JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 25, 1021-1039 (2009)

An Object-Oriented Framework Approach to Flexible Availability Management for Developing Distributed Applications*

HEUNG SEOK CHAE[1], JIAN FENG CUI[1], JIN WOOK PARK[2], JAE GEOL PARK[3] AND WOO JIN LEE[4]
[1]*Department of Computer Science and Engineering*
Pusan National University
Busan, 609-735 South Korea
[2]*A-Shock Studio*
Neople Ltd.
Seoul, 135-873 South Korea
[3]*Busan Branch*
SEBANG Co., Ltd.
Busan, 608-803 South Korea
[4]*School of Electrical Engineering and Computer Science*
Kyungpook National University
Daegu, 702-701 South Korea

With the success of Internet-based business, there is more and more interest in the availability of systems. Availability indicates that a system continuously provides a useful service at any given time. Failure detection and recovery methods are supported for developing application of availability. Requirement on availability may vary with applications and in addition may change over time. For applications to be independent of the change of availability requirement there should be a flexible and extensible architecture for supporting availability management. This paper presents an availability management framework (AMF), which supports the flexible management of availability for large distributed systems using object-oriented framework technologies. We focus on the AMF's flexibility in order to accommodate changing availability requirements which vary with each application. In addition, this paper describes the result of application of the framework to support availability of RFID systems.

Keywords: flexible availability management, object-oriented framework, RFID middleware, failure detection, failure recovery

1. INTRODUCTION

As new technologies enable powerful new services, we should be confident that a system will operate as we expect and the system will not fail in normal use. Along with reliability, safety, and security, availability is one of the four principal dimensions to dependability of systems [1]. Informally, the availability of a system is the probability that the system will be up and running and able to deliver useful services at any given time.

Received September 20, 2007; revised April 30, 2008; accepted January 8, 2009.
Communicated by Jonathan Lee.
* This work was supported by the Korean Ministry of Education, Science and Technology grant (The Regional Core Research Program/Institute of Logistics Information Technology).

Computer systems vary widely in their tolerance of downtime. Some systems cannot handle even the briefest interruption in service without catastrophic results. On the contrary others can even handle extended outages while still delivering on their required returns on investment [2]. In other words, there is a whole range of possible availability levels that range from an absolute requirement of 100 percent down to a low level, where it just does not matter if the system is running or not.

Failure detection and recovery methods vary in their operational characteristics such as precision of detection, level of recovery, and performance overhead. Therefore, some appropriate failure detection and recovery methods should be applied in order to satisfy the required level of availability. In addition, as Lehman's laws suggest [3], the change on the level of availability is inevitable. At current customers are satisfied with the current level of availability. However, they may hope that the system will provide higher level of availability. In this case, the system must be changed for improving the availability by choosing more precise and more efficient detection and recovery methods. That is, the architecture of distributed systems should support the change of failure detection and recovery methods [4].

To address these problems, we propose an object-oriented framework based approach to supporting flexible and extensible management of availability for distributed applications. A framework is a partial, common design and implementation which can be extended to be reused for several specific applications [5]. Frameworks have been considered a promising technique for reusing the already developed software. This is supported by the evidence that there are many frameworks for various domains [6-10].

The proposed framework provides a flexible and extensible availability management by supporting various failure detection and recovery mechanisms that are specific to the systems using object-oriented framework technologies. The framework embeds failure detection and recovery code which are common to various applications and provides extension points by which the framework can be extended to accommodate the specific availability requirements of each application.

The rest of the paper is organized as follows: Section 2 makes a brief description of availability and framework-based approach to the availability. Section 3 presents the proposed framework for supporting availability for distributed systems. In section 4, we describe the results of a case study where the proposed framework is applied to an RFID middleware. The related work and conclusion are given in sections 5 and 6, respectively.

2. FRAMEWORK-BASED APPROACH TO AVAILABILTY

In contrast to earlier object-oriented reuse techniques based on class libraries, framework is targeted for particular business units and application domains. Frameworks like MacApp, Interviews, ACE, RMI, and implementation of OMG's CORBA play an increasingly important role in contemporary software development [9]. For example, Toshiya *et al.* presented an application framework system in [11], the B2.S framework, to facilitate the development efficiency of Web-based systems. The practical implementation suggests that there is an evident improvement on system's scale, person-power, and period of development, and the total number of bugs are reduced a lot compared with the development process without implementing framework. In order to avoid some misun-

derstanding of framework concepts, we follow the notion of framework concepts proposed in [5]: core framework design, framework internal increment, application-specific increment, and application.

- The *core framework design* describes the typical software architecture for applications in a given domain. The core framework design consists of both abstract and concrete classes in the domain. The concrete classes are intended to be invisible to the framework user. On the contrary, an abstract class is generally intended to be subclassed by the framework user to support the variation which is specific to each application. This case is also referred to as *hot spots* [12].
- The *framework internal increment* captures common implementations of the core framework design. The internal increment may have two forms: subclasses representing common realizations of the concepts captured by the superclasses, or a collection of subclasses representing the specifications for a complete instantiation of the framework in a particular context.
- The *application-specific increment* means new classes needed to be developed to fulfill the actual application requirement. That is, the application-specific increment implements those parts of the application design not covered by the core framework design and the internal increment.
- An *application* is composed of one or more core framework designs, each framework's internal increments, and an application-specific increment.

A highly available system is one that will most likely be working at a given instant time. Failure detection and failure recovery are the main activities for providing highly available systems.

For prompt recovery of system failures, the first action is to recognize the occurrence of them. There are typically two approaches to detecting failures in distributed systems: *Ping/echo* and *Heartbeat*. In the case of ping/echo, one component (so called, monitoring component) issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny (so called, monitored component). This can be organized in a hierarchy, in which a lowest-level detector pings the monitored components, and the higher-level detectors ping lower-level ones. This uses less communications bandwidth than a remote detector that pings all components. In the case of heartbeat (so called, dead man timer), the monitored component emits a heartbeat message periodically and the monitoring component listens for it. If the heartbeat fails, the monitored component is assumed to have failed [13].

The failed service should be recovered in order to provide a continuous service. Failure recovery includes any actions taken to restore the system to service from failure. These actions can cover a wide range of activities from restart of the failed service to failover to other available service. The availability management server should contain the knowledge of the appropriate recovery action for the failure of each managed service in the system, whether it involves restoring that service to full operation or switching over to a redundant service.

Availability management is very common feature for applications, especially for distributed software. The two concepts of failure detection and recovery are common to every application which is required to be highly available. In addition, the specific meth-

ods for failure detection and recovery may depend on the requirement on the availability of each application. Thus, framework is a natural solution to providing availability management service for distributed applications. That is, the common strategy and methods of failure detection and recovery are captured by the core framework design and parts of them are implemented by the framework internal increment. And then, application-specific increment defines some application specific functionality such as the specific way to detect failures in the application and recover from those failures.

3. THE AVAILABILITY MANAGEMENT FRAMEWORK

This section presents the proposed framework for providing availability management for distributed applications. Section 3.1 describes the client-server model of the proposed availability management framework and essential roles of the client and the server part of the framework. The static and dynamic models of entities managed by the framework are described in section 3.2. Finally, section 3.3 describes failure detection and recovery mechanisms that the framework supports and the way to support application-specific availability requirement.

3.1 The Client-Server Model of the Framework

The Availability Management Framework (abbreviated as AMF) provides the common design and implementation such as registration, failure detection, and failure recovery. Fig. 1 shows the high-level system interactions of the AMF and the roles of the constituents.

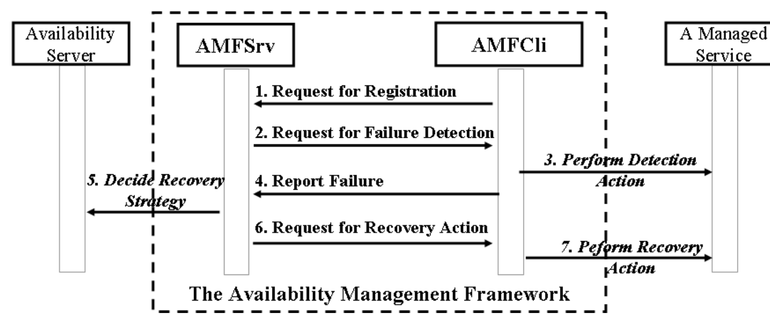


Fig. 1. The system interactions of the AMF.

The AMF consists of two parts: *AMFSrv* package and *AMFCli* package. The *AMFSrv* is for the availability management server and *AMFCli* for various client applications. That is, the *AMFSrv* monitors the occurrence of failures in clients, determine appropriate recovery strategy, and request the chosen recovery action to each client involved in the failure. The *AMFCli* supports the implementation of client applications that are managed by the AMF. The *AMFCli* implements the generic functions for failure detection and failure recovery actions. Each client can extend the predefined generic failure detection and recovery action by supplying its applications-specific increment.

First, a client should be registered in order to be monitored and recovered from failure. Once the client is registered to the AMF Server, the *AMFSrv* requests the *AMFCli* to start failure detection. The *AMFCli* invokes the actual failure detection action that is provided by the client. When some failures are detected by the *AMFCli*, they are reported to the *AMFSrv*. The *AMFSrv* chooses an appropriate recovery strategy depending on the situation. And then, the *AMFSrv* requests each involved client's *AMFCli* to perform the appropriate recovery action. The *AMFCli* invokes the actual recovery action that is provided by the client.

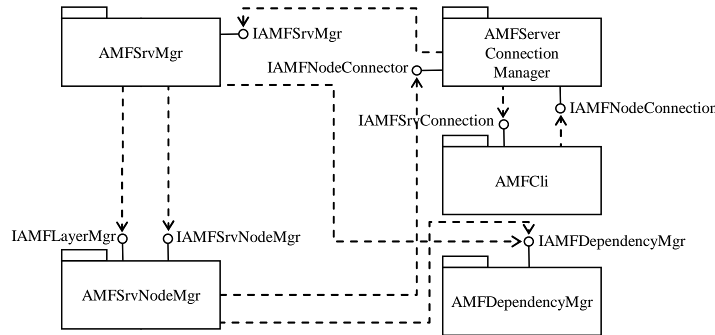


Fig. 2. *AMFSrv* package diagram.

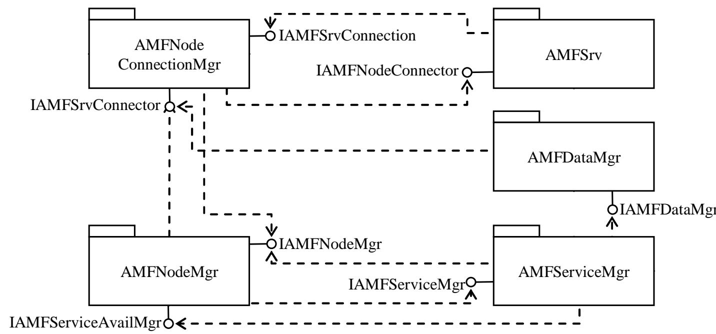


Fig. 3. *AMFCli* package diagram.

Figs. 2 and 3 show the package diagrams of *AMFSrv* and *AMFCli*. To effectively implement availability management, *AMFSrv* is composed of four sub-packages: *AMF-ServerConnectionMgr*, *AMFSrvMgr*, *AMFSrvNodeMgr*, and *AMFDependancy Mgr*. Each sub-package also provides interfaces to other sub-packages and/or *AMFCli*.

- *AMFSrvMgr* monitors client's health status. In case that some failures are detected, *AMFSrvMgr* requests an appropriate recovery action to the client where failure occurs.
- *AMFSrvNodeMgr* and *AMFDependancyMgr* are used to configure and manage the services in a hierarchical structure, which is discussed in section 3.2.
- *AMFServerConnectionMgr* provides communication support between AMF server and clients.

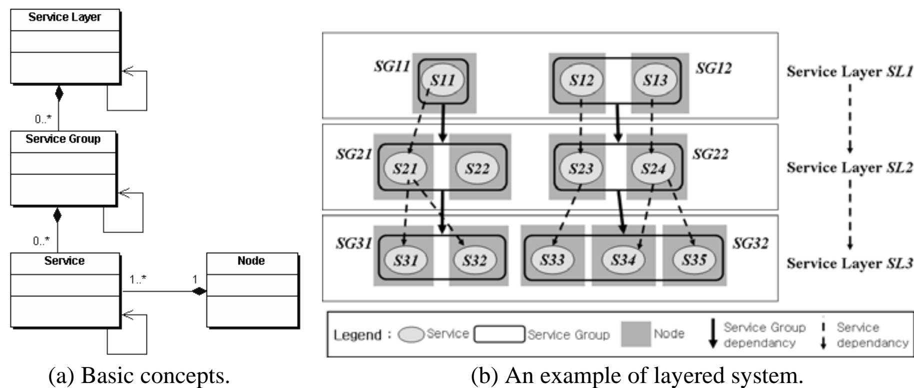
To support the generic implementation of availability in the client side, *AMFCli* is also composed of several sub-packages: *AMFNodeConnectionMgr*, *AMFNodeMgr*, *AMF-ServiceMgr* and *AMFDataMgr*. Each sub-package also provides interfaces to other sub-packages and/or AMFSrv.

- *AMFServiceMgr* provides management for component which intends to join into availability management.
- *AMFNodeMgr* defines typical generic functions for failure detection and recovery actions.
- *AMFDataMgr* keeps a record of working service status for AMF which can be used to support failure recovery.
- *AMFNodeConnectionMgr* provides communication support between AMF server and clients.

Since the AMF only implements the common features of failure detection and recovery, the actual functions that are specific to each client should be provided by the client and invoked by the framework. The AMF provides three hot spots that should be extended by the framework's users in order to satisfy their needs. These hot spots are concerned with failure detection, failure recovery, and failure recovery strategy, respectively. A case study is given in section 4 to illustrate the way to use these hot spots in detail.

3.2 The Conceptual Model of Availability Management

This subsection presents the conceptual model and the state model of logical entities that are managed by the AMF. The class diagram in Fig. 4 (a) shows the logical entities managed by the AMF and the relationships between them.



(a) Basic concepts. (b) An example of layered system.

Fig. 4. The conceptual model of the AMF.

- A *service* is the smallest unit on which the AMF performs failure detection and recovery. A service can be associated with several different states depending on the characteristics with respect to its availability. Typically, a service has two different states:

Active and *Failed*. A service of active state provides the intended function and a service of failed state cannot perform its function due to some kind of failures. We will present in more detail the state model for a service in Fig. 6.

In distributed systems, a service is associated with other services to provide meaningful operation requested by users. In this paper, service *S*2 is defined to depend on service *S*1 when service *S*1 is required for successful operation of service *S2*.

- A *service group* is a cluster of the equivalent services; that is, all the services in a service group provide the same behavior to the request from clients. Each service belongs to only one group. A service group can have one of the *Active*, the *Partially active*, or the *Failed* states, depending on the states of the services that belong to the group:
  - A service group is active when all the services in the group are active.
  - A service group fails when all the services in the group fail.
  - A service group is partially active when the group is not active and any service in the group is active.

  The concept of service group is intended for failure recovery strategy in such a way that an available service in the same group is chosen as a take over in stead of a failed service unless the service group fails.

  There is a dependency between two service groups. A dependency between service groups allows dependency between the services that belong to the services groups. For example, consider a configuration of distributed systems in Fig. 4 (b). There are six service groups, named *SG*11, *SG*12, *SG*21, *SG*22, *SG*31, and *SG*32, respectively, each of which consists of one, two, or three services. There are four dependencies between service groups, which are denoted by solid line with arrow between the corresponding rectangles with rounded corner. It is valid for service *S*21 to depend on services *S*31 and *S*32 because service group *SG*21 depends on *SG*31. However, it is not allowed for *S*21 to depend on services outside group *SG*31 such as *S*33 or *S*34.
- A *service layer* is a cluster of service groups. Each service group belongs to only one layer. There is a dependency between two service layers, which enforce that dependencies between service groups could be defined in a layered fashion. Consider the configuration in Fig. 4 (b) where three service layers *SL*1, *SL*2 and *SL*3 are defined and *SL*1 depends on *SL*2 and *SL*2 on *SL*3. Because *SL*1 depends on *SL*2, a service group (*i.e*., *SG*11 and *SG*12) in *SL*1 can depend on any service group (*i.e*., *SG*21 and *SG*22) in *SL*2. However, a service group cannot have dependency on a service group in the same layer or in upper layers.

  Similar to states of a service group, the state of a service layer is defined to be *Active*, *Partially active*, or *Failed* depending on the states of its composing service groups. Unless a service layer fails, the service layer can provide a service by any active service in an active or partially active service group.
- A *node* is a processing unit which can run services deployed on it. As seen in Fig. 4 (a), a node can have multiple services. For simplicity, there is one service on a node in Fig. 4 (b).

The concepts of service layer and service group are useful for availability management of distributed system of multiple homogenous services. In particular, the concept of service layer is introduced to simplify the definition of the dependency between multiple common services in distributed systems. Especially, service layers are very useful when

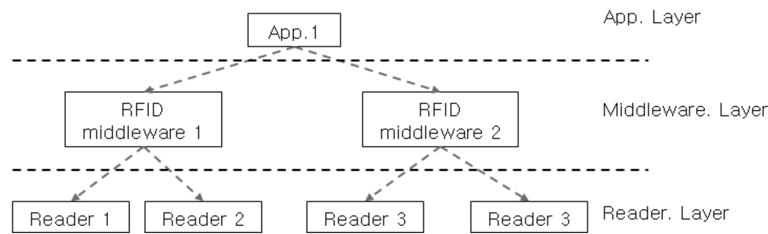


Fig. 5. A simple layered distributed system.

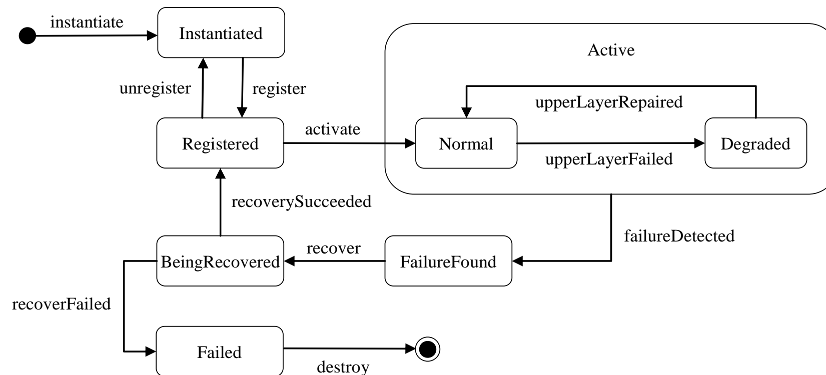


Fig. 6. The state model of a service.

distributed systems are configured in a regular way such as layered configuration or hierarchical configuration.

For example, consider RFID systems (see Fig. 5). In general, an RFID system consists of three components: readers, RFID middlewares, and application servers. And there exist dependencies between each of them. An RFID middleware depends on operations of many readers, and an application server can operate properly only if they collect tag data from the RFID middlewares. Although there are a number of readers, RFID middlewares, and application servers, the configuration can be simply captured by the concept of service layer from the viewpoint of availability. That is, a reader layer is positioned at the bottom, an RFID middleware layer is on the reader layer, and finally an application layer is on the RFID middleware layer. In addition, the concept of dependency between service layers supports the degraded operation of a service when the dependent services at the upper layer fail. The concept of the degraded operation is captured in the state model of a service.

Fig. 6 shows the state model of a service in UML state diagram. An instantiated service is registered into the AMF as an entity for availability management. By activating the registered service, the service starts to provide service (*Active* state). When a failure is detected by the AMF during the operation of a service, the state of the service is changed into *FailureFound* and then the AMF attempts to recover the failed service into *BeingRecovered* state. The state of the service is changed into *Registered* or *Failed* depending on the success of the recovery.

The *Active* state is refined into *Normal* and *Degraded* states, which capture the de-

graded operation of a service due to failures of dependent services. Initially, the active service is operated under the state of *Normal*, which indicates that all its dependent services at the upper layer are successfully operating. When the AMF detects a failure in some dependent services, it changes the service into *Degraded* state. During the *Degraded* state, the service should run a special mode. For example, consider *Reader*1 of an RFID system in Fig. 5. When a failure is detected with the *Reader*1, the AMF just attempts to recover by restarting the reader. However, when its dependent service at the upper layer, *RFID middleware*1, fails, the reader cannot send the information collected from RFID tags. Therefore, the AMF changes the state of the reader into *Degraded* state so that *Reader*1 can perform an appropriate action, *i.e.*, storing the collected information until the *RFID middleware*1 is recovered.

3.3 The Flexibility of Failure Detection and Recovery Support

It is important to support various detection and recovery mechanisms since the selection of detection and recovery actions depends on the changing requirement on availability for each application. Therefore, the AMF is designed to provide a general framework for failure detection and recovery mechanisms. As can be known from Fig. 3, the typical generic functions for failure detection and recovery actions are implemented in the subpackage *AMFNodeMgr* of package *AMFCli*. Figs. 7 and 8 are the low-level diagrams of Fig. 3, and they show the designs for failure detection and recovery mechanisms, respectively.

Fig. 7 shows a class diagram that illustrates the part of the AMF with respect to failure detection mechanisms. The AMF supports four kinds of failure detection mechanisms: passive detection, external active detection, internal active detection, and user-defined detection. On four types of failure detections, we have embraced an approach by SAF's AIS [14]. (SAF's AIS is a standardized interface for availability implementation and it has proposed failure detection approaches to define standard interface to availability management.)

- Passive detection (*AMFPassiveDetectionMechanism*): The service is not involved in the detection, and mostly operating system features are used to decide the health of a service. This includes monitoring the death of processes comprising the service.
- External active detection (*AMFExternalActiveDetectionMechanism*): There is no need to include special code for detecting the health of a service. Instead, the AMF examines its health by invoking the service and checking that the service operates in a timely manner.
- Internal active detection (*AMFInternalActiveDetectionMechanism*): The service can provide the code for checking its own health for more precise and correct detection. The AMF utilizes this code to investigate the health of the service. The AMF basically provides two detection mechanisms of this kind: *AMFPingEchoMechanism* and *AMFHeartbeatMechanism*. *AMFHealthCheckProvider* is used as a placeholder to indicate the audit code for monitoring the health of the service. Two providers *AMFPingEchoProvider* and *AMFHeartbeatProvider* are already provided as framework internal increment.

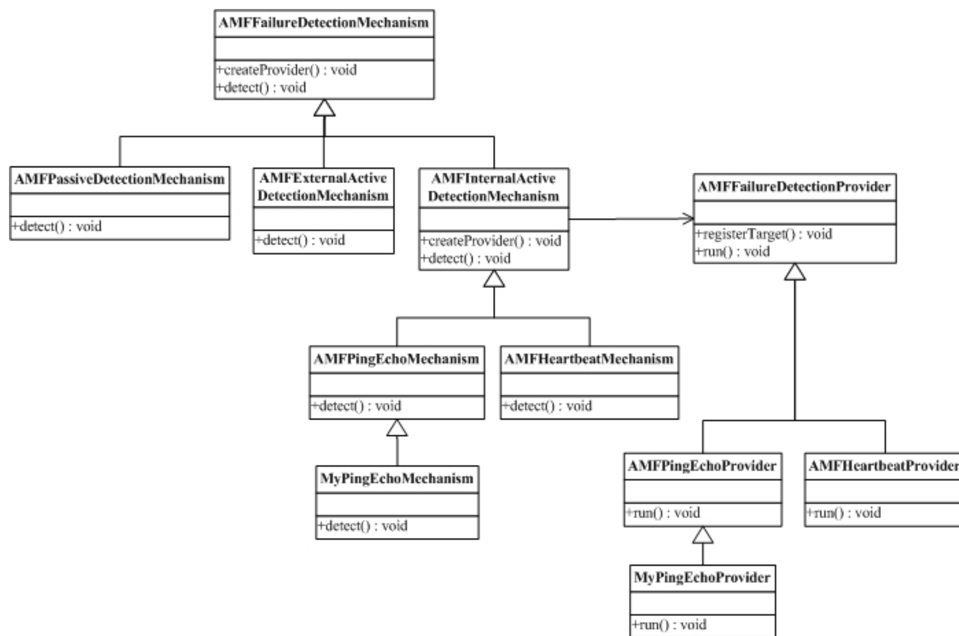


Fig. 7. Design for failure detection mechanism.

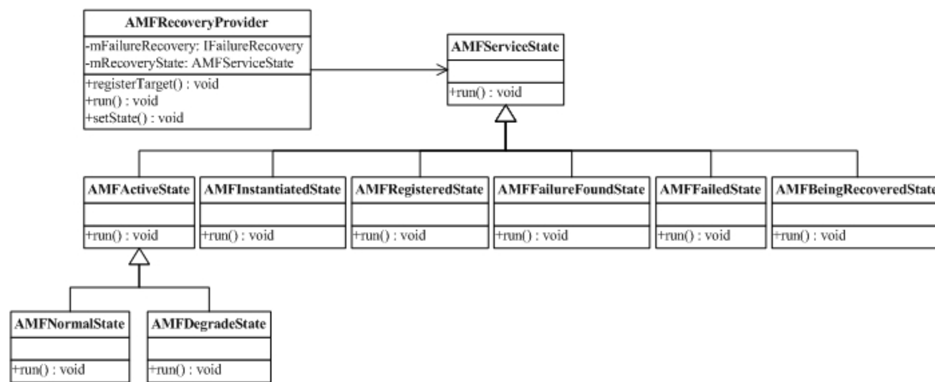


Fig. 8. Design for failure recovery mechanism.

- Application-defined Detection: Developer can define his own detection mechanism that is specific to his application. Application-defined detection mechanism can be designed by extending the provided mechanisms; by subclassing any of classes *AMFPassiveDetectionMechanism*, *AMFExternalActiveDetectionMechanism*, and *AMFInternalActiveDetectionMechanism*.

As the actual code for monitoring its state is provided by the service, the recovery action naturally depends on each application. To support application-specific recovery action, the AMF enables an application to implement its recovery action, which depends on the state of the service.

Recovery mechanism will perform recovery actions when a service is found not in the state of *Active*. According to different recovery strategies and the real application situation, the service may be changed into the state of *Registered* or *Failed*, and its upper layer services have to be changed to *Degraded* or *Normal*. Therefore, the state of service is a critical consideration for recovery mechanism. When performing recovery actions, the state of service should be taken into account to decide proper recovery actions. In the light of this approach, the AMF defines recovery action model (see Fig. 8) based on the state model of Fig. 6. *AMFRecoveryProvider* sends a recovery request message to a generic *AMFServiceState* class. The *AMFServiceState* is specialized into several classes that are specific to each state of a service. Depending on its specific recovery strategy, developers can implement the required recovery action by subclassing the appropriate classes.

3.4 The Processes of Registration, Failure Detection and Failure Recovery

In order to give a clear understanding of availability management in our proposed framework, in this subsection, the processes of registration, failure detection and failure recovery are described.

Registration is a necessary behavior to support failure detection and failure recovery in the proposed AMF. Registration is a process composed of service layer creation, service group creation, node registration and service registration. According to viewpoint of our framework, each service belongs to a service group, and service group should be assigned to a service layer. Before the service is registered into the AMF server, the AMF server should configure such hierarchical structure of service by creating a service group followed by creating a server layer. And then, a node is registered to *AMFSrvMapping Manager*. The objects of *AMFPingEchoMechanism* and *AMFRecoveryProvider* should also be created for the service. Finally service is registered to availability server with parameters *layerID* and *groupID*, which are predefined from previous steps, and *nodeID* and *serviceID* will be generated by *AMFSrvRgisterManager*, which will be notified to *AMFServiceLayer* and *AMFServiceGroup*.

Once failure occurs at some service, which means *AMFServiceManager* cannot receive response from it within a pre-defined interval, *AMFServiceManager* will notify the *serviceID* of that service to *IAMFNodeMgt* by operation *notifyFailure* (*serviceID*), and *IAMFSrvConnection* will be notified by the same operation from *IAMFNodeMgt* also. Afterwards the operation *notifyFailure* (*nodeID*, *serviceID*) of *IAMFNodeConnection*, *IAMFSrvMgt*, and *AMFSrvManager* will be invoked subsequently. *AMFSrvManager* will give a recovery strategy to *AMFSrvFailureDetectionManager*, which comes from *AMFSrvRegisterManager*, delegated by *recoveryID*. Finally, the operation *notifyFailure* (*nodeID*, *serviceID*, *recoveryID*) of *AMFSrvFailureDetectionManager* is invoked to get failure report. *AMFSrvFailureDetectionManager* gets a failure report about node and service name, and recovery strategy.

In case *AMFFailureDetectionManger* receives a failure *serviceID*, it will invoke operation *recoveryFailure* (*nodeID*, *serviceID*, *recoveryID*) from *AMFFailureRecovery-Manager*. For example, we want to take *AMFRecoveryRestart* as our recovery strategy, *AMFFailureRecoveryManager* will take three steps. Firstly, *AMFRecoveryRestart* will create an object of *AMFRecoveryRestart*, and then *AMFRecoveryRestart* decides which

kind of *failureType* and invoke operation *createRecoveryAction* (*failureType*, *nodeID*, *serviceID*) of *AMFRecoveryRestart*, an *AMFRecoveryActionList* is returned, so that *AMF-FailureRecoveryManager* invokes operation recover (*AMFRecoveryActionList*) of *IAMF-LayerMgt*. *IAMFLayerMgt* decides the *AMFRecoveryAction* and invokes operation *get-DependencyInfo* (*AMFRecoveryAction*) of *AMFDependencyManager*. Afterwards, the operation recover of *AMFNodeAvailManager*, *AMFNodeAgent*, *IAMFNodeConnection*, *IAMFSrvConnection*, *IAMFNodeMgt*, *IAMFServiceAvailMgt*, *AMFServiceRecoveryManager* and *IAMFServiceMgt* will be invoked subsequently. Finally, the failed service is restarted.

4. A CASE STUDY

This section describes the result of a case study where the proposed AMF is applied to implement availability management for RFID system. Section 4.1 is the illustration of necessary procedures for RFID system to implement availability management based on AMF. Section 4.2 describes in detail how to implement availability for RFID middleware systems based on our framework.

4.1 An Overall Illustration of RFID System with AMF

We have developed availability management system for an RFID system, named LITware. LITware is an RFID system, developed by Research Center for Logistics Information Technology (for short, LIT) [16]. LIT has been developing the core components of RFID systems such as RFID tags, RFID readers, and RFID middlewares. LITware supports various kinds of RFID readers such as Alien, Intermec, Savi, and Hi-G-Tek including barcode reader. Refer to [16] for more information about LITware.

In order to provide availability management for RFID systems, we have implemented failure detection and recovery functions for RFID readers and RFID middlewares using the proposed AMF. Take LITware for instance. Fig. 9 shows the architecture of RFID systems from the viewpoint of availability.

The availability server has been implemented on the top of the *AMFSrv* package of the AMF that provides basic functions such as node/service management and recovery strategy selection. Application, EdgeManager, and Reader are the entities that are managed by the AMF and thus each of them is implemented using the *AMFCli* package of the AMF where some basic functions of failure detection and recovery classes are implemented.

Fig. 10 shows a configuration of RFID system which consists of five readers (*R*1, …, *R*5), four EdgeManagers (*EM*1, …, *EM*4), and three applications (*A*1, …, *A*3)[1]. For each type of components, three service layers are defined: *App. Layer*, *EdgeManager Layer*, and *Reader Layer*. Within each layer, two service groups and dependencies between them are defined; that is, *App. Layer* depends on *EdgeManager Layer*, and *EdgeManager Layer* depends on *Reader Layer*. Each service group consists of one (*SG*1 in *App. Layer*) to three services (*SG*2 in *Reader Layer*).

Initially, this configuration is registered into and maintained by the availability server. Once registered, all the services are managed by the server. There are two cases

[1] In actual RFID systems, an EdgeManager will manage much more readers.

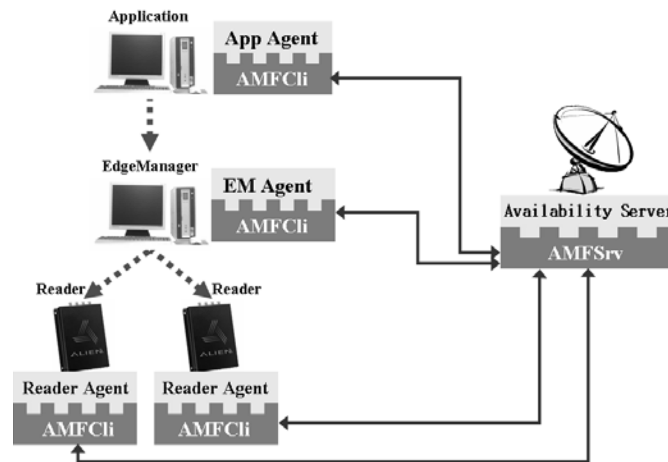


Fig. 9. An RFID system with the AMF.

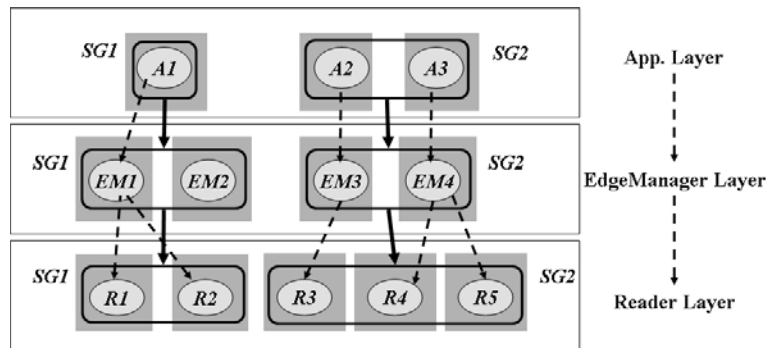


Fig. 10. A configuration of RFID system.

of failure recovery when *EM*1 fails. First, the server attempt to recover by restarting *EM*1. At the same time the server notices that *EM*2 in the same group is available at that time, and then the server requests *R*1 and *R*2 to run in a degraded operation since the two readers cannot deliver the collected tag information any more to the failed *EM*1. When requested for a degraded operation, the reader service stores the pending tag information within itself or sends it to the availability server. After the successful recovery of *EM*1, the server notifies *R*1 and *R*2 of the recovery of *EM*1 so that they continue to deliver tag information to *EM*1 followed by the delivery of the stored tags. Second, if the server fails to recover *EM*1, then it attempts to recover by using *EM*2 in the same group. That is, *R*1 and *R*2 are requested to start a transmission of tag information to *EM*2 followed by the delivery of the pending tags.

The availability server has been implemented on the top of the *AMFSrv* package that provides a skeleton implementation for node/service management and recovery strategy selection. Using the availability server, users can monitor the configuration of RFID systems and understand the process of failure detection and recovery.

4.2 Providing Availability Management to EdgeManager with AMF

Availability management has been implemented by extending the client part of the AMF, *AMFCli*. Our framework enables developers to use the AMF with ease. Fig. 11 shows the class diagram for the design of *EMAgent*.

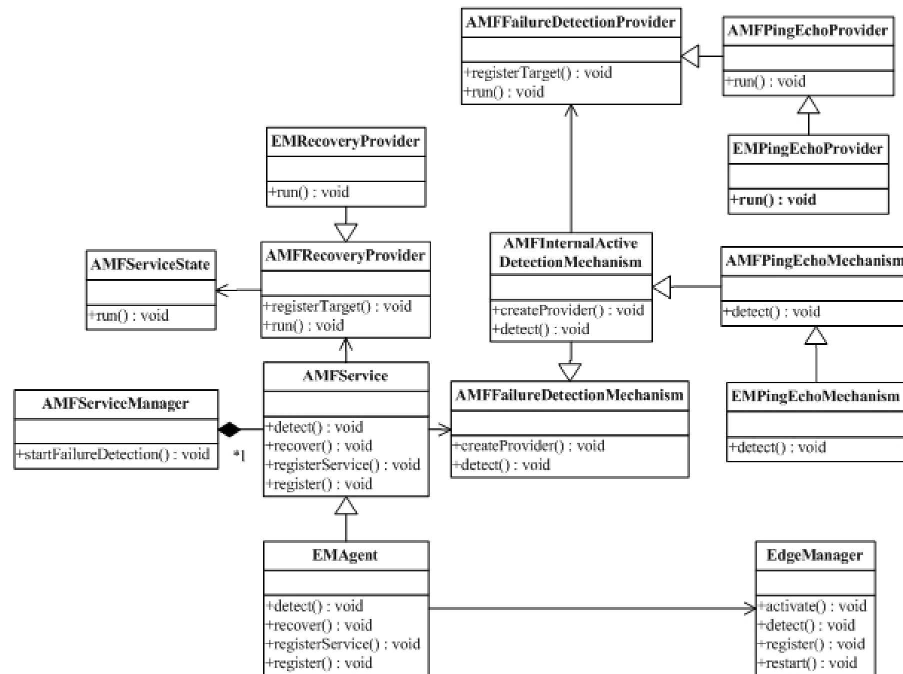


Fig. 11. Implementation of *EMAgent* for *EdgeManager* with AMF.

EMAgent is needed to be implemented for EdgeManager as an application specific increment. First, developers implement their specific failure detection and recovery actions by extending the framework, and then, *EMAgent* is registered into the availability server. *EMAgent*, the subclass of *AMFService*, defines failure detection and recovery actions by implementing *EMPingEchoProvider* and *EMRecoveryProvider*, which are subclasses of *AMFPingEchoProvider* and *AMFRecoveryProvider* respectively. Failure detection mechanism should be implemented by subclassing *AMFPingEchoMechanism*.

In order to provide application specific recovery actions, developers should implement failure recovery mechanism by extending the generic recovery action model defined in Fig. 8. This extended action model will be used by *AMFRecoveryProvider* to provide failure recovery mechanism.

In addition, our framework can be efficient to support modification and evolution of middlewares from change of availability requirement. With the AMF, developers can accommodate the new detection mechanism with little modification to the existing code. Fig. 12 shows the new class diagram developed under the newly defined failure detection actions. In comparison to Fig. 11, we can recognize that developers just implement *An-*

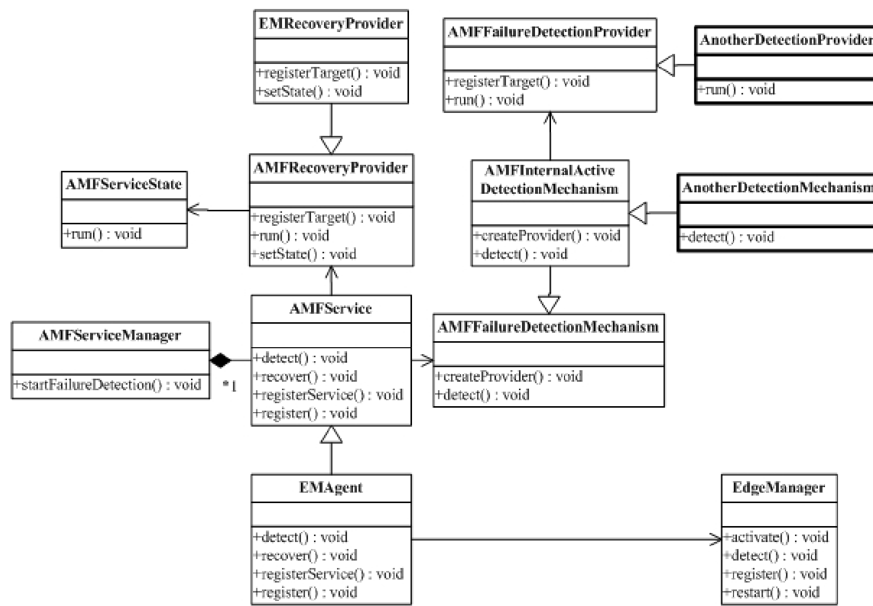


Fig. 12. The class diagram after the change.

otherDetectionProvider and *AnotherDetectionMechanism*. That is, there is no extra effort except adding two corresponding classes at the level of design to support new failure detection actions.

5. RELATED WORK

There have been several projects, researches and standardization activities for developing methods and frameworks for service availability. Some projects have developed or are developing solutions for available or dependable systems in such various and specific domains, such as telecommunication [18], embedded systems [19], web services framework [20], telematics [21], wireless communication networks [22], information systems [23], database [24], and grid computing [25]. The purpose of GST [21] is to create an environment in which innovative telematics services can be developed and delivered cost-effectively, and hence to increase the range of economic telematics services available to manufacturers and consumers. HIDENETS [22] is a specific targeted research project which aims at development and analysis of end-to-end resilience solutions for distributed applications and mobility-aware services in ubiquitous communication scenarios with critical dependability requirements in the context of selected use-cases with infrastructure service support. DESEREC project [23] aims to improve the dependability of new and existing Information Systems by leveraging their capabilities of failure-proof, self-healing, and short recovery time. With a three-tiered approach, DESEREC keeps every incident local, sustains or quickly resumes the critical services and reallocates optimally the resources to recover the full range of services. All of them deal with availability management as a critical issue in their projects. However, limited by specific application domains, those projects did not develop general failure detection and

recovery mechanisms, and they did not propose an extensible and flexible availability management framework.

The service availability forum (SAF) [14] is developing and publishing high availability and management software interface specifications. The availability management framework proposed by SAF provides a set of APIs to enable highly available applications. The availability management framework drives the HA state and monitors the health of a component by invoking callback functions of the component defined in this API. It manages internally also the readiness state, without exposing it to components. There have been several projects and researches on SAF's AIS. OpenAIS [26] project implements current research on virtual synchrony, which provides stateful failover resulting in 100-percent correct operation. The OpenAIS Framework provides highly configurable support for turning newly developed services highly available. For legacy services, when source code is not accessible, the AMF specification proposes so-called proxy components. The function of the proxy components is to mediate between the AMF Framework and the legacy services. Presently OpenAIS has been implemented to support for building highly available Session Initiation Protocol Registrar. In [24], the authors proposed a reference division of responsibilities in a system supporting highly available data services based on SAF's AIS.

6. CONCLUSTION AND FUTURE WORK

We have described the high-level design of a framework for flexible availability management for developing distributed applications. The proposed framework provides flexibility that enables distributed applications to accommodate various availability requirements using object-oriented framework technologies. In addition, the concept of service layers supports the simplified management of dependencies between services and degraded operation of services. We have also performed a case study with an RFID system where the availability of RFID middlewares and readers were efficiently implemented by reusing and extending the framework.

Complex systems are typically structured hierarchically in multilevel organizations. Such systems can be viewed as large distributed systems consisting of interconnected networks of computing nodes [17]. We have adopted from SAF's AIS [14] a few concepts such as the classification of failure detection mechanisms. However, AIS supports only simple layered dependency at service level, which is not applicable for hierarchical distributed systems. Our framework supports a dependency management between service layers where the dependencies between services are constrained in a layered fashion, which simplifies the dependency model of services, and in addition allows the degraded operation of a service at the failure of an upper layer service. Due to the development of the global communication infrastructure and new technologies emerging to power new services, availability becomes more and more important for distributed applications. We believe the proposed framework will give developers convenience for implementing various availability management to distributed applications.

Two research directions are considered future work. First, we plan to enhance our framework by providing framework internal increment for some specific domain such as RFID systems. Second, more optimization considerations can be incorporated into the

framework. For example, when the AMF determines a take over service instead of a failed service in the same group, it can consider the scalability issue additionally. That is, the AMF can be enhanced to choose the service with the least load among several available services in the same group.

REFERENCES

1. J. C. Laprie, "Dependable computing: Concepts, limits, challenges," in *Proceedings of the 25th IEEE Symposium on Fault-Tolerant Computing*, 1995, pp. 42-54.
2. E. Marcus and H. Stern, *Blueprints for High Availability*, Wiley, New York, 2003.
3. M. M. Lehman, J. F. Ramil, and D. E. Perry, "On evidence spporting the FEAST hypothesis and the laws of software evolution," in *Proceedings of the 5th International Symposium on Software Metrics*, 1998, pp. 84-88.
4. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Boston, 2003.
5. M. Fayad and D. Schmidt, "Object-oriented application framework," *Communications of the ACM*, Vol. 40, 1997, pp. 32-38.
6. R. H. Campbell and N. Islam, "A technique for documenting the framework of an object-oriented system," in *Proceedings of Object Orientation in Operating Systems*, 1992, pp. 288-300
7. D. C. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software," *Handbook of Programming Languages*, Vol. 1, 1997.
8. E. T. Birrer, "Frameworks in the financial engineering domain: An experience report," in *Proceedings of the 7th European Conference on Object-Oriented Programming*, 1993, pp. 21-35.
9. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, *Object-Oriented Application Frameworks: Problems and Perspectives*, Wiley, New York, 1997.
10. M. E. Fayad, D. C. Schmidt, and R. E. Johnson, *Object-Oriented Application Frameworks: Implementation and Experience*, Wiley, New York, 1997.
11. T. Hanamori, N. Kurumai, and T. Shima, "Promotion of development efficiency by using framework," *FUJITSU Scientific and Technical Journal*, Vol. 42, 2006, pp. 333- 346.
12. W. Pree, "Meta patterns − A means for capturing the essentials of reusable object-oriented design," in *Proceedings of the 8th European Conference on Object-Oriented Programming*, LNCS 821, 1994, pp. 150-162.
13. M. Bertier, O. Marin, and P. Sens, "Implementation and performance evaluation of an adaptable failure detector," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2002, pp. 345-363.
14. SA Forum, *Application Interface Specification*, SAI-AIS-AMF-B.01.01.
15. EPCglobal Network, http://www.epcglobalinc.com/news/EPCglobal_Network_Overview_10072004.pdf.
16. Research Center for Logistic Information Technology, http://lit.pusan.ac.kr.
17. Y. B. Shieh, D. Ghosal, P. R. Chintamaneni, and S. K. Tripathi, "Modeling of hierarchical distributed systems with fault-tolerance," *IEEE Transactions on Software*

Engineering, Vol. 16, 1990, pp. 444-457.
18. Carrier Grade Linux, http://www.osdl.org/lab_activities/carrier_grade_linux.
19. SelfReliant, GoAhead Software Inc., http://www.goahead.com.
20. P. Verissimo, J. Clarke, W. Donnelly, Z. Dooly, W. Fitzgerald, S. Johnstone, S. Shiu, V. Lotz, M. Melideo, A. Pasic, and P. Soriarodriguez, "Research requirements for security and dependability for services," in *Proceedings of International Service Availability Symposium*, 2006.
21. G. Segarra, "Global system for telematics (GST)," in *Proceedings of International Service Availability Symposium*, 2006.
22. HIDENETS, "High dependability in IP based networks and services project," 2006, http://www.hidenets.aau.dk.
23. DESEREC, "Dependable security by enhanced reconfigurability project," http://www.deserec.eu.
24. A. Wolski and B. Hofhauser, "A self-managing high-availability database: Industrial case study," in *Proceedings of the 21st International Conference on Data Engineering*, 2005, pp. 1210.
25. P. Chen, Z. Xu, and B. Zhang, "A solution to QoS control and availability promotion in complex grid computing," in *Proceedings of the 12th IEEE International Conference on Networks*, 2004, pp. 403-407.
26. Sun and Carrier-Grade Linux, "Accentuating choice in the telecom market," Technical White Paper, 2005.

Heung Seok Chae received the B.S. degree in Nuclear Engineering from Seoul National University in 1994, the M.S. and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1996 and 2000, respectively. Since 2004, he has been on the faculty of the Department of Computer Science and Engineering, Pusan National University, Busan, Korea. His current research interests include object-oriented analysis and design, object-oriented frameworks, component-based software development, software metrics, middleware architecture for availability and scalability.

Jian Feng Cui received the M.S. degree in Software Engineering from NanKai University, China, in 2004. He is currently pursuing the Ph.D. degree in Computer Science and Engineering at Pusan National University, Korea. His current research interests include mobile agent technology for distributed system, component-based software development and software reengineering.

Jin Wook Park received the B.S. and M.S. degrees in Computer Science from Pusan National University in 2005 and 2007, respectively. He is currently a software engineer at Neople Ltd., Korea. His current research interests include software architecture and design, game server programming and UML.

Jae Geol Park received the B.S. degree in Computer Education from Silla University in 2005 and the M.S. degrees in Computer Science and Engineering from Pusan National University, Korea, in 2007. He is currently a software engineer at SE-BANG Co., Ltd. His current research interests include adaptive middleware load-balancing, object-oriented frameworks and service-oriented architecture.

Woo Jin Lee is currently an Associative Professor in the School of Electrical Engineering and Computer Science at Kyungpook National University, Korea. He received the Ph.D. degree in Computer Science from Korea Advanced Institute of Science and Technology in 1999. His main research interests include formal methods, requirements engineering, embedded systems, and component-based software development.