number of components managed by the framework. Here, each component has one provided interface, one required interface, and one attribute (except the first component which only has one provided service interface). The newly installed component is structure-satisfied when installed. The size of each component is the same – 20.6 KB. In order to avoid the impact of component execution towards simulation results, all these component execution parts are disabled during the experiment. Here, as we are only interested in the framework performance, in the newly installed component, the component initialization time is not counted as it may vary according to different implementations.

DSM here is implemented to perform a simple admission control algorithm. It checks that the arrival component will only be enabled when the following constraints holds true

$$\sum \frac{\text{Executiontime}}{\text{Period}} < 1 \text{ for all enabled components.}$$

As hardware platform, we use a Dell D630 laptop with 2.2 GHz dual core T7500 CPU, 2GB RAM and 80 GB 7200RPM HDD. The JVM we adopted is SUN JAVA 1.6.0.2 SDK on Windows. Of course, it is not a real embedded platform. Currently we are actively migrating this platform to other embedded environments.

Installing a new component normally consists of five main steps: component loading, meta-data processing, structural modelling, context modelling, and model fusion for merged adaptation plan. The actuation time is not shown here as it depends on each component's implementation. Figure 5-11 shows the absolute times spent in each step of the process. Each value is the arithmetic mean of 250 runs of the experiment. In order to better illustrate the trend of different steps, we use two Y direction axes in expressing the data. Values in stacked column use the main Y axis (left) and those values in marked lines use the secondary Y axis (right) . The time scale used in both axes is micro-seconds (μs).

As we can see from Figure 5-11, component installation time grows slowly with n, the number of system-managed components. This is mainly due to the fact that two key elements – component loading time and meta-data processing time, which count about 60% ~ 80% of total time –keep comparably stable when n grows. In contrast, the other three key elements will grow with the number of managed components. The structure modelling process mainly deals with matching composability between installed components which has computational complexity O (n *m), in which n is the number of installed components and m is the number of all required interfaces of the newly installed component. This is

because all the required interfaces will be checked. However, in our simulation, each component only has one provided interface and one required interface, so the complexity becomes O(n).

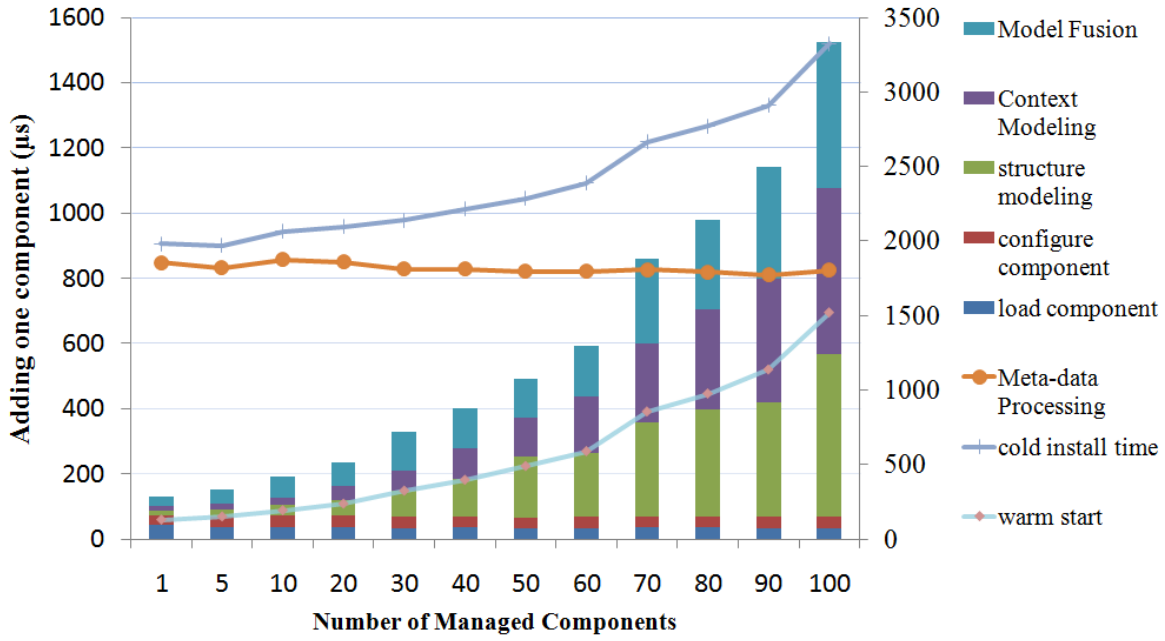


Figure 5-11. Framework performance on adding one new component

DSM modeller here will check whether the new component can satisfy the resource requirements which also have complexity O (n) (stateless implementation, no optimization). Here, the model fusion processing time is for post-processing the modelling results from two modelling processes. The fusion process mainly uses set operations, for instance, finding enabled component configurations by calculating the intersections between two enabled component sets. If we call m as the cardinality of the intersection, then this operation has computation complexity O (n*m) on average.

As most of the installation time results from the large meta-data processing task, we optimized the installation process by parsing a component's meta-data prior to its usage (without initiating the component). We call this a "warm-start". Compared to normal cold installation process, this approach can greatly reduce a component's response time – more than 800µs can be saved in our platform – of course, at the expense of extra memory usage.

Simulation results show that our framework scales well when the number of managed components grows. However, the DSM processing time confines to the simple algorithm described here in which only one DSM is used. In the next section, we evaluate the performance when multiple DSM are involved in the adaptation process.

5.6.4 Multiple DSM performance evaluation

In this section we test the overhead of managing multiple adaptation logics into separate modules as compared to standalone approaches. Each domain-specific adaptation

modules is tested for execution time. As this execution time is affected by the execution route, we also compared the execution time in two different conditions, that is "TV application Structure Check" and "faulty Component detected". Each simulation is performed 1000 times and the average values are calculated to soften the impact introduced by e.g. Java's garbage collection.

Table 5-4. Adaptation modules' performance

Context	**With fault detection**		**TV application Structure Check**	
	TV (μs)	Self-Healing (μs)	TV (μs)	Self-Healing (μs)
Standalone	7.587	21.8612	7.992	18.288
Transformer	7.966	22.3552	8.335	19.083

Execution Time: As we can see from Table 5-4, in different process scenarios, there is no significant overhead in terms of execution time. For instance TV application construction (recomposition process), about 26.28μ are needed for standalone solution while DSM based solution needs about 27.418μa. The separation of adaptation logics into discrete adaptation modules only introduces little overhead mainly from service calls. Less than 2μ overhead are needed for the separation of adaptation modules. If an error is identified, about 30 ov are needed for adaptation, in which self-healing modules take about 22μa and TV optimization process need about 8μa. We can also see that the execution times for two different adaptation modules are quite fast. This is because each adaptation module only has simple adaptation logics and there is no adaptation action performed inside these logics. Of course, more complex adaptation logic is likely to require more time to generate their adaptation plans.

Model Fusion Time: Another key overhead introduced in the separation of adaptation is due to the model fusion process. In order to analyse the performance of such process, we introduced one simple DSM, which only iterates through all installed components and gets their properties. This DSM tags all installed components as "enabled" and performs one setProperty action. This DSM is deployed multiple times (from 1 to 9) to simulate multiple DSM adaptations. They compute actually exactly the same adaptation plan. This design actually creates the worst-case scenario for model fusion. Each set fusion operation needs about O(n*m) for the fusion complexity. In this worst case scenario m is equal to n.

Then, the plan generated by different DSM will be fused into one adaptation plan according to the fusion rule set specified in Section 5.3.4. From Figure 5-12, we can see that the model fusion time depends on two basic factors: one is the number of managed components, and the other factor is the number of utilized DSM. The more DSM are used, the more model fusion time is required. However, such time grows about linearly with the number of utilized DSM. According to our programming practices, for any given context, normally less than 5 DSM will be used. This means that for each round of adaptation, less than 30μs fusion time can be expected.

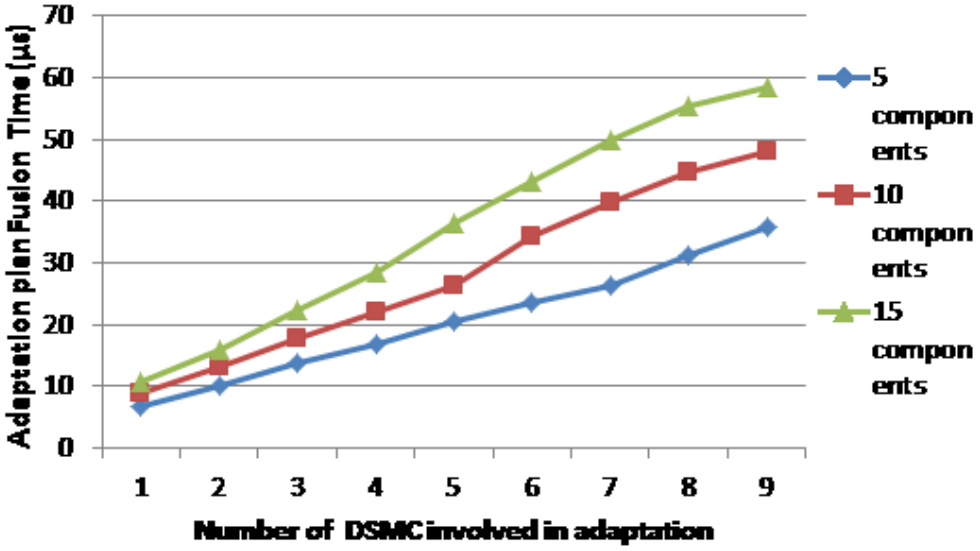


Figure 5-12. Performance of Model Fusion

Here, although the Model Fusion time deteriorates with the number of DSM as it has shown in Figure 5-12, in practice, it won't increase linearly with the number of system resolved DSM. That is because each DSM has its own set of interested change events. For instance, when *Event monitor* gets a "Decoder frame dropping rate > 2%", it raises an adaptation event with three available modellers – *Structural modeller, DSM for TV* and *DSM for Self-healing*. However, the *Structural modeller* and the DSM for Self-Healing are not interested in this event, only *DSM for TV optimization* will be invoked and generate adaptation plans. From our current experience, this approach can greatly decrease the system overhead compared to the one with indiscriminately DSM invoking. However, in what extent it will impact on conflict detection and model fusion is still not quite clear. Future work is needed to have formal analysis on this impact. It is also one of our key future research topics.

5.6.5 Resource usage (comparison with Juliac)

Certain component frameworks provide tools to help programmers to automatically generate auxiliary codes. Examples include Juliac - a Fractal tool chain backend, which generates Java source code corresponding to the application architecture specified by the designer. Such code includes membrane source code, a framework glue code and a bootstrapping code. In the following section, we compare our approach with Juliac's [11].

In the Juliac approach, ADL language is used to generate the glue code and the codes for introspection. The simplest "hello world" example uses two components, Client and Server. The Client will try to invoke the exposed service interface to print the "hello world" string. The reason for selecting Juliac for comparison is that it is a typical application based development platform. It is ready available and well documented and also based on the Java language.

Table 5-5. Line of codes

	Application size	Lines of code(business)	Lines of code (generated)
Juliac	95.7 KB	100	3500
Transformer	4.7 KB	140	0

Table 5-5 shows that, for such simple application with only two functional components, the business code is about 100 lines, including import and interface definitions. With Juliac, about 3500 lines of Java codes will be generated. Such code mainly contains the glue code and basic system run-time. In comparison, in Transformer, no process for on-line auxiliary code generation is needed, as it dynamically manages component's reference together at run-time. In our framework, an application mainly contains its business code, simple and easy to manage.

Resource consumption: We implemented the simple \hello world" application in two components models, and executed it with different number of instances. We can see the difference from Figure 5-13.

With the Juliac approach, the memory consumption will increase considerably with the number of applications, while in our framework it will increase of about 42Kbyte for each installed application (with 2 components, including system management overhead). Of this amount, for each installed component, about 13Kbyte memory are used by our framework to store the parsed meta-data information and reference relations. This overhead is comparably small with respect to the more than 430Kbyte required by the Fractal model.

This discrepancy comes from the different models employed. Juliac focuses on using Fractal component technology to build adaptive applications. For successful execution, each application has to carry a full set of the system run- time. If a system has many applications installed, the overhead from the basic system service will be high. In contrast, Transformer framework is designed to support a set of managed components and is decoupled from application business logics. No matter how many applications are deployed, only one set of basic services is needed. It is worth highlighting here that, being Juliac an application based development platform, the glue code and system management run-time is custom generated for each target application. This means that the generated codes normally cannot be easily adapted to work on other applications. For this reason we say here that, in Juliac, for each application, a new run-time is needed.

As for the total resource consumption, the basic OSGi framework contributes to about 987Kbyte of memory usage, and about 1650 loaded classes. Our frame- work in total adds up to about 101Kbyte of memory and about 160 loaded classes (with one installed component, when no component is managed, over- head is about 7KB). The OSGi implementation we used, Equinox v3.3.2, is a general-purpose platform and not optimized for resource usage. Other compatible OSGi implementations, such as Concierge OSGi [131], achieve a memory consumption of less than 200Kbytes. In other words, by simply changing OSGi implementation, the resource consumption can be further reduced. Other memory compression techniques can be also applied to reduce its memory usage.

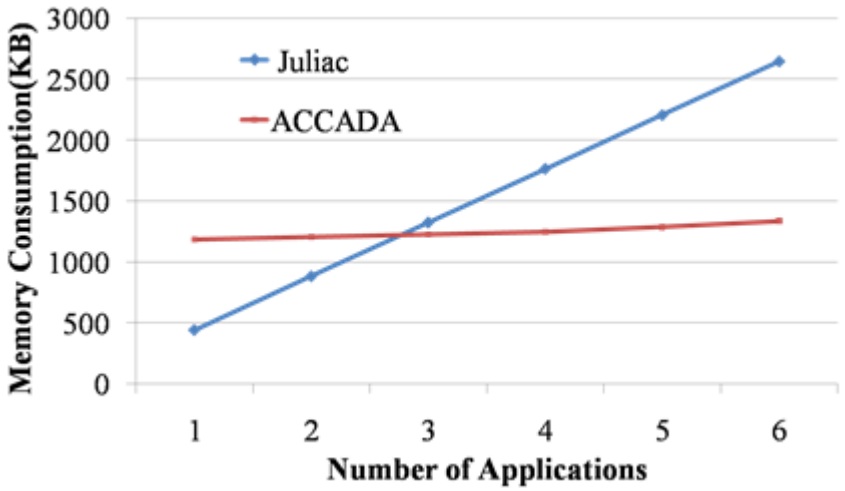


Figure 5-13. Memory Consumption

5.6.6 Self-healing experiments

Figure 5-14 compares the processed video frames of the TV application, in the presence of node failures, when the overall infrastructure is managed by a human administrator versus our multiple DSM architecture. This TV application is composed as in Figure 5-9. In order to avoid its impact towards system adaptation, the processing part is set to be empty. Our framework is triggered by time-trigger failure monitor at rate of 6 times / minute. To compare the approach in the same level, we assume a human being will also check system correctness every 10 seconds (which is rather frequent for human beings). Error is introduced at 15, 205, 445 seconds.

Under human administration, when a failure occurs in our system (simulated by setting the Decoder component to unavailable), the Recording application becomes unavailable and all the video data gotten from remote site could not be handled which results in Video data loss. In order to handle this failure, the system operator must react as follows: (1) first, he/she must detect the failure; (2) the operator must know how the application is composited; (3) based on application construction knowledge, he/she should understand the failures and decide what reconfiguration adaptation is needed. (4) The operator has to unload the failed component instance and install a new component instance. (5) The newly initialized component‘s property are restored to the values just before error happens. (Whether this requirement can be fulfilled depends on whether the system has a logging service). In our simulation, it takes at least 50 seconds for the whole process. During this time, video data (at least 1500 frames of video data until the end of the recovery) will be permanently lost.

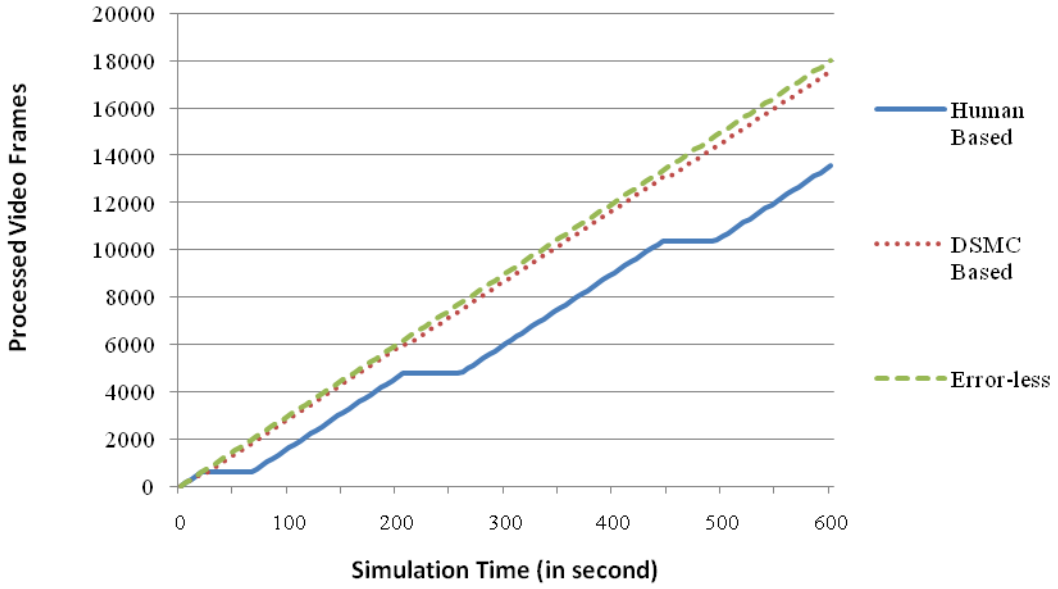


Figure 5-14. Process Video Frames with Human/DSM based Healing

By way of contrast, the frame loss is much less than in the human-based adaptation, when DSM for Self-healing is used. In this application, less than 10 ms is needed to repair the errors (for the whole adaptation process, 3 rounds of adaptation, although for each round of adaptation, about 22μs is needed for reasoning), system recovery-time is mainly attributed by the 10 second interval of system error check, which contributes about 5 seconds delay.

In our experiments, the human administrator is assumed to be an expert and constantly monitoring for failures. He is able to perform all the repairing tasks. However, ordinary users normally could not do all the repairing tasks and would not check for errors every 10 seconds. Oppenheimer et al. [115] have shown that in real Web-service maintenance conditions, system operator delay is the largest contribution to the Mean Time To Repair (MTTR) and might reach up to 9 hours. In our case, as each round of adaptation only needs 1ms for adaptation, system adaptation can be performed much more frequently. Better results can be achieved. When making use of a more powerful actuator, the "repair a component" action can reduce the three rounds of adaptation to one round thus greatly reducing the adaptation time (less than 1ms in our simulation).

5.6.7 Adaptation with DSM TV optimization

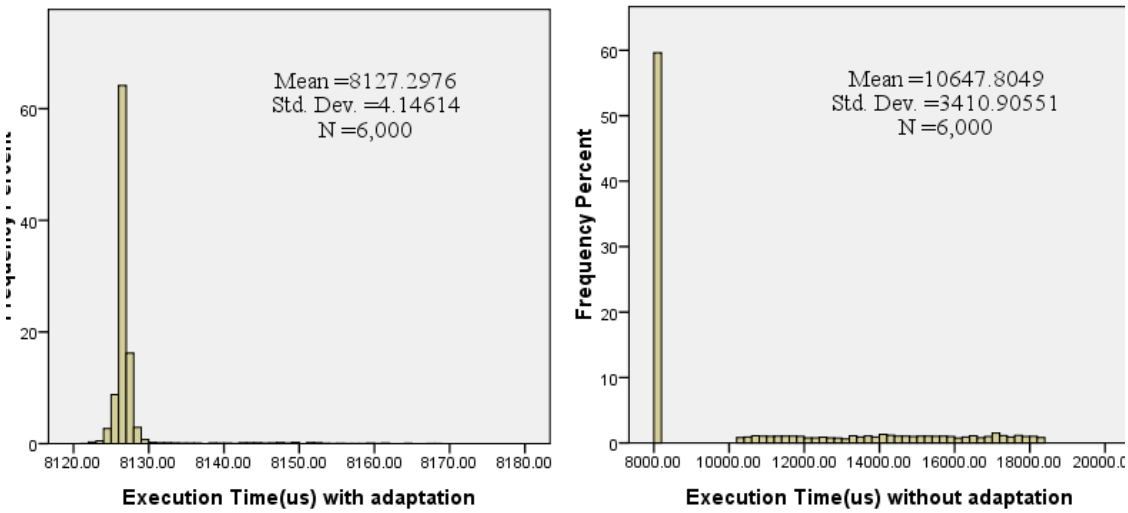


Figure 5-15. Execution time (Adaptation vs. no-Adaptation)

In order to validate this framework both from a qualitative and a quantitative point of view, we implemented the adaptation scenario described in motivation example (see Section 4.1 and the adaptation algorithm as described in Section 5.5.3. Hardware and software configuration is the same as in Section 5.6.3. In order to support real-time task, the software system is run on the Fedora 7 kernel 2.6.23 with Real Time Application Interface (RTAI) version 3.6, a real-time patch for Linux[30].

As shown from Figure 5-15, all six modules' execution parts are implemented as periodic tasks. The Decoding component of TV application is implemented with a real-time task with period 33.37 ms with priority 2 (the smaller the value, the higher the priority), execution time of about 8 ms while deadline is 12 ms. The execution part of Transcoder module is implemented as a real-time task with period 50ms, priority 1, execution time of approximately 10 ms and a deadline of 30ms. The schedule policy used by the underlying RTAI system is FIFO. In order to show the interference between these two components only, all other components use priority 6 and make use of asynchronous communication only, so that we can focus on the two coding module's performance.

As the video decoding execution time may vary according to the contents of video streams, in order to more clearly demonstrate the mutual influence among applications, we substituted the decoding function with a mock function using comparably constant CPU time for each round of calculation.

We performed 6000 observations for the execution time of Decoding Module after system enters stable state. Figure 5-15 shows the execution time distribution with and without the DSM. The time scale for execution is µs. In the former case, the execution time of decoder is mostly about 8000µs, while the biggest execution time is about 8180µs when the context knowledge is used (as this disables the transcode module). In contrast, if no context-specific knowledge is available, the system will try to run these two applications simultaneously. The Standard deviation of execution time is 3410.9 µs, much higher than 4.15µs, the one with adaptation. The jitter of decoder task's execution time is very big, as about 31.3% (1880/6000) runs exceed the deadline specified in the component's meta-data

(12 ms), which can result in a lot of lost frames. As can be seen from Figure 5-15, with DSM adaptation knowledge, the decoding modules can achieve much better performance in term of mean execution time and standard deviation.

5.7 **Discussion**

In this chapter, a middleware architecture is presented to support the Transformer adaptation framework and the software construction methodology presented in Chapter 4. Its major goal is to provide architecture support to enhance adaptation module reusability. At the same time, the declarative and reflective component model extends OSGi native component model and is able to provide more complex contextual dependences description and managements. This DRCom model also supports structural and behavioural reflection to facilitate compositional adaptation during run-time.

Together with a hybrid real-time component model, the earlier version of this middleware architecture has been implemented and is used in the ARFLEX project [3], where it served as the basis for real-time application run-time reconfiguration support. The early version has been extensively used in real-time component installation and reconfiguration phase with simple real-time tasks related concerns. It was later enhanced with multiple DSM support – for instance, self-healing and performance-related. At the same time, conflict detection and resolution support was introduced in our paper [72]. This middleware is constructed by using service-oriented component model, which means each component used can be replaced later with a more enhanced implementation. For instance, currently the Adaptation *Actuator Model* which represents relationships among actuators currently is hardcoded. This can be enhanced with more powerful conflict detection by using e.g. a S*emantic-based Actuator Ontology Model.* The same holds true for the DSM context matching calculation, etc. Benefited from the modular design and service oriented architecture our middleware adopted, these enhancements can be easily injected to the system without major change in the rest of the system.

In our modular middleware design, although it demonstrates a powerful and flexible characteristics for DSM dynamicity support and run-time adaptation evolutions, more features and improvements are planned to further improve its adaptation capability as well as reduce system overhead. For instance, the DSM Fusion mechanism described in the Section 5.3.4 is being revised to enable more efficient adaptation plan calculation. At the same time, a semantic based description language is being designed to describe the relationship between installed Actuators as well as their effects. Furthermore, additional functionality is being designed for the adaptation interface for DSM plug-ins. For instance, add more refined interface for DSM preconditions and support adaptation exception.

In this chapter, scenario-based simulations are used to demonstrate our middleware flexibility and the reuse of adaptation behaviours. In the next chapter, our adaptation framework and modular middleware is evaluated by applying them to implement a practical project – an Autonomous control platform for NXT robots.

Chapter 6

Autonomous NXT Robot Control Platform

This chapter evaluates the proposed adaptation framework and its supporting middleware architecture in a practical use case. The Autonomous Robot Control platform [73] is implemented based on our middleware system. Based on this robot control platform, a demonstrative application – an application to explore an unknown territory – is designed. This application can be incrementally enhanced with adaptation behaviours that can be introduced during run-time by adding new DSM.

In this chapter, two DSM are designed and developed together with their supporting sensors. The much simplified application development process as well as the incremental development process for adaptation behaviour composability provides the foundations for the evaluation in Chapter 7. In the following section, the background on autonomous robot controls is firstly presented which explains why, for the autonomous robot control, adaptation behaviour evolution is an important need.

6.1 **Introduction**

Autonomous robots can perform their intended tasks in unstructured environments without (or with minimal) human guidance. They are designed to learn or gain new capabilities like adjusting strategies for accomplishing their task(s) or adapting to changing surroundings. Such a high degree of autonomy is particularly desirable for

environments that are impossible to fully model, such as space exploration [15] or that are uneconomical to fully model such as floor cleaning robot [82] in home automation environments.

A basic concept that is applied in autonomous robot control is the closed control loop. Each autonomic system consists of managed resources (controllable hardware or software components) and an autonomic manager for steering the underlying managed resources. In other words, this basically follows the MAPE adaptation reference model. The way a software system is designed to support these modules, it is argued, can greatly influence a robot's adaptation capabilities.

A typical example is given by the Mars Rovers [15]. These robots, 170 to 320 million kilometres away from the Earth, are able to receive and send information, either directly or via the Mars Orbiter (satellite around Mars). One problem with its software platform is its static nature. When the software needs an update to fix some problem or when some new features are needed to face unprecedented conditions, the whole robot software needs to be rewritten. The other problem is the static planning capabilities in the Planner. To add a new adaptation behaviour for a new environment, a robot administrator needs to manually design a list of actions and uploads them to the robot when the communication link is available.

In order to introduce more flexible adaptation capability for an autonomous robot, several approaches make use of AI based techniques – examples include Hashimoto et al. [77] , who use evolutionary computation and fuzzy systems, or Inamura et al [79], who use Bayesian networks. This learning process is highly coupled with the robot targeted execution environments. Although these approaches are effective in adapting under their target domains, however, with these pure AI based approaches, it is very hard or sometimes impossible, it is argued, to provide adaptation behaviours across multiple contexts.

The work described in this chapter applies our Transformer framework in introducing adaptation behaviour to autonomous robot systems. In a nutshell, our architecture model realizes an adaptation loop, which can be run-time, revised so as to better match the current context. This strategy allows the application configuration to be modified outside of the application business logics. In order to deal with changing environments or/and robot status, our adaptation framework is designed to systematically support multiple adaptation DSM. An adaptation plan is generated by run-time selected adaptation modules according to context to date. Robot applications, built from individual component instances, are composed and reconfigured by these run-time generated policies.

In order to seamlessly integrate the NXT robot into the Transformer framework, a remote NXT model is designed to represent native NXT sensors and motors as DRCom components in the modular middleware architecture.

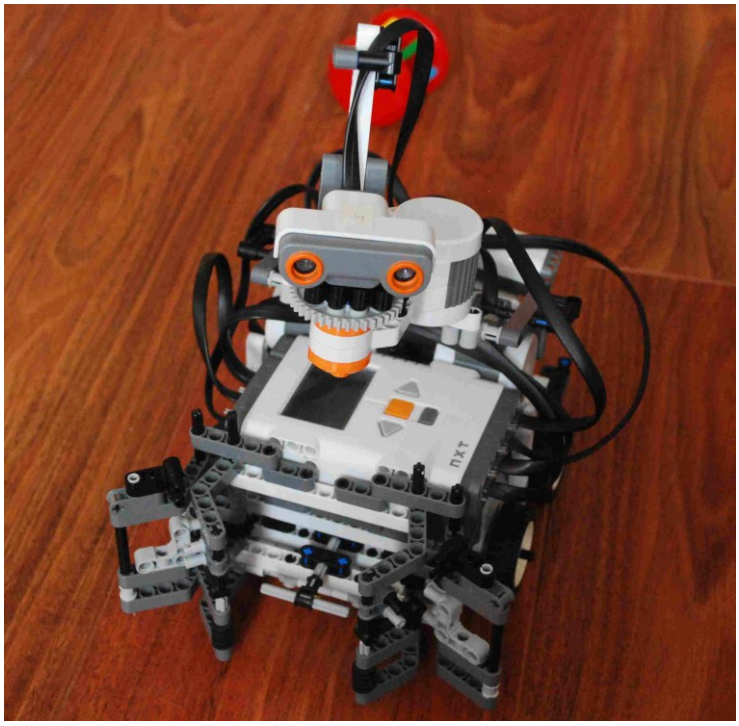


Figure 6-1. A picture of an assembled NXT robot, the front side has a real touch sensor and the back side has a "touch sensor" created by the light sensor

One robot exploration application, which aims at discovering unknown territory by using system available sensors and actuators, is designed. This exploration application should be optimized/reconfigured according to the system current context (battery voltage is used as context factors). At the same time, self-healing capabilities are introduced to deal with the failure of certain components (recovering from a touch sensor failure during run-time).

In order to provide these two different domain-specific adaptation goals – Battery-based adaptation behaviour and Self-healing adaptation behaviour – two different DSM and their corresponding *Event Reasoners* are designed. A demonstration is given to show how to inject a new DSM into the system to equip the robot with new adaptation capabilities. This on-line adaptation behaviour evolution capability makes this robot control platform capable of dealing with unforeseen environments.

6.2 **Integrate NXT robot into Transformer framework**

In order to build a control platform for autonomous NXT robot with the Transformer framework, it is mandatory to let these two systems communicate with each other, which means a communication protocol should be designed. The other important factor to be taken into account is that our Transformer framework only performs adaptation on top of DRCom component model. A wrapper is needed to represented native NXT sensors/motors as DRCom components.

Firstly, a brief introduction of NXT robot is introduced to give better understanding of the robot platform.

6.2.1 Introduction of NXT robot

As our robot platform, we use the Lego Mindstorm [13]. This robot contains 4 sensors, 3 motors, a so-called NXT brick working as controller and lots of Lego components. These components can be put together to build a robot with perception and navigation capabilities. Figure 6-1 shows one picture of the final assemble of a NXT robot. In the following section, the hardware of NXT robot as well as its software programming platform will be introduced.

6.2.1.1 Sensors & Actuators

As we already mentioned, a NXT robot is equipped with four different types of sensors and three identical motors. In this control platform, two of the three motors are configured into a Pilot actuator with a set of standard navigation functions provided to simplify the robot movement management. Table 6-1 provides a list of all available sensors and actuators.

Table 6-1. NXT robot available sensors & actuators

Touch sensor	Sensor to detect hits; one of the most usable sensors of the package.
Light sensor	Sensor to detect light strength. In what follows, it is used to form a second touch sensor.
Sound sensor	Sensor to detect sound. In this thesis, not used in what follows
Ultrasonic sensor	Sensor to measure the distance to an obstacle. This sensor is probably the most powerful sensor in the package, but it is not immune from problems. The theoretical maximal distance of 255cm is just not realistic. In best case it can measure distances up to 150cm. It also has problems with object that have shiny surfaces. This kinds of objects might not be found at all. From time to time it also detects object that aren't there.
Motor for turn	This motor is designed to work with ultrasonic sensor, it turns the sensor attached to it by specific angels
Pilot	It is a combination of two motors. It is used to control two motors collectively. So the robot can drive forwards, backwards or turns by certain angles.

Compared to a commercial robot, NXT robot provides limited resolution and stability on the sensor retrieved values. Among these sensors, touch sensor is intensively used because the sensor data is useful for exploration purpose. As it is shown in Table 6-1, although judging the specification, ultrasonic sensor appears to be the most powerful sensor in exploring environments, however, after thoughtful testing, it is proved that this sensor provides very limited sensing accuracy. The light sensor is not quite useful for the

exploration application due to its collected data type and its limited resolution. This sensor later is transformed into a back-up touch sensor which is located at the opposite side of the primary touch sensor.

6.2.2 Integrating the NXT Robot into the Transformer framework

In order to enhance the programming capability of NXT robot's, native NXT software platform in the NXT brick was erased and reburned with the LoJOS system [14]. LeJOS runs a special Java VM which was originally created from the TinyVM project [17].

However, mainly due to the hardware limitations of NXT robot, the LeJOS platform supports only a small subset of the minimum requirements for OSGi. It is therefore not feasible to run the Transformer framework directly on top of the LeJOS platform. To tackle this problem, the robot software was split into two parts. The robot itself runs a static part which receives and executes commands from its PC counterpart and sends sensor value on demand or periodically, while another more powerful computing device (a PC, in current version) runs the exploration application on top of the Transformer framework.

6.2.2.1 Remote NXT model

In Figure 6-2, the left block represents the NXT robot, which runs a static program on the LeJOS VM. Its major goal is to expose most of the devices of the NXT to the Transformer framework over the network. A remote NXT model is created to perform this task. A program that runs on top of the NXT robot executes a custom designed protocol to receive commands and send corresponding responses. On the PC side, represented by the right block, a set of proxy service components are created to represent exactly the same functional parts of the NXT robot. By using the remote NXT protocol, all sensors and actuators on the NXT become available in the PC side too.

6.2.3 Simple example – touch sensor

The easiest way to fully understand the protocol and all involved components is to use a

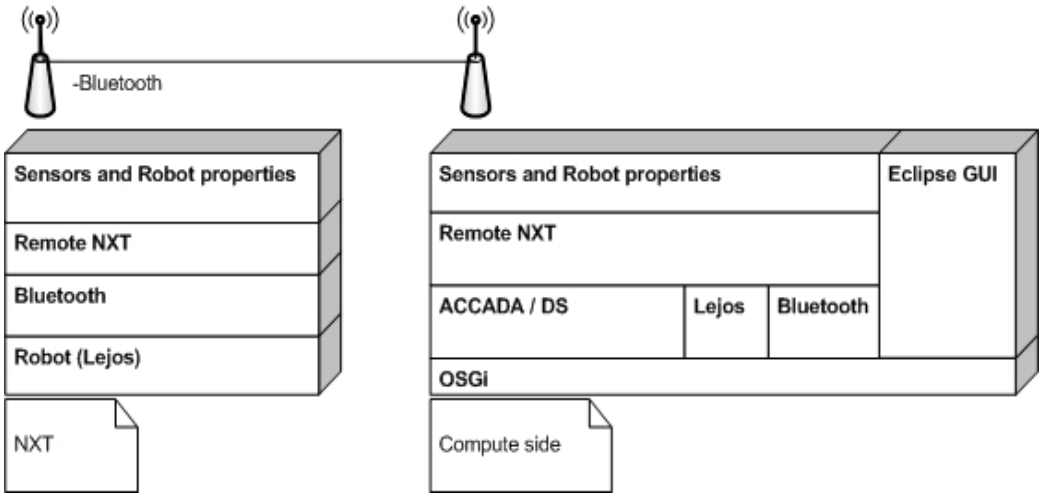


Figure 6-2. Remote NXT model[125]

very simple example. This subsection describes how the remote NXT robot model is used to make the touch sensor remote controllable. A simplest sensor –*Touch Senor* is used. In the robot side, The *TouchSensor* class provided by LeJOS has only one method – isPressed(), which tests if a touch sensor was pressed. Since we want to expose the class over the network, touch sensor classes in both PC and Robot side need to implement the *ICommandable* interface. The *ITouchSensor* interface is introduced to represent *Touch Sensor* related methods. The following diagram shows the class diagram structure from both PC side as well as Robot side.

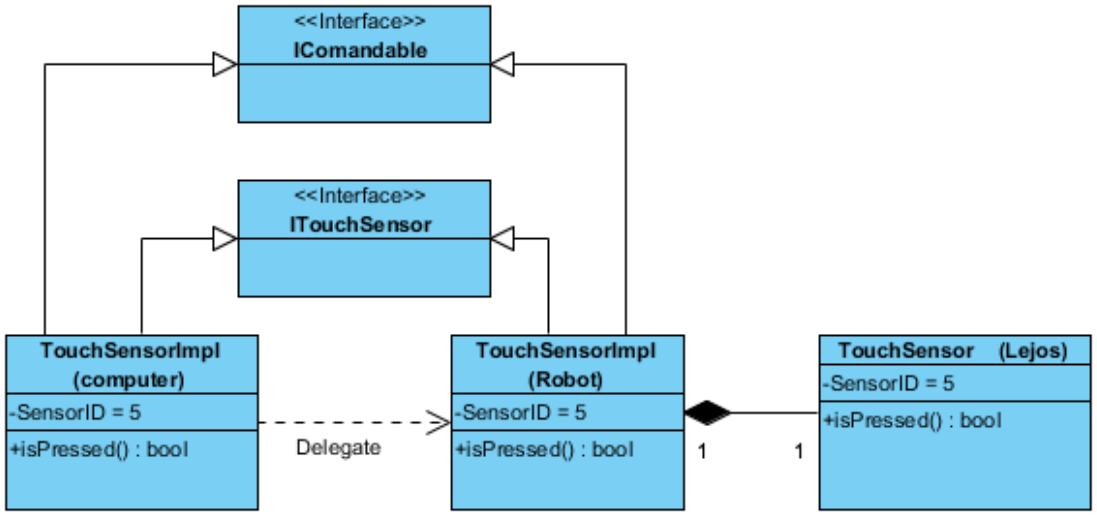


Figure 6-3. Remote NXT model – Touch Sensor

6.2.3.1 Robot side implementation

The implementation of robot side *Touch Sensor* is quite simple. It extends AbstractCommandable class and implemented *TouchSensor* interface. The AbstractCommandable is a helper class that implements ICommandable interface for the remote NXT model. The implementation of the isPressed method on the robot side only forwards the call to an instance of the LeJOS lejos.nxt.TouchSensor class. The following snippet (Listing 6-1) shows the most important code of this class.

In our remote NXT model, in order to send commands between the two counterparts, Command classes are designed to transmit commands and receive response. For the touch sensor, the IsPressed – a serializable command class – is implemented to be sent between PC and robot side by the touch sensor.

```
public class TouchSensorImpl extends AbstractCommandable
implements TouchSensor {
private lejos.nxt.TouchSensor touchSensor;
    public TouchSensorImpl( lejos.nxt.TouchSensor touchSensor ) {
        super();
        this.touchSensor = touchSensor;
        addCommand(new IsPressed());
    }
   public Boolean isPressed() {
        return touchSensor.isPressed();
    }
    public Integer getId() {
        return TouchSensor_ID;
    }
```

Listing 6-1 Touch Sensor – Robot side

6.2.3.2 Command interface

As we already explained, in this NXT remote model, each sensor/actuator must implement the ICommandable interface to transfer commands and receive response from its counterpart.

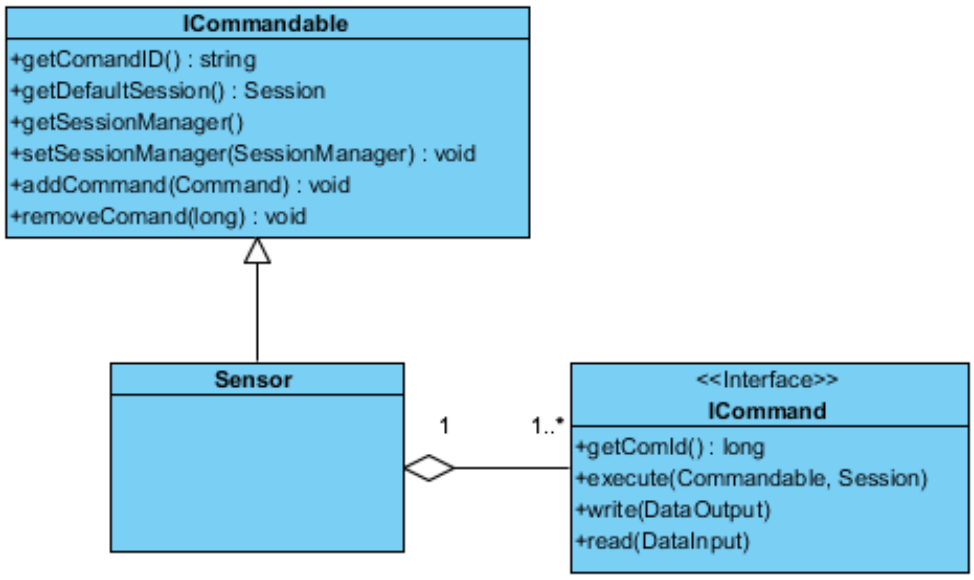


Figure 6-4. Sensor, ICommand and their relationship

In the NXT robot, different sensor might have totally different commands and number of commands. In order to support this diversity, the ICommand interface is designed to support different types of commands. As it is shown in Figure 6-4, one sensor might have more than one functional command. As an example For the *Touch Sensor*, it has only one

functional command – IsPressed; however, other sensors might have more functional commands. For instance, *Ultra-sonic Sensor* has five different commands. Implementation of this ISPressed command is shown in Listing 6-2. The command has two major fields, one is used for the result (pressed/not pressed) and the other one is used to check if the command is a request or an answer.

The command is sent from the compute side with the isRequest flag set. When the command arrives on the robot, the commandable parameter then will be cast to a *TouchSensor* and stored in a reference. The robot side uses the corresponding LeJOS sensor class to poll the real data (touchsensor.isPressed()). The result of this poll is sent back as an IsPressed command by using the connection session. The computer side only needs to put the command into connection session which is read by the corresponding robot side.

```
public class IsPressed implements Command {
   ….
   private Boolean mIsRequest;
   private Boolean mIsPressed;
 ….
 public Boolean getIsPressed() {
        return mIsPressed;
    }
    public void execute(Commandable commandable, Session session)
throws Exception {
        if ( mIsRequest ){
            TouchSensor touchsensor = (TouchSensor) commandable;
            session.send(new IsPressed(touchsensor.isPressed()));
        } else {
            session.buffer(this);
        }   }
…
}
```

Listing 6-2. IsPressed command (robot side)

6.2.3.3 PC-side implementation

In order to provide session-based command support, a session manager is used in the remote NXT model. Whenever a new poll is needed, the PC side inquires the session manager to acquire a new session. The createSession method, a helper function, is designed for creating a new session from the session manager. The IsPressed command with requested command will be sent by the session class. Then, it waits for a reply. When the execute method on the client side has put the command in the buffer as described in Listing 6-2, the session.read method returns the response. After having successfully gotten the isPressed command reply, the session is destroyed and the result is returned. When

```
public class TouchSensorImpl extends AbstractCommandable
implements TouchSensor {
    public TouchSensorImpl() {
        super();
        addCommand(new IsPressed());
    }
    public Boolean isPressed() throws RemoteException {
        boolean result = false;
        Session session = createSession();
        session.send(new IsPressed());
        IsPressed command = (IsPressed) session.read();
        destroySession(session);
        result = command.getIsPressed();
        return result;
    }
    public Integer getId() {
        return TouchSensorID;
    }
}
```

Listing 6-3. Touch Sensor – PC side

there are errors during transmission or timeout, this method stops and raises an exception. The snippet of implementing code is shown in Listing 6-3.

6.2.3.4 Discussions

Due to the fact that the proxy devices are implemented as service components, these components can be directly used to compose a robot application by using the Transformer framework. This remote NXT model lets the remote devices, such as sensors and motors on a NXT, be remote manageable as local OSGi service components. Although this model is used in controlling the remote NXT robot, this design can be easily extended to support more complex external sensors and/or actuators that are not natively supported by a NXT robot. In this way, we can easily increase a robot's capability without the need to use a more powerful robot. Exploiting from this model, we successfully integrated a video camera sensor from a Sony Ericsson K750 into our autonomous robot control platform [125].

This remote NXT model is also designed as a layered architecture which separates the communication layer from the upper level of components such as sensors and actuators. Users can easily change the underlying transport medium with another transmission protocol without the need to change the upper layer implementation. Currently, either Bluetooth or USB transmission components can be used interchangeably for the transmission in the proxy model.

Of course, this design introduces additional complexity in implementation and brings additional performance overhead. If the targeted robot is powerful enough to run OSGi

upon, then, this structure might not be needed. But it can still be useful when additional sensors and actuators are needed to be integrated to expand existing robot's capabilities.

6.3 **Developing the robot explorer application**

In this section, a self-adaptive robot exploration application is developed. This application consists of a typical robot exploration application whose software structure is able to be contextually optimized, so as to automatically change its exploration behaviours. At the same time, this case study also demonstrates how to add self-healing capabilities to existing exploration application during run-time without changing the application business logic (see Section 6.4.2).

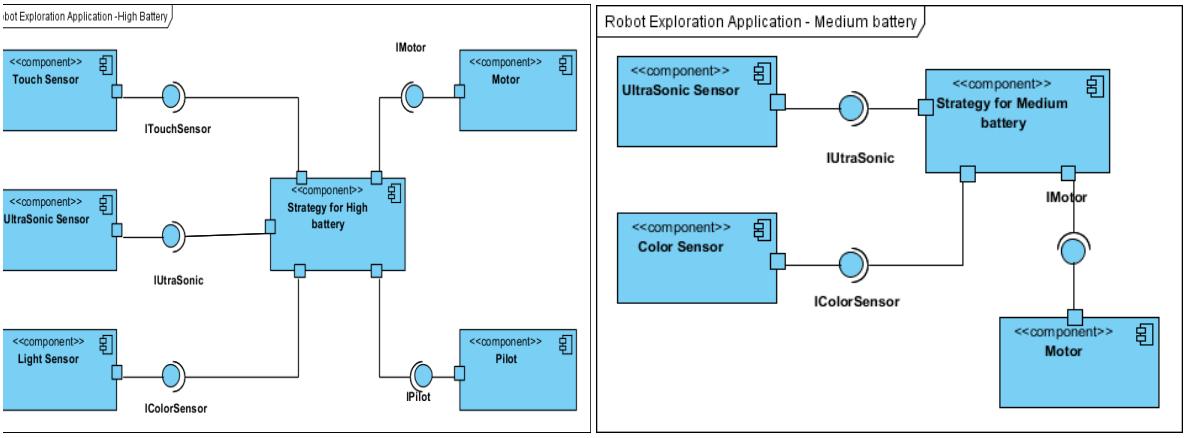
In designing this application, benefited from our architecture-based adaptation framework, we have been able to separate our business logics from those adaptation strategies described above. These two parts can be developed individually. In this section, we focus on the business logic design. For the application, its business logics are quite straightforward: discovery of unknown territory with its available sensors and actuators. The result of application structure is quite clear.

6.3.1 Robot explorer application

This application consists of one central exploration strategy component which makes use of a subset of the available sensors (touch sensor, ultrasonic sensor) as well as the robot's actuators (one motor and one pilot). The sensors are used to get environment data and the actuators are used to change sensors direction (via actuator pilot) or change position of the robot (via actuator motor) to get more data from the robot environment. The strategy component is the "brain" of the robot and it controls the robot behaviour.

As we already described in Section 6.1, this robot control platform is designed to work in changing environments. The robot system might encounter unexpected circumstances. Thus, it is not possible to have one strategy components fit for all possible environments. Therefore, rather than using the one-solution-for-all strategy which is rather unrealistic, several strategies are developed, each of them optimized for only one context. Each component shown here is implemented with our DRCom model and packaged as a single OSGi bundle. Different strategies will build different applications.

Figure 6-5 shows two different explorer application structures. The left part demonstrates the application structure for high battery context while the right of this figure shows that when the battery states changes, the strategy for medium battery can be used to construct a much simplified explorer application which best matches a context in which battery voltage is not very high.



(a) High battery (b) medium battery state

Figure 6-5. Application structures for different contexts

As we can see from Figure 6-5, the Explorer application can be built during the run-time by simply composing available sensors, actuators and strategies with the help of the structural modeller. By identifying the meta-data which describes each component's characteristics, this process can be done without the need to manually write composition code.

In this framework, in order to build the Explorer application, all the sensors and actuators are developed with the DRCom model and installed in the system simultaneously. Some of the components are developed by wrapping physical sensor and actuators, while others, such as the fake battery sensor, are implemented just for test purpose. Among these components, many of them provide the same type of the service contracts while being optimized for different environments. Specifically, there are three strategy components optimized for three different battery voltages ranges.

6.3.2 Supported component repository

As described in the previous section, this system supports different version of service component providers to provide the same service contract. Different developer can develop their own components – general or contextual optimized components and install these components into the systems.

In order to develop the context-aware exploration application, a set of components is developed and is put into our component repository. In Table 6-2, a list of the components available in this repository is shown. The same table also shows the descriptions for each component. Most of these sensors can be directly mapped to physical sensors. However, one special sensor – backup Touch Sensor, is created by reconstructing the robot light sensor so as to mimic touch sensor's characteristics. This *Backup Touch Sensor* is used in our run-time self-healing adaptation that will be described in Section 6.4.2. In order to illustrate the Explorer application development process, in the following section, the development of two basic business components – Touch Sensor and Strategy for high battery, is demonstrated.

Table 6-2. Components available for the explorer application

Component Name	Description
Light Sensor	The Light sensor plug-in uses a hardware light sensor adapter to detect the light level in one direction. This hardware also includes an LED for illuminating an object
Touch Sensor	The touch sensor component uses NXT robot Touch sensor to detect and report any collision sensed in the areas covered by its front arms. This Touch Sensor is also placed in the front part of this robot.
Back-up Touch Sensor	The back-up touch sensor component uses NXT robot Light sensor to detect and report any collision sensed in the areas covered by its back arms. This backup Touch Sensor is placed in the back side of this robot.
Sound Sensor	The sound sensor plug-in uses a microphone to detect and report the sensed noise.
Ultrasonic Sensor	The Ultrasonic sensor plug-in uses a hardware ultrasonic sensor adapter to can measure the distance from the sensor to something that it is facing, and detect movement.
Motor	This actuator is a component that controls a servo motors. This Motor is placed on top of robot with ultrasonic sensor. So it can turn ultrasonic sensor into different direction.
Pilot	This Pilot actuator can drive robot forward and backward, it can also turn robot into different angels.
Strategy High battery	This strategy gets light sensor and sound sensor data and move robot in a circle.
Strategy Medium battery	This strategy keeps the robot stay stationary. It rotates the motor over 360 degrees scanning its environment with the ultrasonic sensor.
Strategy Low battery	This strategy keeps the robot stay stationary. It only polls the data from light sensor.
Battery Sensor	This component polls the sensors value from robot's internal battery voltage meter.
Fake Battery Sensor	This component is for testing and demonstration. It implements the battery sensor interface and return a user-defined battery value to its clients.

6.3.3 Implementing basic modules in DRCom

The first business component – TouchSensor, is the one with the simplest DRCom implementation. This DRCom is packaged as a normal OSGi bundles however with management extension and meta-data. Firstly, we need to define its manifest file to specify the static java package based dependence (i.e., which libraries are used, with the exception of those packages already defined in the OSGi profile).

6.3.3.1 Meta-data for touch senor

The manifest is shown in Listing 6-4, which shows that component has two static Java package based dependences. One is *ua.mw.communication.commandprotocol,* which is the transmission library it needed to transfer commands between the two sides. The other one – *ua.mw.mw.robot.touchsensor* – is the class for touchsensor service interface definition.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ua.mw.mw.robot.TouchSensorImp
Bundle-SymbolicName: ua.mw.mw.robot.touchsensor.TouchSensorImp
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: ua.mw.communication.commandprotocol,
 ua.mw.mw.robot.touchsensor
DRCom-Component:  DR-INF/component.xml
…
```

Listing 6-4. Manifest File for TouchSensor DRCom model

This TouchSensor is also implemented as a DRCom component. Thus, the next step in its development is to design the meta-data for structural information – the service descriptor file. This descriptor specifies information related to the DRCom such as the name, the description, implementation class and provided/required services etc. (see Section 5.2)

Listing 6-4 shows the DRCom descriptor for the Touch sensor implementation. This meta-data shows that a single OSGi service – the ua.mw.communication. commandprotocol.SessionManager service, is required by the *Touch Sensor*. Cardinality of this service is 1, that means exactly one such service is mandatory required.

This meta-data also shows that this component provides two different services – one is the ua.mw.robot.touchsensor.TouchSensor service, the other service – ua.mw.communication.commandprotocol.Commandable – is the service for session management. The latter service is required by the Remote NXT model to serialize the commands to the sensors. Of course, the sensor should also implement the management interface. As it is native supported in the DRCom model, it is not necessary to specify it in the meta-data.

In order to simply the development of the DRCom component, one abstract class – AbstractCommandable – is provided. It can be extended by using class inheritance. Besides realizing the IManagement interface which enables the middleware to make "reversion of control", the AbstractCommandable class also provides predefined methods for business logics code, which provide the session management with support to get latest sensor values from the connection.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component
xmlns:drcr="http://win.ua.ac.be/~ninggui/drcr/v1.0"
immediate="true" name="ua.mw.robot.touchsensor.TouchSensorImp">
   <implementation
class="ua.mw.robot.touchsensor.impl.TouchSensorImpl"/>
   <reference bind="setSessionManager" cardinality="1..1"
interface="ua.mw.communication.commandprotocol.SessionManager"
name="SessionManager" policy="static"
unbind="unsetSessionManager"/>
   <service>
      <provide interface="ua.mw.robot.touchsensor.TouchSensor"/>
     <provide
 interface="ua.mw.communication.commandprotocol.Commandable"/>
   </service>
 </drcr:component>
```

Listing 6-5. The meta-data description for Touch Sensor DRCom

The TouchSensor component and its position in the class hierarchy are demonstrated in Listing 6-5. The implementation of TouchSensor is completed by realizing the logics that collect the sensor data from remote NXT robot side and process the response to generate sensor-specific values. In order to effectively communicate with remote NXT, a sensor needs to check the communication session status which is provided by the session service.

One benefit of this system design is that dependences are managed by the system run-time instead of by each individual component. System run-time will call the methods defined in the bind/unbind attributes to inject/unbind this dependence. As an example, when the SessionManager service becomes available, the setSessionManager will be called. Benefited from the service component model, component developers do not need to manually control these complex dependences. Thus, the development complexity can be greatly reduced.

6.3.4 Implementing the Strategy component

In this section, the meta-data for strategy component is demonstrated. As we might have many different strategy component installed, each optimized for a different situations, here only the strategy optimized for high battery conditions is shown. This strategy will drive robot around in a circle while polling sensor values. In this case, the strategy for high battery needs pilot actuator as well as touch, light and ultrasonic sensors, which are shown in Listing 6-6.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component xmlns:drcr="http://win.ua.ac.be/~ninggui/drcr/v1.0"
name="ua.mw.strategy.HighBattery">
   <implementation class="ua.mw.strategy.battery.HighBatteryStrategy"/>
   <reference bind="setPilot" cardinality="0..1"
interface="ua.mw.robot.pilot.Pilot" name="Pilot" policy="dynamic"
unbind="unsetPilot"/>
   <reference bind="setLightSensor" cardinality="0..1"
interface="ua.mw.robot.lightsensor.LightSensor" name="LightSensor"
policy="dynamic" unbind="unsetLightSensor"/>
  <reference bind="setUltraSoundSensor" cardinality="0..1"
interface="ua.mw.robot.ultrasensor.UltraSoundSensor"
name="UltraSoundSensor" policy="dynamic" unbind="unsetUltraSoundSensor"/>
  <reference bind="setTouchSensor" cardinality="0..1"
interface="ua.mw.robot.touchsensor.TouchSensor" name="TouchSensor"
policy="dynamic" unbind="unsetTouchSensor"/>
</drcr:component>
```

Listing 6-6. The meta-data description for Strategy DRCom- high battery

As we can see from this list, this component is implemented with qualified Java class name – ua.mw.strategy.battery.HighBatteryStrategy. It needs four different types of service interface. Firstly, as it requires the ua.mw.robot.pilot.Pilot service to drive robot around, the system run-time will find an appropriate service provider and inject/unbind its reference to the strategy component via its defined methods – setPilot /unsetPilot. Reverse-of-Control model is used here to allow component container to inject dependence between its managed components. Similarly, the other required sensors are specified in its meta-data. By parsing this meta-data, system run-time can build a global software architecture. By connecting selected installed components, different applications can be constructed during run-time.

6.4 **Adaptation behaviours design**

As we can see from Table 6-2, multiple components might provide the same functionalities while targeting different optimized application environments. These components can be installed into the Transformer system simultaneously, for instance multiple exploration strategy components optimized for different contexts are supported. Here, one of the key challenges for context-aware adaptation is to how to compose and configure an application and alter its structure and configuration in response to its changing environments or system errors. However, as discussed in Section 1.1.2, this challenge could not be effectively solved by an application-based adaptation.

In the previous section, an explorer application's business components are described and developed without the need to care about adaptation behaviours. This section will

illustrate how to use DSM to inject adaptation capabilities with multiple concerns outside the application business logics.

Two different adaptation behaviours are demonstrated in this section. Firstly, in order to optimized explorer‘s battery performance while getting maximum environmental data, applications should be recomposed according to the context status. This concern is implemented via a DSM for Battery-based Optimization. At the same time, this robot system is required to be fault-tolerant – recovering from errors of certain components. For instance, if the touch sensor in the front of the robot failed to response to poll commands, this system should be able to replace the touch sensor with the back-up touch sensor. Actually, this can be done in many component frameworks in which run-time compositional adaptation is supported, as these two components provide the same service interface. However, in order to keep application operate normally, domain-specific knowledge is needed to take accurate adaptation actions. For instance, as the backup touch sensor at the opposite side with respect to the position of the normal touches sensor, self-healing requires reversing the pilot actuator direction and making a 180 degree turn. As this adaptation requires domain-specific knowledge – the position of sensors, this kind of adaptations cannot be expressed by general self-healing algorithms [138, 155].

6.4.1 DSM With Battery-Based Context-Aware Adaptation

As each DSM is modelled as a Finite State Machine, only if it is able to receive those events in which it might interests, a DSM can successfully come out with its adaptation plans. As described in Section 4.4.3, this notification task is performed by an Event Reasoner. The DSM Battery will make context-specific adaptation when it is triggered by the change event sent by the Battery Event Sensor.

In order to demonstrate how the Event Reasoner send event out and how the DSM get notified by its assigned events from the system run-time, the implementation of a battery event reasoner is shown in the following section.

6.4.1.1 Battery Event Reasoner Plug-in

In order to trigger the adaptation actions, the adaptive middleware allows the user-custom *Event Reasoner* to be developed and late injected into system for monitoring unforeseen system metrics and raising notifications to the system run-time.

Meta-data for Battery Event Reasoner

The meta-data specifies that this battery event reasoner provides three types of event – ua.mw.event.battery.high, ua.mw.event.battery.medium and ua.mw.event.battery.low. When this EventReasoner plug-in is registered into system, these properties will be automatically read and monitored by system run-time. The source code for the meta-data is shown in Listing 6-7.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component xmlns:drcr="http://win.ua.ac.be/~ninggui/drcr/v1.0"
immediate="true" name="ua.mw.eventreasoner.batteryreasonerImp">
  <implementation class="ua.mw.eventreasoner.BatteryReasoner Impl"/>
  <reference bind="setBatterySensor" cardinality="1..1"
      interface="ua.mw.robot.BatterySensor" name="batterysensor"
      policy="static" unbind="unsetBatterySensor"/>
  <service>
    <provide interface="ua.mw.eventreasoner.BatteryReasoner"/>
    <provide interface=”ua.mw.eventreasoner.IEventReasoner”/>
  </service>
  <property name=”PROVIDED_EVENT_TYPES” value
    =”ua.mw.event.battery.high; ua.mw.event.battery.medium;
    ua.mw.event.battery.low” />
</drcr:component>
```

Listing 6-7. Meta-data for Battery Event Reasoner

Class diagram of Battery Event Reasoner

As described in Section 4.4.1, there are two levels of sensors – the simple sensors that only (proactively or passively) monitor systems events. The Battery Event Reasoner is in the second level. The abstractEventReasoner class is provided that can be extended to realize Event Reasoner component. It needs to implement the IEventReasoner interface which defines a methods – fireChaneEvent, to fire the event. It also provides methods getEventTypes to allow system reflect its raised Event type. As different DSM have different sets of interested events, these methods can help the system know whether a DSM required Event Reasoners are satisfied or not.

When the Event Reasoner read its interested sensor values, it uses the fireChangeEvent method to communicate context events to the middleware. Plug-in realizations also implement the activate and deactivate methods, which are automatically controlled by the middleware to optimize resource usage. The sensing process is resumed when activated and suspended when deactivated. The AbstractEventReasoner is a helper class that implements the basic methods, for instance, automatically reading this components' metadata. (i.e., the provided/required Event types, as specified in the service descriptor file, etc.)

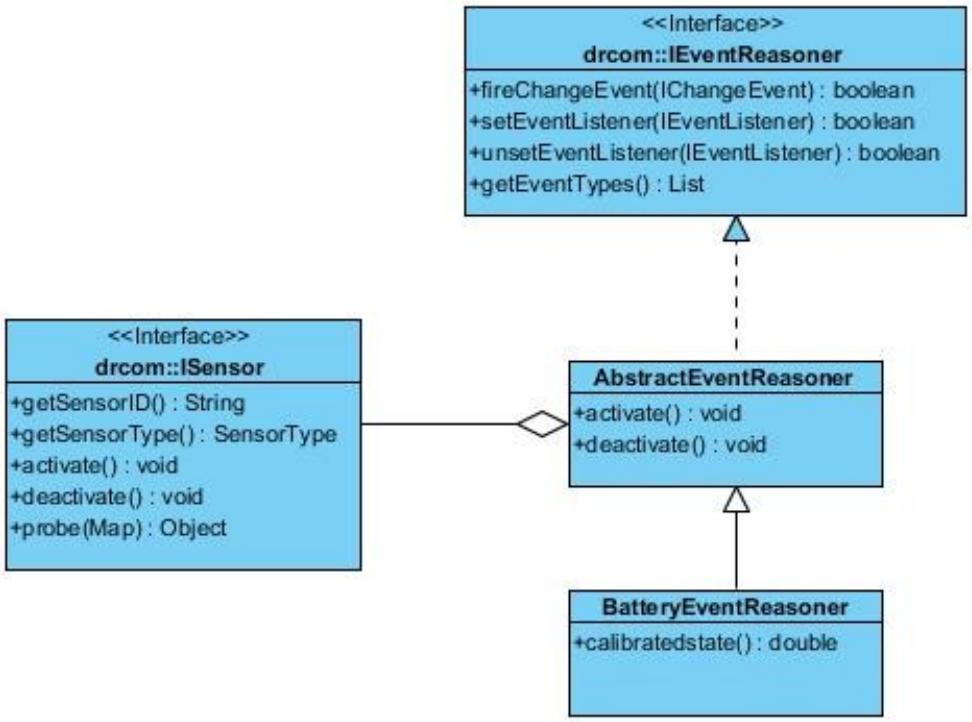


Figure 6-6. The class hierarchy of the Battery EventReasoner

Depending on various design considerations, one Event Reasoner might be interested in other change events. In this case, it can implement IEventListener interface to listen other change events.

The battery-reasoner plug-in class diagram is illustrated in Figure 6-7. The implementation of a battery-reasoner is completed by realizing the logic that monitors the battery sensors value for every 5 seconds. If the new sensors value respectively fits in one of the following three different domains – $(0,6)$, $[6,9)$ and $[9,12]$ – it will respectively raise the battery low, medium, and high event. This simple logic is shown in Listing 6-8.

```
public void run() {
    while ( true ){
    Battery s = mBatterySensor.get();
    float volatage = -1;
    if ( s != null )
        try {
        volatage = s.getVoltage();
        float volatage_cal= calibratevoltage(volatage);
        BatteryState cs_new =currentstate(volatage_cal);
            if (cs_new!= current_state)
                { current_state= cs_new;
                fireChangeEvent(getChangeEvent(current_state));
                }
                    Thread.sleep(Battery_Check_Interval);
            } catch (Exception e1) {
                    …
                    }
            }
```

Listing 6-8. Sample code for Battery Event Reasoner

In the Battery Event Reasoner component, when the battery voltage migrates from current range to another range, a corresponding event will be raised. However, as the battery voltage will also be influenced by the current load, in order to reduce the fluctuation, the *calibratedstate* method is used to come out with filtered results. Some efficient filtering algorithm, such as Kalman filter [84], can be used for this purpose. However, detailed discussion is out of the scope of this thesis. When the calibrated result differs from the old state, the *fireContextChangeEvent* will be called to send notifications to the system run-time.

6.4.1.2 Battery DSM Plug-in

As shown in the previous section, different composition of exploration programs are optimized for specific battery conditions which changes during run-time. Hence, a context-specific adaptation logic is needed to express the strategies and how to select the right application composition.

Firstly, three different battery-optimized exploration program structures are designed. Two of them – the structure for high battery and the structure for medium environment – are shown in Figure 6-5.

Battery optimized software architecture

High Battery: this application drives robot around in a circle while polling the touch, light as well as ultrasonic sensors. This modeller uses touch sensor, ultrasonic sensor and light sensor, as well as motor actuator and pilot actuator. The constructed application has a good data fetching rate, with high power consumption though.

```
public AdaptationPlan  resolveAdaptationPlan (ISystemContext
systemcontext, ContextChangeEvent ev)
    {
      AdaptationPlan plan = new AdaptationPlan(systemcontext);
      List newCompsProps =new ArrayList();
      List newlyenabledComps;
    if ev.getEventTopic.toString.equals(”ua.mw.event.battery.high”)
    newlyenabledComps = resolveBatteryHigh
(systemcontext.getInstalledComponents());
   else if
 (ev.getEventTopic.toString.equals(”ua.mw.event.battery.medium”))
  newlyenabledComps =
  resolveBatteryMedium(systemcontext.getInstalledComponents());
  else if
 (ev.getEventTopic.toString.equals(”ua.mw.event.battery.low”))
  newlyenabledComps =
  resolveBatterylow(systemcontext.getInstalledComponents());
       plan.setEnabledComps(newlyenabledComps);
       plan.setCompProps(newCompsProps)
          return plan;
    }
 resolveBatteryHigh(List enabledComponentConfigurations)
if (enabledComponentConfigurations.isEmpty()) {
           return Collections.EMPTY_LIST;
        }
    List resolvedSatisfiedComponentConfigurations   ;
    //Select the right components and put them into
 //resolvedSatisfiedComponentConfigurations  ;
If(cc.getComponentDescription().getName().contains("ua.mw.strategy.")
)
    {if (cc.getComponentDescription().getName().indexOf("HighBattery")
== -1 ) {
    it.remove();  }
     return     resolvedSatisfiedComponentConfigurations.isEmpty()     ?
Collections.EMPTY_LIST : resolvedSatisfiedComponentConfigurations;

    }
```

Listing 6-9. DSM for battery based application reconstruction

Medium Battery: The robot stays stationary; it rotates the motor over 360 degrees scanning its environment with the ultrasonic sensor, and collect data from ultrasonic sensor and light sensor. It uses comparably less power as it only needs one motor. Results are less accurate.

Low Battery: The robot stays stationary; it polls with the light sensor and sound sensor. In this model, not much information will be retrieved.

As different exploration strategies might fit for different environments, one natural requirement is to construct the exploration application by selecting the most appropriate composition for current environment. However, although several solutions which try to provide a general component solution are proposed, without domain-specific knowledge, their partial usage is rather limited. In our Transformer framework, DSM is used to express such domain-specific adaptation knowledge, which is denoted as DSM Battery.

As described in previous Section 4.5.1, this DSM Battery is able to control installed components' lifecycle as well as perform property configuration. Listing 6-9 shows the code for this simple modeller implementation. Three main methods are designed for different battery status – low, medium and high.

The reasoning process of this DSM can be summarized as follows: While DSM for battery's resolveAdaptationPlan is called, it firstly checks whether the event type is among its supported events. If the change event is among its support events, for instance, if the change event is ua.mw.event.battery.high, then, it will call resolveBatteryHigh to calculate which components should be enabled. The input parameter systemcontext is the reference towards system meta-object model. By using systemcontext. getInstalledComponents() method, the DSM is able to get a list of currently installed components. For instance, for the strategy components, there are three different strategy components installed – strategy.highbattery, and strategy.mediumbattery and strategy.lowbattery. In the DSM design, the resolveBatteryHigh method selects the strategy for high battery context(strategy.BatteryHigh) component and disables the strategy for medium battery and low battery. To do so, this method just simply removes these two strategy components from the enabledComponentList. It then sends this list to the DSM Battery and this DSM will return the new adaptation plan with selected enabled components. Likewise, the resolveBatteryMedium method only selects the strategy for medium battery and disables others. If the battery reduced from high state to medium state, the resolveBatteryMedium method will be invoked to generate new system adaptation plan. The configuration of exploration application will be switched from the high battery structure (shown in the left side of Figure 6-5) into the structure demonstrated in the right side of Figure 6-5.

As displayed here, benefited from *Separation of Concerns*, an DSM implementation only needs to select the right components to be activated and set appropriate properties, which can be implemented in a rather simple and cost-effectively way.

6.4.2 DSM with self-healing mechanism

As DSM - battery is focused on the battery-based contextual application structure optimization, there is no explicit fault-tolerance supports for this application. For instance, in this system, two different touch sensors are deployed to provide the same functional interface. Due to the fact that they have totally different location on the robot, the exploration application cannot simplistically replace the primary touch sensor with the

back-up one. Other domain-specific knowledge – the pilot actuator' direction, need to be used to make correct self-healing adaptation. In this section, DSM – self-healing and its corresponding event monitor are introduced.

```
<?xml version="1.0" encoding="UTF-8"?>
<drcr:component
xmlns:drcr="http://win.ua.ac.be/~ninggui/drcr/v1.0"
immediate="true" name="ua.mw.eventreasoner.selfhealing.Event">
   <implementation class="
ua.mw.eventreasoner.selfhealing.event.EventImpl"/>
    <service>
     <provide interface=”ua.mw.eventreasoner.IEventReasoner”/>
     <provide
interface=”ua.mw.eventreasoner.selfhealing.TimeEvent”/>
   </service>
  <property name=”PROVIDED_EVENT_TYPES” value=
   ”ua.mw.event.Selfhealing.TimeEvent”/>
 </drcr:component>
```

Listing 6-10. Meta-data for Self-healing Timer

6.4.2.1 Monitor for Self-healing

In order to monitor a component's status, a self-healing component needs to periodically check all enabled components. The status monitor plug-in is designed to send periodic event for the system to check those components. Implementation of this component is quite straightforward. A simple Java thread is used to send ua.mw.event.selfhealing.TimeEvent to the system run-time. It also provides the IEventReasoner interface so as allow the system run-time to register for the event notification. Listing 6-10 shows the meta-data for this self-healing reasoner.

6.4.2.2 DSM for self-healing

The Adaptation logic of this subsection demonstrates how domain-specific knowledge – the position of the primary touch sensor and back-up touch senor and their relationship with the pilot actuator – is used to guide the self-healing adaptation process. As we already pointed out in Section 6.2.1.1, due to the space limitation, the physical configuration of the primary touch sensor and the backup touch sensor locate in opposite to n the primary touch sensor goes down. Otherwise, the Explorer application will not function correctly. In this chapter, the DSM for touch sensor self-healing is implemented.

The DSM contains the following adaptation design requirements:

Triggering Events

1. The DSM should be able to get the periodic event from *Monitor for Self-healing* to periodically check component state. It will emulate all enabled components and save their run-time status into its local cache.

   2.It must be able to monitoring the state of all installed touch sensor – enable, disable, depending on the available components. In this demonstration, the two different touch sensors are selected.

Adaptation Logics

1. On receiving the TimerEvent, it will emulate all enabled components and save their run-time status into its local cache. By doing so, it can check whether a component is working correctly or not (we assume that when a component does not respond to the getProperties inquiry via the IManagement interface, then this component need to be repaired).
2. When the primary touch sensor is disabled or has error in responding the periodic check via IManagement interface, if the backup touch sensor exists, the component should be replaced by the backup touch sensor;
3. After the replacement of this touch sensor, the pilot direction should first make a turn and its direction should be reversed;

6.4.2.3 Meta-data of DSM self-healing

As we can see from , the DSM Self-healing will listen for not only TimerEvent but also the event from the framework, but also for the three kinds of ServiceEvent –REGISTERED, MODIFIED and UNREGISTERING. By monitoring these service events, this DSM gets up-to-date information of system status. The meta-data description file for this self-healing DSM is shown in Listing 6-11 .

In order to get an up-to-date knowledge of system current installed components and make a run-time Explorer application, this DSM also monitors basic system events, for instance, adding a new component, removing certain component. As the system run-time works as an event bus, for each DSM, it won't need to be manually coupled with one *Event Monitor*s: any *Event Monitor* which provides this event type can be directly used. As shown in Listing 6-11, this DSM interested four types of events.

```
….
<reference bind="setTimerSensor" cardinality="1..1" interface=
“ua.mw.eventreasoner.selfhealing.TimeEvent" name="batterysensor"
policy="static" unbind="unsetTimerSensor"/>
    <service>
       <provide interface=”ua.mw.DSMResolver.IDSMPlugIn”/>
   </service>
  <property name=”REQUIRED_EVENT_TYPES” value=
   ”ua.mw.event.Selfhealing.TimeEvent;
   org.osgi.framework.ServiceEvent.REGISTERED;
   org.osgi.framework.ServiceEvent.MODIFIED;
   org.osgi.framework.ServiceEvent.UNREGISTERING ”
   />
…..
```

Listing 6-11. Snippet of Meta-data for Self-healing DSM

6.4.2.4 Healing logics

While the TimeEvent is received, the system run-time checks the event topics with existing DSM requirements. It firstly queries the DSM Manager about enabled DSM, then it invokes all the enabled DSM which interested in the event. For the event with ua.mw.event.Selfhealing.TimeEvent type, only DSM Self-healing is interested in this event. Its resolveAdaptationPlan() method will be called to generate adaptation plans. The Listing 6-12 shows how DSM Self-healing for the touch sensor healing is constructed:

The DSM first checks the event type, and then it checks the primary touch sensor’s status. If the touch sensor returns false in the checkcomponentruntimestate methods, the backup-touch sensor will then be selected. The new properties of the pilot actuator are also set. The method setProperty(“turn”, “true”)makes the robot make a turn and setProperty(“direction”, “-180”) make the pilot actuator change direction. Then the new aptation plan is created and returned to the system run-time. Of course, the code shown here is only a fraction of the whole adaptation logics. For instance, system will go back to primary touch sensor if touch sensor goes back to work. This logic is handled by the service.REGISTERED event.

```
 public AdaptationPlan  resolveAdaptationPlan (ISystemContext
systemcontext, ContextChangeEvent ev)
   {
     AdaptationPlan plan = new AaptationPlan(systemcontext);
     Map newCompsProps =new TreeMap ();
  if ev.getEventTopic.toString.equals
        (”ua.mw.event.Selfhealing.TimeEvent”)
     {
      List enabledComps= systemcontext.getEnabledComponents();
      Iterator it = enabledComps.iterator();
       while (it.hasNext()) {
     ComponentConfiguration cc=(ComponentConfiguration)it.next();
     If (checkcomponentruntimestate(cc)==fasle)
     {
      if (cc.getname.equals(ua.mw.sensor.primarytouchsensor))
      {
      Property pilot=getPilotProperties(enabledComps);
      Property newpilot= new Property();
      newpilot. setProperty(“turn”, “true”);
      newpilot. setProperty(“direction”, “-180”);
      plan.addproperties(pilot.getname, newpilot);
     }
     }
    }
     if ev.getEventTopic.toString.equals
        (”org.osgi.framework.BundleEvent.STARTED”)
       ….
  if ev.getEventTopic.toString.equals
        (”org.osgi.framework.BundleEvent.STOPPED”)
       …
  return plan;
}
```

Listing 6-12. Snippet of Self-healing DSM code

6.4.3 Dynamic UI

In order to visualize certain parts of the application on the computing side and enhance its usability, some extra components are designed to have better control and debugging capabilities. This user interface is also built based on our Transformer framework. For each sensor, an optional UI component can be deployed into the system to visualize its control interface. User can directly monitor sensor’s data and set its properties, for

instance, for sensor calibration purpose. for each functional sensors/motors and strategy component, an UI component is designed.

```
….
<reference      bind="setLightSensor"      unbind="unsetLightSensor"
cardinality="1..1"  interface="ua.mw.robot.lightsensor.  LightSensor"
name="LightSensor" policy="dynamic" />
    <service>
       <provide interface="ua.mw.gui.view.View"/>
    </service>
….
```

Listing 6-13. Excerpt of LightSensorUI component meta-data declaration

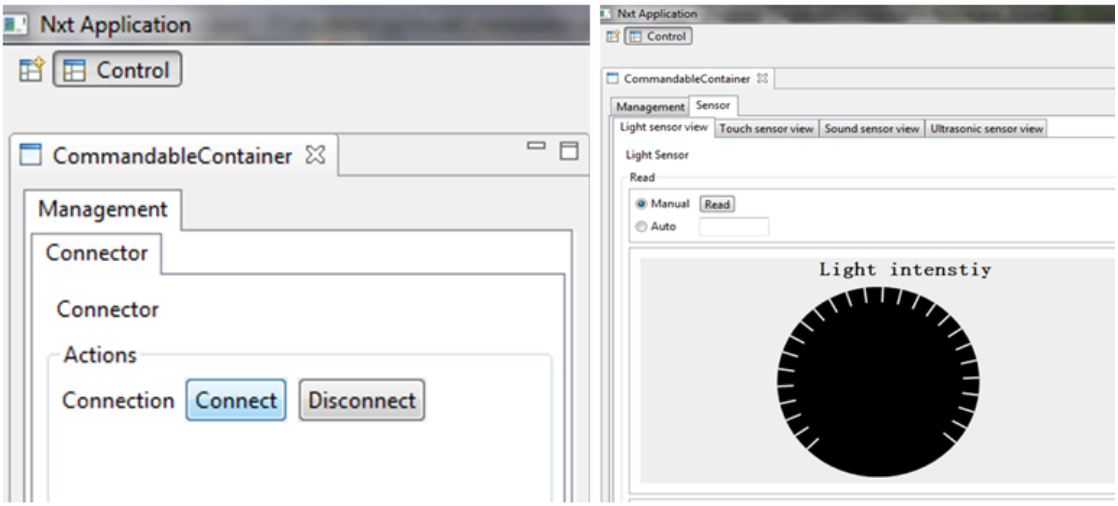
Listing 6-13 is an excerpt of meta-data attached to a Light Sensor UI component. This component provides the *ua.mw.gui.view.View* service to the system, while it requires LightSensor service.

When functional dependence of LightSensorUi is satisfied, it means the LightSensor is activated. Thus, its corresponding UI component will be shown. In other words, when its required sensor/actuator is disabled, the corresponding UI component will be automatically disabled. This dynamicity is automatically managed by the structure modeller. In resource-scarce environments, a DSM can selectively disable certain sensor/actuator components to save resources. The structure modeller will then disable corresponding UI components which depend on those disabled components. Figure 6-8 shows two screenshots of our UI. When NXT application has not built the connection with NXT robot yet, no required connection service provider exists. As sensors and actuators declare their dependence on the connection services, they will not be activated by system run-time. As UI components depends on their corresponding sensors and actuators, these UI will not be shown, just as shown in Figure 6-8a. While the connection is built, the UI for the activated sensors will be initialized and shown as illustrated in Figure 6-8b.

6.4.4 Deployment

The two DSM plug-ins, their corresponding Event Reasoners and the robot exploration components – sensors, actuators and strategies – are packaged as individual JAR-based OSGi bundles. These bundles are installed along with the modular middleware (which is packaged as an OSGi bundle itself).

When those DRCom plug-ins are installed, their meta-data will be automatically checked and parsed by the Transformer system. Their provided interfaces and required interfaces will be checked to see whether they are functionally satisfied. The DSM plug-ins can be also installed during run-time and treated as normal OSGi bundles.



(a) UI before connection (b) UI after connection

Figure 6-7. Dynamic UI

Eclipse Application and GUI

The demonstration application is built by Eclipse and Equinox OSGi implementation [7]. The Eclipse application framework integrates well with OSGi as it uses the OSGi specification as its underlying supporting platform. The goal of this application was demonstrate how Transformer framework and its supporting middleware can be integrated into an existing Eclipse application – turning traditional fixed application into a run-time composed one. Each component can be individually installed and used to build the exploration program. Section 6.4.3 demonstrates how dynamic UI is achieved by designing two major GUI widgets

Bundle control and monitoring

To easily control the bundles a widget was created that list all bundles and their current status. Another widget is designed to allow bundle enabling and disabling via a single button. The widget main purpose is to provide a substitute for the command line based bundle management – the interface provided by Equinox to do the bundle management. The widget thus speeds up the testing and simplifies the bundle management for end-users. Next to this bundle monitor system, another flexible control panel was developed which works as container for possible UI components. This control panel itself is empty but can be dynamically filled by custom GUI components provided by other bundles for instance those that provide the GUI for the sensors. This was is already be shown in the previous section.

At right side of this screenshot, you can find the “Services” view that lists all installed bundles which can be individually enabled/disabled. It also shows the status of each component.

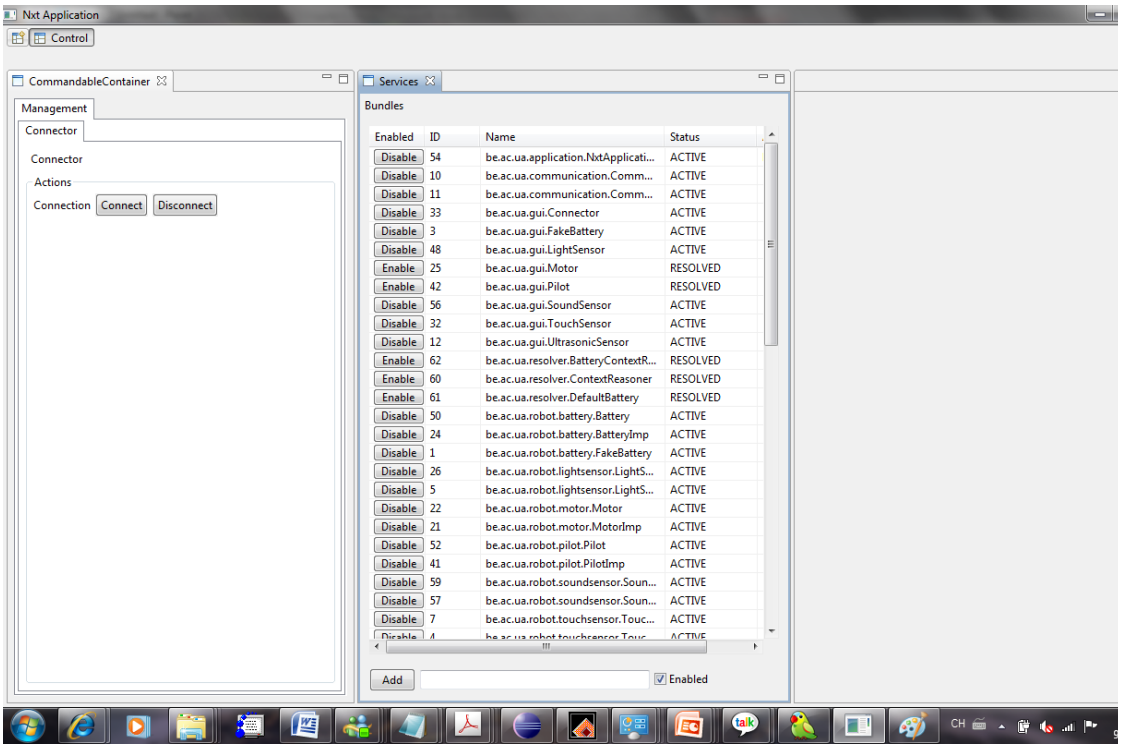


Figure 6-8. Bundle control and monitoring

Run-time installation

In this application, a new component can be run-time installed directly though this control platform. To introduce a new UI element, the user only needs to type in the new component URL and click the install button. The URL based installation means that component can be in the local disk or in a remote server. This component will be automatically downloaded, unzipped and added into the installed component list.

6.4.5 Experiences

In the reported experience, three different aspects are demonstrated. Firstly, the dynamic UI is demonstrated. A UI component will be automatically changed of status by system run-time when its corresponding sensors are enabled or disabled. Secondly, the run-time deployment of application business components – Sensors, Actuators, Strategy components – is illustrated to demonstrate the scalability and flexibility in introducing new capabilities into the system. Finally, the DSM are deployed during run-time, first with DSM Battery then with DSM Self-healing, the Robot Explorer application structure and configuration change with the context and installed DSM. Adaptation videos can be downloaded from http://win.ua.ac.be/~ninggui/video/Final.m2v.mpeg.

6.5 Conclusions and future work

This chapter has presented how to apply our Transformer framework and its supporting middleware architecture into the design and implementation of an autonomous NXT robot control platform. It demonstrates the design benefits of the adaptation framework

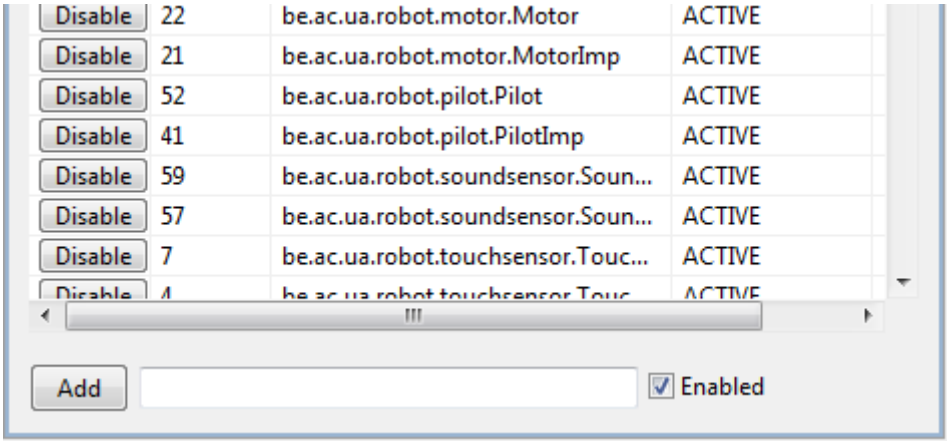


Figure 6-9. Run-time installing new components

described in chapter 4 and the modular middleware architecture described in chapter 5. Followed on the detailed implementation process and decoupled DSM modular support, an autonomous robot explorer software application with multiple adaptation behaviour capabilities is built. Later, it also described the dynamic user interface and the flexibility in installing new components during run-time.

The demonstrated case study application—the Context-aware Robot Exploration application— was used to demonstrate how domain-specific adaptation logics can be developed and run-time integrated into existing exploration application. Firstly, in order to integrate resource-limited NXT robot into Transformer framework, a remote NXT model is introduced to build a proxy between NXT robot and its corresponding DRCom components. Then, a DSM for battery optimization demonstrates the software run-time recomposition characteristic. It selects the most appropriate components to be activated and *Structure Modeller* manages the dependence between components. It also shows how a DSM can be notified with user-customized event (battery event), and how to develop the corresponding *Event Reasoner*.

The second DSM demonstrated is DSM for self-healing. This DSM will periodically checks component run-time properties and stores them into its local repository. When it identifies certain problems, for instance, the primary touch sensor going down, it switches to the back-up touch sensors constructed via a modified light sensor. As the back-up touch sensor has different location with respect to the primary touch sensor (backup touch sensor locates at opposite side of robot), in order to keep exploration strategy unchanged, this DSM needs to change the pilot component's configuration. This case study demonstrates how to use the most appropriate touch sensor as well as component parameter-based adaptation.

In this chapter, the dynamic user interface is also implemented as value-added features of our framework. This dynamic UI can be automatically reconfigured during run-time according to system current enabled sensors/actuators.

Chapter 7

Thesis Evaluation

In this Chapter, the work introduced in this thesis in respect to both framework design as well as middleware implementation is evaluated. In the previous chapters, in Chapter 4, the Transformer framework was presented to extend the SoC paradigm into adaptation modules design and integration, while in Chapter 5 we detailed how to design a modular middleware system to support such framework. Chapter 6 described a practical application in using this framework and its supporting middleware.

In the following sections, key metrics using in several projects are used to evaluate with the proposed framework design and middleware architecture, According to the publications and projects that were surveyed, the functional requirements for adaptive frameworks with multiple and evolving adaptation concerns have been identified[121]. It should be pointed out that it is very hard to provide an extensive quantitative comparison towards software design and implementation (as many implementations are not available or they are used in different environments). Therefore, this section mainly provides qualitative analysis rather than quantitative analysis. For the non-functional requirements, in order to provide a complete and fair analysis, this thesis selectively chooses the evaluation many commonly used metrics, more specifically metrics used in the De Florio's PhD thesis[56] and Paspallis's PhD thesis[121]. The base of the selected metrics is from the claim presented in Chapter 1: "This thesis claims that systematic combination of adaptation logics modularity, appropriate adaptation evolution method and adaptation composition mechanism can make the adaptation module development cost-effective, efficient and easy of usage".

From this claim, functional requirements are identified as follows: modularity, adaptation composition methodology and conflict resolution mechanism. While the

non-functional requirements include metrics such as cost-effective , light-weighted and easy of usage. In order to make more complete comparison, some implementation features will also be discussed and evaluated.

7.1 Functional requirements

7.1.1 Modularity

As different applications have different requirements and capabilities for different execution environments and deployment platforms [11, 74, 138], in order to facilitate the system deployment and configuration, a modular approach becomes a nature choice.

As our thesis targets changing and possibly unforeseen environments, it is important to have new adaptation modules deployed that match the new environment. Modular architectures provide more resource-efficient customization solutions to the evolution of the adaptation behaviour of self-adaptive applications.

Two level of Modularity

The modularity requirement is one the basic design principles for the modular middleware implementation. The modular design is exhibited in two different aspects.

Firstly the modular design is used to construct application business logics that are important for easy application construction. A DRCom component model is proposed as a basis on top of which to build various applications. At the same time, the design guideline for modularity is used for the middleware itself. As the middleware is constructed by SOA component model, the middleware can be easily configured to different system configurations by installing, updating or uninstalling middleware components. In Section 5.4.3, it is shown that this modular design enables the middleware to have different configurations. This feature makes our solution especially fit for fast changing environments, e.g. mobile computing.

Another additional feature of this modularity is that the system adaptation behaviour is also modular. This adaptation modularity is achieved by extending the SoC design paradigm from application business logics to the design of adaptation components. Due to this modularity, system global adaptation behaviours can be constructed both at design-time and at run-time. For the former one, this solution allows DSM components to be started together with system run-time and perform adaptation tasks. DSMs can also be installed during run-time and selected according to the context requirements. System administrator just needs to install the right set of DSMs with the required adaptation functions, without the need to manually compose those adaptation behaviours. The *DSM Manager* together with system basic run-time chooses the right set of DSMs to guide system adaptation behaviour.

In addition to the simplified development complexity, this modular approach can more efficiently perform unit tests for adaptation components as DSM only implements a small set of adaptation behaviours rather than provides a full-fledged adaptation solution. It also

facilitates the reuse of existing adaptation research works that focus on particular contexts as they can be easily integrated as DSM.

7.1.2 Adaptation evolution support

In our platform, a systematic mechanism for contextual adaptation behaviour evolution is provided. Rather than providing a standalone adaptation modeller to deal with multiple contexts, a system adaptation modeller is constructed by contextual selected DSM (see Section 4.2). This run-time adaptation composition provides the foundation for adaptation evolution.

At the same time, modular middleware architecture is designed to provide software engineering support for contextual adaptation evolution. Each DSM is implemented as a pluggable component which can be added, updated and removed during run-time (see Section 5.2.3.2).

This adaptation evolution is checked with both simulation scenarios (see Section 5.6) and an autonomous NXT control platform (see Section 6.4). This shows that such contextual adaptation evolution can be effectively supported by our Transformer framework as well as its supporting middleware implementation.

7.1.3 Adaptation composition

Adaptation composition is support by two different aspects: the ontology of used adaptation actions and a contextual adaptation fusing algorithm.

7.1.3.1 Ontology of adaptation actions

In our system design, application specific sensors and actuators are supported in this framework. In order to effectively identify the relationships between different actions, our middleware provides an *IActuatorModel* interface to allow the programmer to specify their relationships.

In current prototype, the actuator model presented in Section 5.3.4.1 provides a basic, semantic-dependent solution with three actuators. It works fine in our simulation scenarios and robot control platform. However, further improvements are scheduled to support more complex actuator models. In the software engineering approaches, the SOA architecture adopted in our middleware allows future more complex actuator models to be easily inserted into the middleware; one of our ongoing works is to use OWL language and toolset to describe actuators' characteristics and the dependence among them. Further Inference of the relationships can also be done with existing OWL reasoning engines [9, 16].

7.1.3.2 DSM fusion algorithm

In the Transformer framework, the model fusion algorithm is an important part of the system. In our current implementation, the algorithm is designed with respect to three key factors: DSM features, context matching degrees and the ontology of adaptation actions.

Compared to the utility-based solutions, this approach can provides adaptation behaviours better meeting the contextual requirements.

In the Section 5.3.4, the basic rule set of conflict resolution logic are provided and the performance of the DSM fusion algorithm is tested in term of fusing time. Furthermore, in our system design, rather than providing a full-fledged solution towards this problem, we identified this problem and explicitly separated this process from other decision processes and by using our service oriented model, made it exchangeable to allow future changes. Compared to the adaptation approaches which hide the complexity of conflict resolution in the binary code, our approach provides clearer separation and more flexibility.

7.2 Non-functional requirements

The non-functional requirements identified here are: cost-effective, light-weighted and easy of usage. Some of these requirements were greatly facilitated by the underlying OSGi component framework.

7.2.1 Cost-effective

The requirement of cost-effective is demonstrated from two aspects: code reuse and adaptation behaviour building efficiency.

7.2.1.1 Code Reuse

For code reuse of our solution, two different types of reuse are investigated. In one aspect, we assess the extent to which the common middleware infrastructure can be reused for different self-adaptive applications. In the other aspect, we check to what extent the customized parts (DSM modellers) of our Transformer framework can be reused during context changes.

A basic measure of engineering effort, the "source line of code" (SLoC), gives an indication of how the Transformer framework can alleviate the engineer from implementing the basic adaptation functionality and thus the development time. As of Aug. 2010, SLOCCount measured 9,754 total physical source lines of Java code, excluding comments, blank, and non-essential lines. These ~ 10 kSLoC were developed over a period of two years, excluding the initial prototyping and research time. In the beginning of our research, the framework was used for the TV & Recording scenarios (see Section 5.5), and then later reused in the NXT robot control platform (see Chapter 6). It was almost entirely reused as binary code to add self-adaptation capabilities to the NXT robot control system except for the DSM module and application-specific monitoring modules.

For adaptation logic reuse, as described in Chapter 5, during context changes, adaptation logics are created by composing DSM modules. Only missing adaptation behaviours will need to be developed. Thus, this enhanced reusability can greatly simplify the adaptation development complexity. For instance, in order to change from the TV

optimization & Self-healing adaptation behaviour to Recording Optimization & Self-healing, only one Recording optimization DSM needs to be developed which accounts for about 200 SLoC compared to the more than 2500 SLoC required when the adaptation codes are mixed together.

7.2.1.2 Efficiency in adaptation behaviour development

One of the major goals of this research is the simplification of adaptation behaviour developments. Simplified adaptation development makes our solution capable deal with the changing contexts in an easy and efficient manner. In order to fulfil this purpose and reduce the complexity in building adaptation modules, several design choices are made. Firstly, the ontology of adaptation actions is built. This ontology allows adaptation conflicts to be easily identified. Secondly, the Model Fusion module is proposed to explicit deal with adaptation module fusing problem. Thirdly, a contextual conflict resolution algorithm is proposed to simplify the fusing process. All these design allows DSM developers only need to focus on one particular adaptation behaviours. Thus, the complexity of adaptation behaviour development can be largely reduced.

As it is not feasible to evaluate the middleware with a well accepted quantitative measure, this thesis uses case based study to demonstrate the efficiency in adaptation developments. In our NXT robot case study, many adaptations DSM are developed with less than 200 lines of code. Due to this simplicity, after the students get familiar with Transformer, they can develop and test similar DSM within several man-hours. These experiences show that the complexity in bringing/removing a new adaptation feature is much simpler than in the traditional approaches.

7.2.2 Light-weighted

This requirement is evaluated from two different perspectives: resource efficiency and adaptation module complexity.

7.2.2.1 Resource efficiency

In the adaptation framework and its supporting middleware, the “resource efficiency” is demonstrated in two different aspects.

First, supported by the modular design, our middleware solution provides a lightweight and easily tailorable framework, capable of introducing adaptation behaviours with little resource consumption. In Section 5.6.5, it is shown that the memory overhead introduced from our framework is fewer than 300KB. It can be further reduced by using small memory techniques such as data compression [113].

Second, as described in Section 5.3.3.3, not all the DSMs will be initialized and used. The DSM Manager will dynamically activate and deactivate when their context matching degree above certain threshold. As the context matching degree can be directly calculated from the meta-data, this calculation does not requires the initialization of corresponding DSM., this mechanism can also effectively reduce the memory usage.

7.2.2.2 Light-weighted adaptation module

In this thesis, by extending SoC rationale to the development process of adaptation modellers, the adaptation logic can be implemented by light-weighted components. This is achieved from both framework design and the middleware implementation.

The Transformer adaptation framework described in Chapter 4 constructs system adaptation logics by using small, domain-specific modellers. Each modeller only implements typically one or limited domain-specific adaptation behaviours. In Chapter 5, these domain-specific modellers are implemented as run-time pluggable components which can be developed and deployed individually.

For a typical DSM component, as it only needs to deal with small domain-specific adaptation behaviour, their implementation can be very light-weighted. For instance, the DSM-TV described in Chapter 5 can be implemented within 100 lines of code and the size of this adaptation module is less than 3KB. The other more complex DSM, the DSM used in the NXT robot control platform is also less than 6KB. This is because the complex system architecture maintenance is implemented by the system run-time and structural modeller. This design allows light-weight adaptation modules to be developed and deployed.

7.2.3 Easy of usage

The Transformer framework and its supporting middleware have been used for two master projects. In both project, the students have no pre-knowledge even for the basic concepts such as what is middleware, self-adaptive software. They are given the thesis project to develop self-adaptive software based on Transformer framework.

The results proved to be encouraging. Within 2 months, they effectively can use Transformer to develop adaptive software. In the NXT robot control platform, Pieter-Jan, the master student developed four adaptation modules within 2 weeks. This, to some extent, proves our solution is easy of usage. Furthermore, this framework also supports component dynamicity and provides a continuous deployment and reconfiguration supports to simplify its usage.

7.2.3.1 Dynamicity support

In this middleware, dynamicity support is provided in three different aspects. The application business component management, the dynamic architectural model generation, and the DSM dynamicity support.

Firstly, our middleware architecture allows the application business component to be installed and activated at run-time. In our implementation, this dynamicity support is implemented by reusing the services provided by OSGi framework in which bundles can be installed, uninstalled during run-time.

Secondly, our middleware architecture allows the architectural model to be dynamically generated by structural modeller as it constructs the system architectural model – installed components, connections between components by analyzing

component's meta-data as well as monitoring component lifecycle events (see Section 5.3.2).

Thirdly, DSM dynamicity is another example of our system's ability to fulfil this requirement. The DSM Manager takes control of the lifecycle of individual DSM modellers. It monitors the changes of available DSM and dynamically enables those applicable to the current context.

7.2.3.2 Deployment and configuration support

In order to facilitate the end-users, the adaptive system should be easy to deploy and simple to configure/reconfigure. This was accomplished by developing and package system modules as OSGi bundles. This enables our middleware directly reuse the continuous deployment service provided by the OSGi platform. Thus, system deployment process can be largely simplified. In the NXT case study, a UI for bundle management is developed to allow end-user to use graphical user interface to install, activate and deactivate components (see Section 6.4.4).

Each component has meta-data to describe its functional dependence and properties. This design makes it much easier for an end-user to change the configuration of a specific component before its installation. During run-time, the general management interface can help users to change the component run-time values (see Section 5.2.1).

7.2.4 Scalability

Here, the scalability refers to the ability of architecture to gracefully accommodate for an increasing number of components. As it was already mentioned, the design of a distributed adaptation framework is beyond the scope of this architecture.

With regard to local scalability (i.e., in terms of the number of installed components as well as installed DSM), it is argued that the middleware architecture offers a scalable solution. The DSM is designed in a non-intrusive way, as it only performs simple lifecycle management as well as simple property manipulation. It will not intercept the functional calls between functional depended components – an intrusive designed used by Quo project [102]. Experience shows that this solution introduces little performance overhead during application execution. Section 5.6.3 also shows that it can still have good performance when more than 100 components installed.

Supported by the underlying OSGi component framework, numerous components can be deployed and handled, constrained only by the resources and the capabilities of the deployment platform. The adaptation mechanism is triggered only when a component lifecycle is changed or there are required events sent by Event Reasoners. Both mechanisms are normally triggered infrequently. Thus, from the perspective of local-scalability, the middleware architecture is highly scalable.

From the discussion of the previous sections, it has been shown that our framework and the modular middleware implementation have satisfied most of requirements

identified during Section 3.4. Then, in the following section, design issues and possible limitations are discussed.

7.3 **Comparisons with existing projects**

We compare Transformer with five other different adaptation frameworks/solutions from two major perspectives: framework realization and adaptation concerns.

As can be seen from Table 7-1, except for the AAOP project that uses aspects to achieve adaptation, the target for adaptation is focused on the component level. All the considered projects use external adaptation loops that isolate adaptation logics from the application business logic. Table 1 also shows that most solutions are generic and can be used for different adaptation requirements. These similarities show that these design choices have been adopted by most approaches and are becoming a general practice.

However, in terms of the problem of how to build adaptation behaviour, much less agreement has been reached among researchers. In terms of separation of concerns, different levels of separation are being used. Many approaches (Self-healing, AAOP, Gravity) do not support the adaptation composition. Except for Transformer, only Rainbow provides explicit conflict resolution support with utility-based solution. As for

Table 7-1. Transformer v.s. adaptation projects: Taxonomy Facets: "-" (feature not supported,) "E/I" (External/Internal), "S/G" (Specific/Generic solution), "SoC" (Separation of Concerns), "AC" (Adaptation composition)

	Framework Realization			Adaptation construction			
	Target	E/I	S/G	SoC	AC	Adaptation Reuse	Meta-Adaptation
Self-healing	Comp.	E	Specific	Business -> healing	-	-	-
Rainbow	Comp.	E	Generic	Composable adaptation	Utility function	Yes, source code level	-
Gravity (Hall and Cervantes 2003)	Service Comp.	E	Generic	Business ->Dynamicity	-	-	-
Transformer	Service Comp.	E	Generic	Composable adaptation	Actuator semantic	Yes, components	Supported
AAOP	Apps./ Comp	E	Generic	Business ->adaptation	-	Yes, aspects	Select one adaptation aspect
Opportunistic Integration[45, 105]	Service Comp.	E	Generic	Composable adaptation	Opportunistic	Yes, Component	-

the meta-adaptation, only AAOP provides limited support; however, it only allows one adaptation aspect to be used at any given time.

For the adaptation reuse comparison, in Transformer, programmers can directly reuse adaptation modules in binary form. This feature makes the implementation of adaptation logics much simpler. Although other two approaches (AAOP and "Opportunistic Integration") also provide certain level of adaptation reusability, their adaptation modules normally are not designed for composition. Rainbow do provides a certain level of adaptation reuse. However, in Rainbow, adaptation logics have to be written in a custom, non-standard language ("Stich"). Binary reuse of adaptation modules cannot be achieved. The dependence to a custom language also brings additional complexity to the end users.

Transformer, as shown in Table 7-1, provides a general adaptation framework that can be used for different adaptation logics. It allows different adaptation logics to be reused and composed into more complex adaptation module and reused in different contexts. It supports meta-adaptation by means of the DSM Selector and allows conflicts to be systematically identified and resolved by providing its Actuator Model.

7.4 **Discussion of design issues and limitations**

In this section, the potential limitation of this framework and current implementation are discussed. The following issues are addressed:

- Centralized control model
- Hard-coded adaptation action model
- Model fusion algorithm
- Asynchronous adaptation interactions

7.4.1 Centralized control model

In the Transformer framework design, centralized control model is used which allows for a single point of adaptation decision-making. This central control model simplified the adaptation process while still providing a powerful adaptation mechanism.

The central model provides a simple approach towards global knowledge management. This architectural-wide and updated system knowledge allows accurate adaptation decisions to be taken. However, this central control introduces single point-of-failure and might introduce possible scalability issues. In order to deal with this problem, the Transformer framework has been designed to distribute responsibility across multiple modules.

Firstly, the event sensors and actuators are implemented as individual components and can be deployed on target, possibly distributed system nodes. The interaction of a remote sensor and actuator are implemented via Apache CXF Distributed OSGi [5] framework in our previous approaches [125].

Rather than using a predefined controller which might create single point of failure, in the Transformer framework, the adaptation controller is run-time selected. Failure of one DSM will not influence the correctness of other DSM executions.

The DSM manager appears to be the single-point of failure. Failure of the DSM manager can stop the system adaptation management. However, two designs limit the effects of such failure. Firstly, when DSM manager stops working, the Structural Model can still keep application structure coherence. Then, this system will go back to the adaptation capability of declarative service in OSGi specification, which is still very flexible. Secondly, the DSM manager stores minimal internal state – the *IDSMRopository* stores this information. If it fails, it can be easily restarted by simple commands and system can thus regain adaptation capability again.

7.4.2 Hard-coded adaptation action model

In our system design, application specific sensors and actuators are supported in this framework. In order to effectively identify the relationships between different actions, our middleware provides an *IActuatorModel* interface to allow the programmer to specify their relationships.

In current prototype, the actuator model presented in Section 5.3.4.1 provides a basic, semantic-dependent solution with three actuators. It works fine in our simulation scenarios and robot control platform. However, further improvements are scheduled to support more complex actuator models. In the software engineering approaches, the SOA architecture adopted in our middleware allows future more complex actuator models to be easily inserted into the middleware; one of our ongoing works is to use OWL language and toolset to describe actuators' characteristics and the dependence among them. Further Inference of the relationships can also be done with existing OWL reasoning engines [9, 16].

7.4.3 Model fusion algorithm

In the Transformer framework, the model fusion part plays an important role to automate conflict resolution. In our current solution, the model fusion algorithm is designed according to DSM context matching degree. On the contrary, in several researches, utility function is used to resolve conflicting actions. Adaptation actions are tagged with certain utility values. Those actions will be chosen according to these values. However, as our system targeted on the adaptation in the changing context, for each context, the one adaptation action might have totally different utility values. It is not appropriate to use such function when dealing with changing environments.

As the problem of reconciling conflicting adaptation goals is known to be a fundamentally hard problem, the model fusion trade-off is thus an essential (vs. accidental) problem that cannot be easily automated. Although this thesis provides basic rules for conflict-resolving, however, it is by no means the only possible or best solution. Further work is needed to design better or more intelligent fusion algorithms. For instance, one of

work ongoing work is to design an AI-based fusion algorithm that will dynamic evolve with the users' feedback. Thus, more accurate and efficient adaptation behaviour can be expected.

7.4.4 Asynchronous interaction and uncertainty

By design, the adaptation mechanisms of the Transformer framework interact asynchronously with the target adaptable software. This design reflects our adoption of a control systems approach to self-adaptation.

This design decouples the control logic from the target system, making the self-adaptation infrastructures reusable over different applications. On the other hand, the monitoring sensors and adaptation actuators are implemented as individual components, which are loosely coupled with target adaptable software and the adaptation modules.

We are interested in systems that continue to operate in the face of context changes, which mean that the system does not need to be offline while the adaptation actions are taken. Hence, adaptation mechanisms must interact asynchronously with the target system. In the Transformer framework, asynchronous interaction is used in many part of system design, from monitoring, reasoning and actuation.

For instance, the Event reasoner raised asynchronous changes events to the system run-time without blocking run-time normal execution. The run-time query for the adaptation plans from individual DSM is implemented asynchronously to reduce adaptation response time. The module fusion part is synchronized. The Actuator carries out adaptation actions synchronically by blocking until each action completes. However, it does not block for changes to take effect in the system.

The primary disadvantages of asynchronous, concurrent interactions are issues of race conditions and deadlocks. Therefore, the adaptation actions are taken synchronously to contain the potential source of concurrency problems. The primary source of race conditions would be to keep system meta-model and their corresponding components coherent. Here, our implementation keeps the model updates strictly inside system global model manager to limit the impact of inconsistence.

7.5 Summary

In this chapter, the research in adaptation framework and its supporting middleware is evaluated from both functional and non-functional perspectives. The requirements are identified from the thesis claim presented in Chapter 1 and evaluated individually for the framework and middleware. The results of evaluation show that the adaptation framework and the middleware make the adaptation module development cost-effective, efficient and easy of usage.

In this chapter, a number of potential issues and limitations with the Transformer approach are also identified. It strikes an important balance of combining a centralized

controller with a set of decoupled, possibly distributed infrastructures. This architecture makes the controller easier to implement while resilient to failures. While the interaction between Transformer adaptation phases is asynchronous to allow concurrent activities, the possible uncertainty can be mitigated with carefully designed mechanisms. The one major limitation is the current solution for the model fusion which is not capable to make complex, context-specific conflict resolution decisions.

Chapter 8

Conclusions and Future Work

This chapter summarize the research contributions of this thesis chapter by chapter. Then, discussions for current limitations and future improvements are provided.

8.1 **Summary of contributions**

As discussed in Chapter 1, this thesis studies the problems in how to effective build adaptation behaviours with multiple adaptation concerns. In thesis, a software engineering way is provided to tackle this problem from different perspective. Firstly, guided from the conceptual structure proposed in Chapter 1, an adaptation framework was proposed. This framework, rather than use one standalone adaptation module, constructs adaptation behaviours by using reusable DSM. It allows the developers more efficiently create adaptation modules. Secondly, in order to better support the dynamicity of environments and system configurations, a service-oriented middleware architecture was presented. New adaptation features can be dynamically deployed.

As shown in Chapter 7, compared to existing adaptation frameworks, the proposed framework and the middleware architecture can achieve better support for adaptation with multiple concerns. In our approaches, key problems in providing adaptation behaviour evolution were identified. The way of global adaptation behaviour is constructed allows the development of self-adaptive applications across multiple contexts. Furthermore, a supporting service-oriented middleware is designed and implemented to support Transformer. This middleware provides a highly dynamic and modular solution for both application construction and adaptation behaviour composition. This feature makes our middleware much easier to deal with unforeseen environments.

Here, the contributions of this thesis are listed by chapter.

Chapter 2 introduces the basic concepts for adaptive software and provides the conceptual structure for the concept of adaptation behaviour composition. This chapter firstly provides the definitions of key concepts, for instance context, adaptation, system configuration migration, domain-specific adaptation, etc. Based on these definitions, a conceptual model was presented. This model introduces a different aspect in building adaptation modules. System adaptation module is composed rather than pre-developed. This vision allows the construction of adaptation module more flexible, better support of multi-concern adaptation.

Chapter 3 surveys related work from different perspective. This survey firstly examined the contributing disciplines of software architecture and analysis, control theory, and artificial intelligence. A survey of middleware-based approaches for developing self-adaptive applications was presented, along with discussions of their advantages and limitations. It provided insights into the community's evolving view of the problem of adaptation evolution and identified the major challenges in reusing existing mature adaptation solutions. After having studied existing approaches in the context of adaptation evolutions and compared them with requirements in adaptation in changing environments, an extensive list of requirements was identified to the adaptation evolution enabling frameworks.

Chapter 4 introduced the adaptation framework. It first presented a framework that enables the adaptation behaviour to be developed and constructed with separation of concerns. This framework described a systematic approach for adaptation behaviour composition and for conflicts detection and resolution. This chapter also identified the importance to check the correctness of the composed adaptation strategy as it might not be thoughtfully checked. In order to identify and break possibly unlimited adaptation loops, an on-line loop detection and resolution algorithm was proposed. A formal proof of its convergence criteria was also provided in this chapter.

Chapter 5 presented a supporting middleware architecture for Transformer. This middleware is service-oriented and supports the deployment of self-adaptive applications with run-time, composed adaptation behaviours. This middleware has two important features. 1) DRCom component model is used for both application and middleware construction. 2) DSM dynamicity support, DSM can be run-time deployed and used. By using different implementation, this middleware can be easily reconfigured to match various platform characteristics.

This middleware is constructed via a service-oriented component model – DRCom. The service oriented feature make it comparable easy to integrate other middleware components which better fits for the new environments or future enhanced version of existing components.

This modular and pluggable middleware architecture constructs adaptation behaviours via run-time composed DSM. This process is controlled by the DSM Manager. In order to

reduce system resource usage, the middleware activates or deactivates the DSM according to the dynamically changing context needs.

In Chapter 6, the use of the adaptation framework and the middleware architecture was illustrated in a practical adaptation application – autonomous NXT robot control. This chapter demonstrated how to construct a self-adaptive exploration application that is capable to deal with changing requirements with the run-time pluggable DSM. The business logics of this application were identified and selected business components were introduced. This process showed that our new framework largely keeps targeted application business development process unchanged. Then, a DSM for battery-optimization was developed and deployed during run-time to the system. This DSM could reshuffle application structure according to changing context values. In order to demonstrate the addition of new adaptation features during run-time, DSM for touch sensor self-healing was introduced. This DSM contains the domain-specific self-healing adaptation logics. With the help of new domain-specific adaptation behaviour, the exploration application keeps working correctly. In the later part of this thesis, the dynamic UI and flexible deployment support were discussed to simplify the platform deployment and configuration complexity.

Finally, in chapter 7, evaluations of this adaptation framework and middleware architecture were provided. This evaluation checked both functional and non-functional metrics identified from the thesis claim. It showed that the proposed solution has effectively achieved the claim. This chapter also raised several design issues for discussion and points on limitation of our existing approach.

8.2 Future work

In the Chapters 4, 5 and 7, a number of possible directions for improvements have been discussed from different perspectives. This section presents three major directions for future work from a more general view.

8.2.1 Enhanced context knowledge retrieval & reasoning

In this thesis, the author has taken first steps in using an explicit context model to enable self-adaptation. For instance, in the Battery-based adaptation for NXT robot, we explored a performance-oriented optimization by using a simple context model.

However, focuses have been put on providing adaptation capabilities for multiple adaptation behaviours rather than providing context-awareness to the applications. In this thesis, context knowledge is expressed as simple name-value pairs that can be simply retrieved via context manager. However, there is no systematic design on context knowledge management. Further research is required to investigate how to express context information, how to characterize the relationship between these pieces of context information and how to reason about these pieces of context information to get higher level of knowledge.

One of our future works is to provide a systematic context management mechanism which would require resolving the following three problems: 1) describing how context information can be modelled; for instance, in [130] a context meta-model was defined with three concepts – entity, scope and representation, by using Ontology-based Web Language (OWL) [152]; 2) developing pluggable context sensors to acquire context information, which may be challenging if information is distributed and the entities and resources monitored, are not in one control domain; 3) developing simple ontology knowledge base. This can help to acquire higher level of context information. For instance, in reference [66], together with moving sensors values and door sensors value, this knowledge base can infer whether e.g. the target user is sleeping.

8.2.2 Ontology for adaptation actions

This thesis has provided research results primarily in the area of adaptation behaviour composition by using multiple DSM to guide the adaptation process. In this framework, customized atomic actuators are supported by orchestrating system basic adaptation actions. This extension makes DSM more flexible and much easier to implement. In comparison, as we can see from Section 5.5.1.2, when only basic adaptation actions can be taken, a component healing process might need three rounds of adaptation compared to the single one atomic action required in case a customized *healing* action is supported.

However, this extension also brings additional complexity in the conflicts detection and resolution process. For the three native system adaptation actions, the relationships are comparably simple and fixed. However, for the customized adaptations, no clear semantics of adaptation interaction is defined. This limitation hinders the framework from automatic identifying relationships between those actions. As current implementation statically defines actuators' relationships, it could not effectively deal with the custom actuator. That is because whenever there is a new actuator available, a new Actuator Model is needed.

One possible solution towards this problem is to provide ontology definition for each type of actuator [90]. When a new actuator is installed into the system, its ontology information can be automatically parsed and integrated into Actuator Model for future conflict detection and resolution. In this way, implementation of Adaptation Model can keep function without the need for redesigning when a new actuator becomes available. Further researches are needed on ontology-based actuator definition. In order to effectively describe an actuator, many complex semantic factors as well as complex relationships among them need to be considered, such as preconditions, effects, outputs etc.

8.2.3 Learning from user-feedback

One of the most important ingredients in the adaptation process is human being. However, in current model, how to integrate human's decisions into the adaptation process has not been studied.

In Transformer, the Model Fusion module makes important decisions on balancing the adaptation requirements from different DSM and leave the users the burden of taking all adaptation decisions when changes happens. However, as pointed out by Paspallis: "although the users might enjoy reduced workload in adaptation management, they are often reluctant to fully delegate all decisions to machines. It is actually believed that their reluctance to have the control taken from their hands is one of the main hurdles preventing widespread adoption of self-adaptive systems[121]. Moreover, our prototypical conflict resolving mechanism is predefined and might not fully reflect end-user's current preference.

In order to deal with this problem, the model fusion process should allow the users to oversee the decision process and change the decisions if they do not match their preference. For example, in the case of the TV & recording scenarios, the users can have the control to assign higher priority to application they like most. In the current implementation, the reflective adaptation service allows the users to see adaptation plans generation and conflict resolving processes. It is not possible, in current stage, to allow end-user to change final adaptation plan. This can be improved in our future work.

The other way, probably a better way to introduce human feedback to Transformer is to allow the model fusion polices to be refined by the user-feedback. In current implementation, the Model Fusion module use fixed fusion logics which might not appropriate when the environments or users' preferences changes. Users' feedback can be an important data source to tune the model fusion logics to optimize itself at run-time, aided by some machine-learning algorithms.

One possible step towards this solution was to inform a user whenever there is a conflict resolution action to be taken. My advisor, Dr. De Florio has developed a smart user interface that can collect users' feedback. This feedback then can be used for adjusting the model fusion algorithm to achieve more accurate model fusion behaviours.

Appendix A
Structural dependence maintenance algorithm

As pointed in the Chapter 5, the structure dependence maintenance is wildly used and implemented in most run-time architecture-based adaptation framework. Here the general process of this structure maintenance is introduced.

A.1 Event-based Triggering

As it was mentioned in the Chapter 5, the execution of this functional dependence is triggered by changes to the available components as well as services – for instance,

```
if (event == null) {
                resolveCycles();
                adaptationWithDSM(event);
            }
        // if service registered
        else if (event.getType() == ServiceEvent.REGISTERED) {
        resolveCycles();
        List dynamicBind =
selectDynamicBind(event.getServiceReference());
            if (!dynamicBind.isEmpty()) {
                workQueue.enqueueWork(this, DYNAMICBIND,
dynamicBind);
            }
            adaptationWithDSM(event);
        }
        // if service modified
        else if (event.getType() == ServiceEvent.MODIFIED){
      ….
     }
    …..
    }
```

Listing A-1. Excerpt of event-based adaptation invocation

installing a new component, uninstalling an already deployed component, or events in the service registry – for example, *Service.Registering*, *Service.Updating* and *Service.unRegistering*. These service events are also used to check whether a DRCom was successfully installed and initialized as our system run-time automatically registers a component's provided service interface into service registry. If the initialization is successful, system will get corresponding service change events.

A.2 Structural dependence resolution

As pointed in the Chapter 5, the structure dependence maintenance is wildly used and implemented in most run-time architecture-based adaptation framework. The key logic in this module is to check whether a given component's structural dependence are satisfied. Listing A-2 shows the excerpt of code in how to check which set of components are functional satisfied. In this algorithms, the enabledComponentConfigurations is a set which contains all the installed & enabled plug-ins, including DSM plug-in and custom Event monitors and actuators. The resolvedSatisfiedComponentConfigurations records all the component configurations that are derived as structural satisfied. ComponentDescription is the meta-class that records all component-specific information.

```
private List resolveSatisfied() {
       List resolvedSatisfiedComponentConfigurations = new ArrayList();
       Iterator it = enabledComponentConfigurations.iterator();
       while (it.hasNext()) {
           ComponentConfiguration componentConfiguration =
            (ComponentConfiguration) it.next();
           ComponentDescription cd =
            componentConfiguration.getComponentDescription();
  // check if all the services needed by the component configuration
   //are available
           List refs = componentConfiguration.getReferences();
           Iterator iterator = refs.iterator();
           boolean hasProviders = true;
           while (iterator.hasNext()) {
               Reference reference = (Reference) iterator.next();
               if (reference != null) {
                             if
                (reference.getReferenceDescription().isRequired()
                && !reference.hasProvider(componentConfiguration.
               getComponentDescription().getBundleContext())) {
                      hasProviders = false;
                      break;
                  }   }   }
       if (!hasProviders)
              continue;
   …
// we have providers and permission - this component configuration is
//satisfied
   resolvedSatisfiedComponentConfigurations.add(componentConfiguratio
n);
       } // end while (more enabled component configurations)
       return resolvedSatisfiedComponentConfigurations.isEmpty() ?
Collections.EMPTY_LIST : resolvedSatisfiedComponentConfigurations;
    }
```

Listing A-2. Structural resolution process

A.3 **Adaptation action identification**

As demonstrated in the Listing A-2, all the components that are functional satisfied can be identified by checking with all provided interfaces and required interfaces among components. After those components are identified, the structure modeller needs to specify which components should be enabled and which components should be disabled. As there are two general principles for software architecture maintenance,

1) makes structure-satisfied components "activated";
2) deactivate those "activated" components when their functional dependences are no longer satisfied.

According to these two principles, the adaptation algorithm is defined in Algorithm A-1. Here, the form of algorithm representation from Dr. Paspallis's thesis[121] is used.

Algorithm A-1. The algorithm used by the Structure Modeller (pseudo-code)

Basic data-structures

[enabledComponentConfigurations] - set containing all installed DRCom involving adaptation process
[satisfiedComponentConfigurations] - subset of the [all DRCom]; contains only resolved DRCom
[provided] - map of service interfaces to set of resolved components
[ServiceSet] – set containing all available services registered in the service registry

Algorithm

```
1.  # Newly resolved DRCom will be added to [resolved] set
2.  for all p in [enabledComponentConfigurations] - [satisfiedComponentConfigurations] do
3.        if requiredServiceTypes(p) ⊆  [ServiceSet] then
4.          [satisfiedComponentConfigurations]= [satisfiedComponentConfigurations] ∪ {p}
5.        #   ps is the set of component p all provided services
6.            ps= [providedServiceTypes(p)]
7.            [ ServiceSet ]= [ ServiceSet ] ∪ ps
8.            end for
9.       end if
10.     end for
11.   # newly unresolved DRCom should be removed from the
      [satisfiedComponentConfigurations] set
12.     for all p in [satisfiedComponentConfigurations] do
13.           if requiredServiceReferences(p) ! ⊆ [ServiceSet] then
14.           [satisfiedComponentConfigurations]= [satisfiedComponentConfigurations] -
      {p}
15.           [ServiceSet] = [ServiceSet] - {providedServiceTypes(p)}
16.           end if
17.      end for    end for
```

```
AdaptationPlan structurePlan=new
   SimpleAdaptationPlan(newlySatisfiedComponentConfigurations,
   new
ArrayList());
plans.add(structurePlan);
AdaptationPlan fusedplan= FuseAdaptationPlans(plans,
   drcrContext);
if (fusedplan==null)
        return ;
List newlySatisfied=fusedplan.getDSMSatisifiedComponents();
List newlyUnsatisfied=fusedplan.getUNSatisifiedComponents();
   if (!newlySatisfied.isEmpty()) {
   satisfiedComponentConfigurations.addAll
(newlySatisfiedComponentConfigurations);
// add to satisfiedComponentConfigurations before dispatch
   workQueue.enqueueWork(this, BUILD, newlySatisfied);
   }
   if (newlyUnsatisfied.size() > 0)
    {

    satisfiedComponentConfigurations.removeAll(newlyUnsatisfie
   d); // add to satisfiedComponentConfigurations before dispatch
        workQueue.enqueueWork(this, DESTORY, newlyUnsatisfied);
    }
…
```

Listing A-3. Adaptation action identification

In this algorithm, there are two phases of resolution. The first phase is to make sure that any resolved component is marked as resolved and accounted into the [satisfiedComponentConfigurations] set. Whenever there is an event that triggers adaptation, this algorithm iterates through each unresolved components to see whether its dependencies are satisfied. As each component might have more than one required functional interfaces, each interface has to be check. If it is found to be resolved it is marked as such and added into [satisfiedComponentConfigurations] set. When such an iteration of the unresolved DRCom terminates, it means that all resolved components are correctly calculated.

The second phase achieves a similar goal but reverse goal: it ensures that all components marketed as resolved, are indeed “structure-satisfied” after system changes. To achieve this goal, a loop is designed, where each “structure-satisfied” component is checked against its dependencies. If it is found to be unresolved, it is deleted from the [satisfiedComponentConfigurations] set and all its provided services will be removed from service registry. These actions will trigger additional adaptation until a full iteration is completed without changes.

Here, the changes-detected flag shows whether there is any change detected during each round; such changes might trigger additional adaptation actions. Here, the algorithm is merely demonstrative rather than a real implementation. That is because in Algorithm A-1, it is assumed that the change of a component state in the *Structural Modeller* (meta-model changes) will immediately change a component state, which is not the case in the practical implementation. In order to achieve separation of concerns, in the *Structural Modeller*, no adaptation actions will be taken until the adaptation plans has been actuated. So, marking a component as either resolved or unresolved will actually not trigger additional actions. Only when, for instance, a component is initialized or a service is unregistered from registry, system will then continue adaptation steps.

Thus, the real adaptation actions will be taken after the adaptation plan fusing process. The excerpt of fusing code and adaptation actuation is demonstrated in Listing A-3. In the this listing, the adaptation plans are firstly got fused by the FuseAdaptationPlans method to get fused newlysatisfied and newlyUnsatisfied set of components. Then, the system will perform the adaptation actions be issuing adaptation actions to the task queue for later execution. Please note, the task queue is synchronized and the order of adaptation actions are keeps as the same order as in the fused adaptation plan.

Appendix B Simulation on Adaptation with loop detection

B.1 Settings of targeted abstract applications

In this set of experiments, 25 components, named A1~A5, B2~B5, C1~C5, M1~M5, N1~N5, are deployed in a simulated environment. The components in group A can provide the service interface required by the components in group B. Similarly, the components in group B provide the service to the components in group C and the components in group M provide the service to the components in group N. The first group of components forms App_A while the component M, N forms application App_B. For application App_A, the possible configurations range from A1→B1→C1 to A5→B5→C5, which adds up to 125 possible configurations. Similarly, for application App_A, there are 25 possible configurations. In Figure C.1, two sample configurations are shown. In configuration A, the configuration is {{A2, B4, C1}, {M2, N1}} while after one adaptation step, it is changed to configuration B {{A2, B4,∅}, {M4, N1}}.

Using those components, a user might request to synthesize a system configuration with these components. The target configuration v_{pref} is simulated by choosing a specific configuration to be the one preferred (preferred configuration) by the current user at the present moment. Each user, upon being presented a synthesized architecture configuration, decides to agree on the presented configuration if it is equal to his currently preferred configuration. In this case, adaptation will stop. Otherwise, system will resume adaptation loop. At each round of adaptation, system starts adaptation from initial state {{∅, ∅, ∅}, {∅, ∅} . Each simulation is repeated 100 times, and the average numbers of results are calculated.

B.2 Adaptation modules

Using this abstract environment, the performance of the four different DSM is evaluated. “random selection without user preference”, “random selection without user preference” “decision tree based selection without user preference” and “decision tree

based selection without user preference". The use preference is set as "with APP_A" only.

To focus on the loop detection performance, only one DSM will be used in one time. The system performance is evaluated in term of numbers of adaptation actions to be performed in order to reach user preferred state.

During simulation, at each adaptation step, a modeller will select **one adaptation action** from all possible action set. For simplicity, we only consider two lifecycle actions: "enable and disable one component". Structural modeller here is used to check which adaptation can be performed according to system functional dependence. All the actions reasoned by these modellers are converged with functional modeller as the fusion rule describe in Algorithm A.1. For instance, it is impossible to create a component configuration as {A1→ ∅ → C2, M1→N2} because C2 depends on a component instance from group B which, however, does not exist. So, system will always transit among all functional satisfied state.

Random choice DSM uses a very simple strategy. Each round, the modeller randomly chooses one possible adaptation action that can be taken in current system configuration. It can choose to enable / disable one component or replace one instance with another implementation, for instance A1 → A3 (it implemented as disable component A1 and enable component A3). These actions will lead the system into different configurations. For each round of adaptation, this modeller can only choose an action that can be taken in current state. Whenever the preferred state is reached, simulation stops, and the number of system migrations is recorded.

There are 6 possible ways for component selection from any component group, for instance for group A, 6 possible choices can be made, selecting any component in {A1, A2, A3, A4, A5} or no component at all. Thus there are in total 6*6*6*6*6 = 7776 possible combinations in choosing components and their states (on or disabled). However, some of the configurations might not satisfy the functional constraints and converged to nearby functional satisfied state, the total number is 5201.

Learning-based DSM learning module records the components in the system global configuration along with the contexts of the user and of the components state in the configuration. Once it accumulates a certain amount of such records, the Learning module derives application configuration from the accumulated records.

By recording conditions under which the components have been chosen, a decision tree can be built to represent the decision process. The generated decision tree consists of leaf nodes that correspond to the components and other non leaf nodes that specify the conditions in which the underneath leaf nodes have been chosen. The decision tree is built by providing recorded context information and the records of the component configuration into a decision tree building algorithm; here C4.5 [126] is used. Each path from a root node to a leaf node represents a condition. Under this condition, the component corresponding to the leaf node will be chosen, and thus becomes the system

configuration. If any decision tree was built, it will be reused in the following adaptation process.

DSM with user preference

With the help of our framework, it is possible to add external context adaptation logics to achieve domain-specific optimization goal. In this simulation, domain-adaptation is expressed as user's preference in current context. Due to the complexity for normal user to specify all the rules, this modeller might not be able to specify a configuration which exactly matches the preferred configuration. They normally describe a general preference, for instance, user Tom may specify that "only App_A is to be used". It, by no means, could determine the preferred configuration. This domain-adaptation knowledge will be used with random selection based adaptation and learning-based algorithm to find targeted configurations. So, two types of DSM will be created. The **Random based DSM** and **learning based DSM.** In these two types of modellers, user specified rule are firstly used to filter out impossible configurations and then, the random / learning based module will be used to generate further adaptation actions. Here, two DSM are implemented for dealing with two different user's preference – "only App_A is to be used" and "only App_B is to be used". DSM matched with current context will be selected during run-time by *DSM Selector* to meet the changing environments.

B.3 Adaptation in stable environment

Here, the definition of comparably stable environment means there is no significant context change during the whole simulation. In the whole simulation, there is only one preferred configuration. The context-specific modeller will try to form a configuration that fits the preferred one.

Figure C.1 shows the result of this experiment. It shows that the pure learning-based adaptation needs a lot of tries to get a preferred configuration. Normally, about 8001rounds of adaptation are needed. As we can see from Figure C.1, compared to random choice algorithm, user preference based modeller need much less adaptation steps – about 238 to reach the preferred configuration. This is due to the facts that the user's preference rule (run App_A only), can effectively reduce the possible satisfied configurations, to about (125+25+5) times, about 1/36 from all possible 5201 configurations. A user needs far less tries to get the correct configuration. As both random choosing algorithm and user preference-based scheme use stateless algorithm, their performance will not change with the number of simulation cycles.

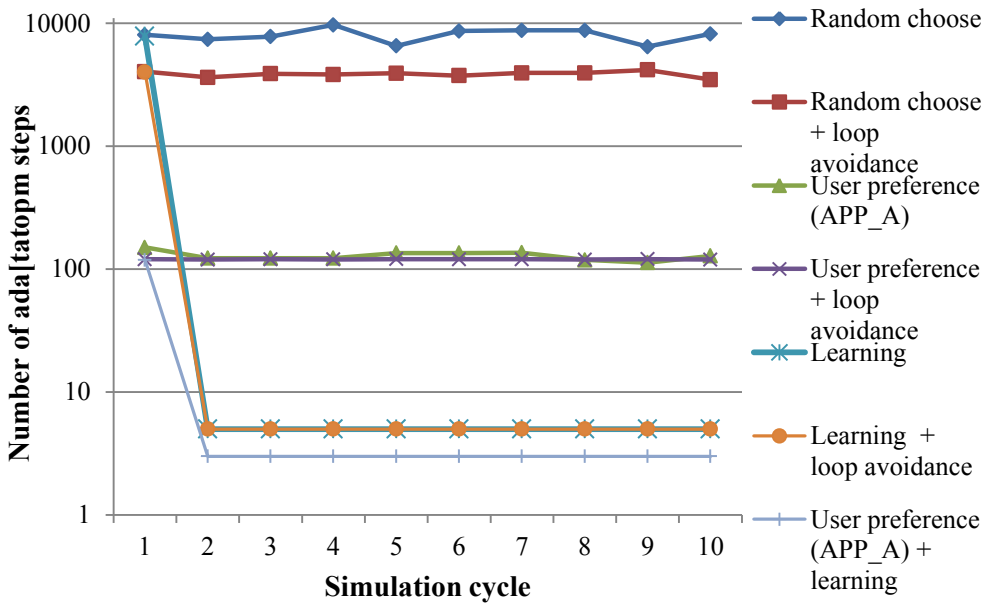


Figure B-1. Adaptation steps under static context environment

We can also see that the learning based scheme can achieve more and more accurate results with the number of simulation cycles. Once a preferred configuration is identified, only several steps are needed to reach the preferred configuration, as the adaptation modeller has already constructed the decision tree to represent current user's preference. The modeller combined with user preference and learning have even better performance, as the user preference reduced the solution space, hence the learning algorithm can more effectively construct the decision tree. For instance if the user preference is to enable App_A only, then on average, around 238 adaptation actions are performed to generate the corresponding decision tree. This is much less than in the pure learning based modeller, where about 8000 steps are needed.

The loop avoidance scheme described in Annex B allows even better results to be achieved: for the random choice scheme, about 3600 steps are required, while the modeller with user preference only needs 120 steps to construct the decision tree.

Appendix C
Varying CMD and DSM Selection

As shown in the motivational example, each DSM can only effectively conclude its adaptation actions when the current context (to a certain extent) matches its requirements. In order to provide accurate adaptation behaviour, for a particular context, only a part of the installed DSMs should be used for the adaptation process. DSM Selector handles this selection process.

In order to select the "right" set of DSMs to be used in the adaptation process, one important responsibility of the DSM Selector is to calculate the similarity between each DSM's preferred context requirements and system's current context. We refer to this similarity as to the Context Matching Degree (CMD). Many methods have been proposed to calculate this value. For instance, Fujii et al. [59] propose the use of context matching condition to calculate the similarity. Liu et al.[100] propose the use of inverse distance to calculate the context matching degree.

C.1 Distance based CMD

In this section, we use the inverse distance between system's current context and a DSM's preferred context to calculate the CMD. As different context factors might normally have different impact factors towards the DSM's usability, the inverse distance is weighted by the impact factor. The context matching degree for any given DSM, say DSM_A, is calculated by the following formula:

$$CMD(CAP\_A) = \sum_{i=1}^{N} \frac{1}{1+K*Abs(C_i^{current}-C_i^{opt})} * Weight^i \qquad (1)$$

in which the C_i^{opt} denotes the preferred value of the context factor C^i, $Weight^i$ denotes the impact factor of context factor C^i towards DSM_A, and N is the number of context factors that have impacts towards DSM_A. The abs function provides the absolute distance between C_i^{opt} and current value $C_i^{current}$ of C^i. K is a constant that is used to adjust context sensitivity and it is currently arbitrarily set to 9. According to the calculated CMD, if the CMD of a DSM is higher than a certain threshold, this DSM will be selected. In this prototype, an arbitrary value – 0.3 – is assigned. One special case for the DSM

Selector is that of the DSM for software structural maintenance, that will be always chosen, as maintaining the integrity of the software structure is needed whatever the context. Of course, the function provided here is only one possible solution for CMD calculation. Other algorithms, such as the semantic graph matching schemes can also be used in our framework.

C.2 CMD Migration and DSM selection

When context factor values change, CMDs of different DSMs also change. In this case study, we assume system context to be expressed by three major context factors, 1) fault-tolerance, which expresses user's preference on whether to keep fault-tolerant adaptation capabilities or not; 2) current system CPU usage; 3) system remaining battery. Of course, in a real-life system, it is likely to have more factors. In order to simplify the case study, only these DSM-related factors are discussed.

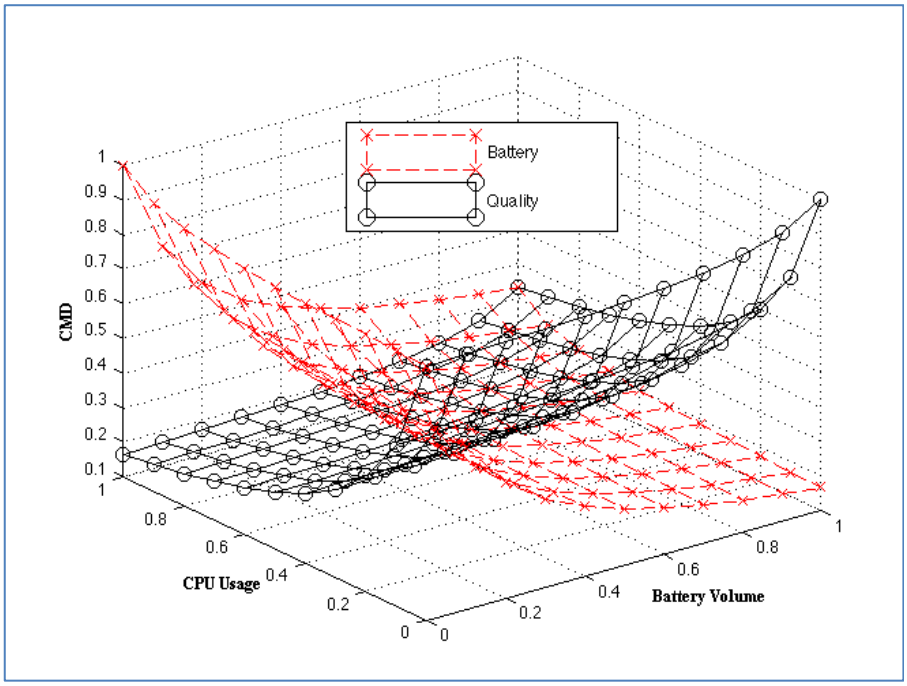


Figure C-1. CMD for two DSMs with two context factors: CPU and Battery

As in the motivational example, during the whole adaptation process, John will always want to have the self-healing feature. So the DSM^{repair} will be always chosen. From *Actuator Model*, we can see this DSM has no conflicts with other adaptation modules. So, CMDs of $CAP^{quality}$ and $CAP^{battery}$ are introduced here to demonstrate conflicts resolution process. Such CMDs are influenced by the two factors: Battery and CPU.

The choice of the preferred context factors and their weights is a problem in itself that is outside the scope of our paper. In this paper, we chose the following arbitrary values:

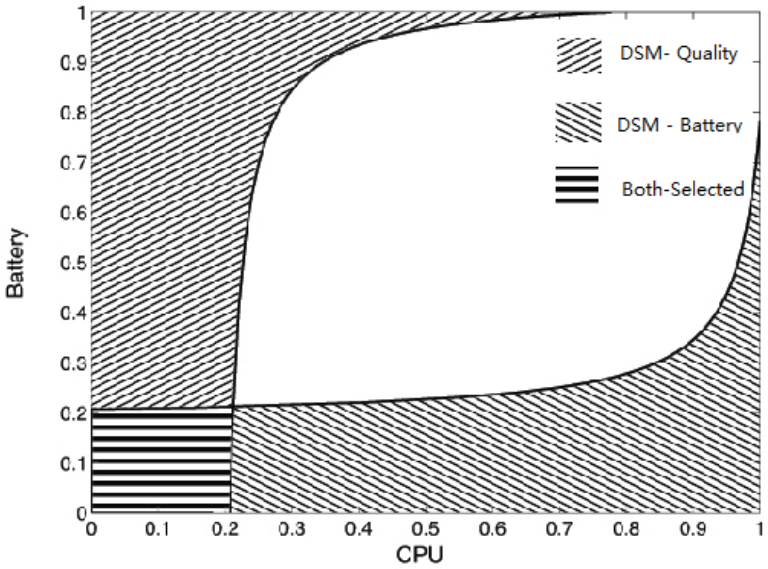


Figure C-2. DSM selection zone with selection threshold 0.3

$\text{DSM}^{\text{quality}}$: $C_{cpu}^{opt} = 0$, $C_{battery}^{opt} = 1$, $\text{Weight}^{\text{cpu}} = 0.8$, $\text{Weight}^{\text{battery}} = 0.2$

$\text{DSM}^{\text{battery}}$: $C_{cpu}^{opt} = 0$, $C_{battery}^{opt} = 1$, $\text{Weight}^{\text{cpu}} = 0.2$, $\text{Weight}^{\text{battery}} = 0.8$.

From these values we can see that $\text{DSM}^{\text{quality}}$ fits better with environments where CPU usage is low and battery volume is still abundant, while $\text{DSM}^{\text{battery}}$ will be used when battery volume is low and CPU usage is high. The CMDs for both DSMs calculated by Formula 1 is shown in Fig. C-1.

The selection of DSMs is based on their CMD values. Figure C-2 shows the selection zone for $\text{DSM}^{\text{quality}}$ and $\text{DSM}^{\text{battery}}$. When the values of two context factors falls into the area with horizontal straps, both DSMs will be selected. In this area, as both DSMs will be used to guide adaptation, possible conflicts might arise.

Bibliography

[1] *Apache Axis 2, Apache Foundation*. Available: http://ws.apache.org/axis2/

[2] *Apache Felix - OSGi R4 Service Platform*. Available: http://felix.apache.org

[3] *The ARFLEX Project*. Available: www.arflexproject.eu

[4] *CORBA IDL, Object Management Group™ (OMG™) , 2005 , http://www.omg.org/gettingstarted/omg_idl.htm*.

[5] *Distributed OSGi, Apache CXF Project*. Available: http://cxf.apache.org/distributed-osgi.html

[6] *Dynamic systems initiative, Microsoft Corporation*. Available: http://www.microsoft.com/windowsserversystem/dsi/

[7] *Equinox - OSGi implementation, Eclipse Fundation*. Available: www.eclipse.org/equinox/

[8] *eXtensible Access Control Markup Language (Version 2.0 ed.)*. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20

[9] *FaCT++*. Available: http://owl.man.ac.uk/factplusplus/

[10] *JBoss Aspect-Oriented Programming*. Available: http://www.jboss.org/jbossaop

[11] *Juliac -Fractal toolchain compiler, OW2 Consortium*. Available: http://fractal.ow2.org/juliac/

[12] *Knopflerfish OSGi - open source OSGi service platform*. Available: http://www.knopflerfish.org

[13] *LEGO, Lego Mindstorm Robot*. Available: http://mindstorms.lego.com

[14] *LeJOS: Java for LEGO MindStorms*. Available: http://lejos.sourceforge.net/

[15] *Nasa Mars Rover*. Available: http://marsrover.nasa.gov/home/.

[16] *Pellet: OWL 2 Reasoner for Java*. Available: http://clarkparsia.com/pellet

[17] *TinyVM*. Available: http://tinyvm.sourceforge.net/

[18] *UniCon - an architectural description language*. Available: http://www.cs.cmu.edu/~UniCon/

[19] *Web Services Policy Framework, W3C, v1.5*. Available: http://www.w3.org/TR/ws-policy/

[20] *xAcme, Carnegie Mellon University and University of California, Irvine*. Available: http://www.cs.cmu.edu/~acme/pub/xAcme/

[21] *xArch, University of California, Irvine and Carnegie Mellon University*. Available: http://www.isr.uci.edu/architecture/xarch/

[22] M. Alia, S. Hallsteinsen, N. Paspallis, and F. Eliassen, "Managing Distributed Adaptation of Mobile Applications," in *Distributed Applications and Interoperable Systems*. vol. 4531, J. Indulska and K. Raymond, Eds., ed: Springer Berlin / Heidelberg, 2007, pp. 104-118.

[23] M. Alia, G. Horn, F. Eliassen, M. U. Khan, R. Fricke, and R. Reichle, "A component-based planning framework for adaptive systems," *Proceedings of the On the Move to Meaningful Internet Systems,* vol. 4276, pp. 1686-1704, 2006.

[24] K. J. Åström and B. Wittenmark, *Adaptive control*. Reading, Mass ; Wokingham: Addison-Wesley, 1989.

[25] D. Ayed, C. Taconet, G. Bernard, and Y. Berbers, "CADeComp: Context-aware deployment of component-based applications," *Journal of Network and Computer Applications,* vol. 31, pp. 224-257, Aug 2008.

[26] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Reading, Mass.: Addison-Wesley, 1998.

[27] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2 ed.: Addison-Wesley, 2003.

[28] W. J. Bert Lagaisse, Bart De Win, "Managing semantic interference with aspect integration contracts," in *Proceedings of the International Workshop on Software-Engineering Properties of Languages for Aspect Technologies*, England, 2004.

[29] V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed, "Enabling Self-Managing Applications using Model-based Online Control Strategies," presented at the Proceedings of the 2006 IEEE International Conference on Autonomic Computing, 2006.

[30] E. Bianchi, L. Dozio, and P. Mantegazza, "RTAI Programming Guide," 1 ed, 2006.

[31] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao, "ABLE: A toolkit for building multiagent autonomic systems," *IBM Systems Journal,* vol. 41, pp. 350-371, 2002.

[32] J. Bonér and A. Vasseur. *AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java*. Available: http://aspectwerkz.codehaus.org/

[33] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, Newport Beach, California, 2004, pp. 28-33.

[34] L. Capra, W. Emmerich, and C. Mascolo, "Reflective Middleware Solutions for Context-Aware Applications," in *Metalevel Architectures and Separation of Crosscutting Concerns*. vol. 2192, A. Yonezawa and S. Matsuoka, Eds., ed: Springer Berlin / Heidelberg, 2001, pp. 126-133.

[35] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-aware reflective mlddleware system for mobile applications," *IEEE Transactions on Software Engineering,* vol. 29, pp. 929-945, Oct 2003.

[36] H. Cervantes and R. S. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, 2004, pp. 614-623.

[37] H. Cervantes and R. S. Hall, "A framework for constructing adaptive component-based applications: Concepts and experiences," *Component-Based Software Engineering,* vol. 3054, pp. 130-137, 2004.

[38] C. Charignon, M. Kostka, N. Koning, P. Jaikumar, and R. Ouyed, "r-Java: an r-process code and graphical user interface for heavy-element nucleosynthesis," *Astronomy & Astrophysics,* vol. 531, Jul 2011.

[39] B. H. C. Cheng, R. de Lemos, S. Fickas, D. Garlan, M. Litoiu, J. Magee*, et al.*, "SEAMS 2007: Software engineering for adaptive and self-managing systems," in *Porceedings of the 29th International Conference on Software Engineering: ICSE 2007*, 2007, pp. 152-153.

[40] S. Cheng, "Rainbow: Cost-effective, Software Architecture-based Self-adaptation," PhD Thesis, Departments of Computer Science and Electrical and Computer Engineering, CMU 2008.

[41] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, 2001, p. 103.

[42] G. David, "Project Aura: Toward Distraction-Free Pervasive Computing," *IEEE Pervasive Computing,* vol. 1, pp. 22-31, 2002.

[43] A. K. Dey, "Providing Architectural Support for Building Context-Aware Applications," PhD Thesis, Georgia Institute of Technology, 2000.

[44] A. K. Dey, G. D. Abowd, and D. Salber, "A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications,"

Human-Computer Interaction, vol. 16, pp. 97-163, 2001.

[45] A. Diaconescu, Y. Maurel, and P. Lalanda, "Autonomic management via dynamic combinations of reusable strategies," in *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*, Turin, Italy, 2008, pp. 1-10.

[46] E. W. Dijkstra, "Structure of the-Multiprogramming System," *Communications of the ACM,* vol. 11, pp. 341-346, 1968.

[47] E. W. Dijkstra, "On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective,* pp. 60-66, 1982.

[48] S. Dobson, S. Denazis, A. Fernandez, D. Gaiti, E. Gelenbe, F. Massacci*, et al.*, "A Survey of Autonomic Communications," *ACM Transactions on Autonomous and Adaptive Systems,* vol. 1, pp. 223-259, Dec 2006.

[49] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, "Fulfilling the Vision of Autonomic Computing," *Computer,* vol. 43, pp. 35-41, Jan 2010.

[50] J. Dowling, "The decentralised coordination of self-adaptive components for autonomic distributed systems," PhD Thesis, Department of Computer Science, Trinity College Dublin, 2004.

[51] J. Dowling and V. Cahill, "Self-managed decentralised systems using K-components and collaborative reinforcement learning," presented at the Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, Newport Beach, California, 2004.

[52] J. Dowling and V. Cahill, "Self-managed decentralised systems using K-components and collaborative reinforcement learning," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, Newport Beach, California, 2004, pp. 39-43.

[53] A. Elkhodary, N. Esfahani, and S. Malek, "FUSION: A Framework for Engineering Self-Tuning Self-Adaptive Software Systems," in *Proceedings of the 18th Internatioal Symposium on the Fundation of Software Engineering*, 2010.

[54] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language (AADL) A standard for engineering performance critical systems," in *Proceedings of the IEEE Conference on Computer-Aided Control System Design*, 2006, pp. 302-307.

[55] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, "Using architecture models for runtime adaptability," *IEEE Software,* vol. 23, pp. 62-70, Mar-Apr 2006.

[56] V. D. Florio, "A Fault-Tolerance Linguistic Structure for Distributed Applications," PhD Thesis, Dept. of Electrical Engineering, Katholieke Universiteit Leuven, 2000.

[57] K. Fujii and T. Suda, "Semantics-based dynamic service composition," *IEEE*

Journal on Selected Areas in Communications, vol. 23, pp. 2361-2372, Dec 2005.

[58] K. Fujii and T. Suda, "Semantics-based dynamic Web Service composition," *International Journal of Cooperative Information Systems,* vol. 15, pp. 293-324, Sep 2006.

[59] K. Fujii and T. Suda, "Semantics-based Context-aware Dynamic Service Composition," *ACM Transactions on Autonomous and Adaptive Systems,* vol. 4, pp. 12-42, May 2009.

[60] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, "Evolution Styles: Foundations and Tool Support for Software Architecture Evolution," in *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2009, pp. 131-140.

[61] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer,* vol. 37, pp. 46-49, Oct 2004.

[62] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven*, et al.*, "A comprehensive solution for application-level adaptation," *Software: Practice and Experience,* vol. 39, pp. 385-422, 2008.

[63] K. Geihs, M. U. Khan, R. Reichle, A. Solberg, and S. Hallsteinsen, "Modeling of component-based self-adapting context-aware applications for mobile devices," *Software Engineering Techniques: Design for Quality,* vol. 227, pp. 85-96, 2006.

[64] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain*, et al.*, "The SmartFrog configuration management framework," *SIGOPS Operation System Review,* vol. 43, pp. 16-25, 2009.

[65] J. Gray, T. Bapty, S. Neema, and J. Tuck, "Handling crosscutting constraints in domain-specific modeling," *Communication of ACM,* vol. 44, pp. 87-93, 2001.

[66] T. Gu, H. K. Pung, and D. Q. Zhang, "A service-oriented middleware for building context-aware services," *Journal of Network and Computer Applications,* vol. 28, pp. 1-18, Jan 2005.

[67] N. Gui, V. De Florio, G. Caporaletti, and C. Blondia, "Adaptive robot design and applications in flexible manufacturing environments," in *13th IFAC Symposium on Information Control Problems in Manufacturing, INCOM'09, June 3, 2009 - June 5, 2009*, Moscow, Russia, 2009, pp. 2149-2154.

[68] N. Gui, V. De Florio, H. Sun, and C. Blondia, "A framework for adaptive real-time applications: the declarative real-time OSGi component model," in *Proceedings of the 7th Workshop on Adaptive and Reflective Middleware(ARM)*, Leuven,Belgium, 2008.

[69] N. Gui, V. De Florio, H. Sun, and C. Blondia, "A hybrid real-time component model for reconfigurable embedded systems," in *23rd Annual ACM Symposium on Applied Computing, SAC'08, March 16, 2008 - March 20, 2008*, Fortaleza, Ceara, Brazil, 2008, pp. 1590-1596.

[70] N. Gui, V. De Florio, H. Sun, and C. Blondia, "ACCADA: A framework for continuous context-aware deployment and adaptation," in *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2009, November 3, 2009 - November 6, 2009*, Lyon, France, 2009, pp. 325-340.

[71] N. Gui, V. De Florio, H. Sun, and C. Blondia, "An architecture-based framework for managing adaptive real-time applications," in *EUROMICRO2009 - 35th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2009, August 27, 2009 - August 29, 2009*, Patras, Greece, 2009, pp. 502-507.

[72] N. Gui, V. De Florio, H. Sun, and C. Blondia, "Toward architecture-based context-aware deployment and adaptation," *Journal of Systems and Software,* vol. 84, pp. 185-197, Feb 2011.

[73] N. Gui, P. Pieter-Jan, V. D. Florio, and C. Blondia, "Run-time Reconfiguration Software Platform for Autonomous Robot," in *Proceedings of ADAMUS 2010, held in the 7th ACM International Conference on Pervasive Services*, 2010, pp. 19-24.

[74] R. S. Hall and H. Cervantes, "Gravity: supporting dynamically available services in client-side applications," *SIGSOFT Software Engineering Notes,* vol. 28, pp. 379-382, 2003.

[75] R. S. Hall and H. Cervantes, "Challenges in building service-oriented applications for OSGi," *IEEE Communications Magazine,* vol. 42, pp. 144-149, 2004.

[76] S. Hallsteinsen, J. Floch, and E. Stav, "A middleware centric approach to building self-adapting systems," *Software Engineering and Middleware,* vol. 3437, pp. 107-122, 2005.

[77] S. Hashimoto, F. Kojima, and N. Kubota, "Perceptual system for a mobile robot under a dynamic environment," *Porceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation,* pp. 747-752, 2003.

[78] J. Hillman and I. Warren, "Meta-Adaptation in Autonomic Systems," in *FTDCS*, 2004, pp. 292-298.

[79] T. Inamura, M. Inaba, and H. Inoue, "User adaptation of human-robot interaction model based on Bayesian network and introspection of interaction experience," in *Proceedings of the International Conference on Intelligent Robots and Systems*, 2000, pp. 2139-2144.

[80] A. Janik and K. Zielinski, "AAOP-based dynamically reconfigurable monitoring system," *Information and Software Technology,* vol. 52, pp. 380-396, Apr 2010.

[81] A. Janik and K. Zielinski, "Adaptability mechanisms for autonomic system implementation with AAOP," *Software: Practice and Experience,* vol. 40, pp. 209-223, 2010.

[82] J. L. Jones, "Robots at the tipping point - The road to the iRobot roomba," *IEEE Robotics & Automation Magazine,* vol. 13, pp. 76-78, Mar 2006.

[83] K. Kakousis, N. Paspallis, and G. A. Papadopoulos, "Optimizing the Utility Function-Based Self-adaptive Behavior of Context-Aware Systems Using User Feedback," in *Proceedings of the OTM 2008 Confederated International Conferences*, Monterrey, Mexico, 2008, pp. 657-674.

[84] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME – Journal of Basic Engineering,* pp. 35-45, 1960.

[85] G. Karsai and J. Sztipanovits, "A model-based approach to self-adaptive software," *IEEE Intelligent Systems & Their Applications,* vol. 14, pp. 46-53, May-Jun 1999.

[86] E. P. Kasten and P. K. McKinley, "Perimorph: Run-time composition and state management for adaptive systems," *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops,* pp. 332-337, 2004.

[87] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer,* vol. 36, pp. 41-50, Jan 2003.

[88] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," presented at the Proceedings of the 15th European Conference on Object-Oriented Programming, 2001.

[89] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier*, et al.*, "Aspect-oriented programming," *Ecoop'97: Object-Oriented Programming,* vol. 1241, pp. 220-242, 1997.

[90] G. H. Kim, D. H. Kim, X. Hoang, and Y. H. Lee, "Group-aware service discovery using effect ontology for conflict resolution in ubiquitous environment," in *Proceedings of the 10th International Conference on Advanced Communication Technology*, 2008, pp. 1811-1816.

[91] C. Klein, R. Schmid, C. Leuxner, W. Sitou, and B. Spanfelner, "A Survey of Context Adaptation in Autonomic Computing," in *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008)*, 2008, pp. 106-111.

[92] M. H. Klein, R. Kazman, L. J. Bass, S. J. Carrire, M. Barbacci, and H. F. Lipson, "Attribute-Based Architecture Styles," in *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999, pp. 225-244.

[93] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas, "Design, implementation, and performance of an automatic configuration service for distributed component systems," *Software-Practice & Experience,* vol. 35, pp. 667-703, 2005.

[94] J. Kwon, O.-H. Choi, C.-J. Moon, S.-H. Park, and D.-K. Baik, "Deriving similarity for Semantic Web using similarity graph," *Journal of Intelligent. Information System,* vol. 26, pp. 149-166, 2006.

[95] R. Laddaga, "Self-adaptive software," DARPA BAA, Technique Report, 1997.

[96] R. Laddaga, "Active Software," in *Self-Adaptive Software*. vol. 1936, P. Robertson, H. Shrobe, and R. Laddaga, Eds., 1st ed: Springer Berlin / Heidelberg, 2001, pp. 11-26.

[97] A. Lapouchnian, S. Liaskos, J. Mylopoulos, and Y. Yu, "Towards requirements-driven autonomic systems design," *SIGSOFT Software Engineering Notes,* vol. 30, pp. 1-7, 2005.

[98] W. S. Levine, *The Control Handbook*. New York: CRC Press, 1996.

[99] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *SIGSOFT Software Engineering Notes,* vol. 30, pp. 1-7, 2005.

[100] L. Liu, F. Lecue, N. Mehandjiev, and L. Xu, "Using Context Similarity for Service Recommendation," presented at the Proceedings of the 2010 IEEE Fourth International Conference on Semantic Computing, 2010.

[101] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas*, et al.*, "QoS Aspect Languages and Their Runtime Integration," in *Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1998, pp. 303-318.

[102] J. P. Loyall, R. E. Schantz, J. A. Zinky, and D. E. Bakken, "Specifying and measuring quality of service in distributed object systems," in *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, 1998, pp. 43-52.

[103] Z. Ma, *Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogeneous platforms*. Dordrecht: Springer, 2007.

[104] P. Maes, "Situated agents can have goals," in *Robotics and Autonomous Systems*, 1990, pp. 49-70.

[105] Y. Maurel, A. Diaconescu, and P. Lalanda, "Creating Complex, Adaptable Management Strategies via the Opportunistic Integration of Decentralised Management Resources," in *Proceedings of the 2009 International Conference on Adaptive and Intelligent Systems*, 2009, pp. 86-91.

[106] D. McIlroy, "Mass-produced Software Components," in *Proceedings of Software Engineering Concepts and Techniques*, Garmisch, Germany, 1969, pp. 138-155.

[107] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer,* vol. 37, pp. 56-64, Jul 2004.

[108] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "A Taxonomy of Compositional Adaptation: Technical Report MSU-CSE-04-17," Michigan State University2004.

[109] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style," *SIGSOFT Software*

Engineering Notes, vol. 21, pp. 24-32, 1996.

[110] M. K. Mieczyslaw, "Control Theory-Based Foundations of Self-Controlling Software," *IEEE Intelligent Systems,* vol. 14, pp. 37-45, 1999.

[111] M. Montes-y-Gómez, A. F. Gelbukh, and A. López-López, "Comparison of Conceptual Graphs," in *Proceedings of the Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*, 2000, pp. 548-556.

[112] M. Moriconi and R. A. Reimenschneider, "Introduction to SADL 1.0: A language for specifying software architecture hierarchies," *Technical Report SRICSL-97-01,* 1997.

[113] J. Noble and C. Weir, *Small memory software: patterns for systems with limited memory*: Addison-Wesley Longman Publishing Co., Inc., 2001.

[114] Open Service Gateway Initiative. (2008). *OSGi Service Platform, Release 4*. Available: www.osgi.org

[115] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do Internet services fail, and what can be done about it?," in *Proceedings of the 4th Usenix Symposium on Internet Technologies and Systems*, 2003, pp. 1-15.

[116] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic*, et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems & Their Applications,* vol. 14, pp. 54-62, May-Jun 1999.

[117] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th international conference on Software engineering*, Kyoto, Japan, 1998, pp. 177-186.

[118] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime Software Adaptation: Framework, Approaches, and Styles," *ICSE'08 Proceedings of the Thirtieth International Conference on Software Engineering,* pp. 899-909, 2008.

[119] OSGi. (2007). *Declarative Service Specification*. Available: www.osgi.org

[120] G. A. Papadopoulos and F. Arbab, "Coordination models and languages," *Advances in Computers,* vol. 46, pp. 329-400, 1998.

[121] N. Paspallis, "Middleware-Based Development of Context-Aware Applications with Reusable Components," PhD Thesis, 2009.

[122] N. Paspallis, R. Rouvoy, P. Barone, G. A. Papadopoulos, F. Eliassen, and A. Mamelli, "A Pluggable and Reconfigurable Architecture for a Context-Aware Enabling Middleware System," *On the Move to Meaningful Internet Systems: Otm 2008, Part I,* vol. 5331, pp. 553-570, 2008.

[123] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Engineering Notes,* vol. 17, pp. 40-52, 1992.

[124] M. Peterson, *An introduction to decision theory*. New York: Cambridge University

Press, 2009.

[125] P.-J. Pintens, "An adaptive OSGi robotic application," Master Thesis, Mathmatics and Computer Science, University of Antwerp, Antwerp, 2010.

[126] J. R. Quinlan, *C4.5 : programs for machine learning*. San Mateo, Calif.: Morgan Kaufmann Publishers, 1993.

[127] C. Reade, *Elements of Functional Programming*. Boston, MA, : USA:Addison-Wesley Longman Publishing Co., Inc., 1989.

[128] P. P. D. Redondo, A. F. Vilas, M. R. Cabrer, and J. J. Pazos, "Exploiting OSGi capabilities from MHP applications," *Journal of Virtual Reality and Broadcasting,* vol. 16, 2007.

[129] R. P. D. Redondo, A. F. Vilas, M. R. Cabrer, J. J. P. Arias, J. G. Duque, and A. G. Solla, "Enhancing residential gateways: A semantic OSGI platform," *Ieee Intelligent Systems,* vol. 23, pp. 32-40, 2008.

[130] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, J. Lorenzo, M. Valla*, et al.*, "A comprehensive context modeling framework for pervasive computing systems," in *Proceedings of the 8th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, Oslo, Norway, 2008, pp. 281-295.

[131] J. S. Rellermeyer, "Concierge: A Service Platform for Resource-Constrained Devices," *Operating Systems Review,* vol. 41, p. 245, 2007.

[132] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: distributed applications through software modularization," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Newport Beach, California, 2007, pp. 1-20.

[133] A. Restivo and A. Aguiar, "Towards detecting and solving aspect conflicts and interferences using unit tests," in *Proceedings of the 5th workshop on Software engineering properties of languages and aspect technologies*, Vancouver, British Columbia, Canada, 2007.

[134] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo*, et al.*, "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments," in *Software Engineering for Self-Adaptive Systems*. vol. 5525, B. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., ed: Springer Berlin Heidelberg, 2009, pp. 164-182.

[135] R. Sabharwal, "Grid Infrastructure Deployment using SmartFrog Technology," in *Proceedings of the International conference on Networking and Services*, 2006, p. 73.

[136] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape and Research Challenges," *ACM Transactions on Autonomous and Adaptive Systems,* vol. 4, pp. 14-55, May 2009.

[137] M. Shaw and D. Garland, *Software Architecture: Perspectives on an Emerging*

Discipline vol. 6. Upper Saddle River, NJ: Prentice Hall, 1996.

[138] S. Sicard, F. Boyer, and N. De Palma, "Using Components for Architecture-Based Management," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, 2008, pp. 101-110.

[139] C. Smith. (2008). *Jess: the Rule Engine for the Java Platform*. Available: http:// www.jessrules.com/jess/index.shtml

[140] B. Srivastava, "The Case for Automated Planning in Autonomic Computing," in *Proceedings of the Second International Conference on Automatic Computing*, 2005, pp. 331-332.

[141] B. Srivastava, J. P. Bigus, and D. A. Schlosnagle, "Bringing Planning to Autonomic Applications with ABLE," in *Proceedings of the First International Conference on Autonomic Computing*, 2004, pp. 154-161.

[142] N. Stankovic, "An open Java system for SPMD programming," *Concurrency-Practice and Experience,* vol. 12, pp. 1051-1076, Sep 2000.

[143] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *Ieee Transactions on Software Engineering,* vol. 23, pp. 759-776, 1997.

[144] N. Subramanian and L. Chung, "Software Architecture Adaptability: An NFR Approach," in *Proceedings of the International Workshop on Principles of Software Evolution*, 2001, pp. 10-21.

[145] Sun. *Sun Microsystems: Java Management Extensions*. Available: http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[146] J. Sztipanovits and G. Karsai, "Model-integrated computing," *Computer,* vol. 30, pp. 110-111, Apr 1997.

[147] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2 ed.: Addison-Wesley Professional, 2002.

[148] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural Styles for Runtime Software Adaptation," in *Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2009, pp. 171-180.

[149] G. Tesauro, "Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies," *IEEE Internet Computing,* vol. 11, pp. 22-30, 2007.

[150] G. Valetto, G. E. Kaiser, and G. S. Kc, "A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems," in *Proceedings of the 8th European Workshop on Software Process Technology*, 2001, pp. 102-116.

[151] K. Verma and A. Sheth, "Autonomic Web Processes," in *Service-Oriented Computing - ICSOC 2005*. vol. 3826, B. Benatallah, F. Casati, and P. Traverso,

Eds., 1st ed: Springer Berlin / Heidelberg, 2005, pp. 1-11.

[152] W3C. *Web Ontology Language (OWL), W3C OWL Working Group, v 2.0*. Available: http://www.w3.org/TR/owl2-overview/

[153] M. Weiser, "The Computer for the 21st-Century," *Scientific American,* vol. 265, pp. 94-101, Sep 1991.

[154] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest*, et al.*, "Energy-Aware Runtime Scheduling for Embedded-Multiprocessor SOCs," *IEEE Design & Test,* vol. 18, pp. 46-58, 2001.

[155] Q. Yang, X. C. Yang, and M. W. Xu, "A framework for dynamic software architecture-based self-healing," in *Proceedings of the International Conference on Systems, Man and Cybernetics*, 2005, pp. 2968-2972.

[156] S. J. H. Yang, J. Zhang, and I. Y. L. Chen, "A JESS-enabled context elicitation system for providing context-aware Web services," *Expert Systems with Applications,* vol. 34, pp. 2254-2266, May 2008.