The Hong Kong Polytechnic University

Department of Computing

CAMPUS: A Middleware for Automated Context-aware Adaptation Decisions at Run-time

by WEI Jing Yuan

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Philosophy

March 2009

Statement of Authorship

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

____

Name: WEI Jing Yuan

Abstract

So far in most context-aware systems, the decisions of when and how to adapt an application are made *a priori* by developers during the compile time. While such approaches empower developers with sufficient flexibility to specify what they want in terms of adaptation rules, they inevitably place an immense load on developers, especially in an extremely dynamic environment like pervasive computing, to anticipate and formulate all potential run-time situations during development time. In addition, making adaptation decisions in design-time or compile-time makes it difficult for the system to consistently deliver services of an optimal quality. These challenges motivated us to explore an approach to automating context-aware adaptation decisions by a middleware layer at run-time.

The resulting middleware CAMPUS, short for **C**ontext-**A**ware **M**iddleware for **P**ervasive and **U**biquitous **S**ervice, achieves the objective with the confluence of three key technologies: compositional adaptation, ontology, and DL/FOL reasoning. More specially, we have proposed and designed a new programming model called ATM (short for **A**daptable **T**ask **M**odel) to completely separate context-aware adaptation from the functional concerns of applications. A comprehensive ontological model has been developed to capture important knowledge about context-aware applications built on the basis of the ATM model. Importantly, the middleware layer can perform DL and FOL reasoning on these ontologies to derive the important decisions at run-time. We designed and implemented a middleware prototype that served as a platform for us to evaluate the effectiveness of the system in enabling automated context-aware adaptation decisions and to validate the principles underpinning the design. The CAMPUS implementation has been evaluated with a number of case studies to validate the operation of the system on a realistic environment and to provide us with opportunity to obtain experimental results for further analysis. In particular, we have selected and implemented a context-aware instance messenger application to run over the CAMPUS. We systematically traced the application development cycle and validate the effectiveness of the semantic-based approach to capturing contextual, service and adaptation requirements. In capturing the system's performance, we evaluated the potential overheads introduced by deferring the adaptation decision to run-time in the middleware level. The results are significant in that they show that CAMPUS can be adapted to run on resource-constraint portable devices without significant degradation in its performance.

Acknowledgements

There are many people I would like to thank for influencing me and helping me to complete this thesis, which has been a difficult yet rewarding process.

First and foremost, I would like to thank my supervisor, Dr. Alvin Chan, for his valuable guidance and support. He taught me about research and writing, and gave me enough freedom and trust to choose the direction in which I wanted to work. He has shown great patience with me and provided valuable comments on my papers and thesis. I would like to thank Alvin even more for his friendship, and for having been so supportive. It has been my great pleasure to work under his supervision in the past three years.

I would also like to thank Dr. Jiannong Cao and Dr. Henry Chan, for their valuable advice on my study and for graciously agreeing to serve as my referees. Special thanks also go to Mrs. May Chu from the Research Office, for her patience and understanding during my frequent consultations with her. I would further like to express my appreciation for Dr. Richard Moore from the English Learning Center. I learned a great deal in his classes and had fun. Others I would like to thank include Mrs. Miu Tai, and the departmental tech team. All of them extended much help to me during these past three years of study.

Last, but not least, I am indebted to my family, for their love and unconditional support, and for more than I will ever be able to express. Thank you!

List of Publications

Edwin J. Y. Wei, and Alvin T. S. Chan, "Towards Context-Awareness in Ubiquitous Computing," *Proceedings of the International Conference on Embedded and Ubiquitous Computing (EUC 2007)*, LNCS 4808, pp. 706-717, Taipei, Taiwan, Dec. 2007

Edwin J. Y. Wei, and Alvin T. S. Chan, "Semantic Approach to Middleware-driven Run-time Context-aware Adaptation Decision," *Proceedings of the IEEE International Conference on Semantic Computing (IEEE-ICSC 2008)*, pages 440-447, Santa Clara, CA, USA, Aug. 2008.

Edwin J. Y. Wei, and Alvin T. S. Chan, "CAMPUS: A Middleware for Automated Context-aware Adaptation Decisions at Run-time," *submitted to IEEE Transactions on Software Engineering*.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background

Context-aware adaptation refers to the ability of computing systems to adapt their behaviors or structures to highly dynamic environments without explicit intervention from users, with the ultimate aim of improving the user experience of these computing systems. In recent years, we have witnessed a proliferation of context-aware computing platforms that have adapted themselves using situation information. The Active Badge location system, proposed in the early 1990s, was one of the first context-aware systems. Want *et al.* [Want92] developed a phone call redirection application that employed periodic pulse-width modulated infrared signals to determine a user's current location. A couple of location-aware tour guides [Abowd97, Cheverst00] emerged in the middle of the 1990s. They used knowledge of the users' current and past location information to provide services that are expected from a real tour guide; for example, offering information relating to objects and people of interest in the physical world. Location information is by far the most frequently used attribute of context. However, in recent years, other context information is increasingly being employed. For example, a notepad application in the TEA [Schmidt99] project can adapt its display font size to a user's activity so that it changes depending on whether the user is walking or stationary, or adapts to the available light level. SenSay [Siewiorek03] is a context-aware mobile phone that modifies its behavior based on what the user is doing and where he is. For instance, when the user is involved in a conversation or has an important event scheduled in the electronic calendar, all incoming calls are automatically answered with an SMS message.

The need for such context-aware systems has grown because of the emerging paradigm of pervasive computing, or ubiquitous computing. Many computing devices, such as PDAs, cellular smart phones and notebooks, exhibit a high degree of mobility; their computational systems therefore need to adapt to the heterogeneous and dynamic surrounding environments in which they are operating. For example, taking into account the quality of network connectivity, which varies in terms of bandwidth fluctuations and error rates, streaming applications may use different transcoding protocols to guarantee the video quality. Everyday devices, such as digital cameras and watches, are now equipped with computing capabilities. It has thus become necessary for computational systems to consider the contextual attributes of neighboring devices and local resources to optimize the users' experience. For instance, it would be undesirable for a desktop application to present output that is unreadable on a small screen when the application migrates to a handheld computing device [Satyanarayanan04]. Moreover, as sensor technology continues to progress and advance with respect to issues such as size, power consumption, computing capability and cost, existing and evolving classes of contextual information will be made available for software platforms to further improve the experience of users. For example, biosensors, which measure physiological data such as pulse, skin temperature, and galvanic resistance to capture data about the physical states of users, can be employed to recognize the users' emotional information. Applications might then be able to adapt to human emotions, for example, by soothing a user that it perceives to be angry.

1.2 Motivation and Problem Statement

Previous works have demonstrated the potential of context-aware applications, but have also uncovered many challenges in designing, developing, and maintaining such systems. Instead of focusing on coding the actual service logics, developers are often distracted by context-related issues such as how to capture and represent the contextual information concerned, and when and how to adapt to the contextual changes in the operating environment. The emerging paradigm of pervasive computing, which envisions a world in which users can manage their information anywhere, at anytime, and on any device, is further complicating the development of such context-aware applications. The inherent heterogeneity of a pervasive computing environment requires applications to consider many more varieties of contextual information than ever before. Context-aware adaptation will also occur much more frequently than before, due to the high mobility of portable computing systems across a pervasive computing environment. Facilities must be provided to conceal the complexity of these issues from developers of context-aware applications and to ease the development process. However, thus far in most context-aware systems, such as [Dowling01, Yau02, Davis04, Zheng06], the decisions of when and how to adapt an application are made *a priori* by developers during the compile time. In general, developers are provided with a set of declarative scripts and/or programming APIs to dictate which aspects of contexts are relevant to the execution of the applications, and when and how the applications should adapt to relevant changes in context.

While such approaches empower developers with sufficient flexibility to specify what they want in terms of adaptation rules, they inevitably place an immense load on developers, especially in an extremely dynamic environment like pervasive computing, to anticipate and formulate all potential run-time situations during development time. For example, assume a computationally intensive mobile application that can adapt its behavior to various contexts, including CPU usage, memory availability, network speed, and battery level. Assuming that each context can equate to one of the four values of *worst*, *bad*, *good*, or *best*, the adaptation rule pattern of this application may take the following form: *IF (CPU_Usage is* a_i*) AND (Memory_Availability is* b_i*) AND (Network_Speed is* c_i*) AND (Battery_Level is* d_i*) THEN (Action* e_i*)*. In the worst situation, the adaptation policy may have as many as *256* rules. Maintaining such a large base of rules is not an easy task, and will distract the focus of development from actual application logic.

In addition, making adaptation decisions in design-time or compile-time makes it difficult for the system to consistently deliver services of an optimal quality, which is one of the most important motivations for context-aware adaptation. Consider the following scenario: a particular action e_j in the above example is expected to be triggered under the context combination of CPU usage a_j, memory availability b_j, network speed c_j, and battery level d_j. However, due to unstable network connectivity, the action e_j fails to execute. If developers did not take such an exception into account in advance, then no rule will be predefined to respond to such a context situation. As a result, the application will simply have to throw a notification error at best, resulting in an unsatisfactory user experience. Even supposing that developers have considered such a situation, with an appropriate established rule in place as an alternative action to be taken in order to deal with this exceptional case, the alternative action may also fail for other unconsidered factors caused by the fluctuant situation of a mobile environment. At the root of such conflict is the fact that, with adaptation decisions made at the development time, all adaptation strategies have been predetermined by developers based on the approach of looking ahead when formulating rules. Yet, except for a restricted number of contexts and constrained operating environments, it is impractical to consider all possible run-time situations, especially in an extremely dynamic environment such as pervasive computing. At best, developers will tend to cater to common cases rather than consider optimal solutions.

These challenges have motivated us to explore an approach to automating context-aware adaptation decisions by a middleware layer at run-time. The middleware has three main functions:

- Reasoning about context changes.
- Making decisions about what adaptation to perform.
- Implementing the adaptation choices.

Through automated decisions at run-time, developers will be freed from the need to predict, formulate, and maintain adaptation rules, thereby greatly reducing the efforts required to develop context-aware applications. It will also be possible to deliver services of an optimal quality by deferring the adaptation decisions until run-time to account for up-to-date contextual conditions.

1.3 Approach and Contributions

The main objective of this study is to design, implement and evaluate a middleware layer that can be used to automate context-aware adaptation decisions at run-time, based on an analysis of the problems posed by the above-mentioned challenges. Importantly, it aims to provide a balanced level of programming abstractions that will make it easy to develop context-aware adaptation for pervasive applications, while facilitating automated adaptation decisions at run-time. The resulting middleware CAMPUS, short for **C**ontext-**A**ware **M**iddleware for **P**ervasive and **U**biquitous **S**ervice, achieves the objective with the confluence of three key technologies: compositional adaptation, ontology, and DL/FOL reasoning. In particular, CAMPUS proposes a new programming model based on compositional adaptation to construct context-aware applications and facilitate adaptation decisions. CAMPUS also formulates a comprehensive ontology-based model to capture the important concepts and relationships of entities in the programming model, which are necessary for automated context-aware adaptation decisions. Based on these ontologies, CAMPUS makes use of description logic and first-order logic to infer and make context-aware adaptation decisions automatically. The following is an overview of the main research contributions.

- Proposed and designed a new programming model to completely separate context-aware adaptation from the functional concerns of applications. Context-aware adaptation and the functional concerns of applications are often tightly coupled and intertwined. Dealing with adaptation and the basic functional concerns of applications at the same time and at the same level pushes developers into an awkward position mentioned previously. In this study, we investigated the principle of separation-of-concerns to support context-aware applications. This has led to the development of a novel programming model that makes it possible for the middleware layer to automate the decision process of context-aware adaptation.

- Defined a set of ontologies for supporting automated context-aware adaptation decisions. These ontologies are expressed using the OWL-DL language, which captures the knowledge that represents the important concepts and relationships involved in the development of context-aware applications. Based on these ontologies, a middleware layer can make use of description logics (DL) and first-order logics (FOL) to infer and make context-aware adaptation decisions. In particular, this study showed that not only can the OWL be used to express the semantics of information on the web, but that it can be used to express the semantics of entities in the domain of software development, including applications and contextual information.

- Designed and implemented a middleware, CAMPUS, which served as a platform for us to evaluate how effective the system is at enabling automated context-aware adaptation decisions and to validate the principles underpinning the design. We evaluated the CAMPUS implementation by using a number of case studies to validate the operation of the system in a realistic environment and to obtain experimental results for further analysis. In particular, we selected and implemented a context-aware instance messenger application to run over the CAMPUS. We systematically traced the application development cycle and validate the effectiveness of the semantic-based approach to capturing contextual, service, and adaptation requirements. The experiments also presented us with the opportunity to study the interactions between the core modules of the system and their adaptation response to changing contexts. In capturing the system's performance, we evaluated the potential overheads introduced by deferring the adaptation decision to run-time in the middleware level. The results are significant: we found that CAMPUS can be adapted to run on resource-constraint portable devices without significant degradation in its performance.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 describes the work related to this study. Past studies on context-aware middleware are surveyed and compared with the CAMPUS middleware. The aim is to highlight the core contributions of these works and how they are benchmarked against the CAMPUS middleware. Some well-known works on context ontologies are also introduced and compared with the CAMPUS ontologies. The specific characteristics of the CAMPUS middleware and its advantages over other similar works are highlighted.

Chapters 3 to Chapter 7 form the core of this thesis. Chapter 3 is devoted to the architecture of the CAMPUS middleware. Chapter 4 focuses on the programming model. The model effectively separates adaptation from the functional concerns of context-aware applications and makes it possible for the middleware layer to automate context-aware adaptation decisions at run-time. Presented in Chapter 5 is the comprehensive set of ontologies that can be used to describe the necessary semantic information for the middleware to make context-aware adaptation decisions. In Chapter 6, details are given of the mechanisms used in CAMPUS to dynamically derive adaptation decisions. Chapter 7 contains a discussion of the implementation of CAMPUS middleware.

In Chapter 8, a sample application is presented that demonstrates the feasibility and validates the benefits of CAMPUS in providing context-aware adaptation. The performance and evaluation of the system are discussed in Chapter 9. Three operations that show the most promise of bringing about overheads to the system are measured independently. A complete end-to-end application that fully exercises the system components of CAMPUS is set up to evaluate the system performance.

Finally, Chapter 10 presents the conclusions of this study. It also points out some directions for future research on this topic. Such work is necessary to make the CAMPUS more complete, secure, and robust for deployment over a wide-scale wireless and mobile environment.

Chapter 2 Related Work

As presented in Chapter 1, one of the important contributions of our study is to leverage and apply the concept of semantic ontologies to facilitate automated context-aware adaptation decisions. The aim of this chapter is to provide a general review of recent developments on middlewares that provide applications with the support of context-aware adaptation, as well as on some well-known context ontologies that represent, structure, and organize contextual data and relationships between them. Importantly, this chapter serves to provide a comprehensive background of related works that have greatly motivated the design of CAMPUS, while significantly setting it apart from previous systems.

2.1 Middlewares for Context-aware Adaptation

The need for a middleware layer to facilitate context-aware adaptation for applications has been widely reported and acknowledged in the research community [Chan03, Capra03]. Introduced in this section are several typical middleware systems that provide applications with facilities to ease context-aware adaptation. They are compared with CAMPUS in terms of the underlying mechanisms that they use to make adaptation decisions.

2.1.1 Odyssey

Odyssey [Noble97] provides an application-aware approach to adaptation. The essence of this model is a collaborative partnership between the system and individual applications. The system monitors resource levels, notifies applications of relevant changes, and enforces resources-allocation decisions. Each application independently decides how best to adapt when notified. For example, Odyssey itself does not decide that color video frames should be converted to black-and-white, but

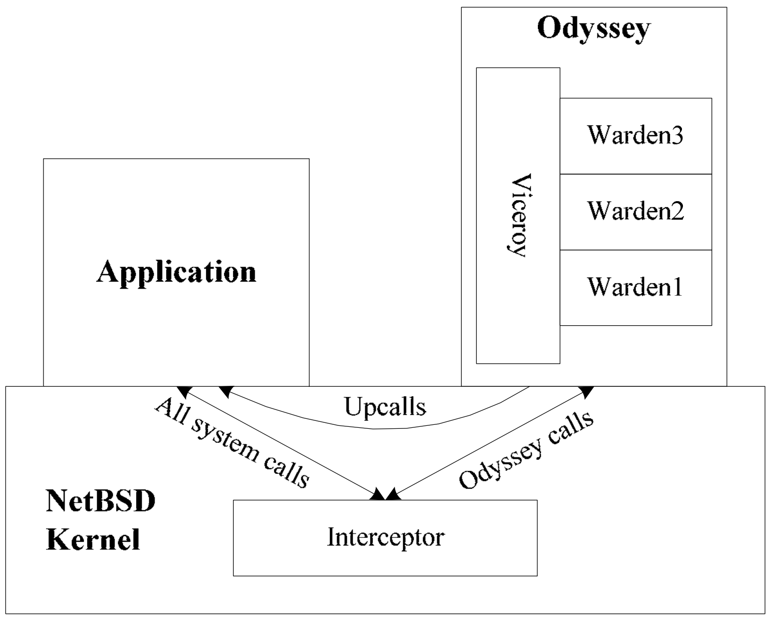


Figure 2-1 The Odyssey Client Architecture. [Noble97]

rather instructs the application that some action is required. The application itself decides how adaptation should occur, and typically instructs the server to make the adjustment.

As shown in Figure 2-1, the Odyssey system consists of a viceroy, an operating system entity in charge of managing the limited resources for multiple processes, a set of data type-specific wardens that handle the intercommunications between clients and servers, and applications that negotiate with Odyssey to receive the best level of services available. Applications request from Odyssey the resources that they need, specifying the window of tolerance required for the desired operation. If resources within that window are currently available, the request is granted and the client application is connected to its server through the appropriate warden for the data type to be transmitted. Wardens can handle issues like caching or pre-fetching in manners specific to their data types, in order to make the best use of the available resources. If resources within the requested window are not available, the application is notified and can subsequently request a lower window of tolerance and corresponding level of service. As conditions change and requests that had previously been satisfied can no longer be met (or, conversely, conditions improve dramatically), the viceroy uses upcalls that had previously been registered by the applications to send notifications, in the form of events, to these applications. These notifications indicate that the applications must operate in a different window of tolerance, which may subsequently potentially alter their behaviors.

2.1.2 MobiPADS

MobiPADS [Chan03] is designed to support context-aware processing by providing an executing platform to enable active service deployment and the reconfiguration of service compositions in response to varying contexts in the operating environment. It supports dynamic adaptation at both the middleware and application layers to provide a flexible configuration of resources to optimize the operations of the mobile applications.

Within the MobiPADS system, a series of *mobilets* is linked together to form a processing chain called the *service chain*, which reacts and adapts to the varying characteristics of a wireless environment. In the MobiPADS service space, mobilets exist in pairs: a master mobilet resides at the MobiPADS client and a slave mobilet resides at the MobiPADS server. Mobilets access the services of the system components through the mobilet API, which also provides interfaces to allow the system components to communicate and configure the mobilets. At the top level of the service space, there is a set of meta-objects that reflects the configuration for the composite events and service chain, as well as the adaptation policies.

The MobiPADS achieves context-awareness by using an event notification model, which monitors the status of all contexts of interest and reports the event to the subscribed entities. These include all of the entities within the platform such as the system components, the mobilets, and the mobile application. On detecting changes in the environment, the MobiPADS system can respond either by reconfiguring the current service chain or by communicating the changes to each of the mobilets. An abstraction of service object interactions and configurations is expressed in a high-level declarative language written in XML format. Based on these profiles, the MobiPADS system can respond to changes of context by adding and removing mobilets within the service chain to select an optimum set of mobilets. Simply adding or removing mobilets within the service chain may not be enough to adapt to contextual changes. To allow a finer-grained adaptation, the MobiPADS system allows the mobilets to subscribe to an event and react to the event message by adjusting its internal parameters to best adapt to the changes.

2.1.3 RCSM

Yau *et al.* [Yau02] proposed a Reconfigurable Context-Sensitive Middleware (RCSM) to facilitate the development and run-time operation of context-aware applications. RCSM models context-aware applications as context-sensitive objects, which consist of two parts: a context-sensitive interface and a context-independent implementation. The interface encapsulates the description of the application's context awareness. More specifically, this interface lists the types of contexts or situations that are relevant to the application, the actions to be triggered, and the timing of these actions. The second part is the actual implementation of the actions that the application must provide.

RCSM provides application developers with a **C**ontext-**A**ware **I**nterface **D**efinition **L**anguage (CA-IDL) that can be used to specify the context-sensitive object interfaces. Figure 2-2 shows such a context-sensitive interface using CA-IDL. The CA-IDL interfaces are compiled into custom-made **A**daptive **O**bject **C**ontainers (ADCs) that communicate with the underlying system to acquire contexts, and then perform periodic context analysis as specified in the context-sensitive interfaces. These ADCs are also responsible for activating different actions whenever they

```
// context source
RCSMContext dc {
        char[] string location;
        boolean light;
}

// beginning of context-sensitive interface
interface instructor_object {
        // context variables
        RCSMContext_var dc C1
                where location="screen";
        RCSMContext_var dc C2
                where light=true;
        RCSMContext_var dc C3
                where light=false;

        // context-sensitive method
        [outgoing]
        [activate when C1^(C2->C3)]
        void distribute (string lectures);
};
// end of context-sensitive interface
```

Figure 2-2 A Context-sensitive Interface Using CA-IDL. [Yau02]

detect suitable contexts as a result of the context analysis. RCSM also provides **R**CSM **O**bject **R**equest **B**rokers (R-ORBs) that perform a proactive device discovery during the execution of the application, and use their R-GIOP (**R**CSM **G**eneral **I**nter-**O**RB **P**rotocol) to establish and maintain a CTC (**C**ontext-**T**riggered **C**ommunication channel) with a remote device, in order to collect the data from sensors and the operating system.

2.1.4 Rocks

Zandy *et al.* [Zandy02] at the University of Wisconsin developed reliable sockets (as known as *Rocks*) to protect socket-based applications from poor network conditions, such as unexpected modem disconnections and IP address changes as a result of mobile device movements or a DHCP lease expiration.

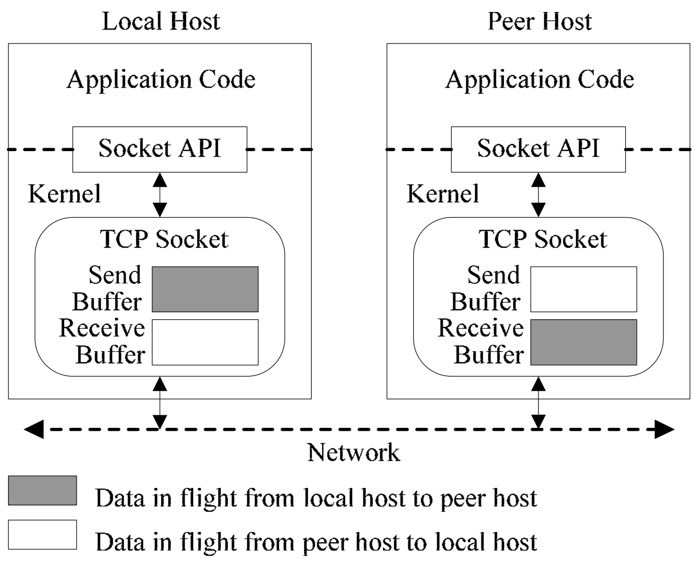


Figure 2-3 The Rocks Architecture. [Zandy02]

Rocks resume sessions automatically after recovering from a period of disconnection. Using the *preloading* feature of the Linux loader, the Rocks library is interposed between the application code and the kernel TCP socket, as shown in Figure 2-3. Rocks monitor the send and receive buffers of TCP socket, and maintain a copy of in-flight packets to prevent data loss in the event of a connection failure. After reconnection, Rocks will initially resend those packets that are cached in the in-flight buffers, and then resume the normal TCP socket operation. The Rocks library exports the socket API, which is the same as the kernel socket API, to be used transparently by the application. This middleware interception approach means that the Rocks reconfiguration is transparent to the application code, as well as to any distribution middleware or virtual machine. The reliability provided in Rocks is independent of specific applications; hence, Rocks is also transparent to the adaptive code.

2.1.5 MUSIC

The MUSIC middleware [Rouvoy08] is an autonomous platform for supporting self-adaptive mobile applications. It allows an appropriate application configuration to be automatically selected among all possible application configurations. The process of adapting applications in respond to changes of context includes a planning procedure and a reconfiguration process. The former decides appropriate application configurations, and the latter deploys them. We introduce these two processes in this section.

The planning procedure is supported by the Adaptation Reasoner, which makes adaptation decisions based on configuration plans and utility functions. A configuration plan in MUSIC defines how the components are connected to each other in order to provide the functionality required by the applications. For any particular application, there may be multiple configuration plans that can achieve its functionality. The Adaptation Reasoner decides an appropriate configuration plan based on the utility it offers to the system. During the planning procedure, the reasoner asks the plan repository for plans that are compatible with a given service type. After that, the reasoner recursively resolves the dependencies of the plans to build a service configuration, and discards configurations whose explicit or implicit dependencies remain unresolved. Finally, the service configurations are ranked by evaluating their utilities. MUSIC uses utility functions to map the user preferences for QoS to a function that defines how a selected plan satisfies the user preference. The input of a utility function includes the user preferences considering the current context and the available resources, while its output is the degree to which a configuration plan satisfies the user goals.

The reconfiguration process is handled by the Configuration Executor, which takes the set of plans selected by the Adaptation Reasoner and reconfigures the application.

During the reconfiguration process, the Configuration Executor set the current service into a quiescence state and deploys the service configuration selected by the Adaptation Reasoner. If the configuration indicates a service instance, the configurator connects this instance to other services; if the configuration describes a composite or an atomic service, the service should be created and deployed using the blueprint descriptions enclosed within the configuration.

2.1.6 Summary

Odyssey represents an early effort to facilitate dynamic adaptation. Odyssey and other similar works, such as [Friday96, Blair00], use an application-aware approach to adaptation. That is, the bulk of the adaptation in this model is, in fact, done by the underlying applications. The middleware monitors context, notifies applications of relevant changes, and enforces adaptation decisions made by the applications independently when notified. Such an application-aware approach presents the opportunity for operating applications to potentially adapt their internal logics in response to contextual changes. With utmost flexibility comes rigidity, such that application developers are required to intricately capture all possible contextual changes of interest and, if necessary, to enforce adaptation policies for the applications. In addition, applications have to be modified and re-compiled for any changes in adaptation strategies or to cater to new and evolving contexts.

MobiPADS is another kind of middleware to support context-aware adaptation. It supports dynamic adaptation at both the middleware and application layers. Similar to Odyssey, MobiPADS monitors the status of contexts of interest and reports any changes of events directly to the applications. The underlying applications are responsible for interpreting the contexts and for deciding how to best adapt to changes of context. To enable more flexible adaptation, MobiPADS employs the reflective mechanism to dynamically reconfigure the service chain based on descriptive adaptation policies regulated externally by XML. Importantly, these policies can be changed after the deployment of applications. Using declarative adaptation policies can be seen as semi-application-transparent, since the adaptation process is transparent to the application, while the decision is handled by the application.

RCSM is another semi-application-transparent example. The adaptation policies are specified using CA-IDL and compiled into an application skeleton. To change the policies, the application developers merely need to change the interface and re-compile it to generate a new adaptive object container (ADC). In comparison with application-aware approaches, semi-application-transparent architectures are much more flexible in that the adaptation process is transparent to developers, and may be modified even after the deployment of applications. However, the challenges of formulating adaptation decisions to operate under an extremely dynamic environment have yet to be addressed.

Unlike MobiPADS and RCSM, the Rocks project uses a completely application-transparent approach to context-aware adaptation. This is achieved by constructing a layer of adaptable common services within the middleware, while applications are required to make explicit calls to these adaptive services. Using such an approach, applications are oblivious of the need to consider how to adapt and when to adapt, and can completely focus on their functional logic. However, such an approach of adaptive services amalgamates adaptation concerns with functional concerns in the middleware layer, so that only programs that are written for these specific services can be supported. Importantly, regardless of the point at which adaptation decisions being made in all of the surveyed middleware, the decisions are inherently static and predefined. At best, these decisions are formulated based on looking ahead and predicting what contexts will be available to drive the adaptation decisions.

CAMPUS goes one step further than previous approaches in advocating automated run-time adaptation decisions instead of predefined adaptation policies that capture limited contextual changes operating in a potentially dynamic environment. The core aim of CAMPUS is to provide a software engineering solution that seamlessly integrates contextual awareness to application development, while providing a high-level semantic-based expression of adaptation policies. Importantly, CAMPUS aims to avoid the formulation of rigid adaptation rule languages, and to make use of the semantic nature of contexts and applications to automatically reason about when and how to adapt applications in response to changes of context. Furthermore, the CAMPUS middleware supports a more general approach to adaptation by completely separating adaptation from computational concerns, while not limiting itself to particular types of applications.

MUSIC and several other middlewares, such as [Ma06, Rouvoy08], share similar objectives with our CAMPUS middleware. They also aim to avoid formulation of complex adaptation rules and advocate automated adaptation decisions. To the best of our knowledge, all of them make adaptation decisions simply based on utility functions. The major differences between these middlewares and CAMPUS, which uses semantic based DL/FOL reasoning to achieve automated adaptation decisions, are manifold. First, unlike CAMPUS, they do not provide a common terminology and shared set of concepts that agents can use when they interact with each other. This problem is especially acute in the realm of pervasive computing environments since different agents could have different understandings of the current context. They might use different terms to describe context, and even if they use the same terms, they might attach different semantics to these terms. In addition, although utility functions are the natural way to represent value, it is often difficult to apply appropriate calculations of expected utilities for various components of a large, complex system [Walsh04, Chang05, Alia07]. Finally, utility function represents a single-phase decision model that may involve the unnecessary processing of a large volume of information, and does not allow for multiple strategies to be utilized within a single decision process [Shao06].

Table 2-1 is a summarized comparison of the CAMPUS system with the middlewares introduced above. The notable points are shown below:

- *Decision Maker* in these middleware systems can be application or middleware, depending on whether the decision of when and how to adapt to changes of contexts are made in the application layer by developers, or the middleware attempts to completely shield the application from such decisions.
- *Decision Range* is the collection of applications supported by these middleware. Some middleware provide general machinery to support collection of unrelated applications, while others probably only support a specific application or narrowly-defned class of applications.
- *Decision Type* describe whether the context decisions are predefined or not. Some middlewares decided the adaptation strategies during the design-time or compile-time. Such decisions are deemed as static. Rather, others make the decisions dynamically during the time to account for the up-to-date contexts.
- *Decision Mechanism* is the primary technology used for the decision maker to determine the adapation decisions. As far as CAMPUS system is concerned, semantic-based DL/FOL reasoning is the major technology used to make decisions.

Table 2-1 Summary of Middlewares for Context-aware Adaptation Decisions.

	Decision Maker	**Decision Range**	**Decision Type**	**Decision Mechanism**
Odyssey	Application	Application-specific	Static	API

MobiPADS	Application	General	Static	API Scripts
RCSM	Application	General	Static	Scripts
Rocks	Middleware	Application-specific	Static	API
MUSIC	Middleware	General	Dynamic	Utility functions
CAMPUS	Middleware	General	Dynamic	Semantic-based DL/FOL reasoning

2.2 Ontologies for Context-aware Adaptation

Using ontologies to facilitate context-awareness is not a new idea. Some forms of ontology-based models have also been applied to capture concepts and relationships in the domain of context-aware applications. This section reviews several representative context ontologies and points out how the CAMPUS ontologies differ from them.

2.2.1 CONON

CONON [Wang04] is an OWL encoded context ontology for modeling context in pervasive computing environments, and for supporting logic-based context reasoning. It provides an upper context ontology that captures general concepts about basic contexts, and also provides extensibility for adding domain-specific ontology in a hierarchical manner. Figure 2-4 shows the upper context ontology. The context model is structured around a set of abstract entities, each describing a physical or conceptual object, namely *Person*, *Activity*, *Computational Entity* and *Location*, as

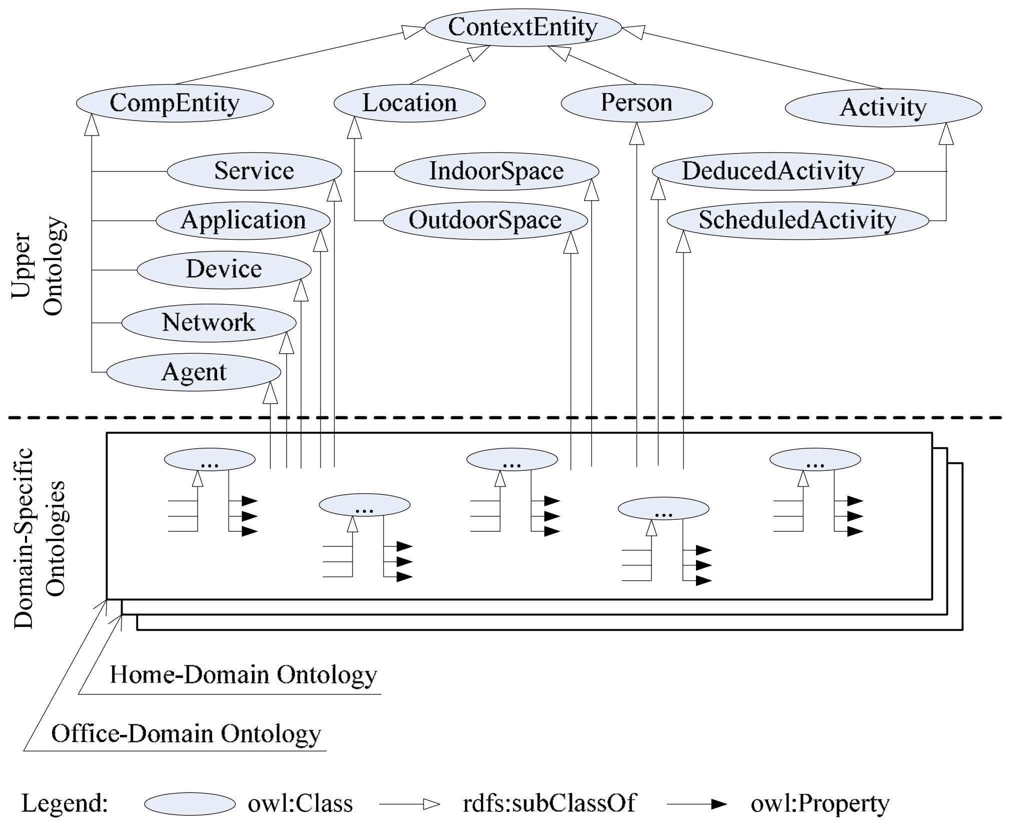


Figure 2-4 The CONON Upper Ontology. [Wang04]

well as a set of abstract sub-classes. Each entity is associated with its attributes and relations with other entities. The built-in OWL property *owl:subClassOf* allows for a hierarchical structuring of sub-class entities, thus providing extensions to add new concepts that are required in a specific domain. Besides general classes defined in the CONON upper ontology, a number of concrete sub-classes are defined to model specific contexts in a given environment. For example, the abstract class *IndoorSpace* of the home domain is classified into the four sub-classes: *Building*, *Room*, *Corridor,* and *Entry*.

2.2.2 COBRA-ONT

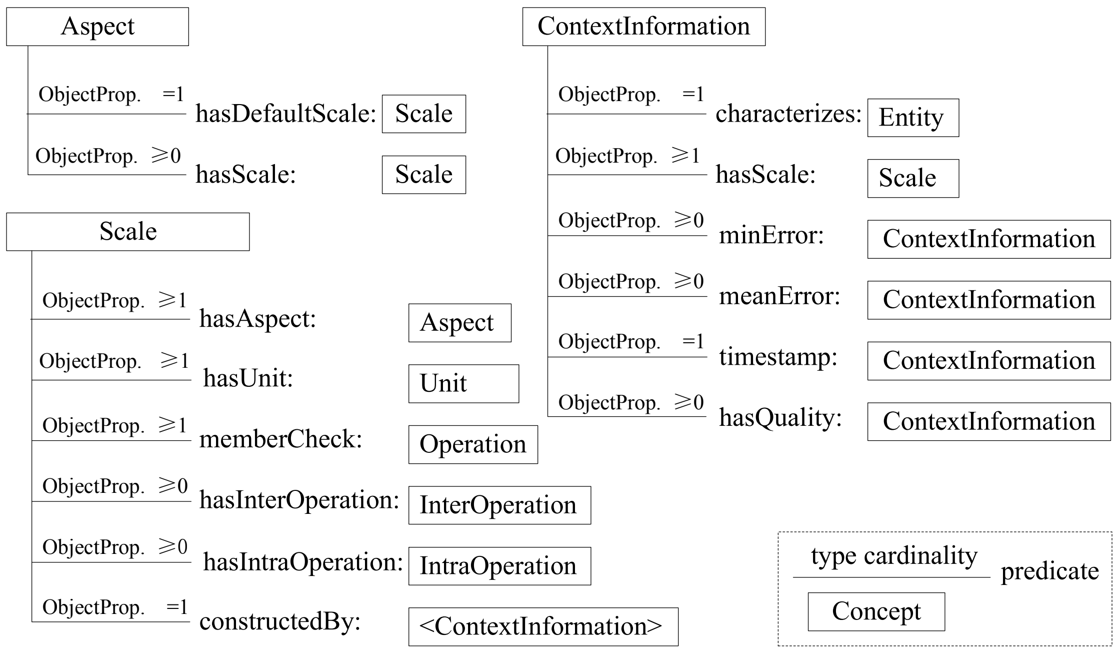
Chen *et al.* [Chen04] used RDF and OWL to define ontologies of context, which provide an explicit semantic representation of context that is suitable for reasoning and for the sharing of knowledge. The COBRA-ONT was designed to support smart meeting room applications for eBiquity group meeting at UMBC. It covers typical concepts associated with information on the geography of the UMBC campus, eBiquity group meetings, and actions performed by the smart meeting applications.

The eBiquity Geo-Spatial Ontology defines vocabularies for modeling certain physical places located on the UMBC campus and their spatial relations and constraints. In particular, it defines ontology classes for symbolic representations of rooms, buildings, campus, states, and countries. It also defines instances of these geo-spatial classes and the associated relations. The eBiquity Meeting Ontology covers key concepts including the modeling of eBiquity group membership, the friends of the eBiquity group members, and the meeting contexts such as descriptions about the speaker of the presentation, the organizer of the meeting, the attendees at the meeting, the presentation video file, event photos, and voice recordings of the discussions. Finally, the aim of the eBiquity Action Ontology is to support the protection of privacy in a context broker. It defines the communication vocabularies between a context broker and other agents.

2.2.3 CoOL

The CoOL (**Co**ntext **O**ntology **L**anguage) [Strang03] is not a single, monolithic language but a collection of several fragments that are grouped into two subsets. The first subset, CoOL Core, is a projection of the Aspect-Scale-Context (ASC) model into two different common ontology languages: OWL/DAML+OIL and F-logic. The second subset, CoOL Integration, is a collection of schema and protocol extensions as well as common sub-concepts of the ASC model, enabling CoOL Core to be used in several service frameworks, particularly Web Services.

Figure 2-5 shows the ASC model that is named after the core concepts of the model, which are aspect, scale, and context information. An aspect is a classification



Aspect
ObjectProp. =1 hasDefaultScale: Scale
ObjectProp. ≥0 hasScale: Scale
Scale
ObjectProp. ≥1 hasAspect: Aspect
ObjectProp. ≥1 hasUnit: Unit
ObjectProp. ≥1 memberCheck: Operation
ObjectProp. ≥0 hasInterOperation: InterOperation
ObjectProp. ≥0 hasIntraOperation: IntraOperation
ObjectProp. =1 constructedBy: <ContextInformation>
ContextInformation
ObjectProp. =1 characterizes: Entity
ObjectProp. ≥1 hasScale: Scale
ObjectProp. ≥0 minError: ContextInformation
ObjectProp. ≥0 meanError: ContextInformation
ObjectProp. =1 timestamp: ContextInformation
ObjectProp. ≥0 hasQuality: ContextInformation
type cardinality predicate
Concept

Figure 2-5 The Aspect-Scale-Context (ASC) Model. [Strang03]

(symbol- or value-range) whose subsets are a superset of all reachable states, grouped in one or more related dimensions called scale. Context information is any information that can be used to characterize the state of an entity concerning a specific aspect. In other words, valid context information with respect to an aspect is one of the elements of the aspects' scales. For example, the aspect "*GeographicCoordinateAspect*" may have two scales, "*WGS84Scale*" and "*GaussKruegerScale*," and valid context information may be an object instance created in an object-oriented programming language such as Java with a new *GaussKruegerCoordination("367032," "533074").*

2.2.4 SeCom

Neto *et al.* [Neto05] used OWL to present a domain-independent ontological context model from contextual dimensions: identity (who), location (where), time (when), activity (what), and device profile (how).

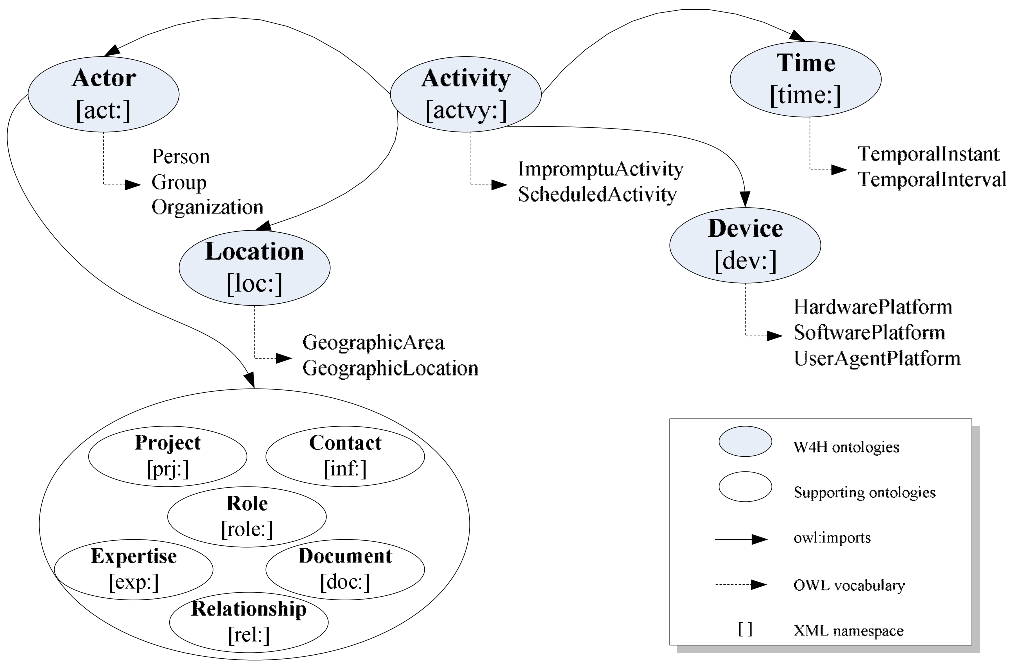


Figure 2-6 An Overview of the SeCom Context Ontologies. [Neto05]

The core of the Who-based ontologies is the Actor ontology. It models the profile of all entities that can perform actions in a pervasive computing environment such as people, groups, and organization, as shown in Figure 2-6. The Actor ontology imports other ontologies, including the role ontology that describes the actors' social role in the real world, the contact ontology that represents the contact information of the different types of actors, the expertise ontology that models areas of knowledge, the relationship ontology that models social relationships between people, the project ontology that describes meta-information associated with projects and the links with actors, and the document ontology that models documents made by actors.

The where-based ontology aims to describe the whereabouts of real world entities. Location information is not only related to such usual information as street, city, and room, but also to geographic coordinates, combined with direction information. Absolute and relative location information is also modeled in this ontology.

The when-based ontology represents temporal information in terms of temporal instants and intervals. A temporal instant is a point on the universal timeline, whereas a temporal interval is delimited by two distinct and convex time instants. Additionally, the when-based ontology models calendar and clock information to represent time in multiple granularities.

The how-based ontology describes computational devices by means of profiles, which includes a set of descriptions that model the features of devices with regard to three platforms: hardware, software, and user agent. The hardware platform describes a device in terms of its input, output, and network features; the software platform represents the application environment, operating system, and installed software; and the user agent platform describes the software browser running on a device.

The what-based ontology describes actions that people do or cause to happen. An activity in this ontology is modeled as of two disjointed types: impromptu and scheduled. The former represents activities that occur in an informal manner, while the latter represents activities planned in terms of time and place.

2.2.5 Summary

To the best of our knowledge, all of the existing ontologies for context-awareness focus on context models that provide abstraction to context entities, and do not aim to provide direct support for adaptation decision. CAMPUS advocates the use of ontology to capture knowledge not only in the context domain, but also the application domain, in order to facilitate adaptation. It has been discovered that the semantics of contextual information and applications in CAMPUS help the middleware understand all of these interacting entities and make important decisions related to adaptation. Table 2-2 offers a comparison of the CAMPUS ontologies and the context ontologies introduced above along three dimensions: *Structure* describes how these ontologies are constructed. This dimension is further divided into two sub-dimensions: *Range* and *Domain*. *Goal* in these ontologies can be general or application-specific, depending on whether they are desinged for specific application-domains or can be used in a general purpose. Finally, *Use* refer to how these ontologies can be use.

Table 2-2 A Comparison of Context Ontologies.

	Structure		**Goal**	**Use**
	Dimension of Range	**Dimension of Domain**		
CONON	Upper ontology Domain-specific ontology	N/A	General	Context modeling Context reasoning
COBRA-ONT	N/A	Geo-Spatial Ontology Meeting Ontology Action Ontology	Application-specific	Context modeling Context reasoning
CoOL	CoOL Core CoOL Integration	N/A	General	Context modeling Context reasoning
SeCom	N/A	Who-based ontologies Where-based ontology When-based ontology What-based ontology How-based ontology	General	Context modeling
CAMPUS	Foundation ontologies Domain-specific ontologies	Context ontologies Tasklet ontologies Service ontologies	General	Context modeling Context reasoning Adaptation decision-making

Chapter 3 The CAMPUS Architecture

As highlighted previously, the main objective of the CAMPUS middleware is to automate context-aware adaptation decisions at run-time for pervasive computing. In order to achieve this objective, the following three important issues need to be considered:

- **Programming model.** This issue refers to the mechanism used to construct context-aware applications and perform context-aware adaptation. The aim of context-aware adaptation decision-making is to choose one or a limited number of application configurations that can be applied in a given context among possibly many alternatives [Alia07]. It is obvious that the challenge of how to make such decisions greatly depends on how the applications are constructed and how they can be reconfigured; thus a suitable programming model directly impacts the automated context-aware adaptation decisions.

- **Knowledge model.** This issue is concerned with the kinds of knowledge that need to be captured and how to represent them. Decisions are made based on the decision maker's knowledge, including knowledge of the characteristics or requirements that each alternative possesses and the effects of each alternative. Therefore, a substantial amount of information needs to be understood by the middleware layer before it can make appropriate decisions. A suitable knowledge model is fundamental to facilitate automated machine understanding.

- **Decision model.** This issue is concerned with the progressive steps in the process of decision-making, how to rate the alternatives, and how to decide on the final one. A suitable decision model that formalizes the process of decision-making in the context of context-aware adaptation will finally determine the quality of the decisions made by the middleware layer.

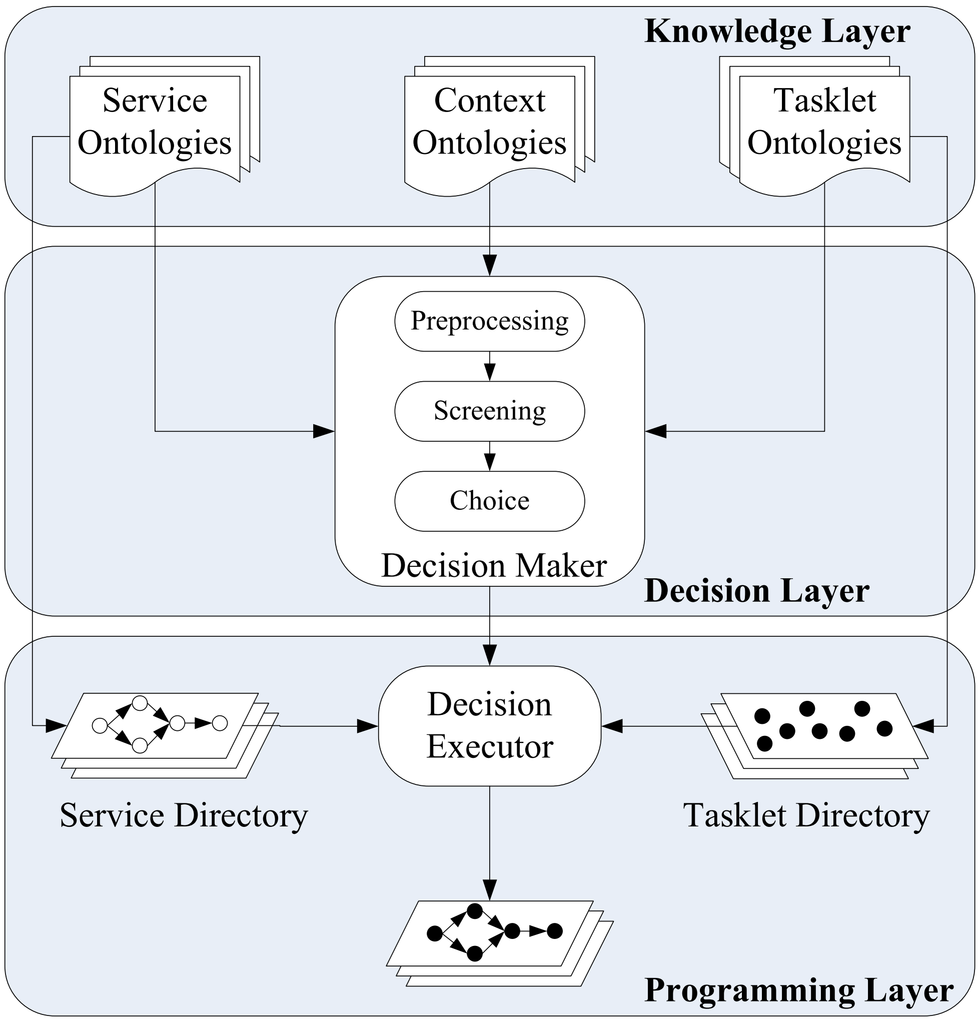


Figure 3-1 The Layered Architecture of the CAMPUS System.

As illustrated in the Figure 3-1, CAMPUS provides solutions to the above issues in a layered architecture. Logically, CAMPUS is divided into a programming layer, a knowledge layer and a decision layer:

- In the programming layer, CAMPUS constructs context-aware applications via a new programming model called ATM (short for **A**daptable **T**ask **M**odel). It relies on two main concepts: *services* and *tasks*. A service is an abstract of a business or a technical process, which is comprised of a series of tasks. Tasks are execution units that perform certain actions to deliver a result to other tasks or the end user. A task is further divided into two parts: a *tasklet* and a *task base*. The former part concentrates on the computational concerns of the task, i.e. how to process data;

and the latter part handles issues involved in coordination and adaptation, such as how to communicate with other modules of the service and how to transfer the states of tasks during adaptation. A service can adapt to contextual information by adding, removing, or replacing the tasklet parts of its tasks.

- In the knowledge layer, CAMPUS captures a comprehensive set of ontologies to describe related entities that are involved in context-aware adaptation, including services, tasklets, and contexts. These ontologies are used by the programming layer to initialize service and tasklet instances, and also used by the decision layer to perform semantic reasoning and to make adaptation decisions.
- In the decision layer, CAMPUS uses description logics and first-order logics to reason about the ontologies in order to make context-aware adaptation decisions. The goal of these decisions is to select the best tasklet alternatives for given tasks. The whole decision-making procedure is divided into three phases: preprocessing, screening, and choice. The preprocessing phase performs several preprocessing tasks to ensure that the ontologies are semantically consistent and to prepare fine-grained information for the subsequent phases. The screening phase filters off tasklet alternatives that are not satisfied by up-to-date contextual information. The remaining filtered tasklets are compared in the choice phase, using the utility function in order to select the best tasklets for given tasks.

A detailed design of CAMPUS is presented in the remaining chapters, while the aim of this chapter is to analyze and justify our major design decisions and approaches.

3.1 The Programming Model

The programming layer of CAMPUS is responsible for constructing and reconfiguring context-aware applications according to the instructions from the decision layer. This is achieved by the ATM programming model that is based on compositional adaptation and separation-of-concerns.

In general, approaches to realize context-aware adaptation can be classified into transformational adaptation and compositional adaptation [Tekinerdogan96, Edwin07]. In transformational adaptation, applications directly modify related specifications and/or implementations to respond to changing contexts. Compositional adaptation, in contrast, responds to contexts through adding, removing, replacing, or even changing the interconnections of the algorithmic or structural parts of applications. For example, to adapt web contents to be rendered on a display constrained mobile device, it is necessary to transcode the image, with the ultimate aim of reducing the amount of data to be transferred. In transformation adaptation, the contents are directly transcoded. The transcoding codes are often deeply embedded within the application. In contrast, in compositional adaptation, the aim is to discover and select the most suitable transcoding component that can be composed into the flow of processed data.

Compositional adaptation is more suitable than transformational adaptation for automated context-aware adaptation decisions. Transformational adaptation statically defines variables to describe context-aware aspects of applications and, if necessary, to tune them to adapt to contextual information. This approach is easier to implement than compositional adaptation; nevertheless, in transformational adaptation, it is necessary for adaptation rules and decisions to be planned and coded before they are deployed. On the other hand, compositional adaptation can be amended for automated context-aware adaptation decisions. First, the decomposition of applications to a set of functional components enables compositional adaptation to exercise greater extensibility, and to effectively respond to changing contexts and react to the operational challenges encountered in dynamic computing. New behavioral or structural parts to address evolving challenges can be non-invasively deployed into applications without affecting the operations of existing units. Moreover, compositional adaptation can effectively separate adaptation-related concerns from the functional behaviors of context-aware applications, and make it possible for adaptation-related issues to be handled in the middleware layer. Through compositional adaptation, application functions are organized and encapsulated in various components, and context-aware adaptation is achieved by the composition of these functional components. Importantly, application developers can focus on programming various components to implement various functions of the application, and leave it to the middleware to handle adaptation-related concerns. Specifically, the middleware will be responsible for deciding how and when to compose different components to adapt to the surrounding and changing contexts.

While there have been many projects that advocate using compositional adaptation to ease the development of context-aware applications [McKinley04], they are insufficient for CAMPUS that requires a fine-grained separation of adaptation concerns from functionalities to enable automated context-aware adaptation decisions. As we argued, dealing with adaptation concerns and the basic functional concerns of applications at the same time and at the same level pushes developers into an awkward position. Adaptation concerns should be studied and tuned independent of the basic requirements of applications. In particular, CAMPUS requires its programming model to separate adaptation concerns at both the application level and component level. In addition, CAMPUS requires its programming model to provide primitives that enable the middleware to automatically make adaptation decisions. However, to the best of our knowledge, previous composition-based programming models do not satisfy the requirements of CAMPUS. Most of them only consider the separation of application functionalities and the definition of adaptive behaviors. For example, MobiGATE [Zheng06] delineates a separation of interdependent parts from the service-specific computational codes by using a separate coordination language, called MobiGATE Coordination Language (MCL), to describe the composition and reconfiguration of the MobiGATE components, referred to as *streamlet*. While such separation is desirable in developing component-based context-aware systems, in these systems, the part of a component that provides the desired functionalities and the part that is involved only during adaptation are amalgamated together, so that developers can not focus on providing functionalities indeed. There also exist a few systems that consider the separation of adaptation and functionality at the component level. For instance, Biyani and Kulkarni propose a design to separate adaptation concerns from component functionality [Biyani05]. Each component in their system is designed to consist of two parts: a *functional* part and an *adapt-active* part. The latter part is involved in actions that are only required when adaptation. Nevertheless, these systems do not provide direct supports for a middleware layer to make adaptation decisions. That is, using these systems, developers are still required to handle concerns of adaptation decisions, while considering application functionalities.

To facilitate automated context-aware adaptation decisions, CAMPUS proposes a new composition-based programming model that fully supports the separation of adaptation concerns from functionality. The ATM programming model distinguishes itself from previous composition-based programming models in two important features. First, it separates adaptation concerns from functionality both at the application level and component level. In addition, it provides primitives that enable a middleware layer to make adaptation decisions. A detailed design of the ATM programming model is presented in Chapter 4.

3.2 The Knowledge Model

The programming layer adapts a service by adding, removing, or replacing the tasklet parts of its tasks. Consequently, context-aware adaptation decisions converge to the decision of how to select appropriate tasklets according to up-to-date contextual information. Therefore, the following information is required by CAMPUS in order to make adaptation decisions:

- The requirements desired by the target service: they capture the functional and non-functional requirements imposed by the target service, such as functionality, performance, structure, security, and reliability, among others.

- The properties of the available tasklets: they specify the functional and non-functional properties of the available individual tasklets, for example, the provided functionality and QoS, and the dependencies on other components.
- The context requirements imposed by tasklets: they capture any assumption made about the environment in which the specified tasklet is expected to execute and the resources required by the tasklet to perform its functionality.
- The properties of run-time contexts: they capture the actual conditions of the execution environment and resources.

It is necessary for CAMPUS to define a new knowledge model that captures and represents the above knowledge that is required to make adaptation decisions. To the best of our knowledge, thus far there is no work that aims to provide an integrated scheme to represent all of the necessary information. Existing description schemes have been developed to capture and represent different aspects of software. For example, architecture description languages [Garlan94, Binns96, Medvidovic99] can be used to describe application properties, especially in the aspect of software architecture. Alternatively, interface and component description languages [Warmer98, Gordon00, Sora07] can be used to describe syntactic interfaces and type systems of software components, while context description languages [Strang03, Neto05] can be used to describe contextual information. These representations are designed to separately capture specific characteristics of aspects of software, with little or no provision to enable integrated sharing of knowledge to facilitate context-aware decision-making. On the other hand, it is not practical to integrate existing languages from various domains in order to provide support for automated adaptation decisions, since these languages lack the intrinsic properties to enable seamless integration across standards.

CAMPUS uses an ontological model to capture and expose the internal semantics of the knowledge required during the process of making context-aware adaptation decisions. The term ontology has its origin in philosophy. In computer science, ontology is a description of the concepts and relationships that can exist for an agent or a community of agents, and is generally written as a set of definitions of formal vocabulary [Gruber93]. Ontologies are commonly used in the fields of artificial intelligence [Lenat90], the semantic web [Berners-Lee01], and software engineering [Kitchenham99] as a form of representing knowledge about the world or some parts of it in order to clarify the structure of knowledge and enable knowledge to be shared.

An ontology consists of a vocabulary used to describe a particular view of some domain, an explicit specification of the intended meaning of the vocabulary, and the constraints on capturing additional knowledge about the domain [Horrocks02]. Contemporary ontologies share many structural similarities, regardless of the languages in which they are expressed. Most ontologies describe individuals, classes, attributes, and relations. Individuals are the basic, “ground level” component of ontology. In an ontology, individuals may include concrete objects such as people and automobiles, as well as abstract individuals such as numbers and words. Strictly speaking, an ontology does not need to include any individuals, but one of the general purposes of ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology. Classes are abstract groups, sets, or collections of objects. Objects in the ontology can be described by assigning attributes to them. Each attribute has at least a name and a value, and is used to store information that is specific to the object to which it is attached. An important use of attributes is to describe the relationships between objects in the ontology. Typically, a relation is an attribute whose value is another object in the ontology.

The CAMPUS ontologies are a set of ontologies that capture concepts and relationships in various domains of interest, in order to expose to CAMPUS the semantics of the knowledge required during the process of adaptation decision-making, as discussed previously. In general, these ontologies can be classified into *context ontologies, tasklet ontologies*, and *service ontologies*.

- *Context ontologies* model various context entities to share contextual information in a dynamic environment.
- *Tasklet ontologies* describe the properties of individual tasklets and their requirements for contextual conditions. Examples include the functionalities provided by a tasklet, the types of data that a tasklet can process and produce, and the computing resources required by a tasklet.
- *Service ontologies* describe the properties of context-aware services and their requirements for tasklets. Some examples of the requirements include the composition of services, the desired functionality of tasks, and their required input and output data types.

There are several reasons to develop ontologies as the underlying description scheme for CAMPUS:

- First, formal ontologies are an efficient solution for managing the inherent heterogeneity present in knowledge from different sources [Ciocoiu00]. Ontologies can be used to explicitly represent the meaning and semantics of contextual and computational entities, and to thereby enable entities to have a common set of concepts while interacting with one another. This is especially important for CAMPUS, since different agents could have different understandings of the current contexts. They might use different terms to describe contexts, and even if they use a set of the same terms, they might attach different semantics to these terms. Similar requirements also exist with heterogeneous tasklets that have been developed independently by different development teams.
- Second, ontologies will greatly simplify the tasks of semantic-based automated reasoning and decision-making. Ontologies include machine interpretable

definitions of concepts in the domain and the relationships among them. They are expressed in a logic-based formal language, so that consistent and meaningful distinctions can be made among the classes, properties, and relations. Importantly, a considerable number of existing reasoning mechanisms can be employed to perform automated reasoning and decision-making on the ontologies. DL and FOL reasoning have been used to reason about the CAMPUS ontologies. In particular, CAMPUS uses DL reasoning to check knowledge consistency, i.e., to ensure that the CAMPUS ontologies do not contain any contradictory knowledge. In addition, CAMPUS uses FOL reasoning to make more complex inferences, such as reasoning about whether a tasklet alternative satisfies a certain task or not.

- Finally, many graphical ontology editors, such as Protégé OWL [Knublauch04], are available to facilitate the development of ontologies. Large-scale context ontologies can also be composed without the need to start from scratch by reusing the well-defined ontologies of different domains.

3.3 The Decision Model

CAMPUS achieves context-aware adaptation by adapting the tasklet parts of tasks, and the decision layer uses a multi-stage normative decision model to choose the best tasklet alternatives for given tasks. In general, most decision theories can be classified into two groups: normative and descriptive. A normative decision theory is a theory about how decisions should be made. It is concerned with identifying the best decision to take, while assuming an ideal decision maker who is fully informed, able to compute with perfect accuracy, and fully rational. A descriptive theory, on the other hand, is a theory about how decisions are actually made. It attempts to describe what people will actually do. It is fairly obvious that a normative decision theory is more suitable for CAMPUS because it aims to choose the best tasklets for given tasks. However, all existing normative decision theories assume a linear compensatory model that suggests a single-stage choice process where the decision makers are to choose from all of the available alternatives. Fischhoff *etc*. [Fischhoff83] described the normative decision rule as:

"List all feasible courses of action. For each action, enumerate all possible consequences. For each consequence, assess the attractiveness or aversiveness of its occurrence, as well as the probability that it will be incurred should the action be taken. Compute the expected worth of each consequence by multiplying its worth by its probability of occurrence. The expected worth of an action is the sum of the expected worth of all possible consequences. Once the calculations are completed, choose the action with the greatest expected worth" (p. 183).

A single-stage decision model does not satisfy CAMPUS. The major weakness of a single-stage model is that it may involve the unnecessary processing of a large volume of information. CAMPUS is designed to operate in a pervasive computing environment where adaptation may occur frequently and many contexts need to be considered, so that the decision-making process will involve a great deal of information. At the same time, a pervasive computing environment requires short break time when performing an adaptation, in order to achieve satisfactory user experience. Therefore, unnecessary processing of information in the single-stage model may result in a performance bottleneck for CAMPUS. In addition, a single-stage decision model does not allow a decision to be omitted, i.e. it always recommends a final alternative, whereas CAMPUS allows the tasklet part of a task to be empty if no suitable tasklet alternative is found for the task. Finally, a single-stage decision model does not allow for multiple strategies to be utilized within a single decision process. In contrast, CAMPUS requires multiple decision strategies in order to decide a final tasklet. For example, a tasklet is rated not only on the basis of its functionality, but also on its contextual requirements.

CAMPUS uses a multi-stage model that is common in descriptive decision theories such as image theory [Beach90]. A multi-stage decision process incorporates multiple qualitatively separable stages, i.e., multiple heuristic-based phases followed by a choice phase. In a multi-stage model, the decision maker first uses less cognitively demanding decision strategies to eliminate unacceptable alternatives, thereby reducing the number of alternatives remaining in the choice decision. In the choice phase, the decision maker may use more cognitively demanding decision strategies to choose between the remaining alternatives. The whole decision-making procedure of CAMPUS is divided into three phases: preprocessing, screening, and choice. In the preprocess phase, several preprocessing tasks are performed to ensure that the ontologies are semantically consistent and to prepare fine-grained information for the following phases. For example, qualified tasklets are registered as alternatives for each task. In the second screening stage, tasklets alternatives that were registered in the first phase are screened out if they are not satisfied by the up-to-date contextual information. If more than one acceptable tasklet alternative survives the screening phase, the choice phase selects the best alternative from among the survivors, using the expected utility function. Expected utility theory is the dominant normative approach to decision-making. The theory proposes that for each option, there are objective payoffs (*x*), and for each level of payoff, there is a corresponding value called the utility of the payoff (*u*). Associated with each level of payoff, there is an objective probability of occurrence (*p*). The expected utility of an option can be expressed by the following mathematical formula:

$$\mathrm{EU} = \sum_{\mathrm{i}=1}^{\mathrm{n}} [p_i][u(x_i)] \qquad \text{--- Equation 3-1}$$

Where *p* = objective probability of outcome *i*; *u* = utility of outcome *i*; and *x* = payoff (gain or loss) associated with outcome *i*. The expected utility of each decision alternative is the sum of the utilities of the potential payoffs, each weighted by its objective probability of occurrence. The decision maker then chooses the option with the highest expected utility.

Chapter 4 The ATM Programming Model

CAMPUS proposes an adaptable task model (ATM) to enable context-aware adaptation. CAMPUS tasks are system elements that offer predefined functionalities and that can be independently developed under given specifications, deployed, and composed through well-defined interfaces by third parties. Such a design supports the large-scale reusing of software by enabling the assembly of “commodity-off-the-shelf” (COTS) tasks from a variety of vendors. This chapter focuses on the design of the ATM programming model, including its conceptual model and semantic model.

4.1 Conceptual Model

The ATM programming model relies on two main concepts: s*ervice* and *task*. A service is an abstract of a business or a technical process. For example, in CyberGuide [Abowd97], a mobile context-aware tour guide, four context-aware services are provided including *Cartographer*, which provides maps of the physical environments that the tourist is visiting; *Librarian*, which provides information relating to objects and people of interest in the physical world; *Navigator*, which delivers accurate information on tourist locations and orientations; and *Messenger*, which conveys messages to related tourists. Since such services provided by a context-aware application are loosely coupled, it is necessary to introduce a separate concept of service to organize context-aware applications.

Internally, each service is comprised of a series of tasks. Tasks are execution units that perform certain actions to deliver a result to other tasks or the end user. For example, a streaming media service may be composed of tasks such as the *codec*, *transceiver*, *jitter buffer*, and *renderer*. A task in CAMPUS transforms or filters data of specified types, and communicates with others solely through the exchange of data instead of by direct method calls. Thus a task has two basic concerns: how to transform or filter data, and how to exchange data. CAMPUS separates these two concerns by dividing a task into two parts: a *tasklet* and a *task base*. The former part concentrates on the computational concerns of the task, i.e. how to process data; and the latter part handles issues involved in coordination and adaptation, such as how to communicate with other modules of the service and how to transfer the states of tasks during adaptation.

The tasklet part of a task can specify the contextual requirements that should be satisfied for it to operate normally and optimally. For example, a tasklet that processes image data may require the terminal to be equipped to display graphics. Such contextual requirements are described using the CAMPUS ontologies that are described in Chapter 5. Importantly, this feature of the tasklet part enables a task to be context-aware. When the surrounding contexts change, the behavior of a task can be adapted by using a different version of its tasklet part. Tasks are further classified into *essential tasks* and *expansion tasks*, according to whether or not the tasklet part of a task can be empty. An essential task always requires a tasklet. If no tasklet is found for an essential task, an exception will be reported. An expansion task, on the other hand, is allowed to have an empty tasklet part. Such classifications reflect the fact that in the composition of a service, some of its tasks are core, and some are non-core. For instance, the *codec*, *transceiver*, and *renderer* are essential to a service of streaming media, while the *jitter buffer* is not always required, especially when the network condition is satisfactory.

The *task base* part of a task serves three purposes. First, the base part is responsible for communicating with other tasks to exchange operating data. A task base receives and sends the operating data through its input and output *ports*, respectively. A task base can own an arbitrary number of input or output ports. In addition to the *port type*, i.e. *input* and *output*, each port has a *data type*, and only data of the specified type can pass the port. Secondly, the task base part acts as a proxy to its corresponding tasklet. Any data that need to be processed by the tasklet will be sent to the base, and the base will cache the incoming data and forward them when the tasklet is ready. Similarly, after processing the input data, the tasklet is required to return the output data immediately to the base, where interested parties can fetch these output data. Finally, the task base part, on behalf of the task, defines the requirements for the tasklet part. In order to perform the provided functionalities, a tasklet has to be inserted into a compatible base, where it receives the desired input data and publishes the output data after processing. To be compatible with a task base, a tasklet must implement the desired functionality of the task, and must be able to process the data of all input data types and produce the data of all output data types of the task base.

Tasks are assembled by setting up channel connections between their compatible ports as illustrated in Figure 4-1. A *channel* is used to connect an output port of a task to an input port of another task. It is necessary for the connected tasks to have ports of a compatible type that will enable data to be seamlessly transferred. A channel represents a reliable, directed flow of information in time. *Reliable* means

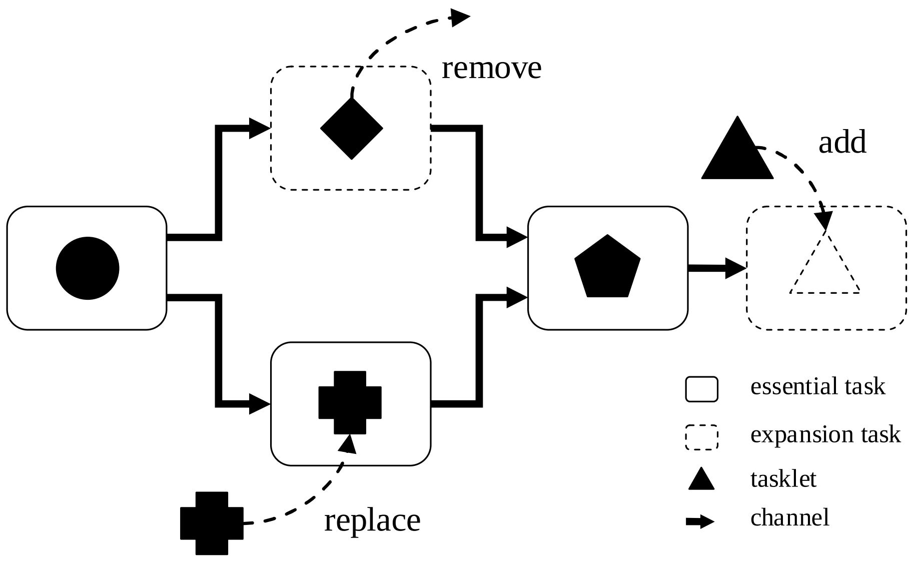


Figure 4-1 The ATM Programming Model.

that all of the data that have been placed into a channel are guaranteed to flow through without loss, error, or duplication, while preserving their order. *Directed* means that a channel always has two identifiable ends: a *source* port and a *sink* port.

The ATM programming model offers three salient features. First, it greatly simplifies the work of maintaining the consistency of the data and migrating states when adaptation occurs. In CAMPUS, tasklets are required to retrieve data from their bases when necessary, and to return the data immediately after it has been processed, i.e., the operating data are stored in bases instead of tasklets. Therefore, when tasklets are unplugged from their bases, the effort to enforce tasklet state consistency is greatly reduced. Moreover, the issue of how to cooperate with other parts of the target application in order to receive and publish data is handled by the task bases, while the tasklet is only responsible for providing the desired functionalities and is treated as a black box that uses predefined interfaces to communicate with its host task. Such a separation greatly facilitates the development and reusing of tasklets. Finally, this separation simplifies verification of the adaptation. To verify that the adaptation of a task is correct, the task needs to continue to correctly perform its functionality after adaptation, and specification during adaptation needs to be satisfied [Biyani05]. The separation of the part related to adaptation from task functionality simplifies the task of specifying and verifying adaptation.

4.2 Semantic Model

This section describes the mathematical model of the ATM programming model using the specification language Z. The Z schemas, which can be regarded as definitions of a generalized type, are used to represent the basic constructs. These schemas provide semantics that permit formal verification of properties of the model. Additional details on Z can be found in [Spivey89].

Service

$serviceId : ENTITY$

$channels : \mathbb{F}\ Channel$

$tasks : \mathbb{F}_1\ Task$

$\forall\ c : channels \bullet \exists\ t1, t2 : tasks \bullet$

$\quad c.source \in t1.base.outputs \wedge c.sink \in t2.base.inputs$

$\forall\ t : tasks \bullet$

$\qquad (\forall input : t.base.inputs \mid input.state = used \bullet$

$\qquad\qquad \exists\ c : channels \bullet c.sink = input)$

$\quad \wedge (\forall output : t.base.outputs \mid output.state = used \bullet$

$\qquad\qquad \exists\ c : channels \bullet c.source = output)$

$\#channel \geq \#task\text{-}1$

Figure 4-2 The Service Schema.

It is assumed that sets *[ENTITY, DTYPE]* exist. The *ENTITY* identifiers represent global names. Name clashes between distinct entities are disallowed. The set *DTYPE* includes different types of data and are introduced as a given set in the model.

Service. A service is identified with a unique id. It can be modeled as a non-empty finite set of tasks connected by channels. As shown in Figure 4-2, the following properties are required to ensure a consistent context-aware service:

- For all channels in the service, the source port of a channel must connect to an output port of a task, and the sink port of this channel must connect to an input port of another distinct task.
- For all tasks, all ports must connect to channels.
- If there is more than one task in the service, then they must be connected by channels.

Task

$taskId : ENTITY$

$function : ENTITY$

$base : TaskBase$

$tasklet : Tasklet$

$base.taskId = taskId$

$tasklet.function = function$

TaskBase

$baseId : ENTITY$

$taskId : ENTITY$

$inputs, outputs : \mathbb{F}\ Port$

$inputs \cap outputs = \emptyset$

$inputs \cup outputs \neq \emptyset$

$\forall\ i1, i2 : inputs \mid i1 \neq i2 \bullet i1.dtype \neq i2.dtype$

$\forall\ o1, o2 : outputs \mid o1 \neq o2 \bullet o1.dtype \neq o2.dtype$

Tasklet

$taskletId : ENTITY$

$function : ENTITY$

$inputTypes, outputTypes : \mathbb{F}\ DTYPE$

$inputTypes \cup outputTypes \neq \emptyset$

Figure 4-3 The Task-Related Schemas.

Task. A task provides desired functionality and consists of a task base part of a tasklet part. The base part of a task owns a number of input and output ports that receive and send data. Each port has a data type, and only data of the specified type can pass the port. The tasklet part of a task provides an implementation of the

Channel

$channelId : ENTITY$

$dateType : DTYPE$

$source, sink : Port$

$source \neq sink$

$source.type = output$

$sink.type = input$

$source.dtype = sink.dtype$

$source.baseId \neq sink.baseId$

Figure 4-4 The Channel Schema.

desired functionality of the task, i.e., transforming or filtering data of specific types. As shown in Figure 4-3, some enforced constraints on tasks and their parts are:

- The tasklet part must provide the desired functionality of the task.
- The task base part must own at least one input port or one output port, and all ports are distinct. Two ports are distinct if and only if both their port types and data types are distinct.

Channel. A channel connects two distinct tasks to pass data of a specific type. Figure 4-4 shows the formal definition of a channel. Importantly, the following constraints are enforced on channels:

- The source port and the sink port are distinct, and their data types are identical.
- The type of the source port is *output* and the type of the sink port is *input*.
- The source port and the sink port belong to distinct tasks.

Chapter 5 The CAMPUS Ontologies

CAMPUS makes use of ontologies to capture the most important concepts and relationships in the process of developing and deploying context-aware applications. These ontologies represent knowledge that enables CAMPUS to reason the underlying semantics of involved entities and help it to decide how to compose target services and select suitable tasklets. The CAMPUS ontologies are split into two dimensions. This makes them more extensible and scalable, and also helps to improve the performance of the system. The first dimension is that of domain. Here, the ontologies are divided into *context ontologies*, *tasklet ontologies*, and *service ontologies*:

- *Context ontologies* model various context entities to share contextual information in a dynamic environment.
- *Tasklet ontologies* describe the properties of individual tasklets and their requirements for contextual conditions. Examples include the functionalities provided by a tasklet, the types of data that a tasklet can process and produce, and the computing resources required by a tasklet.
- *Service ontologies* describe the properties of context-aware services and their requirements for tasklets. Some examples of the requirements include the composition of services, the desired functionality of tasks, and their required input and output data types.

Additionally, from the dimension of range, the CAMPUS ontologies are separated into foundation ontologies and domain-specific ontologies. The foundation ontologies model common objects that are generally applicable across a wide range of domain-specific ontologies. A domain-specific ontology models a particular domain and represents the particular meanings of terms as they apply to that domain.

The separation encourages the reusing of general concepts and provides a flexible interface for defining domain-specific knowledge. On the one hand, all applications belonging to a specific domain can share the ontologies in this domain, and all domain-specific ontologies can reuse the foundation ontologies; on the other hand, new domain-specific ontologies can be flexibly plugged and unplugged to support new domain knowledge.

5.1 Context Foundation Ontology

CAMPUS models a context as a set of descriptions of one particular context entity. It does this by providing a set of properties to describe various aspects of this entity. For example, a *RAM* context entity may use the properties of *hasCapacity* and *hasFreeSpace* to describe its total capacity and current free space. Formally, a context is a set of triples (*entity, property, value*), with each component defined as follows:

- *entity* $\in E$: the set of individuals of context entities, such as a computing terminal, a person, or a noise level.
- *property* $\in P$: the set of properties used to describe various aspects of a context entity. For example, *hasCapacity*, to describe the total capacity of a *RAM* instance.
- *value* $\in V$: the set of all values of the property. For example, *10 MB* for the property *hasCapacity* of a *RAM* instance.

CAMPUS defines a *ContextEntity* class as the base class for all contextual entities, and uses a well-accepted context category [Schilit94, Dey00], i.e. *ComputationalEntity*, *PhysicalEntity* and *UserEntity*, to extend the *ContextEntity* base class, as shown in Figure 5-1. *ComputationalEntity* refers to an application's execution conditions, including its software, network, and hardware conditions. *PhysicalEntity* refers to the circumstances by which applications are surrounded,

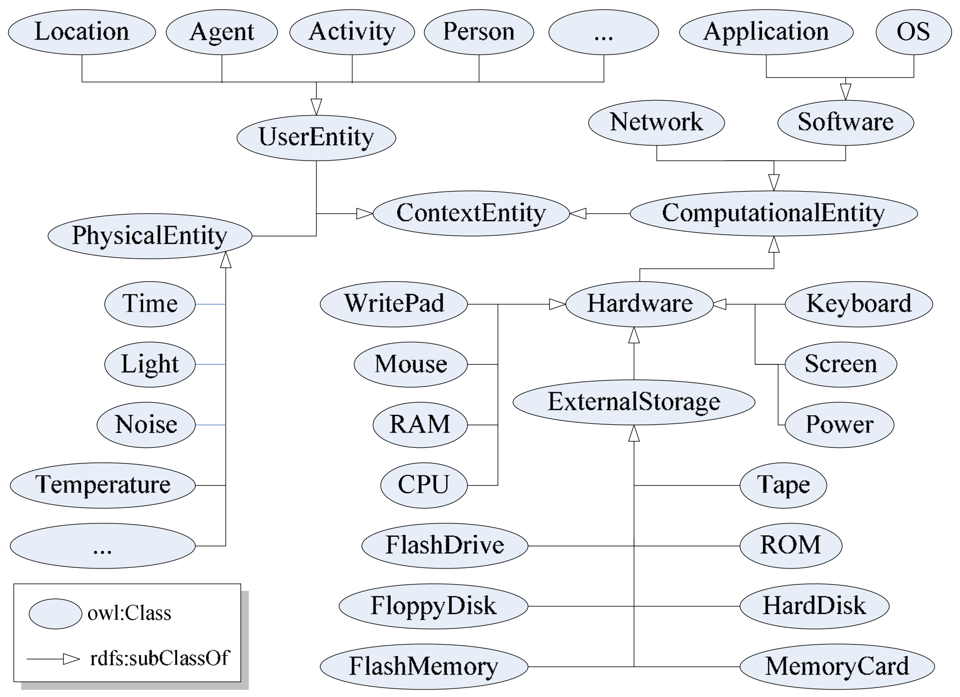


Figure 5-1 The Hierarchy of Context Entities.

such as noise level, temperature and lighting level. Finally, *UserEntity* pertains to characteristics of users, such as user presence, status, activities, and abilities. The context foundation ontology has defined most of the common context entities and their properties for domain-specific ontologies to extend. In particular, the value of a context property can be a physical quantity, a context entity, or a data value of a built-in OWL datatype, as listed in Table 5-1.

Table 5-1 Legal Types of the Value of Context Properties.

Type	**Description**
context:PhysicalQuantity	Represents physical quantities.
context:ContextEntity	Represents context entities.
rdfs:Literal	Represents literal values.
rdf:string	Represents character strings.
rdf:boolean	Has the value space required to support the mathematical

	concept of binary-valued logic: {true, false}.
rdf:base64Binary	Represents Base64-encoded arbitrary binary data.
rdf:hexBinary	Represents arbitrary hex-encoded binary data.
rdf:decimal	Represents arbitrary precision decimal numbers.
rdf:float	Corresponds to the IEEE single-precision 32-bit floating point type.
rdf:double	Correspond to the IEEE double-precision 64-bit floating point type.
rdf:anyURI	Represents a Uniform Resource Identifier Reference (URI).
rdf:dateTime	Represents a specific instant of time.
rdf:time	Represents an instant of time that recurs every day.
rdf:date	Represents a calendar date.
rdf:gYearMonth	Represents a specific gregorian month in a specific gregorian year.
rdf:gYear	Represents a gregorian calendar year.
rdf:gMonthDay	Represents a gregorian date that recurs, specifically a day of the year.
rdf:gDay	Represents a gregorian day that recurs, specifically a day of the month.
rdf:gMonth	Represents a gregorian month that recurs every year.

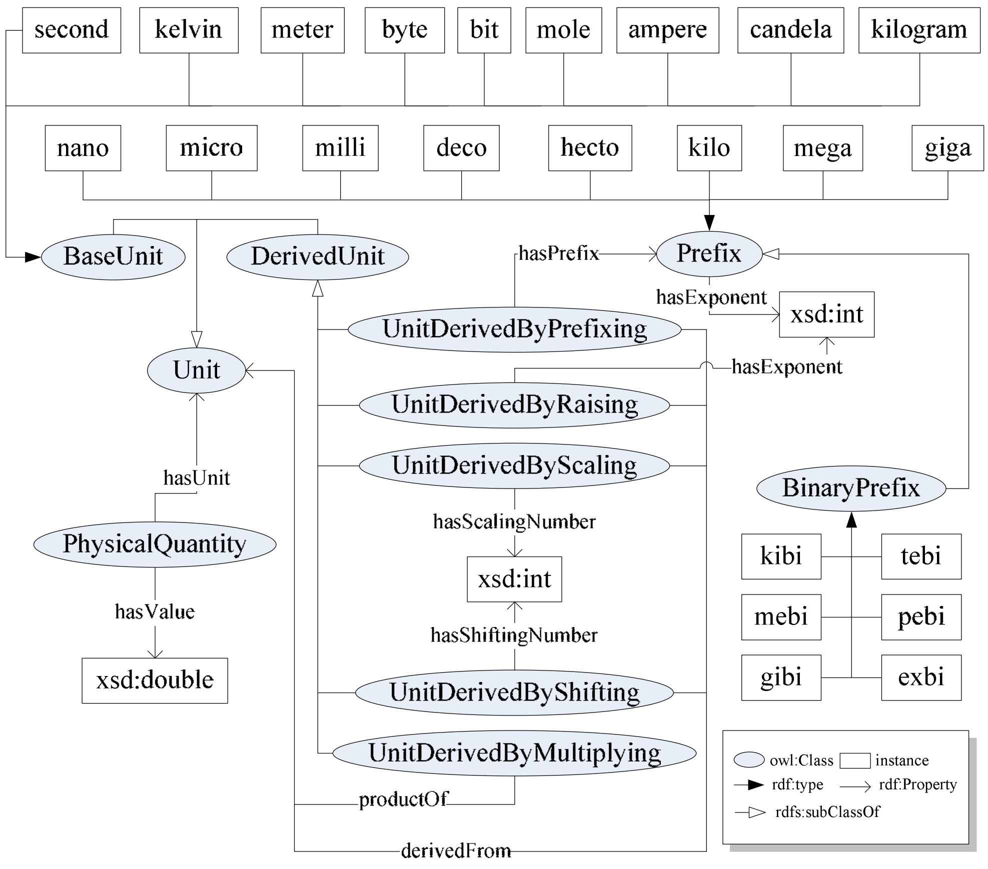


Figure 5-2 *PhysicalQuantity* and *Units*.

Importantly, the context foundation ontology captures the concept of physical quantity. *Physical quantity* is a very important concept in most practical scenarios of context reasoning and context-aware adaptation decision-making. For example, the current availability of a CPU may be inferred from its clock rate and current loading as illustrated using a first-order logic predicate: *(cpu, hasClockRate, <=400MHz)* ∧ *(cpu, hasLoading, >=80%)* → *(cpu, hasAvailability, LOW)*. To achieve this, the reasoner needs to understand exactly what *400 MHz* and *80%* represent, so that these measurements can be compared and inferred. As shown in Figure 5-2, the context foundation ontology defines a class *PhysicalQuantity* that has two properties: *hasValue* and *hasUnit*. The context foundation ontology also defines a comprehensive unit hierarchy that is basically derived according to the International System of Units (SI). In this unit hierarchy, the *Unit* class is the base class, and has two concrete sub-classes: *BaseUnit* and *DerivedUnit*. We define nine base units, namely seven SI base units and two information storage units, *bit* and *byte*. Derived units are further divided into *UnitDerivedByMultiplying*, *UnitDerivedByShifting*, *UnitDerivedByScaling*, *UnitDerivedByRaising*, and *UnitDerivedByPrefixing*. *UnitDerivedByMultiplying* refers to units that are derived from multiple units by multiplication. For example, *coulomb* is an individual of the class *UnitDerivedByMultiplying*, which is the product of second and ampere. *UnitDerivedByShifting* abstracts units that are derived from base units or other derived units by shifting. *DegreeC* is such an example, derived as it is from *Kelvin* by subtracting 273. *UnitDerivedByScaling* is used to capture units that are derived by scaling other units. For instance, *hour* is derived from *second* by scaling 3600. Units that are classified into *UnitDerivedByRaising* are derived by raising other units to an n-th power, such as *area* being a squared *meter*. Finally, individuals of *UnitDerivedByPrefixing* are decimal multiples and submultiples of other units. For example, *mebi-bit* is derived from *bit* by adding a prefix *mebi*. Moreover, we distinguish two kinds of prefixes: *SIPrefix* and *BinaryPrefix*. The former refers to prefixes adopted by SI, which represent powers of 10; and the latter refers to prefixes adopted by the International Electrotechnical Commission (IEC), which represent powers of 2.

5.2 Tasklet Foundation Ontology

The tasklet foundation ontology defines the underlying concepts of the CAMPUS tasklets. Tasklet vendors are required to extend the foundation ontology to provide concrete information about their own tasklets, in order to make available meta-data for the middleware to make appropriate adaptation decisions.

As shown in Figure 5-3, the *Tasklet* class uses the *implementedBy* property to indicate the path of the tasklet implementation that is used by CAMPUS to locate the tasklet codes. The functionalities provided by the tasklet are indicated by the *provides* property, and the types of data available for the tasklet to process and

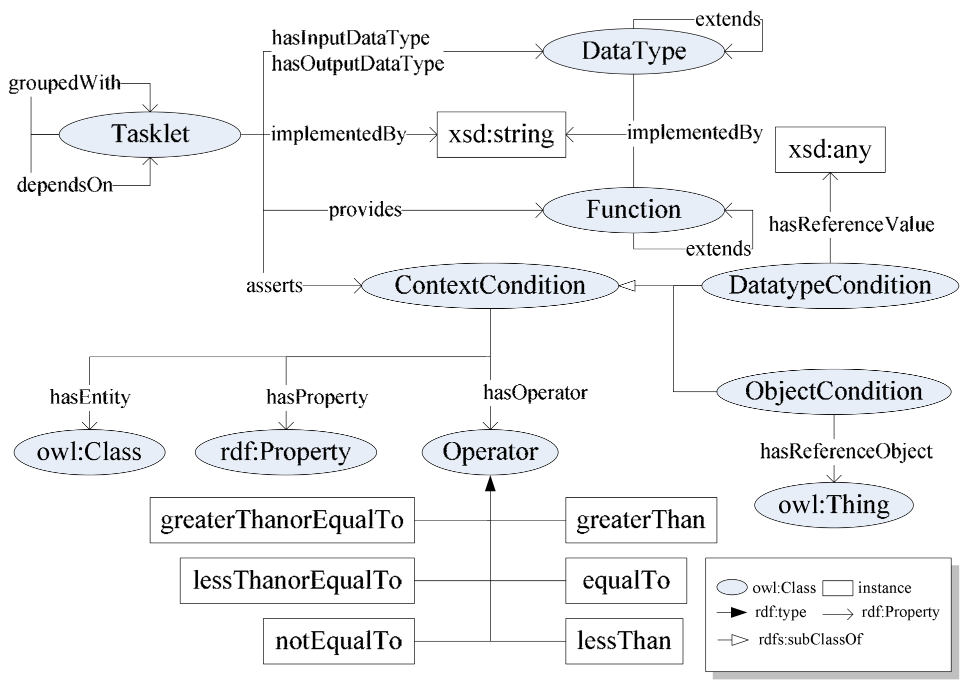


Figure 5-3 The Tasklet Foundation Ontology.

generate are specified by the *hasInputDataType* and *hasOutputDataType* properties. Similar to *Tasklet*, instances of the *Function* and *DataType* classes can use the *implementedBy* property to specify their code paths. In addition, the *extends* property can be used to indicate the inheritance relationship between two functions or two datatypes.

The tasklet foundation ontology introduces a *ContextCondition* class to model the contextual requirements imposed by a tasklet. A context condition statement, for example "*the available RAM capacity needs to be larger than 100 MB*," can be logically separated into four parts: a part on the context entity, which is "*RAM*," a part on the property of the entity, which is "*capacity*," a part on the comparison operator, which is "*larger than*," and a part on the reference value, which is "*100 MB*." In particular, the part on the reference value can be either a literal value or a comparable individual. These four parts are represented in the tasklet foundation ontology by four properties, respectively: *hasEntity*, *hasProperty*, *hasOperator* and *hasReferenceValue* (or *hasReferenceObject* when the reference value is an individual

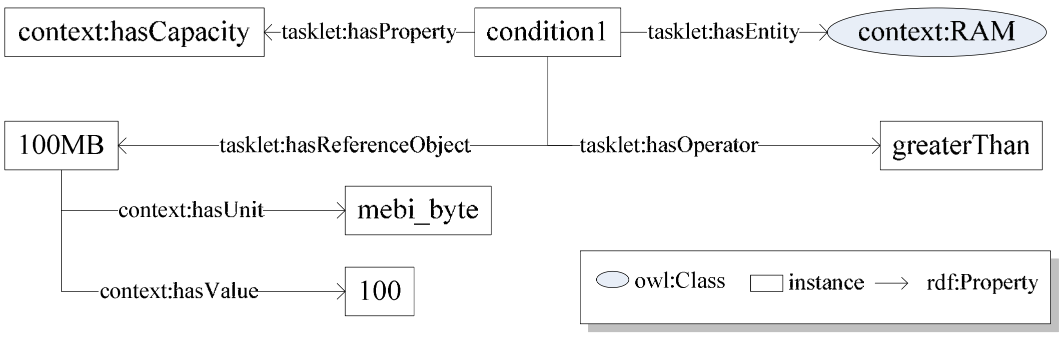


Figure 5-4 A Sample Context Condition.

instead of a literal value). Figure 5-4 illustrates an example condition statement using the CAMPUS ontology.

Context-aware services undergoing dynamic compositional adaptation often require synchronization to ensure that related tasklets are added and removed consistently. For example, in the case of a streaming service with the deployment options of various codecs based on various network situations, the matching encoder and decoder should always be replaced synchronously, otherwise the content may be decoded incorrectly. Thus, the tasklet foundation ontology uses an object property *groupedWith* to capture the synchronization requirements of the tasklets. When two tasklets are related by the *groupedWith* property, they should be added or removed synchronously. Another relationship between tasklets is concerned with dependency. That a tasklet *A* depends on another tasklet *B* means that when the tasklet *A* is about to be deployed, the tasklet *B* should be deployed first. The tasklet foundation ontology uses an object property *dependsOn* to represent such a relationship of dependency.

5.3 Service Foundation Ontology

The defined service foundation ontology formally describes the common concepts of services on CAMPUS. Service developers are required to provide their own domain-specific service ontologies by extending this foundation ontology, in order to

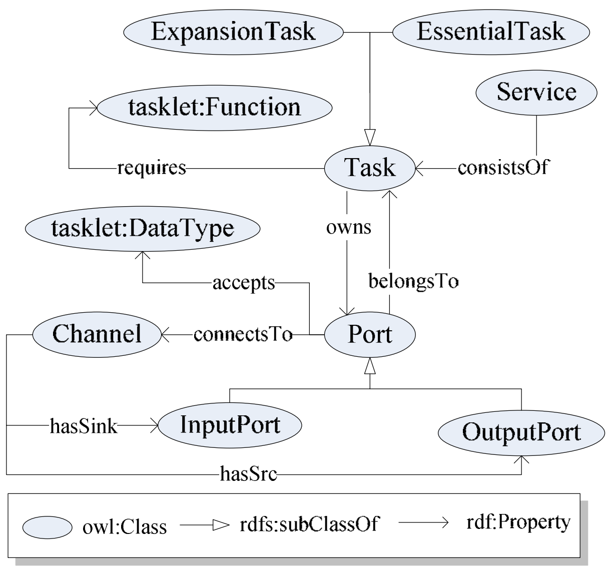


Figure 5-5 The Service Foundation Ontology.

describe the concrete properties related to the service. As shown in Figure 5-5, a service consists of a series of tasks, including essential tasks and expansion tasks. A task specifies its functional requirement for its tasklet by an object property *requires*, whose value is a functionality that has been detailed in the section on tasklet foundation ontology. A task owns a set of ports, including input ports and output ports. Each port accepts only one data type, and is connected to a channel. A channel receives data from an output port of a task, and sends data to a compatible input port of another task.

Chapter 6 Automated Context-aware Adaptation Decisions

CAMPUS achieves context-aware adaptation by adapting the tasklet parts of tasks. The primary objective of adaptation decisions is to choose an appropriate set of tasklet alternatives for given tasks. This chapter details the decision-making mechanism of CAMPUS, which offers two salient features:

- Decisions are made based on the semantics of the involved entities. CAMPUS makes use of description logics and first-order logics to perform semantic reasoning on the ontologies that describe the CAMPUS entities; and tasklets alternatives are compared based on their semantics in order to select the best ones. For example, assume that a particular task *T* requires a function *FuncA*, and a tasklet *Tl* provides another function *FuncB*. Commonly, *Tl* is not a qualified tasklet for the task *T* because it does not provide the desired function of *T*. However, if the function *FuncB* is extended from *FuncA*, or if it has been indicated that it is semantically identical with *FuncA*, then *Tl* will be regarded as being able to satisfy the functional requirement of *T*. Of course, this example is somewhat contrived. A more likely use would be in a case where different vendors develop tasklets independently, and use different terms to refer to the same concepts. With this semantic approach, it is possible to reuse these heterogeneous tasklets.

- Decisions are made through a multi-staged process as illustrated in Figure 6-1. CAMPUS makes adaptation decisions based on the semantic ontologies. Before the decision-making process is launched, these ontologies are preprocessed. The preprocessing stage ensures that the ontologies are consistent in semantics. It also serves to prepare fine-grained information for the decision-making process, which indicates a heuristic-based screening phase followed by a choice phase of

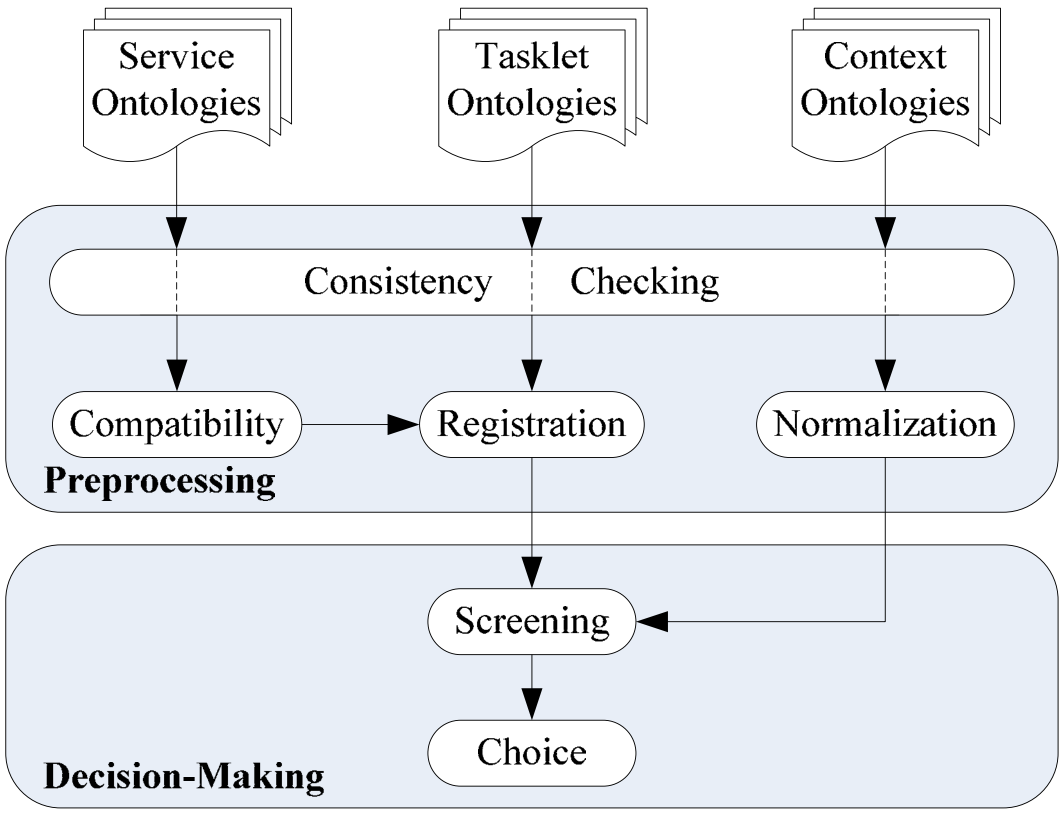


Figure 6-1 The Complete Process of Making Adaptation Decisions.

comprehensive tasklet comparisons. The screening phase addresses the pre-choice process of eliminating unacceptable tasklets, that is, the mechanism that governs which tasklet alternative is rejected and which tasklet enters into the final choice set. Eventually, in the final phase, the tasklet that maximizes the expected utility is chosen.

6.1 Semantic Reasoning

This section presents the technologies used by CAMPUS to perform semantic reasoning on the ontologies. CAMPUS makes use of DL and FOL reasoning to facilitate context-aware adaptation decisions. The CAMPUS ontologies are described using the OWL DL language. The equivalence of description logic and OWL DL allows various DL reasoning tasks to be performed on the CAMPUS ontologies [Wang04]. For example, assume that there are an inverse-functional property *P*, and two individuals *x1*and *x2* such that both pairs *(x1, y)* and *(x2, y)* are instances of *P*. According to the semantics of the inverse-functional property axiom, we can infer that *x1* and *x2* actually refer to the same thing, i.e., *x1 owl:sameAs x2*. Table 6-1 lists the most common OWL DL axioms that can be used for DL reasoning.

Table 6-1 OWL DL Axioms for DL Reasoning.

OWL DL Axiom	**Semantics**
rdfs:subClassOf	(*A* rdfs:subClassOf *B*) ^ (*B* rdfs:subClassOf *C*) -> (*A* rdfs:subClassOf *C*)
owl:equivalentClass	(*A* owl:equivalentClass *B*) ^ (*C* rdf:type *A*) -> (*C* rdf:type *B*)
owl:disjointWith	(*A* owl:disjointWith *B*) ^ (*C* rdf:type *A*) ^ (*D* rdf:type *B*) -> (*C* owl:differentFrom *D*)
rdfs:subPropertyOf	(*A* rdfs:subPropertyOf *B*) ^ (*B* rdfs: subPropertyOf *C*) -> (*A* rdfs: subPropertyOf *C*)
owl:InverseFunctionalProperty	(*P* rdf:type owl:InverseFunctionalProperty) ^ (*A P B*) ^ (*A' P B*) -> (*A* owl:sameAs *A'*)
owl:SymmetricProperty	(*P* rdf:type owl:SymmetricProperty) ^ (*A P B*) -> (*B P A*)
owl:TransitiveProperty	(*P* rdf:type owl:TransitiveProperty) ^ (*A P B*) ^ (*B P C*) -> (*A P C*)
owl:equivalentProperty	(*P* equivalentProperty *P'*) ^ (*A P B*) -> (*A P' B*)
owl:inverseOf	(*P* owl:inverseOf *P'*) ^ (*A P B*) -> (*B P' A*)
owl:sameAs	(*A* owl:sameAs *B*) ^ (*A P C*) -> (*B P C*) (*A* owl:sameAs *B*) ^ (*C P A*) -> (*C P B*)

Furthermore, CAMPUS defines a set of application-independent meta-rules to further semantically reason the ontologies. The CAMPUS meta-rules define the general guidelines to instruct the middleware on how to process ontologies and make adaptation decisions. Compared with adaptation rules in existing context-aware systems like [Chan03, Yau02, Zheng06], which describe concrete adaptation decisions, such general meta-rules do not change frequently and their scale is far smaller. Consequently, it is easier to manage and maintain the CAMPUS meta-rules. Table 6-2 lists the defined CAMPUS meta-rules and their details will be discussed in the following sections.

Table 6-2 CAMPUS Meta-Rules.

Meta-Rule	**Description**
Compatibility	Checks whether a CAMPUS services is composed correctly. It reflects the semantic restrictions and constraints on a service imposed by the ATM programming model.
Registration	Sets up a relation between each task and its qualified alternatives in the knowledge base.
Normalization	Converts the original unit of a physical quantity to a base unit with the same quantity, based on the semantics defined in the context foundation ontology.
Screening	Filters away tasklet alternatives that are not satisfied by the up-to-date contexts.
Choice	Selects the best tasklet from the final tasklet set that passes the screening phase. It computes the fitness utility of each tasklet and selects the tasklet with the maximum fitness utility.

The CAMPUS meta-rules can be defined using the Semantic Web Rule Language (SWRL). SWRL combines sublanguages of the OWL language (OWL DL and OWL Lite) with those of the RuleML (Unary/Binary Datalog). It extends the set of OWL axioms to include Horn-like rules, and thus enables Horn-like rules to be combined with an OWL knowledge base [Horrocks04]. SWRL provides a formally sound way of inferring information in OWL ontologies. Its inherent integration with OWL makes it easy to use when reasoning OWL ontologies. On the other hand, SWRL supports monotonic inferences only. Consequently, negation-as-failure and disjunction are not supported in SWRL. It is also unable to directly model changing information and to update property values. If a property has an existing value and a SWRL rule asserts a new different value, then the property will have two values. If the property is functional, an inconsistency exception will be generated.

The monotonic feature of SWRL is not sufficient for CAMPUS to perform complex reasoning and make adaptation decisions. Therefore, CAMPUS also supports Jess rules [Friedman-Hill03]. The Jess rule language is a superset of CLIPS and supports sufficient expressiveness for CAMPUS to perform more complicated reasoning tasks. The defined Jess rules are processed by the Jess rule engine, which uses the Rete algorithm [Forgy82], a very efficient mechanism for solving the many-to-many matching problem. The entire Jess system consists of a rule base, a fact base, and an execution engine. The execution engine matches facts in the fact base with rules in the rule base. These rules can assert new facts and put them in the fact base. In order to use Jess to reason about OWL ontologies, the relevant knowledge about OWL individuals are represented as Jess facts. After performing an inference using the defined meta-rules, the results of that inference will be reflected in the OWL knowledge base.

6.2 Preprocessing

Before making adaptation decisions, CAMPUS preprocesses the received ontologies in order to facilitate the decision-making process. It first checks the consistency of the received ontologies using the OWL semantics standard, as detailed in Section 6.1. The purpose of this is to ensure that the ontologies do not contain any contradictory facts. If any inconsistency is found, the affected ontologies will not be deployed in the CAMPUS system and an error will be reported. After checking for consistency, the consistent ontologies will be further preprocessed according to their types, i.e., service ontologies, tasklet ontologies, and context ontologies.

If a service ontology passes the check for consistency, its compatibility will be checked. The task of checking for compatibility is carried out to ensure that all services are composed correctly. When an incompatible service is found, it will be marked and removed from the knowledge base. This task checks for service compatibility via the *compatibility* meta-rule, which reflects the semantic restrictions and constraints on a service imposed by the ATM programming model. These restrictions and constraints were detailed in Chapter 4 and are restated below:

- Each task in a service must own at least a port. Furthermore, all of the ports of a task are distinct. Two ports of a task are distinct if both their port type, i.e. *input* and *output*, and data type are distinct.

- The source port of each channel in the service must connect to an output port of a task. In addition, the sink port of this channel must connect to an input port of another distinct task. Furthermore, the data type of the sink port must be equal to, or be a supertype of, the data type of the source port.

- If there is more than one task in the service, then these tasks must be connected by channels.

If a tasklet ontology passes the check for consistency, it will be further used to register tasklet alternatives for tasks defined in the compatible service ontologies. A tasklet can be registered as an alternative for a task if and only if it satisfies the following criteria:

- The tasklet provides the required functionality of that task.
- The tasklet is able to process data of certain types that are specified by the input ports of that task.
- The tasklet is able to generate data of certain types that are specified by the output ports of that task.

The task of registering alternatives is done by the *registering* meta-rule that sets up a relation between each task and its qualified alternatives in the knowledge base.

Context ontologies that pass the check for consistency will be further preprocessed. For example, all of the physical quantities will be normalized using the *normalization* meta-rule, which converts the original unit of a physical quantity to a base unit with the same quantity, based on the semantics defined in the context foundation ontology. Such normalization is critical to the comparison of physical quantities, which are required when making context-aware adaptation decisions. Importantly, this task provides a chance for developers and end users to specify their own strategies to deduce high level and implicit contexts from low level and explicit contexts. Such context deduction strategies are application-dependent. As they are outside the scope of CAMPUS, they will not be detailed here.

6.3 Decision-Making

The decision-making process of CAMPUS involves selecting the best tasklet for a given task under the effect of changing contexts that may compromise the quality of service. The process consists of two phases: a screening phase and a choice phase.

The screening phase uses the *screening* meta-rule to filter away tasklet alternatives that are not satisfied by the up-to-date contexts, while the choice phase uses the *choice* meta-rule to select the best tasklet from the final tasklet set that passes the screening phase. The decision-making process is triggered upon request by CAMPUS under three kinds of situations. First, when activating a task, CAMPUS will use the process to decide an initial tasklet for that task. Second, during run-time, if a task reports an error to the effect that its tasklet part is operating under abnormal conditions, CAMPUS will use the process to select another tasklet for this task. Finally, when CAMPUS receives an updated context, all interested tasks that subscribe to this context will be notified. When such a task subscriber is notified, it will use the process to infer whether adaptation is necessary and which tasklet is the best alternative.

In the screening phase, tasklet alternatives whose context requirements are not satisfied by the up-to-date contexts will be filtered away. Recall that a tasklet can assert a number of context conditions, each of which is composed of four parts: a part on the context entity, a part on the property of the entity, a part on the comparison operator, and a part on the reference value. A tasklet alternative is satisfied if and only if each of its asserted context conditions is satisfied. A context condition is satisfied if the up-to-date context instance of its specified context class semantically matches the condition statement. For example, assume that a context condition states, "*The available RAM capacity needs to be larger than 100 MB*." That is, the entity part of the condition is *RAM*, the property part is *available capacity*, the operator part is *larger than*, and the reference value part is *100 MB*. The condition is satisfied if and only if the value of the property *available capacity* of the up-to-date *RAM* instance is semantically larger than 100 MB. Here *semantically* refers to the fact that the property values are compared based not only on the numerical value but also on the physical unit. That is, *200 MB* and *0.2 GB* both satisfy the condition, "*larger than 100 MB*."

If multiple alternatives for the given task pass the screening phase, they will be compared in the choice phase, in order to determine the most suitable tasklet. This is done by the *choice* meta-rule. Importantly, developers can specify their own choice strategies to overwrite the default *choice* meta-rule that selects a tasklet that best fits the current context. The default *choice* meta-rule computes the fitness utility of each tasklet and selects the tasklet with the maximum fitness utility. The overall fitness utility of a tasklet is computed as the arithmetic mean of the fitness utilities of its context conditions, as shown in Equation 6-1 where the fitness function over a context condition *i* is expressed as U^i.

$$U_{fitness}(tasklet_x) \equiv \frac{\sum_{i=1}^{k} U^i_{fitness}(tasklet_x)}{k} \qquad \text{--- Equation 6-1}$$

The fitness utility over a specific context condition can be computed in terms of a fitness function expressed in a mixed form of mathematical notations and pseudo code as below:

```
Utility_fitness =
    if (operator  is  equal to  or  not equal to) then
        1
    else                                                   --- Equation 6-2
        $e^{-\left(\frac{Cu-Cr}{Cr}\right)^2}$
```

Where *operator* refers to the operator part of the condition, C_r refers to the reference value part, and C_u refers to the up-to-date value of the corresponding context. When the operator is *equal to* or *not equal to*, the function simply sets the utility as *1*. Such functions measure the fitness of up-to-date contextual information for specific context conditions of tasklets. For example, consider two context conditions "*the bandwidth is to be larger than or equal to 256 Kbps*" and "*the bandwidth is to be larger than or equal to 128 Kbps,*" and the fact that the up-to-date bandwidth is *512 Kbps*. In this example, the up-to-date bandwidth is considered as more fit for the first condition, i.e., the fitness utility of the first condition is higher than that of the second one.

Chapter 7 Implementation

To demonstrate the working principle of CAMPUS in a real-world setting and provide a basis for assessing practical issues, we have implemented the system using Java SE 1.6. The reasons for this choice are manifold: Java is a portable language, it has embedded support for logical mobility and reflection, and more and more mobile devices being released are enabled with J2ME technology. The many libraries available, as well as run-time support for Java have further motivated our choice. The CAMPUS system uses Pellet 1.5.1 [Sirin07] to execute DL reasoning tasks on the CAMPUS ontologies detailed in Chapter 5. Pellet is an open-source Java based OWL DL reasoner that is based on the tableaux algorithms developed for expressive description logics. In addition, the Jess 7.1p2 [Friedman-Hill03] is used to perform FOL reasoning on the ontologies. Jess is a Java rule engine that uses an enhanced version of the Rete algorithm to process the CAMPUS meta-rules that are presented in Chapter 6.

This chapter presents the implementation of the CAMPUS system that focuses on providing flexible adaptation support for context-aware applications. The low-level details of the implementation codes are not discussed here. Rather, this chapter introduces the overall architecture of the prototype and highlights the API that facilitates the development of adaptable tasks and their composition.

7.1 The Implementation Architecture of CAMPUS

Figure 7-1 illustrates the implementation architecture of the CAMPUS middleware. The components of the CAMPUS middleware are briefly described as follows:

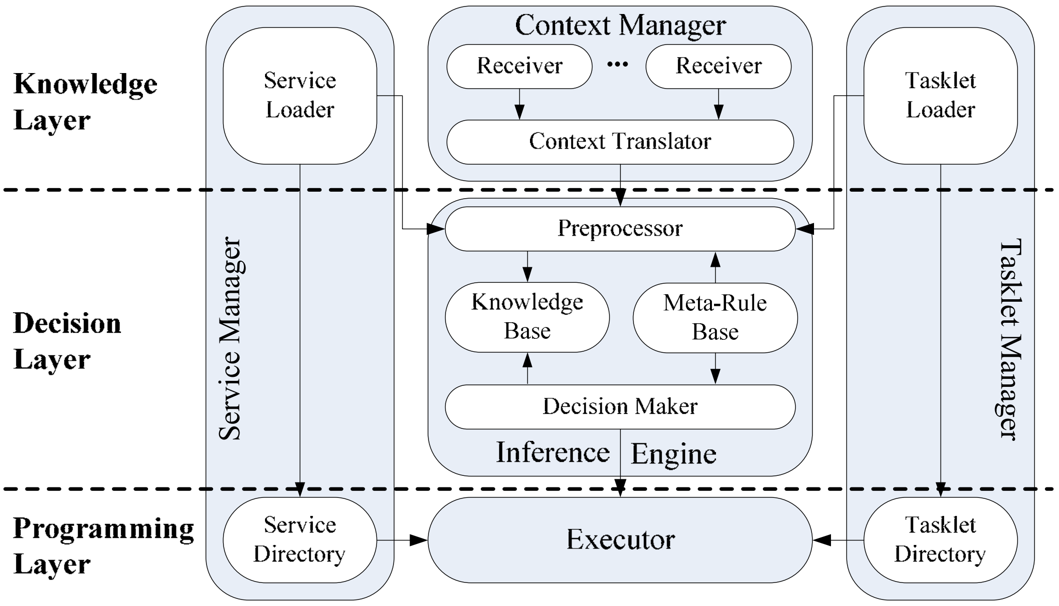


Figure 7-1 The Architecture of the CAMPUS Prototype.

- *Tasklet Manager* provides the basic service for managing a set of CAMPUS tasklets. As part of its initialization, the tasklet manager will attempt to load all available tasklet ontologies in the predefined directories, delegate them to the inference engine for preprocessing, and instantiate corresponding tasklet instances according to consistent tasklet ontologies. These tasklet instances are maintained in its *Tasklet Directory*, which provides mechanisms for querying and accessing tasklets via various conditions.

- *Service Manager* is responsible for managing the lifecycles of services deployed in CAMPUS. At its initialization, it will load and check all service ontologies. In particular, after checking the consistency of the service ontologies, the service loader will also check their compatibility. The compatibility checking process ensures that all services are composed correctly based on the semantic restrictions and constraints imposed by the CAMPUS programming model that is detailed in Chapter 4. Similar to the tasklet manager, it maintains a *Service Directory* that stores the detailed information of all of the services in CAMPUS.

- *Context Manager* is responsible for communicating with external context providers to obtain fine-grained contexts that will be translated into fragments of context ontologies. It also provides facilities for registering the context listeners, so that when new contexts are received, interested listeners such as the *Inference Engine* can be notified.

- *Inference Engine* handles issues related to adaptation decision-making. The major components of the inference engine include a *Meta-rule Base,* which stores all of the meta-rules that are defined for the decision maker to perform FOL reasoning; a *Knowledge Base*, which stores the ontological knowledge; a *Preprocessor*, which preprocesses raw ontologies; and a *Decision Maker*, which uses the two-phase process, as discussed in Chapter 6, to make adaptation decisions.
- *Executor* constructs and reconfigures context-aware applications according to instructions from the decision maker.

7.2 The CAMPUS API

CAMPUS provides a comprehensive set of APIs that provides a default implementation of the ATM programming model detailed in Chapter 4. The APIs comply with the principle of programming to interfaces, i.e., classes rely on collaborators' interfaces instead of on their coding [Flatt98]. The interfaces and classes defined by the CAMPUS APIs are in the *campus.model, campus.model.impl* and *campus.model.impl.desc* packages. The *model* package contains classes and interfaces that reflect the specification of the ATM model. As illustrated in Figure 7-2, each entity involved in the ATM model is represented by a Java interface that extends the super interface *IEntity* and defines the basic operations of this entity. The *impl* package provides a default implementation to these interfaces, in order to minimize the effort required to implement them. Figure 7-3 shows the greatly simplified but representative UML class diagram of the *impl* package. In particular, an entity in the *impl* package is instantiated using a descriptor design pattern [Lott05], and the related descriptors are included in the *desc* package. In the following sub-sections, we introduce the most important classes in the CAMPUS API.

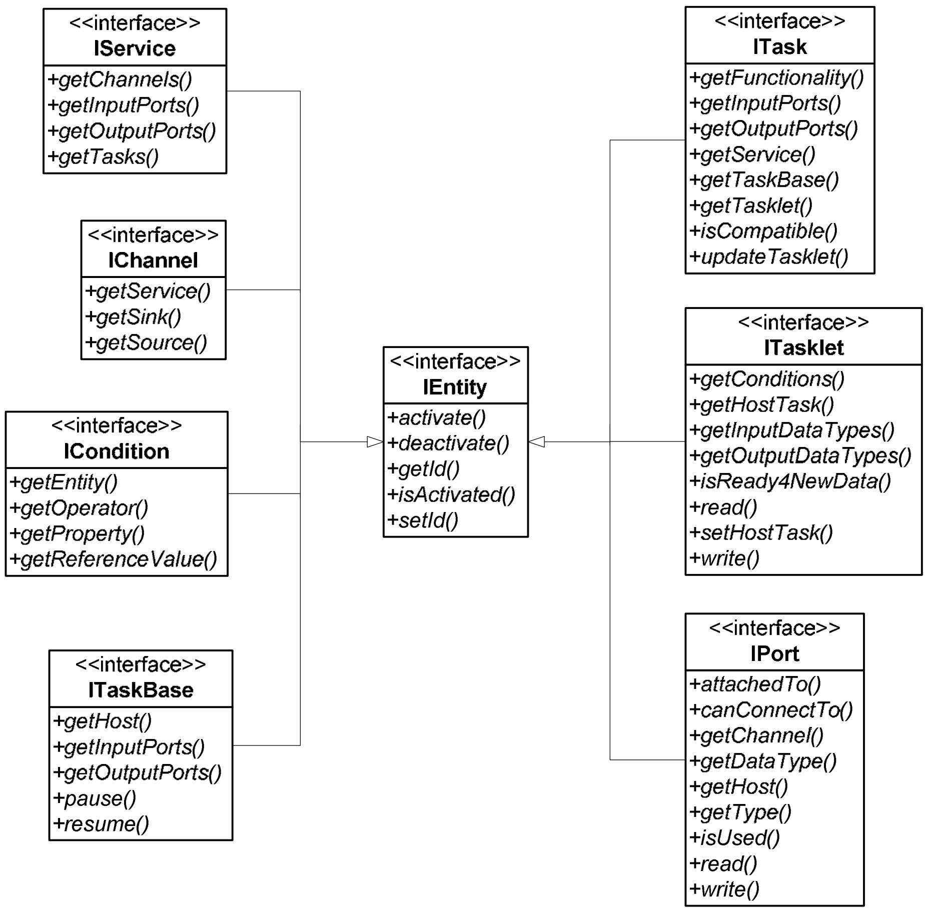


Figure 7-2 An Overview of the *model* Package.

7.2.1 Worker

The CAMPUS API is based on the multi-threaded technique. For example, a channel uses a scheduled thread to periodically transfer data from its source port to its sink port, and task bases use threads to feed and fetch data to and from tasklets. The *Worker* class is designed to more flexibly enable this feature. It implements the *Runnable* interface and can be seen as a Java thread in run-time. Importantly, a worker instance can be scheduled for one-time execution, or for repeated execution at regular intervals. When a worker is created, the execution interval can be specified in milliseconds. If the specified interval is not larger than 0, the worker will be executed only once. In order to be scheduled for execution, a worker needs to be activated via the *activate()* method. If a worker has been scheduled for repeated

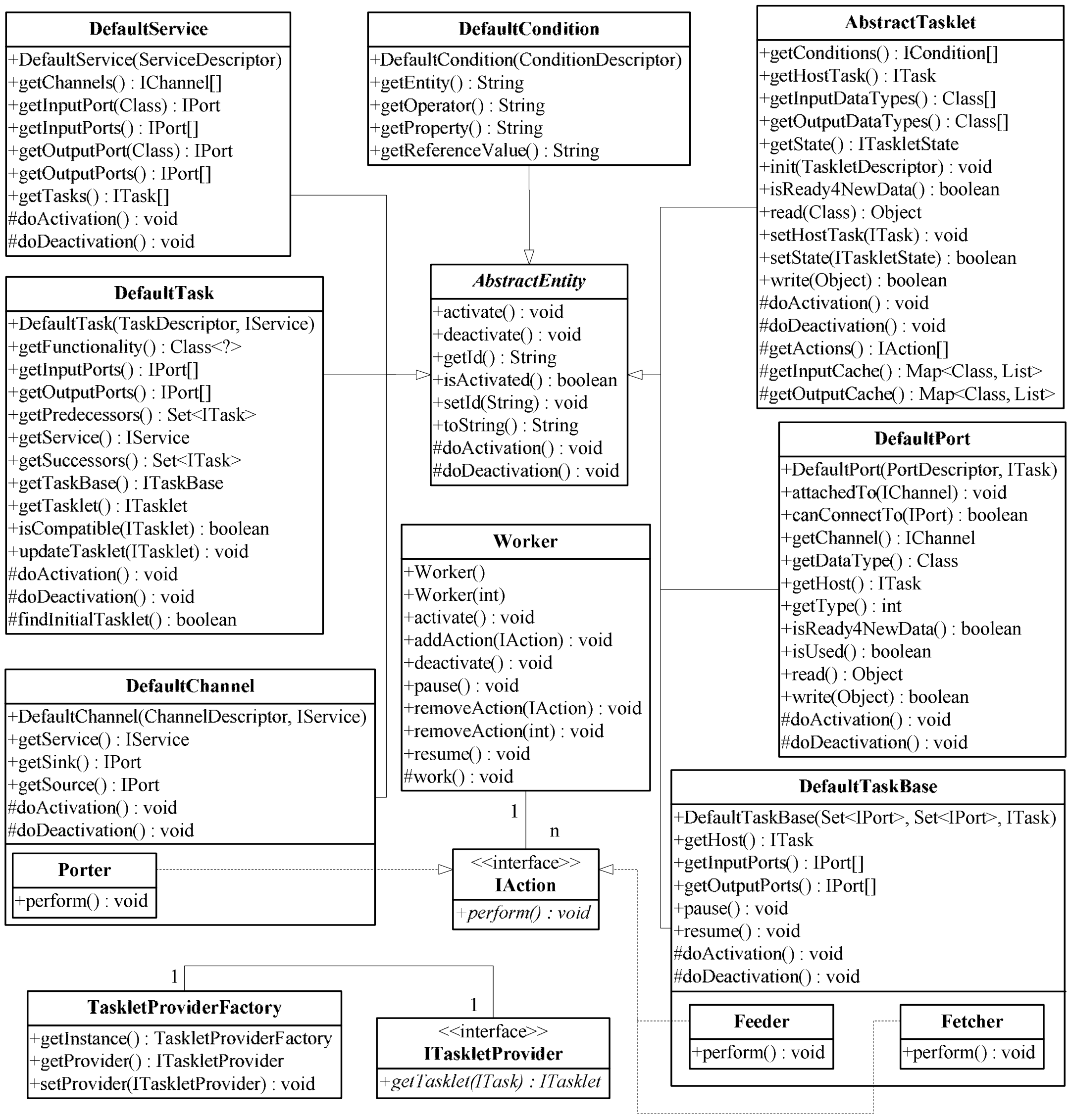

DefaultService
+DefaultService(ServiceDescriptor)
+getChannels() : IChannel[]
+getInputPort(Class) : IPort
+getInputPorts() : IPort[]
+getOutputPort(Class) : IPort
+getOutputPorts() : IPort[]
+getTasks() : ITask[]
#doActivation() : void
#doDeactivation() : void
DefaultCondition
+DefaultCondition(ConditionDescriptor)
+getEntity() : String
+getOperator() : String
+getProperty() : String
+getReferenceValue() : String
AbstractTasklet
+getConditions() : ICondition[]
+getHostTask() : ITask
+getInputDataTypes() : Class[]
+getOutputDataTypes() : Class[]
+getState() : ITaskletState
+init(TaskletDescriptor) : void
+isReady4NewData() : boolean
+read(Class) : Object
+setHostTask(ITask) : void
+setState(ITaskletState) : boolean
+write(Object) : boolean
#doActivation() : void
#doDeactivation() : void
#getActions() : IAction[]
#getInputCache() : Map<Class, List>
#getOutputCache() : Map<Class, List>
DefaultTask
+DefaultTask(TaskDescriptor, IService)
+getFunctionality() : Class<?>
+getInputPorts() : IPort[]
+getOutputPorts() : IPort[]
+getPredecessors() : Set<ITask>
+getService() : IService
+getSuccessors() : Set<ITask>
+getTaskBase() : ITaskBase
+getTasklet() : ITasklet
+isCompatible(ITasklet) : boolean
+updateTasklet(ITasklet) : void
#doActivation() : void
#doDeactivation() : void
#findInitialTasklet() : boolean
AbstractEntity
+activate() : void
+deactivate() : void
+getId() : String
+isActivated() : boolean
+setId(String) : void
+toString() : String
#doActivation() : void
#doDeactivation() : void
DefaultPort
+DefaultPort(PortDescriptor, ITask)
+attachedTo(IChannel) : void
+canConnectTo(IPort) : boolean
+getChannel() : IChannel
+getDataType() : Class
+getHost() : ITask
+getType() : int
+isReady4NewData() : boolean
+isUsed() : boolean
+read() : Object
+write(Object) : boolean
#doActivation() : void
#doDeactivation() : void
Worker
+Worker()
+Worker(int)
+activate() : void
+addAction(IAction) : void
+deactivate() : void
+pause() : void
+removeAction(IAction) : void
+removeAction(int) : void
+resume() : void
#work() : void
DefaultChannel
+DefaultChannel(ChannelDescriptor, IService)
+getService() : IService
+getSink() : IPort
+getSource() : IPort
#doActivation() : void
#doDeactivation() : void
Porter
+perform() : void
1
n
<<interface>>
IAction
+perform() : void
DefaultTaskBase
+DefaultTaskBase(Set<IPort>, Set<IPort>, ITask)
+getHost() : ITask
+getInputPorts() : IPort[]
+getOutputPorts() : IPort[]
+pause() : void
+resume() : void
#doActivation() : void
#doDeactivation() : void
Feeder
+perform() : void
Fetcher
+perform() : void
1
1
TaskletProviderFactory
+getInstance() : TaskletProviderFactory
+getProvider() : ITaskletProvider
+setProvider(ITaskletProvider) : void
<<interface>>
ITaskletProvider
+getTasklet(ITask) : ITasklet

Figure 7-3 An Overview of the *impl* Package.

execution, it may be paused, resumed, or deactivated. When it is paused, it will not run again until it is resumed. When it is deactivated, it will never run again. A worker may be assigned a number of actions via the method *addAction(IAction)*, and when it is executed, it will perform these actions in order.

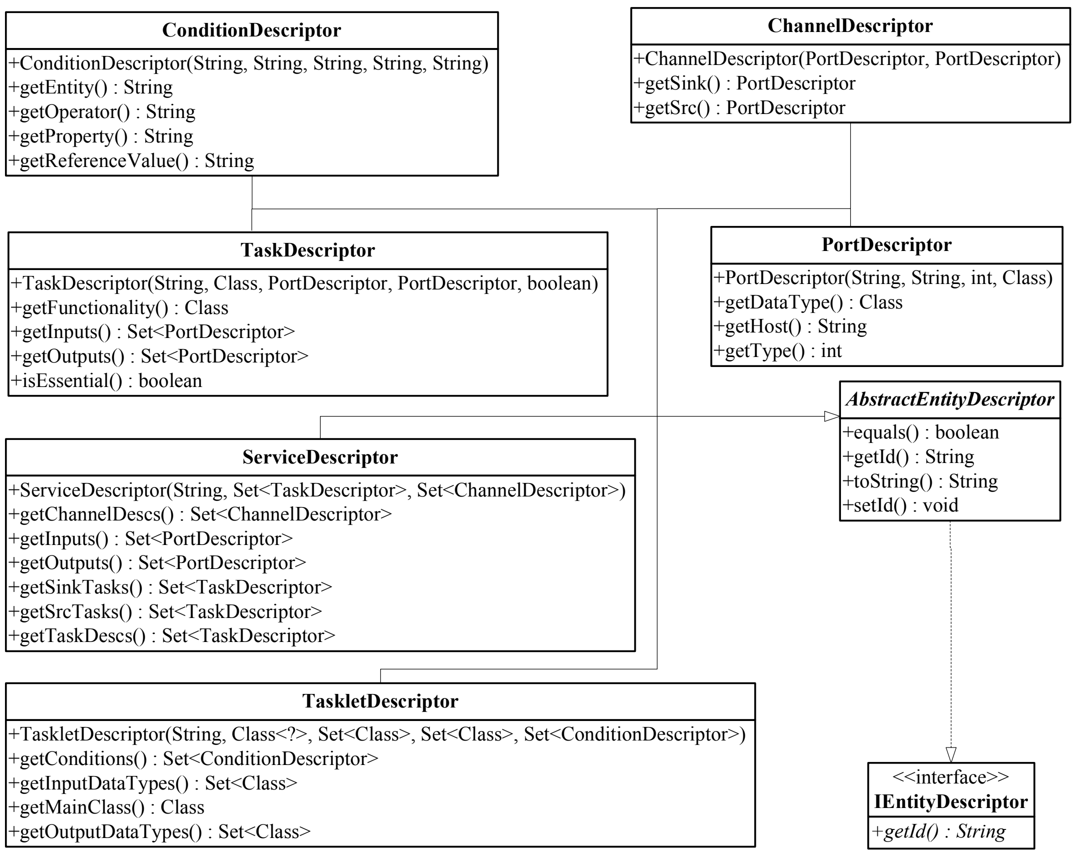

ConditionDescriptor
+ConditionDescriptor(String, String, String, String, String)
+getEntity() : String
+getOperator() : String
+getProperty() : String
+getReferenceValue() : String
ChannelDescriptor
+ChannelDescriptor(PortDescriptor, PortDescriptor)
+getSink() : PortDescriptor
+getSrc() : PortDescriptor
TaskDescriptor
+TaskDescriptor(String, Class, PortDescriptor, PortDescriptor, boolean)
+getFunctionality() : Class
+getInputs() : Set<PortDescriptor>
+getOutputs() : Set<PortDescriptor>
+isEssential() : boolean
PortDescriptor
+PortDescriptor(String, String, int, Class)
+getDataType() : Class
+getHost() : String
+getType() : int
AbstractEntityDescriptor
+equals() : boolean
+getId() : String
+toString() : String
+setId() : void
ServiceDescriptor
+ServiceDescriptor(String, Set<TaskDescriptor>, Set<ChannelDescriptor>)
+getChannelDescs() : Set<ChannelDescriptor>
+getInputs() : Set<PortDescriptor>
+getOutputs() : Set<PortDescriptor>
+getSinkTasks() : Set<TaskDescriptor>
+getSrcTasks() : Set<TaskDescriptor>
+getTaskDescs() : Set<TaskDescriptor>
TaskletDescriptor
+TaskletDescriptor(String, Class<?>, Set<Class>, Set<Class>, Set<ConditionDescriptor>)
+getConditions() : Set<ConditionDescriptor>
+getInputDataTypes() : Set<Class>
+getMainClass() : Class
+getOutputDataTypes() : Set<Class>
<<interface>>
IEntityDescriptor
+getId() : String

Figure 7-4 The Entity Descriptors.

7.2.2 Descriptors

Our implementation uses the descriptor design pattern [Lott05] to instantiate entities. Each entity class has a corresponding descriptor class that provides the detailed information to construct entity instances, as shown in Figure 7-4. Importantly, these descriptor classes provide a means for our programming framework to incorporate various description schemes without changing the codes. For example, on supplying an ontology adapter, entities can be instantiated from ontology files. Moreover, the descriptors check the semantic compatibility of the parameters when they are constructed, so that they can be directly used by their corresponding entities without worrying about whether the parameters are legal. For example, in the constructor of the *ChannelDescriptor* class, the descriptor will test whether the source and sink port parameters are compatible according to the semantic constraints presented in

Chapter 4. Figure 7-5 shows the source code of the *ChannelDescriptor* class, which tests the port constraints of the channel as follows:

- The type of the source port is *output* and the type of the sink port is *input*.
- The source port and the sink port belong to distinct tasks.
- The data types of the source port and sink port are identical.

```
public class ChannelDescriptor extends AbstractEntityDescriptor{

    private PortDescriptor src;
    private PortDescriptor sink;

    public ChannelDescriptor(PortDescriptor src, PortDescriptor sink) {

        setSrc(src);
        setSink(sink);
        if (src.equals(sink) ||
                src.getHost().equals(sink.getHost()) ||
                !sink.getDataType().isAssignableFrom(src.getDataType())) {
            throw new IllegalArgumentException(
                    "the data types of src and sink are incompatible");
        }
        setId(src.getId()+"-->"+sink.getId());
    }

    public PortDescriptor getSrc() {

        return src;
    }

    public PortDescriptor getSink() {

        return sink;
    }

    private void setSrc(PortDescriptor src) {
        if (src == null || src.getType()!=IPort.OUTPUT) {
            throw new IllegalArgumentException(
                    "src should not be null and it should be an output port");
        }
        this.src = src;
    }


    private void setSink(PortDescriptor sink) {
        if (sink == null || sink.getType()!=IPort.INPUT) {
            throw new IllegalArgumentException(
                    "sink should not be null and it should be an input port");
        }
        this.sink = sink;
    }
}
```

Figure 7-5 An Except of the Class *ChannelDescriptor*.

7.2.3 AbstractEntity

AbstractEntity provides a skeleton implementation of the super interface *IEntity*, and all concrete entity classes may directly extend it to minimize the development efforts. *AbstractEntity* captures the characteristics common to all ATM entities:

- Entities are located and retrieved by their unique identifiers. *AbstractEntity* provides two related methods: *getId():String* and *setId(String)*.
- The behavior of an entity is composed of a number of states: *initialized*, *activated*, *deactivated*, and *finalized* as shown in Figure 7-6. An entity is first instantiated by a constructor method. If the initialization process fails, the constructor method is required to throw an exception (*InitializationException*). If it succeeds, the entity state is set to *initialized*. Under this state, only accessor methods of the entity can be invoked, which access the contents of the entity object but do not modify the object. Once a client tries to invoke mutator methods that can modify the entity instance under the initialized state, an *IllegalStateException* should be thrown. An exception is the *activate()* method, which activates the entity and changes its state to *activated,* in which state an entity can be accessed fully. Similarly, when an entity fails to be activated, an *ActivationException* should be raised to indicate possible reasons for the failure. When the *deactivated()* method is called, the state of the entity will change to

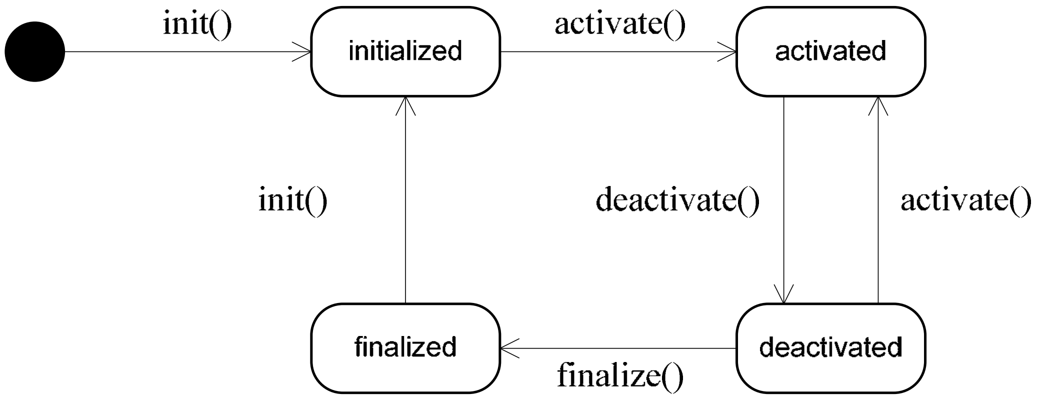


Figure 7-6 The Entity State Diagram.

deactivated, which means the mutator methods should no longer be called. Finally, when the *finalize()* method is invoked, the entity enters the *finalized* state, meaning that the entity instance has been disposed of.

7.2.4 AbstractTasklet

All interfaces defined in the *model* package, except for the *ITasklet*, have corresponding concrete class implementations in the *impl* package. We provided only a skeletal implementation of the *ITasklet* interface, which is called *AbstractTasklet* and cannot be instantiated. The reason we did not provide concrete implementation of *ITasklet* was simply because it is impossible to provide a general solution for all tasklets to process received data. Tasklet vendors need to extend the *AbstractTasklet* and provide their own solutions.

The *AbstractTasklet* has implemented all methods defined in the interface *ITasklet*. For example, methods like *getCondition*, *getInuptDataTypes*, *getOutputDataTypes*, and *getHostTask* can be used to determine whether the tasklet is suitable for a given task. When it is decided that a tasklet will be used for a particular task, the method *setHostTask* may be used to maintain a relationship between the tasklet and the task. In addition, if it is necessary for information on the state of the replaced tasklet to be kept and migrated to the replacement tasklet, the *getState* and *setState* methods can be used, Methods that include *isReady4NewData*, *read*, and *write* are used to exchange operating data between a tasklet and its task base.

In particular, *AbstractTasklet* maintains an input cache for incoming data and an output cache for outgoing data. In each cache, data are classified by their type and sorted according the order in which they are put into the cache. When the *read* method is called, *AbstractTasklet* will return the first data element, if any, of the specified type in the output cache; when the *write* method is called, *AbstractTasklet* checks the legality of the incoming data and then simply puts it into the input cache without any further processing. The task of processing incoming data is done by concrete tasklets that extend the *AbstractTasklet* class. A concrete tasklet class is required to implement the abstract method *IAction[] getActions()*. The returned actions indicate how the data in the input cache are to be handled and how output data for the output cache may be generated. Recall that the CAMPUS API provides a *Worker* class whose instances can be assigned a number of actions and scheduled for repeated execution at regular intervals. The *AbstractTasklet* class creates such a worker when it is activated, assigns the actions that are returned by the *getActions* method to the created worker, and then activates the worker. Through such a scheduled worker, a concrete tasklet periodically processes incoming data in the input cache and generates output data for the output cache. Finally, in order to keep its state consistent, when a tasklet is deactivated, it will first wait for the input cache to be emptied; in other words, when there is no more data that needs to be processed by the tasklet. After that, the tasklet deactivates its worker, and waits for the output cache to be emptied; that is, for all output data to have been fetched by the task base.

7.2.5 DefaultTask

DefaultTask implements the *ITask* interface and exposes the basic operations of a task of the ATM model. In particular, *getInputPorts* and *getOutputPorts* return the input and output ports of the task. Recall that each port only accepts data of one particular type; therefore these defined ports, together with the functionality returned by *getFunctionality*, consist of the requirements of the task. The operation *isCompatible* is used to test whether a given tasklet is compatible with the task, i.e., whether it can be used for the task, while *updateTasklet* is used to replace the existing tasklet with the given one. The given tasklet can be null, which means removing the existing tasklet. The operation *getSuccessors* returns tasks that are connected to the input ports of the task and *getPredecessors* return tasks that are connected to the output ports of the task. In this section, we detail two important sequences of a task: the activation sequence and the adaptation sequence.

As shown in Figure 7-7, when a *DefaultTask* is activated, it will ask the *TaskletProviderFactory* to return an available tasklet provider that implements the interface *ITaskletProvider*. A tasklet provider acts as a decision maker that is responsible for choosing a suitable tasklet for a given task upon request. The CAMPUS API does not provide any concrete implementation of the interface *ItaskletProvider*. Instead, it provides a *TaskletProviderFactory* that implements the Factory design pattern [Gamma95]. Importantly, the design choice makes the API independent of particular decision-making mechanisms, while providing a chance for various decision-making technologies to be adopted without modifying the existing codes. For example, in the current implementation of the CAMPUS middleware, the *Inference Engine* can to some extent be seen as a tasklet provider. On obtaining a tasklet provider from the tasklet provider factory, the *DefaultTask* will ask the provider for a suitable tasklet and activate it. Subsequently, the task will begin to activate the task base part. The latter will then activate the input and output ports of the task and start a worker that is assigned with two actions: *Feeder* and *Fetcher*. The feeder will periodically ask the tasklet if it is ready to accept new data

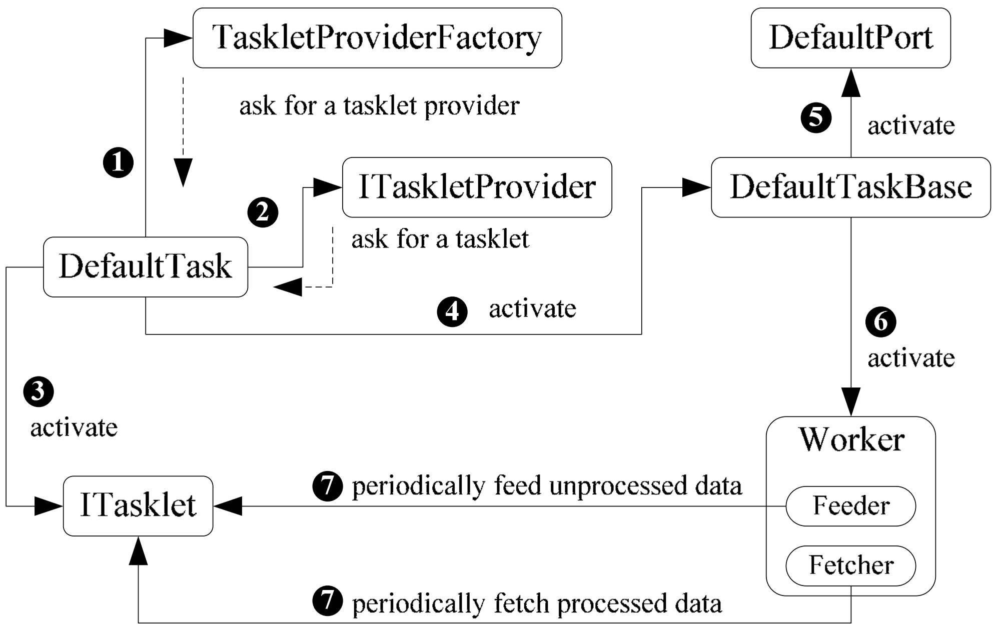


Figure 7-7 The Activation Process of *DefaultTask*.

via the method *isReady4NewData()* of *ITasklet*. If the tasklet's response is positive, the feeder will use the method *write()* of *ITasklet* to forward the operating data to the tasklet, which it has received from the input ports. Similarly, the fetcher of the task base will periodically fetch processed data from the tasklet via the method *read()*.

Another sequence of the *DefaultTask* presented here is the adaptation process, i.e., the process of updating the tasklet. The *DefaultTask* will first check whether the given tasklet is compatible, using the following criteria:

- The tasklet is not null and has not been used for any other task.
- The tasklet can provide the required functionality of the task.
- The tasklet declares that it can process all of the input data that the task has received.
- The tasklet declares that it can generate all of the output data that the task requires.

If the given tasklet is compatible with the task, the task will deactivate its existing tasklet and suspend its task base. Recall that when a task base is paused, its worker will not be executed until it is resumed, i.e., the task base will temporarily not feed and fetch data to and from the tasklet. After that, the existing tasklet is unplugged from the task and its state is migrated to the new tasklet. Next, the new tasklet will be activated and plugged into the task. Finally, the task base is resumed, i.e., it will begin to write and read data to and from the new tasklet. The sequencing of events is necessary to ensure consistency and that no data will be lost during adaptation.

Chapter 8 Case Study

The best demonstration of the middleware's ability to ease the development of context-aware applications is made by example. This section presents a pragmatic example to demonstrate how CAMPUS can be leveraged for the development of context-aware applications. Importantly, it provides us with a platform to understand the complete operational flow of the middleware and to study the complex interactions among the core components in the system. To demonstrate the ease of integrating CAMPUS to existing applications, we have chosen an open-source instant messenger application called Spark[1] to which to add a context-aware service for users to initiate or respond to one-to-one, peer-to-peer voice, and video chats. During the chat, the service can automatically adapt to various situations. For example, when the network bandwidth is low, the service will automatically reduce the quality of the video to ensure that the transfer of data is smooth, or use voice only to further improve the quality of the transfer. The chat service uses XMPP/Jingle[2] as the negotiation protocol, and RTP/UDP as the media transportation protocol. In addition, it uses JMF[3] to capture, transmit, and receive streaming data.

8.1 Developing CAMPUS Services and Tasklets

Developing a CAMPUS service means developing a service ontology that describes the semantics of the service, and collecting tasklet implementations that may satisfy the requirements presented by the tasks of the services. We used Protégé OWL, a popular graphical ontology editor, to develop the required ontology for the chat service. Figure 8-1 shows an excerpt of the service ontology.

[1] http://www.igniterealtime.org/projects/spark/index.jsp
[2] http://xmpp.org/extensions/xep-0166.html
[3] http://java.sun.com/javase/technologies/desktop/media/jmf/

```
<rdf:RDF
        …
        <service:Service rdf:ID="sim">
                <service:consistsOf rdf:resource="#T_AudioDS"/>
                <service:consistsOf rdf:resource="#T_Transcoder"/>
                <service:consistsOf rdf:resource="#T_VideoDS"/>
                <service:consistsOf rdf:resource="#T_Renderer"/>
                <service:consistsOf rdf:resource="#T_JSessionCtor"/>
                <service:consistsOf rdf:resource="#T_Transmitter"/>
                <service:consistsOf rdf:resource="#T_Receiver"/>
                <service:consistsOf rdf:resource="#T_TxSessionCtor"/>
                <service:consistsOf rdf:resource="#T_RxSesseionCtor"/>
        </service:Service>
        <service:EssentialTask rdf:ID="T_AudioDS">
                <service:owns rdf:resource="#out0_AudioDS"/>
                <service:requires rdf:resource="#F_AudioDS"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_JSessionCtor">
                <service:owns rdf:resource="#in0_JSessionCtor"/>
                <service:owns rdf:resource="#out0_JSessionCtor"/>
                <service:owns rdf:resource="#out1_JSessionCtor"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_Receiver">
                <service:owns rdf:resource="#in0_Receiver"/>
                <service:owns rdf:resource="#out0_Receiver"/>
                <service:requires rdf:resource="#F_Receiver"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_Renderer">
                <service:owns rdf:resource="#in0_Renderer"/>
                <service:requires rdf:resource="#F_Renderer"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_RxSesseionCtor">
                <service:owns rdf:resource="#in0_RxSessionCtor"/>
                <service:owns rdf:resource="#out0_RxSessionCtor"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_Transcoder">
                <service:owns rdf:resource="#in0_Transcoder"/>
                <service:owns rdf:resource="#out0_Transcoder"/>
                <service:owns rdf:resource="#in1_Transcoder"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_Transmitter">
                <service:owns rdf:resource="#in0_Transmitter"/>
                <service:owns rdf:resource="#in1_Sender"/>
                <service:requires rdf:resource="#F_Transmitter"/>
        </service:EssentialTask>
        <service:EssentialTask rdf:ID="T_TxSessionCtor">
                <service:owns rdf:resource="#in0_TxSessionCtor"/>
                <service:owns rdf:resource="#out0_TxSessionCtor"/>
                <service:requires rdf:resource="#F_RxSessionCtor"/>
        </service:EssentialTask>
        <service:ExpansionTask rdf:ID="T_VideoDS">
                <service:owns rdf:resource="#out0_VideoDS"/>
                <service:requires rdf:resource="#F_VideoDS"/>
        </service:ExpansionTask>
</rdf:RDF>
```

Figure 8-1 An Excerpt of the Chat Service Ontology.

In particular, the service ontology declares a context-aware chat service that consists of eight essential tasks and one expansion task, each of which describes the specific

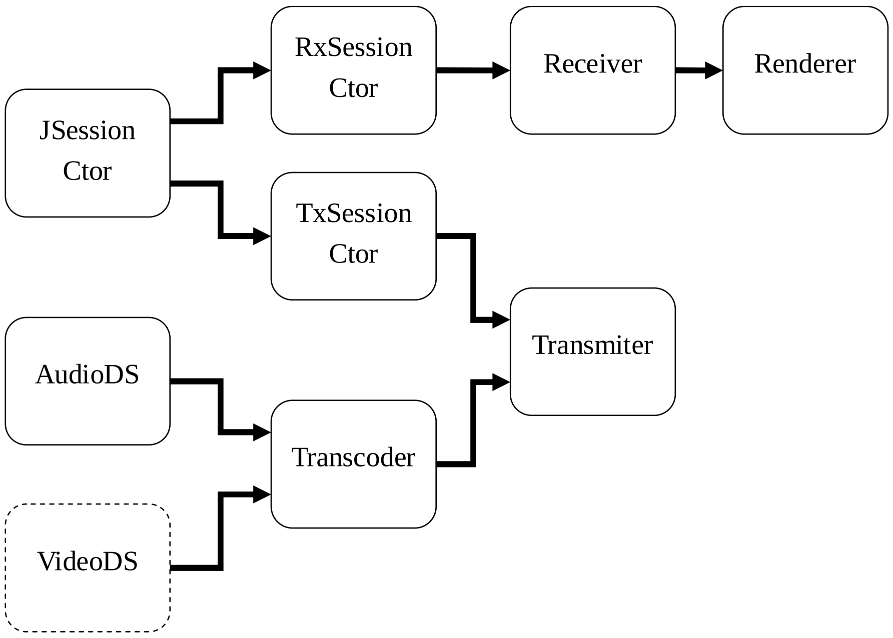


Figure 8-2 The Structure of the Chat Service.

requirements for its tasklet. Figure 8-2 illustrates the composition of these tasks and Table 8-1 lists their requirments.

Table 8-1 Summary of the Tasks Defined in the Chat Service Ontology.

Task	Requirement	Description
AudioDS	**Functionality**: F_AudioDS **Output Type**: AudioData	Captures raw audio data.
VideoDS	**Functionality**: F_VideoDS **Output Type**: VideoData	Captures raw video data. It is an expansion task.
Transcoder	**Functionality**: F_Transcoder **Input Type**: AudioData VideoData **Output Type**: MediaData	Transcodes captured raw media data.
JSession Ctor	**Functionality**: F_JSessionCtor **Input Type**: JSessionReq **Output Type**: RxJingleSession TxJingleSession	Creates jingle sessions.
RxSession	**Functionality**: F_RxSessionCtor	Creates a media transportation

Ctor	**Input Type**: RxJingleSession **Output Type**: RxSession	session for the receiver.
TxSession Ctor	**Functionality**: F_TxSessionCtor **Input Type**: TxJingleSession **Output Type:** TxSession	Creates a media transportation session for the transmitter.
Receiver	**Functionality**: F_Receiver **Input Type**: RxSession **Output Type**: MediaData	Receives media data over the network.
Renderer	**Functionality**: F_Renderer **Input Type**: MediaData	Renders the received media data.
Transmitter	**Functionality**: F_Transmitter **Input Type**: TxSession MediaData	Sends media data over the network.

For demonstration purposes, we have developed one or more compatible tasklets for each task of the chat service. Table 8-2 summarizes the tasklets developed for the chat service. Importantly, these tasklets are reusable since they are only responsible for providing the desired functionalities and are treated as a black box that uses predefined interfaces to communicate with others.

Table 8-2 Summary of the Tasklets Developed for the Chat Service.

Tasklet	Property	Requirement	Description
MicDS	**Functionality**: F_AudioDS **Output Type**: AudioData	MIC.isAailable = “true”	Captures raw audio data from the microphone.
WebcamDS	**Functionality**: F_VideoDS **Output Type**: VideoData	Camera.isAvailable = “true”	Captures raw video data from the web camera.
TC4HighBW	**Functionality**: F_Transcoder **Input Type**: AudioData VideoData **Output Type**: MediaData	Network.isAvailable = “true” Network.bandwidth >= “1.5 Mbps”	Transcodes raw audio and video data using the MPEG-1 audio and MJPEG compression algorithm, respectively.
TC4MedBW	**Functionality**: F_Transcoder	Network.isAvailable = “true”	Transcodes raw audio and video

	Input Type: AudioData VideoData **Output Type**: MediaData	Network.bandwidth >= "512 Kbps"	data using the G723.1 and H.263 compression algorithm, respectively.
TC4LowBW	**Functionality**: F_Transcoder **Input Type**: AudioData VideoData **Output Type**: MediaData	Network.isAvailable = "true" Network.bandwidth >= "64 Kbps"	Transcodes raw audio data using G723.1, and filters off the captured video data.
Jingle SessionCtor	**Functionality**: F_JSessionCtor **Input Type**: JSessionReq **Output Type**: RxJingleSession TxJingleSession	Network.isAvailable = "true"	Creates a jingle session over the RTP protocol.
RTP SessionCtor	**Functionality**: F_RcvrSessionCtor F_SndrSessionCtor **Input Type**: JingleSession **Output Type**: RTPSession	Network.isAvailable = "true"	Creates a media transportation session over RTP protocol that can be used to transmit or receive streaming data.
RTP Receiver	**Functionality**: F_Receiver **Input Type**: RxSession **Output Type**: MediaData	Network.isAvailable = "true"	Receives media data over the network using an RTP session.
Player	**Functionality**: F_Renderer **Input Type**: MediaData	Speaker.isAvailable = "true" Screen.isAvailable = "true"	Processes received media data in a track and delivers it to the screen and the speaker.
RTP Transmitter	**Functionality**: F_Transmitter **Input Type**: TxSession MediaData	Network.isAvailable = "true"	Sends media data over the network using an RTP session.

In order to develop CAMPUS tasklets, tasklet vendors need to provide a compatible implementation by extending the *AbstractTasklet* base class and implementing the abstract method *getActions* that has been discussed in Chapter 7. The returned actions indicate how to handle input data and how to generate output data. Figure 8-3 shows an excerpt from such a tasklet implementation. In addition, a tasklet requires a

```
public class MicDS extends AbstractTasklet implements IAudioDS {

    private IAction[] actions = new IAction[] { new IAction() {
        public boolean perform() {
            Vector devices = CaptureDeviceManager
                    .getDeviceList(
                        new AudioFormat("linear", 44100, 16, 2));
            if (devices.size() > 0) {
                try {
                    CaptureDeviceInfo cdi
                        = (CaptureDeviceInfo) devices.firstElement();
                    getOutputCache().get(AudioData.class).add(
                        new AudioData(
                            Manager.createCloneableDataSource(
                                Manager.createDataSource(
                                    cdi.getLocator()))));
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
                return true;
            }
            return false;
        }
    } };

    protected IAction[] getActions() {

        return actions;
    }
}
```

Figure 8-3 An Excerpt of the *MicDS* Implementation.

corresponding ontology that extends the tasklet foundation ontology to describe the properties and contextual requirements of the tasklet. The tasklet ontologies will be further discussed in Section 8.2.

8.2 Preprocessing Ontologies

Recall that when CAMPUS launches, the service manager will load the service ontologies and use them to construct service instances if they pass the consistency and compatibility checking stage, and the tasklet manager will load the tasklet ontologies and use them to construct tasklet instances. Furthermore, the consistent service and tasklet ontologies will be used by the *registering* meta-rule, detailed in

Chapter 6, to construct a task registry where tasklets that are compatible with a particular task are registered as its alternatives. A tasklet is compatible with a task if and only if:

- It can provide the desired functionality of the task.
- It can process data from the input ports of the task, each of which accepts one particular data type.
- It can generate data for the output ports of the task, each of which requires one particular data type.

According to the above criteria, the task registry of the chat service is listed in Table 8-3. Importantly, when a tasklet is registered as an alternative for a particular task, the contextual aspects that are involved in the conditions of the tasklet will be registered as the contexts of interest for the task. For example, the tasklet *TC4HighBW* is registered as an alternative of the task *Transcoder,* and the contextual conditions of *TC4HighBW* contain two aspects, *isAvailable* and *bandwidth*, of the context class *Network*. Thus, *Transcoder* will subscribe to an update of these two aspects of *Network*. Once these aspects of *Network* (i.e., *isAvailable* and *bandwidth*) are updated, *Transcoder* will be notified.

Table 8-3 The Task Registry of the Chat Service.

Task	**Tasklet**
AudioDS	MicDS
VideoDs	WebcamDS
Transcoder	TC4HighBW TC4MedBW TC4LowBW
JsessionCtor	JingleSessionCtor
RxSessionCtor	RTPSessionCtor
TxSessionCtor	RTPSessionCtor
Receiver	RTPReceiver
Renderer	Player
Transmitter	RTPTransmitter

Table 8-4 Comparison of *RTPSessionCtor*, *RxSessionCtor*, and *TxSessionCtor*.

	Task		**Tasklet**
	RxSessionCtor	**TxSessionCtor**	**RTPSessionCtor**
Functionality	F_RxSessionCtor	F_TxSessionCtor	F_RcvrSessionCtor F_SndrSessionCtor
Input Type	RxJingleSession	TxJingleSession	JingleSession
Output Type	RxSession	TxSession	RTPSession

In particular, the registration of the tasklet *RTPSessionCtor* demonstrates the semantic feature of CAMPUS, i.e., its reasoning about the ontologies is based on the semantics of concepts. *RTPSessionCtor* is registered as a tasklet alternative to the tasks *RxSessionCtor* and *TxSessionCtor*. Although the functionalities and I/O data types of *RTPSessionCtor*, as listed in Table 8-4, are not an exact match as required by the two tasks, they are semantically compatible. *RTPSessionCtor* semantically satisfies the requirements presented by *RxSessionCtor* and *TxSessionCtor* in the following ways:

- It can provide the desired functionality of *RxSessionCtor* and *TxSessionCtor*. As shown in Figure 8-4, the functions *F_RcvrSessionCtor* and *F_SndrSessionCtor* provided by *RTPSessionCtor* are declared, via the *owl:sameAs* axiom, to be semantically the same as the *F_RxSessionCtor* and F*_TxSessionCtor* that are required respectively by *RxSessionCtor* and *TxSessionCtor*. The declaration means that the tasklet *RTPSessionCtor* is able to provide functionalities similar to those required by *RxSessionCtor* and *TxSessionCtor*.

- It can process data from the input ports of *RxSessionCtor* and *TxSessionCtor*. *RxSessionCtor* has one input port, whose data type is *RxJingleSession;* and *TxSessionCtor* has one input port, whose data type is *TxJingleSession*. On the other hand, it has been declared that *RTPSessionCtor* is able to process data with the type *JingleSession*, which is the super class of *RxJingleSession* and *TxJingleSession*. This means that *RTPSessionCtor* is able to process data of the types *RxJingleSession* or *TxJingleSession*.

```
<rdf:RDF
  …
  <tasklet:Function rdf:ID="F_RcvrSessionCtor">
    <owl:sameAs rdf:resource="&sim;F_RxSessionCtor"/>
    <tasklet:implementedBy rdf:datatype="&xsd;string">
                sim.service.phone.RTPSessionCtor</tasklet:implementedBy>
  </tasklet:Function>
    <tasklet:Function rdf:ID="F_SndrSessionCtor">
        <owl:sameAs rdf:resource="&sim;F_TxSessionCtor"/>
        <tasklet:implementedBy rdf:datatype="&xsd;string">
            sim.service.phone.RTPSessionCtor</tasklet:implementedBy>
    </tasklet:Function>
    <tasklet:DataTypeCondition rdf:ID="Network.isAvailable_equals_to_true">
        <tasklet:hasReferenceValue rdf:datatype="&xsd;boolean">
      true</tasklet:hasReferenceValue>
        <tasklet:hasEntity rdf:resource="&context;Network"/>
        <tasklet:hasOperator rdf:resource="&tasklet;equal"/>
        <tasklet:hasProperty rdf:resource="&context;isAvailable"/>
    </tasklet:DataTypeCondition>
    <tasklet:DataType rdf:ID="RTPSession">
        <tasklet:implementedBy rdf:datatype="&xsd;string">
            sim.service.phone.data.RTPSession</tasklet:implementedBy>
        <tasklet:extends rdf:resource="&sim;RxSession"/>
        <tasklet:extends rdf:resource="&sim;TxSession"/>
    </tasklet:DataType>
    <tasklet:Tasklet rdf:ID="RTPSessionCtor">
        <tasklet:implementedBy rdf:datatype="&xsd;string">
            sim.service.phone.impl.RTPSessionCtor</tasklet:implementedBy>
        <tasklet:asserts rdf:resource="#Network.isAvailable_equals_to_true"/>
        <tasklet:hasInputDataTyppe rdf:resource="&sim;JingleSession"/>
        <tasklet:hasOutputDataType rdf:resource="#RTPSession"/>
        <tasklet:provides rdf:resource="#F_RcvrSessionCtor"/>
        <tasklet:provides rdf:resource="#F_SndrSessionCtor"/>
    </tasklet:Tasklet>
</rdf:RDF>
```

Figure 8-4 An Excerpt of the *RTPSessionCtor* ontology.

- It can generate data for the output ports of *RxSessionCtor* and *TxSessionCtor*. *RxSessionCtor* has one output port, whose data type is *RxSession*, and *TxSessionCtor* has one output port, whose data type is *TxSession*. On the other hand, the output type of *RTPSessionCtor* is *RTPSession*, which has been declared to extend *RxSession* and *TxSession*. This means that the data generated by *RTPSessionCtor* can be seen as data of the types *RxSession* or *TxSession*.

8.3 Making Decisions at Run-time

In this section, we explain the decision-making process using the chat service. Recall that the decision-making process is a two-phased process that is triggered upon request by a task under three kinds of situations: when a task is activated, when its running tasklet reports an error, and when it is notified of an updated context of interest. We use three scenarios to illustrate these cases.

When the chat service is activated, each of its tasks will ask the Tasklet Manager for a suitable tasklet. Upon request, the Tasklet Manager will query the Inference Engine for a suitable tasklet id. The decision-making process then starts. Here we use the task *Transcoder* as an example to explain the process step by step:

1. The inference engine searches the task registry to get the registered tasklet alternatives. In the case of *Transcoder*, there exist three tasklet alternatives, as explained previously. Table 8-5 lists these alternatives.

Table 8-5 Tasklet Alternatives for *Transcoder*.

	Task	**Tasklet**		
	Transcoder	**TC4HighBW**	**TC4MedBW**	**TC4LowBW**
Functionality	F_Transcoder	F_Transcoder	F_Transcoder	F_Transcoder
Input Type	AudioData VideoData	AudioData VideoData	AudioData VideoData	AudioData VideoData
Output Type	MediaData	MediaData	MediaData	MediaData
Contextual Requirement		Network.isAvailable = “true” Network.bandwidth >= “1.5 Mbps”	Network.isAvailable = “true” Network.bandwidth >= “512 Kbps”	Network.isAvailable = “true” Network.bandwidth >= “64 Kbps”

2. The contextual requirements of each alternative will be tested in the screening phase. If the contextual requirements of an alternative are not satisfied, the alternative will be screened out in this phase. In the case of *Transcoder*, assuming that the network is available and the available bandwidth is 512 Kbps, the tasklet *TC4HighBW* will be filtered off in the screening phase.

3. The fitness utility of each alternative that passes the screening phase is computed in the choice phase, and the alternative with the maximum fitness utility is chosen. In our example, *TC4MedBW* and *TC4LowBW* pass the screening phase, and their fitness utilities are computed using Equations 6-1 and 6-2, which are defined by the default *choice* meta-rule. The results of the calculation are that the overall fitness utility of *TC4MedBW* is 1.0, and that *TC4LowBW* is 0.5. That is, the tasklet *TC4MedBW* will be chosen since its fitness utility is higher than that of *TC4LowBW*.

4. Using the tasklet id returned by the inference engine, the tasklet manager locates a corresponding tasklet instance in its tasklet directory and returns it to the requesting task. Finally, the task uses the tasklet to continue its activation process, which has been detailed in Chapter 7.

Suppose that the chat service has been successfully activated and the user begins to chat with his friend. Subsequently, let us assume that the tasklet *WebcamDS* encounters a problem and fails to work normally (due, for example, to an interruption of communications involving the web camera). As a result, it reports an error to its host task, i.e. *VideoDS*, and the latter will ask the tasklet manager for another appropriate tasklet. Again, the decision-making process starts. This time, since the task *VideoDS* has only one tasklet alternative, *WebcamDS*, the inference engine will still choose *WebcamDS* as the best alternative for *VideoDS*. Since *WebcamDS* has been marked as an abnormal tasklet in the tasklet manager, the latter returns a *NULL* reference to the task, which means that no suitable tasklet has been found. Note that *VideoDS* is an expansion task that allows an empty tasklet part, therefore the task *VideoDS* will directly remove its abnormal tasklet, i.e. *WebcamDS*, without any problem.

In the last scenario, we assume that the network goes down during the chat and the context manager receives an updated context from one of its context receivers, which indicates that the available bandwidth of the network is 256 Kbps. In that case, the task *Transcoder* will be notified since it has subscribed for an update of the context *Network*. Once informed, *Transcoder* will again ask the tasklet manager to return the potentially best tasklet according to the updated context. This time, only the tasklet *TC4LowBW* passes the screening phase. It is returned to *Transcoder* as the recommended tasklet, which will be used to replace the original tasklet part of *VideoDS*, as detailed in Chapter 7.

8.4 Discussion

This simple, but representative, example demonstrates several important features of CAMPUS. First, CAMPUS frees developers from the need to predict, formulate, and maintain adaptation rules, thereby greatly reducing the efforts required to develop context-aware applications. For example, the chat service in the case study consists of 9 tasks. Suppose that each task has 2 tasklet alternatives. As a result, the service configuration will have as many as $2^9=512$ variants. When using the previous rule-based systems, developers need to formulate *512* adaptation rules in order to reflect all these variants. Obviously, this is not an easy task. Instead, when using CAMPUS, developers need to prepare only $2\times9=18$ tasklet-specific ontologies, each of which describes the required semantics of one tasklet, as detailed previously. That is to say, previous rule-based systems indicate an exponential increase in the size of adaptation rules when the variants increase, and CAMPUS indicates a linear increase in the size of ontologies when the variants increase. Furthermore, assume that we now change the composition of the chat service and add a new task *Jitter* between the task *Receiver* and *Renderer*. In order to reflect the change, rule-based systems need to revise all the *512* adaptation rules defined, while CAMPUS only needs to simply modify the service ontology, as shown in Figure 8-1, to add a new task instance. Admittedly, while being freed from formulating adaptation rules, developers are required to construct domain-specific ontologies when using CAMPUS. However, the efforts of building and maintaining domain-specific ontologies can be greatly reduced by using a graphical editor like Protégé OWL and extending the CAMPUS foundation ontology. In our experience in the case study, building and maintaining domain-specific ontologies is very easy.

Secondly, CAMPUS effectively separates the concerns of adaptation and coordination from the computational concerns of a task. The issues of how to connect tasks and how to transfer operating data are handled by the middleware layer. Importantly, a tasklet based on CAMPUS is solely responsible for realizing its own functional logic. This feature not only further reduces development efforts, but also promotes the reuse of tasklets.

Finally, CAMPUS places few structural or functional limitations on existing applications that want to make use of CAMPUS's facilities for context-aware adaptation. This feature makes it easy to enhance existing applications to become context-aware. In the case study, an existing instant messenger, Spark, was enhanced by CAMPUS to provide a context-aware service of voice and video chats, without affecting its existing codes.

The aim of this chapter was to present a case application of how CAMPUS can be used to develop or extend applications to become context-aware. Importantly, the chapter explained how the concepts discussed in earlier chapters have been systematically mapped out and implemented in a real and practical application. In the next chapter, we will present and analyze some empirical performance results that were collected from the application setup.

Chapter 9 Performance Evaluation

A central goal of CAMPUS is to simplify the development of context-aware applications. While the programming interface and its use, described in the previous chapters, are important to this goal, the performance of the middleware must also be of paramount concern to ensure that the overheads associated with using the middleware are not detrimental to the application's operations. A set of experiments was conducted to measure the potential computational overheads that could be incurred by using the CAMPUS system under different situations. The goal of the evaluation is to provide developers with a better understanding of the performance of CAMPUS under different operating conditions. Importantly, by analyzing and comparing the results, we hope to gain further insights into the characteristics of CAMPUS.

Based on our past experience, semantic reasoning is a computationally intensive task, so that it may act as a possible performance bottleneck for CAMPUS. The first experiment presented in this section was designed to test the overhead incurred by the middleware when reasoning the ontology to make adaptation decisions. Another possible performance bottleneck could come from the additional work involved in transmitting data to and from tasks. The second experiment presented in this section measured the overhead incurred by transferring data between tasks. The third experiment was to measure the overhead incurred by updating a task to adapt to contextual changes, and the last experiment presented in this section measured the overall system performance of CAMPUS from an end-to-end perspective. All of the experiments were conducted on an HP Compaq NC6400 laptop equipped with 1G MB of RAM and an Intel Core Duo T2300E processor rated at 1.66G Hz. The operating system used was Microsoft's WinXP and the Java Virtual Machine version was 1.6.0.

9.1 Semantic Reasoning Overhead

The CAMPUS middleware performs DL reasoning and FOL reasoning on the ontologies to make adaptation decisions. Consequently, the size of the knowledge base, i.e. the loaded ontologies, and the complexity of the defined meta-rules will greatly affect the processing overheads of ontology reasoning. This set of experiments evaluates the impact of the size of loaded ontologies and the complexity of meta-rules on the processing overheads of ontology reasoning. The size of the knowledge base was measured in terms of the number of RDF statements, and the complexity of the meta-rules was measured in terms of the number of rule atoms. The experimental results are shown in Figure 9-1. As expected, the results clearly indicate an increase in the time overheads of ontology reasoning, with a progressive increase in the size of the ontologies and the complexity of the meta-rules. The rate of increase is nearly linear. The experimental results also show that semantic reasoning is a computationally intensive task. However, the time overhead (in the range of a couple of seconds) under current CPU speeds is reasonable for non-time-critical applications, considering the fact that the size of the knowledge base of most

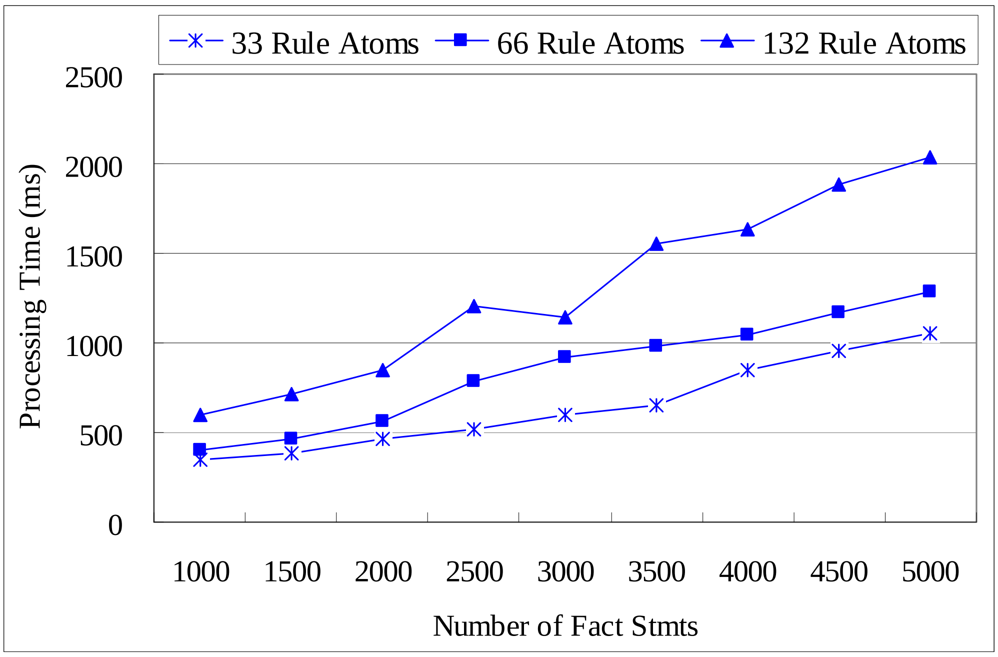


Figure 9-1 The Processing Overheads of Semantic Reasoning.

applications is unlikely to exceed five thousand, and the size of rules is generally less than two hundred.

9.2 Data Transfer Overhead

This set of experiments was designed to measure the processing overheads incurred by a complete CAMPUS service in processing a given chunk of data. Ignoring the function processing time, for a specific service the overheads incurred come primarily from the additional work involved in transmitting data to and from tasks. In this set of experiments, a special service has been configured to be composed of a set of tasks with the same functional requirement. The primary logic of the tasklet instances that are involved is to receive data from their host tasks, and to directly deliver the data back to their host tasks without any processing. Delay time can easily be captured by measuring the time needed for a specific amount of data to pass through a configured number of tasks. Considering the fact that the primary overheads incurred by these tasks are inherent in any task for processing incoming

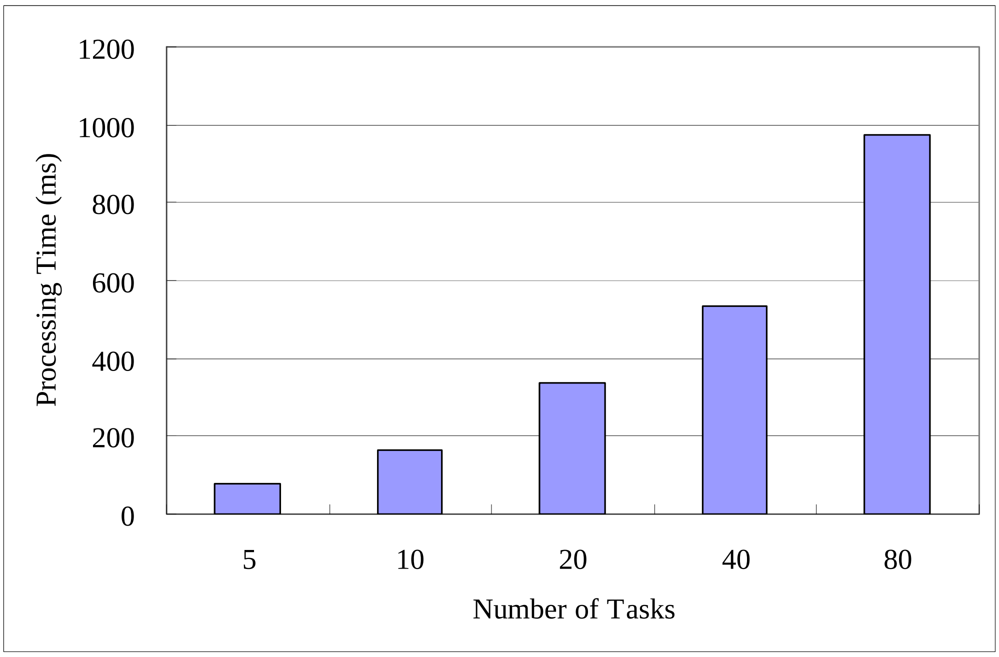


Figure 9-2 The Data Transfer Overhead.

data, it is argued that the setup of the experiment is reasonable and realistic. The results of the experiment are shown in Figure 9-2. They show that the delay overhead increases linearly with the increase in the number of tasks through which the data passes. On average, the overhead is about 15 ms per task. It is believed that the overhead can be further reduced with improved hardware configuration. Furthermore, in the realistic deployment of a service, it is unlikely that more than ten task will be used. That is to say, the overhead brought by these tasks can safely be bound to about 150 ms, which is acceptable compared with the potentially long delays incurred in network transmissions.

9.3 Adaptation Time

This set of experiments was designed to measure the processing overheads incurred by updating a task to adapt to contextual changes. The adaptation process of a task brings a certain number of performance penalties that are unavoidable. Adaptation time is the time taken for the CAMPUS system to update the tasklet part of a task in order to adapt to contextual changes. In other words, adaptation time is the amount of time during which a user will find the CAMPUS system inactive due to adaptation. Before going into the details of the experiment, the steps of the adaptation process are restated below:

1. The existing tasklet part of the task is deactivated.
2. The task base part of the task is suspended.
3. The existing tasklet part of the task is unplugged and its state is migrated to the new tasklet, if necessary.
4. The new tasklet is activated.
5. The new tasklet is plugged into the task.
6. The task base of the task is resumed and the adaptation is finished!

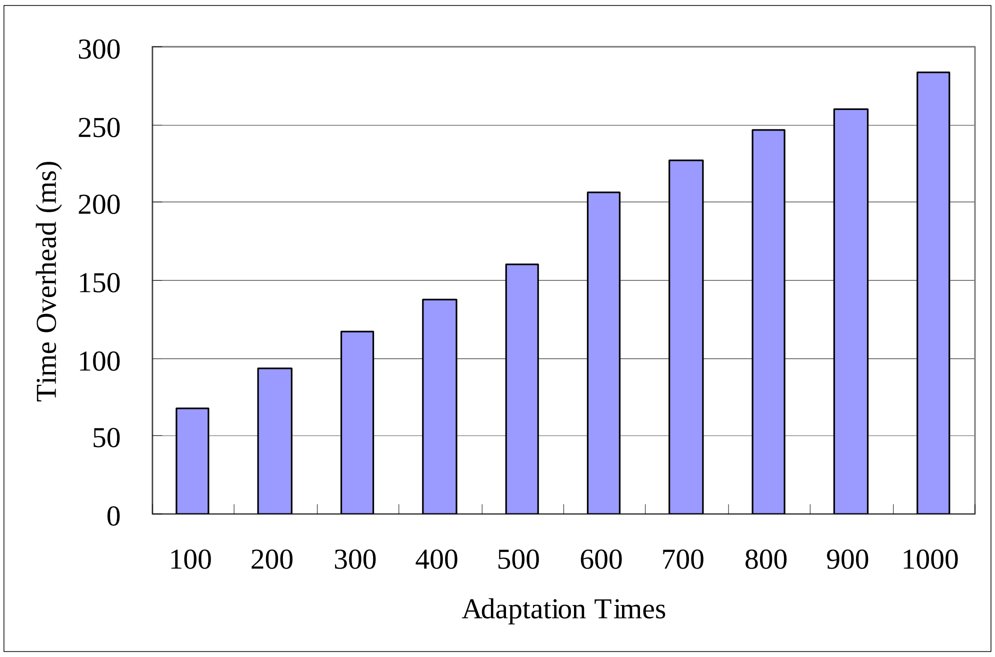


Figure 9-3 The Adaptation Time.

The experiment reused the service designed in section 9.2, although in this experiment the service was configured to repeatedly update a particular task. The time *Ts* is recorded once at the beginning of the adaptation process. Immediately after the adaptation process as detailed previously, the time *Te* is recorded as the ending time of the adaptation process. By varying the number of times that the adaptation is repeated, referred to as *N* here, different numbers of adaptation processes can be measured and $N \times (Te\text{-}Ts)$ will be the resultant time cost. Figure 9-3 shows the result of the experiment. Notice that when the number of adaptation times is less than 100, the adaptation time is less than 70ms. Even when the number of adaptation times reaches 1,000, the adaptation overhead is still less than 300ms. This is a noteworthy and promising result considering that the adaptation rate is likely to be comparatively low (typically ranging from tens of seconds to minutes, depending on the contextual changes of the environment) and the adaptation time is insignificant. A good adaptation performance is the result of extensive use of multi-threading and of the separation of adaptation from computational concerns to accelerate and support ease of adaptation.

9.4 CAMPUS End-to-End Performance

After evaluating the overheads of key CAMPUS mechanisms, this section describes the overall system performance of CAMPUS from an end-to-end perspective. In particular, we aim to verify the benefits of the CAMPUS system by asserting that the operation overhead is small compared to the improvement in performance that comes from using this system.

For this purpose, we reused the video chat service presented in Chapter 8, which reacts to changes in bandwidth to maintain a smoother performance for the video chat service, in terms of frame rates. In the experiment, we measured the frame rates on the receiver side under different bandwidths. Figure 9-4 contrasts the frame rate on the receiver side using CAMPUS with the frame rate when not using CAMPUS. By not using CAMPUS, we mean that the highest video rate is sent according to its transmission plan throughout the duration of the chat session, regardless of network conditions. From the experimental results, we can see that without using CAMPUS,

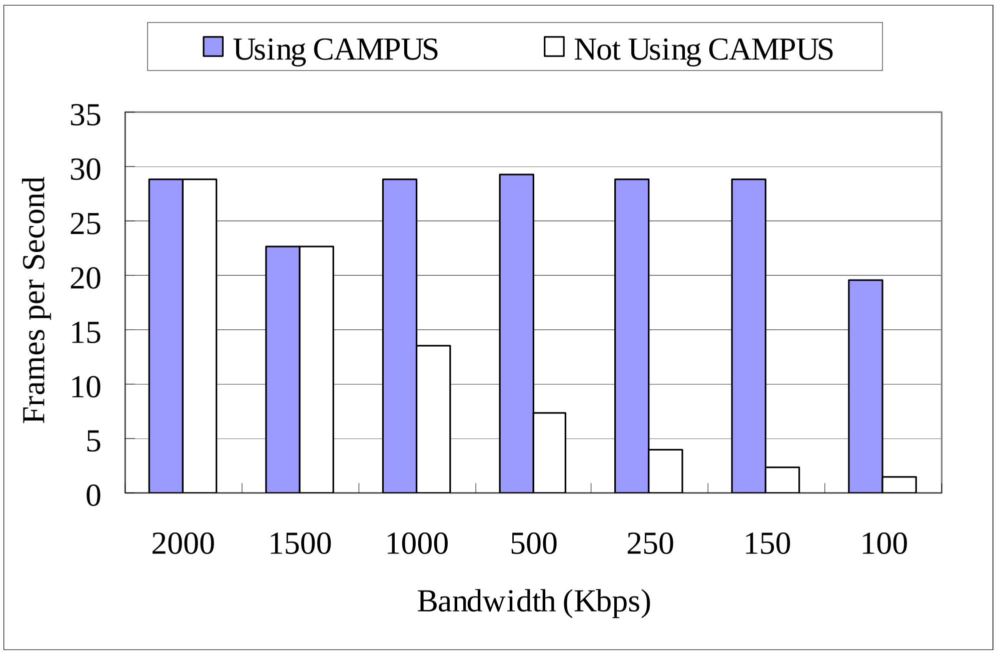


Figure 9-4 The Effectiveness of the CAMPUS System.

when the bandwidth decreases from 2 Mbps (which represents a good network condition for video chat) to 100 Kbps (which represents a bad network condition for video chat), the frame rate drops sharply from an average of 28.8 fps to an average of 1.5 fps. In contrast, with CAMPUS, a significant decrease in frame rate is avoided and the frame rate is always in the range of 19.5 to 28.8 fps. This is because when using CAMPUS, the chat service can automatically adapt to the changes of bandwidth conditions and use appropriate transcoders to reduce the data size of each frame at the sender side, and subsequently reduce the total amount of information transmitted over the network. While compromising the video quality of the receiver side, the chat service maintains a highly smooth performance, which is most important in a scenario of real-time video chatting. In contrast, without using CAMPUS, the chat service did not consider the actual network condition, always using a high video rate to transcode the data. This means that a large amount of information needs to be sent over the network. Therefore, when the bandwidth decreases, the frame rate drops sharply. The experiments clearly suggest the advantages of the CAMPUS system and its ability to offset the processing overheads that may be incurred in deploying applications in dynamic environments.

Chapter 10 Conclusions and Future Work

10.1 Conclusions

As pervasive computing continues to evolve, the need to embed context awareness to future mobile applications becomes more apparent. In contrast to virtual reality, which places physical objects in the virtual world, context-aware computing aims to place computing to work in the physical world. An important goal of the work presented in this thesis has been the development of a context-aware programming abstractions and a paradigm that make it easier to develop and execute context-aware applications in pervasive computing environments. Our work has resulted in CAMPUS, a comprehensive semantic-based context-aware middleware for pervasive computing that can automatically derive context-aware adaptation decisions at run-time. Through automated decisions at run-time, developers will be freed from the need to predict, formulate, and maintain adaptation rules, so that the effort involved in developing context-aware applications will be greatly reduced. It will also be possible to deliver services of an optimal quality by deferring the adaptation decisions until run-time to account for up-to-date contextual conditions.

More notably, we have developed CAMPUS based on a layered architecture that promotes a structured design, with each layer providing a well-defined level of abstraction and role. Importantly, each layer represents an abstraction boundary that provides software engineering solutions that contribute to the overall integration of context-awareness to the system.

- In the programming layer, we have proposed and designed a new programming model called ATM to facilitate the construction of context-aware applications. The ATM model relies on two main concepts: *services* and *tasks*. A service is an abstract of a business or a technical process, which is comprised of a series of tasks. Tasks are execution units that perform certain actions to deliver a result to other tasks or the end user. A task is further divided into two parts: a *tasklet* and a *task base*. The former part concentrates on the computational concerns of the task, i.e. how to process data; and the latter part handles issues involved in coordination and adaptation, such as how to communicate with other modules of the service and how to transfer the states of tasks during adaptation. The ATM model greatly separates context-aware adaptation from the functional concerns of applications, simplifies the work of maintaining the consistency of the data and migrating states when adaptation occurs, and eases the tasks of specifying and verifying adaptation.

- In the knowledge layer, a comprehensive ontological model has been developed to capture important knowledge about context-aware applications that have been built on the basis of the ATM model. The proposed ontological model has been split into two dimensions, in order to make the model more extensible and scalable, and also to help improve the performance of the system. The first dimension is the dimension of domain. Here, the ontologies are divided into *context ontologies*, *tasklet ontologies*, and *service ontologies*. Additionally, from the dimension of range, the CAMPUS ontologies are separated into foundation ontologies and domain-specific ontologies. The foundation ontologies model common objects that are generally applicable across a wide range of domain-specific ontologies. A domain-specific ontology models a particular domain and represents the particular meanings of terms as they apply to that domain.

- In the decision layer, we then investigated the technologies of semantic reasoning to automatically derive context-aware adaptation decisions at run-time. In particular, the decision layer performs DL reasoning and FOL reasoning on the proposed ontologies to derive the important decisions at run-time. The whole decision-making procedure of CAMPUS is divided into three phases: preprocessing, screening, and choice. In the preprocessing phase, several preprocessing tasks are performed to ensure that the ontologies are semantically consistent and to prepare fine-grained information for the following phases. For example, qualified tasklets are registered as alternatives for each task. In the second stage, tasklet alternatives registered in the first phase that are not satisfied by the up-to-date contextual information are screened out. If more than one acceptable tasklet alternative survives the screening phases, in the choice phase, the best alternative from among the survivors is selected using the expected utility function.

The CAMPUS implementation was evaluated with a number of case studies to validate the operation of the system on a realistic environment and to provide us with the opportunity to obtain experimental results for further analysis. In particular, we selected and implemented a context-aware instance messenger application to run over the CAMPUS. In capturing the system's performance, we evaluated the potential overheads introduced by deferring the adaptation decision to run-time in the middleware level. The results are significant in that CAMPUS can be adapted to run on resource-constraint portable devices, without significant degradation in its performance.

In short, we can conclude that the main contributions of this thesis are as follows:

- A new programming model was proposed and designed that can facilitate the development of context-aware applications in order to automate context-aware adaptation decisions.
- A comprehensive ontological model was defined to capture the knowledge and semantics of the entities involved during the process of making context-aware adaptation decisions. This ontological model effectively supports automated context-aware adaptation decisions.
- A middleware, CAMPUS, which provides an integrated solution to automate context-aware adaptation decisions at run-time, was designed and implemented.

CAMPUS was evaluated using a number of case studies to validate the operation of the system on a realistic environment.

10.2 Future Work

The work presented in this thesis establishes a theoretical foundation of middleware primitives for enhancing the development and execution of context-aware applications in a pervasive computing environment. As in most research work, the progress made in this study undoubtedly has not covered all interesting directions, but suggestions for future work to further improve CAMPUS are given below.

10.2.1 Collaborative Decision-Making

One important direction to make CAMPUS more complete and powerful is to enable collaborative decision-making among multiple CAMPUS middleware instances over the network. That is, when a task asks a CAMPUS middleware for a suitable tasklet, the CAMPUS system can cooperate with other CAMPUS systems to collaboratively make the final decision on which tasklet is the most suitable for the task. Collaborative decision-making is a natural extension of CAMPUS, which aims to integrate the capabilities of multiple CAMPUS instances to find potentially better alternatives than local alternatives and consequently to improve the decision outcome. In addition, collaborative decision-making among multiple CAMPUS instances greatly improves the reusability of tasklets. In order to enable collaborative decision-making, a collaboration protocol between CAMPUS systems is necessary. The protocol needs to consider such issues as the discovery of a potential collaborator CAMPUS, and communication and negotiation between them. In addition, mobile code techniques can be used to download remote tasklets from collaborator CAMPUS systems.

10.2.2 Semantic-based Service Composition

Another direction to improve CAMPUS is to enable the composition of CAMPUS services. Unlike tasks, a service is not composable in the current CAMPUS system. The current implementation of CAMPUS assumes that the services provided by a context-aware application are always in a loose relationship, so that direct primitives have not been offered to enable the composition of CAMPUS services. In the future, it will be possible to extend services by making them composable. Composable services make it possible to reuse existing services to develop more complex services, and therefore further facilitate the development of context-aware applications. Web service composition is a very active area of research and development [Benatallah02, Zeng04], and the composition of CAMPUS services will benefit from these existing works. In order to make CAMPUS services composable, CAMPUS needs to be extended in all three layers:

- In the programming layer, the spare input and output ports of the tasks that compose a service can be utilized as the input and output ports of the service in order to connect to other services.
- In the knowledge layer, the service foundation ontology needs to capture more knowledge related to the composition of services; for example, the overall functionality and cost of a service.
- In the decision layer, the inference engine needs to consider how to decide whether two services are compatible and to choose among a set of alternatives. The concrete decision strategies will depend on the knowledge captured by the knowledge layer. However, a general approach is to compare the overall functionalities of a service and its potential cost.

10.2.3 User Preference Model

Another possible extension of CAMPUS is to introduce a comprehensive user preference model. Although CAMPUS advocates automated decision-making by the middleware layer, a suitable user preference model helps to improve the quality of the decision. There are two general approaches to designing a user preference model: filtering and rating. Filtering models like [Mooney00] allow users to choose keywords that describe their preference, and decisions are made based on the matching of these keywords. Rating models like [Candillier07] allow users to rate the alternatives and decisions are made based on the ranking of these ratings.

10.2.4 Security and Power Saving Issues

Security has not been our focus in the current CAMPUS system. There are several issues that must be addressed in this area. First, authentication of the tasklets must be guaranteed. One approach is to digitally sign the tasklets to assure their authenticity. Second, the CAMPUS system should exploit the Java security features [Garms01], so that the CAMPUS platform presents a protected environment for a tasklet.

For small portable devices, one of the major concerns is power consumption. However, as the current version of CAMPUS system runs only on the Java2 standard Edition (J2SE), which runs only on standard PCs and workstation machines, we have been unable to obtain any data on power consumption from the current experimental setup. It is desirable to migrate the CAMPUS system to the Java2 Micro Edition (J2ME), which allows the CAMPUS system to run on PDAs and Java-enabled smart phones. In the future, experiments can be carried out to measure the factors that affect the power consumption of these mobile devices.

References

[Abowd97] G. D. Abowd, C. G. Atkerson, J. Hong, S. Long, R. Kooper, and M. Pinkerton, “Cyberguide: A Mobile Context-Aware Tour Guide,” *Wireless Networks,* vol. 3, no. 5, pp. 421-433, Oct. 1997.

[Alia07] M. Alia, V. S. W. Eide, N. Paspallis, F. Eliassen, S. O. Hallsteinsen, and G. A. Papadopoulos, "A Utility-based Adaptivity Model for Mobile Applications," *Proceedings of 21st International Conference on Advanced Information Networking and Applications Workshops 2007 (AINAW'07)*, vol. 2, pp. 556-563, May 2007.

[Beach90] L. R. Beach, and T. R. Mitchell, "Image Theory: A Behavioral Theory of Decision Making in Organizations," *B. M. Staw & L. L. Cummings (Eds.), Research in Organizational Behavior*, Vol. 12, pp. 1-41, Greenwich: JAI Press, 1990.

[Benatallah02] B. Benatallah, and F. Casati, *Distributed and Parallel Database, Sepcial issue on Web Services*, Kluwer Academic Publishers, 2002.

[Berners-Lee01] T. Berners-Lee, J. Hendler, O. Lassila, “The Semantic Web,” *Scientific American Magazine*, vol. 284, no. 5, pp. 34-43, 2001.

[Binns96] P. Binns, M. Engelhart, M. Jackson, and S. Vestal, “Domain-Specific Software Architecture for Guidance, Navigation, and Control,” *International Journal on Software Engineering and Knowledge Engineering*, vol. 6, no. 2, 1996.

[Biyani05] K. N. Biyani, and S. S. Kulkarni, “Building Component Families to Support Adaptation,” *Proceedings of the 2005 workshop on Design and evolution of autonomic application software (DEAS 2005)*, St. Louis, Missouri, USA, May 2005.

[Blair00] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran, N. Parlavantzas, and K. B. Saikoski, "A Principled Approached to Supporting Adaptation in Distributed Mobile Environments," *Proceedings of International*

Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), pp. 3-12, IEEE Computer Society, Limerick, Ireland. 2000.

[Candillier07] L. Candillier, F. Meyer, and M. Boulle, "Comparing State-of-the-Art Collaborative Filtering Systems," *Journal of Maching Learning and Data Mining in Pattern Recognition, Springer Berlin*, vol. 4571, pp. 548-562, 2007.

[Capra03] L. Capra, W. Emmerich, and C. Mascolo, "CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 929-944, Oct. 2003.

[Chan03] A. T. S. Chan, and S. N. Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing," *IEEE Transactions on Software Engineering*, vol. 29, no. 12, pp. 1072-1085, Dec. 2003.

[Chang05] S. F. Chang, and A. Vetro, "Video Adaptation: Concepts, Technologies, and Open Issues," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 148-158, Jan. 2005.

[Chen04] H. Chen, T. Finin, A. Joshi, F. Perich, D. Chakraborty, and L. Kagal, "Intellegient Agents Meets the Semantic Web in Smart Spaces," *IEEE Internet Computing*, vol. 8, no. 6, pp. 69-79, 2004.

[Cheverst00] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou, "Developing A Context-Aware Electronic Tourist Guide: Some Issues and Experiences," *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 17-24, Apr. 2000.

[Ciocoiu00] M. Ciocoiu, and D. S. Nau, "Ontology-Based Semantics," *Proceedings of the 7th International Conference on the Principles of Knowledge Representation and Reasoning*, pp. 539-548, 2000.

[Davis04] J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, "System Support for Pervasive Applications," *ACM Transactions on Computer Systems*, vol. 22, no. 4, pp. 421-486, Nov. 2004.

[Dey00] A. K. Dey, and G. D. Abowd, “Towards a Better Understanding of Context and Context-Awareness,” *Proceedings of the CHI 2000 Workshop on the What, Who, Where, When, and How of Context-Awareness*, 2000.

[Dowling01] J. Dowling, and V. Cahill, “The K-Component Architecture Meta-Model for Self-Adaptive Software,” *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Sep. 2001.

[Edwin07] Edwin J. Y. Wei, and Alvin T. S. Chan, “Towards Context-Awareness in Ubiquitous Computing,” *Proceedings of the International Conference on Embedded and Ubiquitous Computing (EUC 2007),* LNCS 4808, pp. 706-717, Taipei, Taiwan, Dec. 2007.

[Flatt98] M. Flatt, S. Krishnamurthi, and M. Felleisen, "Classes and Mixins," *ACM Symposium on Principles of Programming Languages (PoPL 98),* 1998.

[Floch06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven, “Using Architecure Models for Runtime Adaptability,” *IEEE Software*, 23(2):62-70, 2006.

[Fischhoff83] B. Fischhoff, B. Goitein, and Z. Shapira, "Subjective Expected Utility: A Model of Decision-Making," *R. W. Scholz (Ed.) Decision Making Under Unvertainty: Cognitive Decision Research, Social Interaction, Development and Epistemology*, vol. 16, pp. 183-207, Amsterdam, Elsevier, 1983.

[Forgy82] C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

[Friday96] A. Friday, N. Davies, G. Blair, and K. Cheverst, "Developing Adaptive Applications: The MOST Experience," *Jounal of Integrated Computer-Aided Engineering*, vol. 6, no. 2, pp. 143-157, 1996.

[Friedman-Hill03] E. Friedman-Hill, *Jess in Action: Java Rule-based Systems*, Manning Publications Company, June 2003, ISBN 1930110898, http://www.jessrules.com/.

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2, 1995.

[Garms01] J. Garms, "*Professional Java Security*," Birmingham, Wrox Press, 2001.

[Garlan94] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments," *Proceedings of SIGSOFT '94: Foundations of Software Engineering*, PP. 175-188, Dec. 1994.

[Gordon00] A. Gordon, *The COM and COM+ Programming Primer*, Microsoft Technology Series, Prentice Hall PTR, New-Jersey, 2000.

[Gruber93] T. R. Gruber, "A translation approach to portable ontologies," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199-220, 1993.

[Horrocks02] I. Horrocks, and J. Hendler, "The Semantic Web," *Proceedings of the First International Semantic Web Conference (ISWC 2002),* Sardinia, Italy, 2002.

[Horrocks04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," Available from http://www.w3.org/Submission/SWRL/, May 2004.

[Kitchenham99] B. A. Kitchenham, G. H. Travassos, A. VonMayrhauser, F. Niessink, N. F. Schniedewind, J. Singer, S. Takado, R. Vehvilainen, and H. Yang, "Towards an Ontology of Software Maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 6, pp. 365-389, 1999.

[Knublauch04] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen, "The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications," *Proceedings of the 3rd International Semantic Web Conference 2004 (ISWC'04),* LNCS 3298, pp. 229-243, Nov. 2004.

[Lenat90] D. B. Lenat, and R. V. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*, Addison-Wesley, Reading, Mass., 1990.

[Lott05] S. F. Lott, *Building Skills in Python - A Programmer's Introduction to Python*, Available in http://www.linuxtopia.org/online_books/programming_books/python_programming/index.html, 2005.

[Ma06] H. Ma, I. L. Yen, J. Zhou, and K. Cooper, “Qos Analysis for Component-based Embedded Software: Model and Methodology,” *The Journal of System and Software*, 79(6):859-870, 2006.

[McKinley04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, “Composing Adaptive Software,” *IEEE Computer*, vol. 37, no. 7, pp. 56-64, Jul. 2004.

[Medvidovic99] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, “A Language and Environment for Architecture-Based Software Development and Evolution,” *Proceedings of the 21st International Conference on Software Engineering (ICSE 99),* pp. 44-53, May 1999.

[Mooney00] R. J. Mooney, and L. Roy, "Content-based Book Recommending Using Learning for Text Categorization," *Proceedings of the 5th ACM Conference on Digital Libraries*, pp. 195-204, ACM New York, 2000.

[Neto05] R. F. B. Neto, and M. G. C. Pimentel, “Toward a Domain-Independent Semantic Model for Context-Aware Computing,” *Proceedings of the 3rd Latin American Web Congress (LA-WEB’05),* 2005.

[Noble97] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, “Agile Application-Aware Adaptation for Mobility,” *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pp. 276-287, 1997.

[Rouvoy08] R. Rouvoy, M. Beauvois, L. Lozano, J. Lorenzo, and F. Eliassen, “MUSIC: an Autonomous Platform Supporting Self-Adaptive Mobile Applications,” *Proceedings of the 1st Workshop on Mobile Middleware: Embracing the Personal Communication Device*, Dec. 2008, Leuven, Belgium.

[Satyanarayanan04] M. Satyanarayanan, “The Many Faces of Adaptation,” *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 4-5, 2004.

[Schilit94] B.N. Schilit, N. Adams, and R. Want, “Context-Aware Computing Applications,” *Mobile Computing Systems and Applications*, pp. 85-90, Dec. 1994.

[Schmidt99] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde, “Advanced Interaction in Context,” *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, pp. 89-101, 1999.

[Shao06] W. Shao, A. Lye, S. Rundle-Thiele, and C. Fausnaugh, “Decision Theory: Poised for the New Millennium,” *Proceedings of the Australian and New Zealand Marketing Academy Annual Conference 2003 (ANZMAC 2003)*, Dec. 2003.

[Siewiorek03] D. Siewiorek, A. Smailagic, J. Furukawa, A. Krause, N. Moraveji, K. Reiger, J. Shaffer, and F. L. Wong, “SenSay: A Context-Aware Mobile Phone,” *Proceedings of the 7th IEEE International Symposium on Wearable Computers (ISWC’03)*, 2003.

[Sirin07] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reason," *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, issue 2, pp. 51-53, June 2007.

[Sora07] I. Sora, P. Verbaeten, and Y. Berbers, “CCDL: the Composal Components Description Language,” *Interntional Jornal on Software Tools for Technology Transfer*, vol. 9, no. 2, pp. 155-168, 2007.

[Spivey89] J. M. Spivey, *The Z Notation, A Reference Manual*, Englewood Cliffs, NJ : Prentice-Hall, 1989.

[Strang03] T. Strang, C. Linnhoff-Popien, and K. Frank, “CoOL: A Context Ontology Language to Enable Contextual Interoperability,” *Proceedings of the 2003 IFIP International Federation for Information Processing*, 2003.

[Tekinerdogan96] B. Tekinerdogan, and M. Aksit, “Adaptability in Object-Oriented Software Development Workshop Report,” *Proceedings of the 10th Annual*

European Conference on Object-Oriented Programming (ECOOP), Linz, Austria, Jul. 1996.

[Walsh04] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, “Utility Functions in Autonomic Systems,” *Proceedings of the International Conference on Autonomic Computing (ICAC’04)*, pp. 70-77, May 2004.

[Wang04] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, “Ontology Based Context Modeling and Reasoning using OWL,” *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops (PERCOMW’04)*, 2004.

[Warmer98] J. B. Warmer, and A. G. Kleppe, *The Object Constraint Language – Precise Modeling with UML*, Addison-Wesley Publishing Company, 1998.

[Want92] R. Want, A. Hopper, V. Falcao, J. Gibbons, “The Active Badge Location System,” *ACM Transactions on Information Systems*, vol. 10, no. 1, pp. 91-102, 1992.

[Yau02] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta, “Reconfigurable Context-Sensitive Middleware for Pervasive Computing,” *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 33-40, 2002.

[Zandy02] V. C. Zandy, and B. P. Miller, "Reliable Network Connections," *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking (MobiCom'02)*, pp. 95-106, Atlanta, Georgia, USA, 2002.

[Zeng04] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, “QoS-aware Middleware for Web Services Composition,” IEEE Transactions on Software Engineering, vol. 30, no. 5, pp.311-327, May 2004.

[Zheng06] Y. J. Zheng, and A. T. S. Chan, “MobiGATE: A Mobile Computing Middleware for the Active Deployment of Transport Services,” *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 35-50, Jan. 2006.

Appendix A The Context Foundation Ontology

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<rdf:RDF xmlns="http://campus.comp.polyu.edu.hk/context-foundation.owl#"
    xml:base="http://campus.comp.polyu.edu.hk/context-foundation.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <owl:Ontology rdf:about=""/>
   <owl:Class rdf:ID="Activity">
      <rdfs:subClassOf rdf:resource="#UserEntity"/>
   </owl:Class>
   <owl:Class rdf:ID="Agent">
      <rdfs:subClassOf rdf:resource="#UserEntity"/>
   </owl:Class>
   <BaseUnit rdf:ID="ampere"/>
   <owl:ObjectProperty rdf:ID="availableBandwidth">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#Network"/>
      <rdfs:range rdf:resource="#PhysicalQuantity"/>
   </owl:ObjectProperty>
   <owl:Class rdf:ID="BaseUnit">
      <rdfs:subClassOf rdf:resource="#Unit"/>
   </owl:Class>
   <owl:Class rdf:ID="BinaryPrefix">
      <rdfs:subClassOf rdf:resource="#Prefix"/>
   </owl:Class>
   <BaseUnit rdf:ID="bit"/>
   <BaseUnit rdf:ID="byte"/>
   <BaseUnit rdf:ID="candela"/>
   <SIPrefix rdf:ID="centi">
      <hasExponent rdf:datatype="&xsd;int">-2</hasExponent>
   </SIPrefix>
   <owl:Class rdf:ID="ComputingEntity">
      <rdfs:subClassOf rdf:resource="#ContextEntity"/>
   </owl:Class>
   <owl:Class rdf:ID="ContextEntity"/>
   <owl:Class rdf:ID="CPU">
      <rdfs:subClassOf rdf:resource="#Hardward"/>
   </owl:Class>
   <SIPrefix rdf:ID="deci">
      <hasExponent rdf:datatype="&xsd;int">-1</hasExponent>
   </SIPrefix>
   <SIPrefix rdf:ID="deka">
```

```
<hasExponent rdf:datatype="&xsd;int">1</hasExponent>
        <hasValue rdf:datatype="&xsd;double">10</hasValue>
    </SIPrefix>
    <owl:ObjectProperty rdf:ID="derivedFrom">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#UnitDerivedByPrefixing"/>
                    <owl:Class rdf:about="#UnitDerivedByRaising"/>
                    <owl:Class rdf:about="#UnitDerivedByScaling"/>
                    <owl:Class rdf:about="#UnitDerivedByShifting"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
        <rdfs:range rdf:resource="#Unit"/>
    </owl:ObjectProperty>
    <owl:Class rdf:ID="DerivedUnit">
        <rdfs:subClassOf rdf:resource="#Unit"/>
    </owl:Class>
    <BinaryPrefix rdf:ID="exbi">
        <hasExponent rdf:datatype="&xsd;int">60</hasExponent>
    </BinaryPrefix>
    <owl:Class rdf:ID="ExternalStorage">
        <rdfs:subClassOf rdf:resource="#Hardward"/>
    </owl:Class>
    <owl:Class rdf:ID="FlashDrive">
        <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
    </owl:Class>
    <owl:Class rdf:ID="FlashMemory">
        <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
    </owl:Class>
    <owl:Class rdf:ID="FloppyDisk">
        <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
    </owl:Class>
    <BinaryPrefix rdf:ID="gibi">
        <hasExponent rdf:datatype="&xsd;int">30</hasExponent>
    </BinaryPrefix>
    <SIPrefix rdf:ID="giga">
        <hasExponent rdf:datatype="&xsd;int">9</hasExponent>
        <hasValue rdf:datatype="&xsd;double">1000000000</hasValue>
    </SIPrefix>
    <owl:Class rdf:ID="HardDisk">
        <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
    </owl:Class>
    <owl:Class rdf:ID="Hardward">
        <rdfs:subClassOf rdf:resource="#ComputingEntity"/>
    </owl:Class>
    <owl:ObjectProperty rdf:ID="hasCapacity">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#ExternalStorage"/>
            <owl:Class rdf:about="#RAM"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:domain>
   <rdfs:range rdf:resource="#PhysicalQuantity"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="hasExponent">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Prefix"/>
            <owl:Class rdf:about="#UnitDerivedByRaising"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:domain>
   <rdfs:range rdf:resource="&xsd;int"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="hasPrefix">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain rdf:resource="#UnitDerivedByPrefixing"/>
   <rdfs:range rdf:resource="#Prefix"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="hasScalingNumber">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain rdf:resource="#UnitDerivedByScaling"/>
   <rdfs:range rdf:resource="&xsd;double"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasShiftingNumber">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain rdf:resource="#UnitDerivedByShifting"/>
   <rdfs:range rdf:resource="&xsd;double"/>
</owl:DatatypeProperty>
<owl:ObjectProperty rdf:ID="hasUnit">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain rdf:resource="#PhysicalQuantity"/>
   <rdfs:range rdf:resource="#Unit"/>
</owl:ObjectProperty>
<owl:DatatypeProperty rdf:ID="hasValue">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain rdf:resource="#PhysicalQuantity"/>
   <rdfs:range rdf:resource="&xsd;double"/>
</owl:DatatypeProperty>
<SIPrefix rdf:ID="hecto">
   <hasExponent rdf:datatype="&xsd;int">2</hasExponent>
   <hasValue rdf:datatype="&xsd;double">100</hasValue>
</SIPrefix>
<owl:DatatypeProperty rdf:ID="isAvailable">
   <rdf:type rdf:resource="&owl;FunctionalProperty"/>
   <rdfs:domain>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Hardward"/>
            <owl:Class rdf:about="#Network"/>
            </owl:unionOf>
          </owl:Class>
        </rdfs:domain>
        <rdfs:range rdf:resource="&xsd;boolean"/>
    </owl:DatatypeProperty>
    <BaseUnit rdf:ID="kelvin"/>
    <owl:Class rdf:ID="Keyboard">
        <rdfs:subClassOf rdf:resource="#Hardward"/>
    </owl:Class>
    <BinaryPrefix rdf:ID="kibi">
        <hasExponent rdf:datatype="&xsd;int">10</hasExponent>
    </BinaryPrefix>
    <SIPrefix rdf:ID="kilo">
        <hasExponent rdf:datatype="&xsd;int">3</hasExponent>
        <hasValue rdf:datatype="&xsd;double">1000</hasValue>
    </SIPrefix>
    <BaseUnit rdf:ID="kilogram"/>
    <owl:Class rdf:ID="Light">
        <rdfs:subClassOf rdf:resource="#PhysicalEntity"/>
    </owl:Class>
    <owl:Class rdf:ID="Location">
        <rdfs:subClassOf rdf:resource="#UserEntity"/>
    </owl:Class>
    <BinaryPrefix rdf:ID="mebi">
        <hasExponent rdf:datatype="&xsd;int">20</hasExponent>
    </BinaryPrefix>
    <SIPrefix rdf:ID="mega">
        <hasExponent rdf:datatype="&xsd;int">6</hasExponent>
        <hasValue rdf:datatype="&xsd;double">1000000</hasValue>
    </SIPrefix>
    <owl:Class rdf:ID="MemoryCard">
        <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
    </owl:Class>
    <BaseUnit rdf:ID="meter"/>
    <SIPrefix rdf:ID="micro">
        <hasExponent rdf:datatype="&xsd;int">-6</hasExponent>
        <hasValue rdf:datatype="&xsd;double">0.000001</hasValue>
    </SIPrefix>
    <owl:Class rdf:ID="Microphone">
        <rdfs:subClassOf rdf:resource="#Hardward"/>
    </owl:Class>
    <SIPrefix rdf:ID="milli">
        <hasExponent rdf:datatype="&xsd;int">-3</hasExponent>
        <hasValue rdf:datatype="&xsd;double">0.001</hasValue>
    </SIPrefix>
    <BaseUnit rdf:ID="mole"/>
    <owl:Class rdf:ID="Mouse">
        <rdfs:subClassOf rdf:resource="#Hardward"/>
    </owl:Class>
    <SIPrefix rdf:ID="nano">
        <hasExponent rdf:datatype="&xsd;int">-9</hasExponent>
        <hasValue rdf:datatype="&xsd;double">0.000000001</hasValue>
    </SIPrefix>
    <owl:Class rdf:ID="Network">
```

```
<rdfs:subClassOf rdf:resource="#ComputingEntity"/>
  </owl:Class>
  <owl:Class rdf:ID="Noise">
    <rdfs:subClassOf rdf:resource="#PhysicalEntity"/>
  </owl:Class>
  <owl:Class rdf:ID="OS">
    <rdfs:subClassOf rdf:resource="#Software"/>
  </owl:Class>
  <BinaryPrefix rdf:ID="pebi">
    <hasExponent rdf:datatype="&xsd;int">50</hasExponent>
  </BinaryPrefix>
  <owl:Class rdf:ID="Person">
    <rdfs:subClassOf rdf:resource="#UserEntity"/>
  </owl:Class>
  <owl:Class rdf:ID="PhysicalEntity">
    <rdfs:subClassOf rdf:resource="#ContextEntity"/>
  </owl:Class>
  <owl:Class rdf:ID="PhysicalQuantity"/>
  <owl:Class rdf:ID="Power">
    <rdfs:subClassOf rdf:resource="#Hardward"/>
  </owl:Class>
  <owl:Class rdf:ID="Prefix"/>
  <owl:ObjectProperty rdf:ID="productOf">
    <rdfs:domain rdf:resource="#UnitDerivedByMultiplying"/>
    <rdfs:range rdf:resource="#Unit"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="RAM">
    <rdfs:subClassOf rdf:resource="#Hardward"/>
  </owl:Class>
  <owl:Class rdf:ID="ROM">
    <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
  </owl:Class>
  <owl:Class rdf:ID="Screen">
    <rdfs:subClassOf rdf:resource="#Hardward"/>
  </owl:Class>
  <BaseUnit rdf:ID="second"/>
  <owl:Class rdf:ID="SIPrefix">
    <rdfs:subClassOf rdf:resource="#Prefix"/>
  </owl:Class>
  <owl:Class rdf:ID="Software">
    <rdfs:subClassOf rdf:resource="#ComputingEntity"/>
  </owl:Class>
  <owl:Class rdf:ID="Speaker">
    <rdfs:subClassOf rdf:resource="#Hardward"/>
  </owl:Class>
  <owl:Class rdf:ID="Tape">
    <rdfs:subClassOf rdf:resource="#ExternalStorage"/>
  </owl:Class>
  <BinaryPrefix rdf:ID="tebi">
    <hasExponent rdf:datatype="&xsd;int">40</hasExponent>
  </BinaryPrefix>
  <owl:Class rdf:ID="Temperature">
    <rdfs:subClassOf rdf:resource="#PhysicalEntity"/>
  </owl:Class>
```

```
<owl:Class rdf:ID="Time">
        <rdfs:subClassOf rdf:resource="#PhysicalEntity"/>
    </owl:Class>
    <owl:Class rdf:ID="Unit"/>
    <owl:Class rdf:ID="UnitDerivedByMultiplying">
        <rdfs:subClassOf rdf:resource="#DerivedUnit"/>
    </owl:Class>
    <owl:Class rdf:ID="UnitDerivedByPrefixing">
        <rdfs:subClassOf rdf:resource="#DerivedUnit"/>
    </owl:Class>
    <owl:Class rdf:ID="UnitDerivedByRaising">
        <rdfs:subClassOf rdf:resource="#DerivedUnit"/>
    </owl:Class>
    <owl:Class rdf:ID="UnitDerivedByScaling">
        <rdfs:subClassOf rdf:resource="#DerivedUnit"/>
    </owl:Class>
    <owl:Class rdf:ID="UnitDerivedByShifting">
        <rdfs:subClassOf rdf:resource="#DerivedUnit"/>
    </owl:Class>
    <owl:Class rdf:ID="UserEntity">
        <rdfs:subClassOf rdf:resource="#ContextEntity"/>
    </owl:Class>
    <owl:Class rdf:ID="WritingPad">
        <rdfs:subClassOf rdf:resource="#Hardward"/>
    </owl:Class>
</rdf:RDF>
```

Appendix B The Tasklet Foundation Ontology

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<rdf:RDF xmlns="http://campus.comp.polyu.edu.hk/tasklet-foundation.owl#"
    xml:base="http://campus.comp.polyu.edu.hk/tasklet-foundation.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:swrl="http://www.w3.org/2003/11/swrl#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
    xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <owl:Ontology rdf:about=""/>
   <owl:ObjectProperty rdf:ID="asserts">
      <rdfs:domain rdf:resource="#Tasklet"/>
      <rdfs:range rdf:resource="#ContextCondition"/>
   </owl:ObjectProperty>
   <owl:Class rdf:ID="ContextCondition">
      <owl:disjointWith rdf:resource="#DataType"/>
      <owl:disjointWith rdf:resource="#Function"/>
      <owl:disjointWith rdf:resource="#Tasklet"/>
      <owl:disjointWith rdf:resource="#Operator"/>
   </owl:Class>
   <owl:Class rdf:ID="DataType">
      <rdfs:subClassOf rdf:resource="&owl;Thing"/>
      <rdfs:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#extends"/>
            <owl:allValuesFrom rdf:resource="#DataType"/>
         </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#ContextCondition"/>
      <owl:disjointWith rdf:resource="#Function"/>
      <owl:disjointWith rdf:resource="#Tasklet"/>
      <owl:disjointWith rdf:resource="#Operator"/>
   </owl:Class>
   <owl:Class rdf:ID="DataTypeCondition">
      <rdfs:subClassOf rdf:resource="#ContextCondition"/>
      <rdfs:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#hasProperty"/>
            <owl:allValuesFrom rdf:resource="&owl;DatatypeProperty"/>
         </owl:Restriction>
```

```
</rdfs:subClassOf>
        <owl:disjointWith rdf:resource="#ObjectCondition"/>
    </owl:Class>
    <owl:ObjectProperty rdf:ID="dependsOn">
        <rdfs:domain rdf:resource="#Tasklet"/>
        <rdfs:range rdf:resource="#Tasklet"/>
    </owl:ObjectProperty>
    <Operator rdf:ID="equalTo"/>
    <owl:ObjectProperty rdf:ID="extends">
        <rdf:type rdf:resource="&owl;TransitiveProperty"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#DataType"/>
                    <owl:Class rdf:about="#Function"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
        <rdfs:range>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#DataType"/>
                    <owl:Class rdf:about="#Function"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:range>
    </owl:ObjectProperty>
    <owl:Class rdf:ID="Function">
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#extends"/>
                <owl:allValuesFrom rdf:resource="#Function"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <owl:disjointWith rdf:resource="#ContextCondition"/>
        <owl:disjointWith rdf:resource="#DataType"/>
        <owl:disjointWith rdf:resource="#Tasklet"/>
        <owl:disjointWith rdf:resource="#Operator"/>
    </owl:Class>
    <Operator rdf:ID="greaterThan"/>
    <Operator rdf:ID="greaterThanOrEqualTo"/>
    <owl:ObjectProperty rdf:ID="groupedWith">
        <rdf:type rdf:resource="&owl;TransitiveProperty"/>
        <rdfs:domain rdf:resource="#Tasklet"/>
        <rdfs:range rdf:resource="#Tasklet"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="hasEntity">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#ContextCondition"/>
        <rdfs:range rdf:resource="&owl;Class"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="hasInputDataTyppe">
        <rdfs:domain rdf:resource="#Tasklet"/>
        <rdfs:range rdf:resource="#DataType"/>
    </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="hasOperator">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#ContextCondition"/>
      <rdfs:range rdf:resource="#Operator"/>
   </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="hasOutputDataType">
      <rdfs:domain rdf:resource="#Tasklet"/>
      <rdfs:range rdf:resource="#DataType"/>
   </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="hasProperty">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#ContextCondition"/>
      <rdfs:range rdf:resource="&rdf;Property"/>
   </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="hasReferenceObject">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#ObjectCondition"/>
      <rdfs:range rdf:resource="&owl;Thing"/>
   </owl:ObjectProperty>
   <owl:DatatypeProperty rdf:ID="hasReferenceValue">
      <rdfs:domain rdf:resource="#DataTypeCondition"/>
   </owl:DatatypeProperty>
   <owl:DatatypeProperty rdf:ID="implementedBy">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain>
         <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
               <owl:Class rdf:about="#DataType"/>
               <owl:Class rdf:about="#Function"/>
               <owl:Class rdf:about="#Tasklet"/>
            </owl:unionOf>
         </owl:Class>
      </rdfs:domain>
      <rdfs:range rdf:resource="&xsd;string"/>
   </owl:DatatypeProperty>
   <Operator rdf:ID="lessThan"/>
   <Operator rdf:ID="lessThanOrEqualTo"/>
   <Operator rdf:ID="notEqualTo"/>
   <owl:Class rdf:ID="ObjectCondition">
      <rdfs:subClassOf rdf:resource="#ContextCondition"/>
      <rdfs:subClassOf>
         <owl:Restriction>
            <owl:onProperty rdf:resource="#hasProperty"/>
            <owl:allValuesFrom rdf:resource="&owl;ObjectProperty"/>
         </owl:Restriction>
      </rdfs:subClassOf>
      <owl:disjointWith rdf:resource="#DataTypeCondition"/>
   </owl:Class>
   <owl:Class rdf:ID="Operator">
      <owl:disjointWith rdf:resource="#ContextCondition"/>
      <owl:disjointWith rdf:resource="#DataType"/>
      <owl:disjointWith rdf:resource="#Function"/>
      <owl:disjointWith rdf:resource="#Tasklet"/>
   </owl:Class>
   <owl:ObjectProperty rdf:ID="provides">
      <rdfs:domain rdf:resource="#Tasklet"/>
```

```
<rdfs:range rdf:resource="#Function"/>
  </owl:ObjectProperty>
  <owl:Class rdf:ID="Tasklet">
    <owl:disjointWith rdf:resource="#ContextCondition"/>
    <owl:disjointWith rdf:resource="#DataType"/>
    <owl:disjointWith rdf:resource="#Function"/>
    <owl:disjointWith rdf:resource="#Operator"/>
  </owl:Class>
</rdf:RDF>
```

Appendix C The Service Foundation Ontology

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
   <!ENTITY tasklet "http://campus.comp.polyu.edu.hk/tasklet-foundation.owl#" >
]>
<rdf:RDF xmlns="http://campus.comp.polyu.edu.hk/service-foundation.owl#"
    xml:base="http://campus.comp.polyu.edu.hk/service-foundation.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
    xmlns:tasklet="http://campus.comp.polyu.edu.hk/tasklet-foundation.owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <owl:Ontology rdf:about="">
      <owl:imports rdf:resource="http://campus.comp.polyu.edu.hk/tasklet-foundation.owl"/>
   </owl:Ontology>
   <owl:ObjectProperty rdf:ID="accepts">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#Port"/>
      <rdfs:range rdf:resource="&tasklet;DataType"/>
   </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="belongsTo">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#Port"/>
      <rdfs:range rdf:resource="#Task"/>
      <owl:inverseOf rdf:resource="#owns"/>
   </owl:ObjectProperty>
   <owl:Class rdf:ID="Channel"/>
   <owl:ObjectProperty rdf:ID="connectsTo">
      <rdf:type rdf:resource="&owl;FunctionalProperty"/>
      <rdfs:domain rdf:resource="#Port"/>
      <rdfs:range rdf:resource="#Channel"/>
   </owl:ObjectProperty>
   <owl:ObjectProperty rdf:ID="consistsOf">
      <rdfs:domain rdf:resource="#Service"/>
      <rdfs:range rdf:resource="#Task"/>
   </owl:ObjectProperty>
   <owl:Class rdf:ID="EssentialTask">
      <rdfs:subClassOf rdf:resource="#Task"/>
      <owl:disjointWith rdf:resource="#ExpansionTask"/>
   </owl:Class>
   <owl:Class rdf:ID="ExpansionTask">
      <rdfs:subClassOf rdf:resource="#Task"/>
      <owl:disjointWith rdf:resource="#EssentialTask"/>
   </owl:Class>
<owl:ObjectProperty rdf:ID="hasSink">
```

```
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Channel"/>
        <rdfs:range rdf:resource="#InputPort"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="hasSrc">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Channel"/>
        <rdfs:range rdf:resource="#OutputPort"/>
    </owl:ObjectProperty>
    <owl:Class rdf:ID="InputPort">
        <rdfs:subClassOf rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#OutputPort"/>
    </owl:Class>
    <owl:Class rdf:ID="OutputPort">
        <rdfs:subClassOf rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#InputPort"/>
    </owl:Class>
    <owl:ObjectProperty rdf:ID="owns">
        <rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>
        <rdfs:domain rdf:resource="#Task"/>
        <rdfs:range rdf:resource="#Port"/>
        <owl:inverseOf rdf:resource="#belongsTo"/>
    </owl:ObjectProperty>
    <owl:Class rdf:ID="Port">
        <owl:disjointWith rdf:resource="&tasklet;ContextCondition"/>
        <owl:disjointWith rdf:resource="&tasklet;DataType"/>
        <owl:disjointWith rdf:resource="&tasklet;Function"/>
        <owl:disjointWith rdf:resource="&tasklet;Operator"/>
        <owl:disjointWith rdf:resource="&tasklet;Tasklet"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </owl:Class>
    <owl:ObjectProperty rdf:ID="requires">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Task"/>
        <rdfs:range rdf:resource="&tasklet;Function"/>
    </owl:ObjectProperty>
    <owl:Class rdf:ID="Service">
        <owl:disjointWith rdf:resource="&tasklet;ContextCondition"/>
        <owl:disjointWith rdf:resource="&tasklet;DataType"/>
        <owl:disjointWith rdf:resource="&tasklet;Function"/>
        <owl:disjointWith rdf:resource="&tasklet;Operator"/>
        <owl:disjointWith rdf:resource="&tasklet;Tasklet"/>
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </owl:Class>
    <owl:Class rdf:ID="Task">
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#owns"/>
                <owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality>
            </owl:Restriction>
        </rdfs:subClassOf>
        <owl:disjointWith rdf:resource="&tasklet;ContextCondition"/>
<owl:disjointWith rdf:resource="&tasklet;DataType"/>
```

```
        <owl:disjointWith rdf:resource="&tasklet;Function"/>
        <owl:disjointWith rdf:resource="&tasklet;Operator"/>
        <owl:disjointWith rdf:resource="&tasklet;Tasklet"/>
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
    </owl:Class>
    <rdf:Description rdf:about="&tasklet;ContextCondition">
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </rdf:Description>
    <rdf:Description rdf:about="&tasklet;DataType">
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </rdf:Description>
    <rdf:Description rdf:about="&tasklet;Function">
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </rdf:Description>
    <rdf:Description rdf:about="&tasklet;Operator">
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </rdf:Description>
    <rdf:Description rdf:about="&tasklet;Tasklet">
        <owl:disjointWith rdf:resource="#Port"/>
        <owl:disjointWith rdf:resource="#Service"/>
        <owl:disjointWith rdf:resource="#Task"/>
    </rdf:Description>
</rdf:RDF>
```