The Journal of Systems and Software xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Quality-driven architecture development using architectural tactics

Suntae Kim [a,b,*], Dae-Kyoo Kim [a], Lunjin Lu [a], Sooyong Park [b]

[a] Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309, USA
[b] Department of Computer Science, Sogang University, Seoul, South Korea

ARTICLE INFO

Article history:
Received 27 July 2008
Received in revised form 24 February 2009
Accepted 31 March 2009
Available online xxxx

Keywords:
Architectural tactics
Feature composition
Feature modeling
Quality-driven
Role-based metamodeling language
Software architecture, UML

ABSTRACT

This paper presents a quality-driven approach to embodying non-functional requirements (NFRs) into software architecture using architectural tactics. Architectural tactics are reusable architectural building blocks, providing general architectural solutions for common issues pertaining to quality attributes. In this approach, architectural tactics are represented as feature models, and their semantics is defined using the Role-Based Metamodeling Language (RBML) which is a UML-based pattern specification notation. Given a set of NFRs, architectural tactics are selected and composed, and the composed tactic is used to instantiate an initial architecture for the application. The proposed approach addresses both the structural and behavioral aspects of architecture. We describe the approach using tactics for performance, availability and security to develop an architecture for a stock trading system. We demonstrate tool support for instantiating a composed tactic to generate an initial architecture of the stock trading system.

1. Introduction

Software development is initiated with application requirements which consist of functional requirements (FRs) and non-functional requirements (NFRs). Functional requirements describe the functionality of a system that supports user needs, while non-functional requirements impose constraints on how the system should accomplish the system's functionality, determining the quality of the system. The interdependency between FRs and NFRs requires consideration of both FRs and NFRs throughout the development. However, in practice, NFRs are often deferred for consideration until the late phase (Cysneiros et al., 2003; Zou et al., 2007; Xu et al., 2005), which makes it difficult to satisfy NFRs and often necessitates changes to early development artifacts such as the architecture and the design. This is mainly attributed to the lack of techniques that help systematic embodiment of NFRs in the early development phase.

There has been some work on addressing NFRs at the architectural level (e.g., see Khan et al., 2005; Chung et al., 1999; Cysneiros and Leite, 2004; Franch and Botella, 1998; Metha and Medvidovic, 2002; Rosa et al., 2000; Xu et al., 2006; Zarate and Botella, 2000). There are two major streams. One is to specify NFRs into the functional architecture of the application as architectural constraints. In this approach, NFRs and functional architectures are usually described in the same language for better compatibility. While this approach helps to check the satisfaction of NFRs, it does not provide a solution for NFRs. In the other approach, quality attributes (e.g., security, performance) in a specific domain are specified with domain constraints as reusable architectural building blocks which are used in building an application architecture or an architectural style for that domain. However, their building blocks are not general enough (i.e., domain-specific), and their granularity is coarse (e.g., one architecture building block per quality attribute), and hence significant tailoring work is required in order to satisfy specific requirements.

To address the above issues, we propose a systematic approach for building a software architecture that embodies NFRs using architectural tactics (Bass et al., 2003). An architectural tactic is a reusable architectural building block that provides a generic solution to address issues pertaining to quality attributes. In this approach, architectural tactics are selected based on a given set of NFRs, and the selected tactics are composed to produce a tactic that exhibits the solutions of the selected tactics. The composed tactic is then instantiated to create an initial architecture of the application into which the NFRs are embodied.

We use feature modeling (Czarnecki and Eisenecker, 2000) to represent tactics and their relationships. We chose featuring modeling for the natural correspondence between the classification feature of feature modeling and the variable nature of tactics. In feature modeling, tactics and their variations are captured as features in a hierarchical form. Use of feature modeling helps to make architectural decisions for NFRs by offering an explicit set of solutions. Also, the relationships of tactics assure accompanied

* Corresponding author. Address: Department of Computer Science, Sogang University, Seoul, South Korea.
E-mail addresses: jipsin08@sogang.ac.kr (S. Kim), kim2@oakland.edu (D.-K. Kim), l2lu@oakland.edu (L. Lu), sypark@sogang.ac.kr (S. Park).

0164-1212/$ - see front matter © 2009 Published by Elsevier Inc.
doi:10.1016/j.jss.2009.03.102

requirements (which are not always obvious) when a decision is made on a tactic.

The structure and behavior of tactics are described using the Role-Based Modeling Language (RBML) which is a UML-based pattern specification language developed in our previous work (France et al., 2004; Kim, 2007). The RBML is chosen for its capability of capturing variations of architectural tactics. We present architectural tactics for availability, performance and security, and describe how the tactics can be used to embody NFRs of a stock trading system into the architecture of the system. We also demonstrate tool support for instantiating a composed tactic to generate an initial architecture of the stock trading system.

The major contributions of this paper are (1) the analysis of architectural tactics' relationships for availability, performance and security, (2) the semantic specifications of tactics for quality attributes, and (3) the mechanism for composing tactics to build a high quality architecture for a specific application that embodies the NFRs of the application.

The remainder of this paper is structured as follows. Section 2 gives a background on architectural tactics, feature modeling and the RBML, Section 3 presents feature models for availability, performance and security tactics and their semantic specifications in the RBML, Section 4 describes how availability, performance and security tactics can be composed, and how the composed tactic can be used to develop an architecture that satisfies NFRs of a stock trading system, Section 5 demonstrates tool support to instantiate the composed tactic to generate an initial architecture of the stock trading system, Section 6 gives an overview of related work, and Section 7 concludes the paper.

2. Background

In this section, we give an overview of architectural tactics which form the basis of this work, feature modeling for categorizing architectural tactics and the RBML for defining the semantics of an architectural tactic.

2.1. Architectural tactics

An architectural tactic is a fine-grained reusable architectural building block that provides an architectural solution built from experience to help to achieve a quality attribute. A tactic is a design decision that influences the concerned quality, and a collection of tactics forms an architectural strategy (Bass et al., 2003). Related to other reusable architectural artifacts, architectural tactics may be viewed as foundational building blocks from which architectural patterns and styles are created (Bass et al., 2003).

There are many architectural tactics (Bachmann et al., 2002; Bass et al., 2003; Ramachandran, 2002) for various quality attributes such as availability, performance, security, modifiability, usability and testability. The following are brief descriptions of some example tactics for availability, modifiability and security:

- Exception – An availability tactic for recognizing and handling faults.
- Ping/Echo – An availability tactic for checking the availability of a component by sending ping messages.
- Heartbeat – An availability tactic for checking the availability of a component by listening to heartbeat messages from the component.
- Semantic Coherence – A modifiability tactic for lowering architectural coupling via inheritance while increasing cohesion.
- ID/Password – A security tactic for user authentication using IDs and passwords.
- Maintain Data Confidentiality – A security tactic for protecting against unauthorized modifications of data using encryption.

A tactic may be either required or optional for a quality attribute. For example, a fault detection tactic is required for availability because it is a prerequisite for other tactics (e.g., fault recovery or recovery reintroduction) to recover the system (Bass et al., 2003). Optional tactics provide the architect maneuverability in building an architecture according to his architectural strategy. The architect can choose optional tactics in various ways to strategically build an architecture. Tactics may be related to one another. A tactic may require another complementary tactic or exclude a conflicting tactic. For example, the *Exception*, *Ping/Echo* and *Heartbeat* tactics are often used together for rigorous fault detection. A tactic may also be refined into more concrete tactics that provide a specific solution to help to achieve the quality attribute related to the tactic. In some cases, use of tactics for one quality attribute may affect another quality attribute. For instance, use of both the *ID/Password* and *Maintain Data Confidentiality* tactics for security often decreases performance due to the encryption of login information.

2.2. Feature modeling

Based on our analysis in Section 2.1, we found feature modeling (Czarnecki and Eisenecker, 2000) suitable for representing the relationships among architectural tactics. Feature modeling is a design methodology for modeling the commonalities and variations of an application family and their interdependencies. A feature model consists of mandatory features capturing commonalities, optional features capturing variations, feature relationships representing logical groupings of features and configuration constraints. There are two types of groups, *exclusive-or* and *inclusive-or*. An *exclusive-or* group specifies that only one feature can be selected from the group, while an *inclusive-or* group denotes that one or more features can be selected from the group. Configuration constraints are feature relationships constraining feature selection. There are two types of relationships, *requires* and *mutually exclusive*. A *requires* relationship constrains that selection of one feature in the relationship requires the other feature. A *mutually exclusive* relationship specifies that the two features in the relationship cannot co-exist. In this work, we introduce another type of relationship, *suggested* to specify complementary relationships for a synergistic combination of tactics. Unlike *requires* relationships which are mandatory, *suggested* relationships are only suggestive.

Fig. 1 shows an example of a feature model for cellular phones. The feature model has the root feature of *Cellular Phone* which has

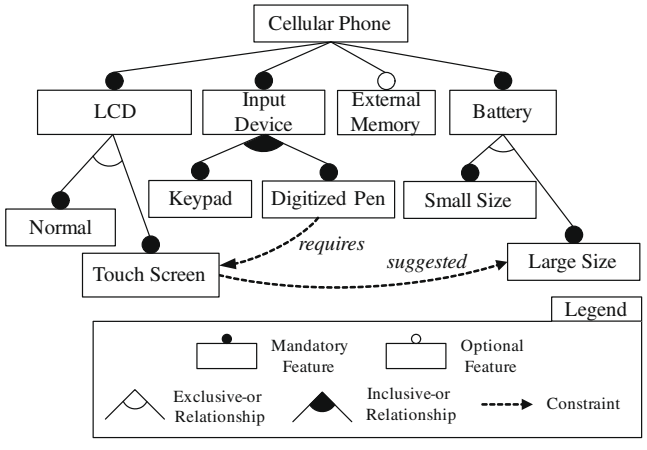


Fig. 1. A feature model.

three mandatory subfeatures of *LCD*, *Input Device* and *Battery* and one optional subfeature of *External Memory*. An LCD can be either a normal screen or a touch screen, but cannot be both. This is specified by the *exclusive-or* group of the *Normal* and *Touch Screen* features which is denoted by the empty arc underneath the *LCD* feature. An input device can be a keypad or a digitized pen, or both. This is specified by the *inclusive-or* group of the *Keypad* and *Digitized Pen* features which is denoted by the filled arc underneath the *Input Device* feature. The *requires* constraint specifies that use of the *Digitized Pen* feature requires the *Touch Screen* feature. The *Battery* feature can be described similar to the *LCD* feature. The *suggested* relationship specifies that a large size battery is suggested when a touch screen is used due to more power consumption.

2.3. Role-Based Metamodeling Language

We use the Role-Based Metamodeling Language (RBML) to define the semantics of an architectural tactic. The RBML is a UML-based pattern specification language developed in our previous work (France et al., 2004; Kim, 2007). The RBML specifies tactic semantics in terms of roles (Kim et al., 2003) which are defined at the UML metamodel level and played by UML model elements (e.g., classes, messages). A role has a UML metaclass as its base and defines a set of constraints on the metaclass to tailor the type of elements that can play the role. Only the instances of the base metaclass that satisfy the constraints can play the role. Every role has a realization multiplicity which constrains the number of elements that can play the role. If it is not specified, the default multiplicity *1..** is used, specifying that there must be at least one element playing the role. Realization multiplicity can also be used to specify the number of instances of a role to create in role instantiation. Major benefits of using the RBML for defining architectural tactics are that (1) the RBML facilitates the use of tactics at the model-level, and (2) the RBML is capable of capturing the generic structure of a tactic, and (3) the RBML enables instantiating a tactic in various structures through the realization multiplicity of roles, which facilitates reuse of tactics.

A Structural Pattern Specification (SPS) is a type of RBML specification, characterizing the structural aspects of an architectural tactic in a class diagram view. An SPS consists of *classifier* and *relationship* roles whose bases are the *Classifier* and *Relationship* metaclasses, respectively, in the UML metamodel. A classifier role is associated with a set of feature roles that determine the characteristics of the classifier role. A classifier role is connected to other classifier roles through relationship roles. An Interaction Pattern Specification (IPS) is another type of RBML specification, defining an interaction view of tactic participants in terms of *lifeline* and *message* roles whose bases are the *Lifeline* and *Message* metaclasses, respectively. Similar to the UML, IPSs are dependent upon an SPS.

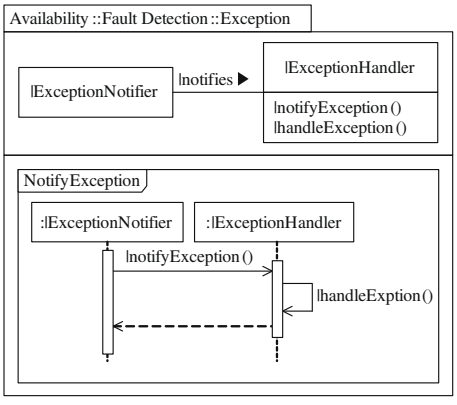


Fig. 2. An RBML specification for the *Exception* tactic.

For example, a lifeline role characterizes a set of lifelines that are instances of a classifier that plays the corresponding classifier role in the associated SPSs of the IPS.

Fig. 2 shows an RBML specification for the *Exception* tactic for availability. In the figure, the upper compartment presents an SPS, and the lower compartment shows an IPS. Roles are denoted by the "|" symbol. The SPS has two class roles, |*ExceptionNotifier* and *ExceptionHanlder* which are associated via an association role |*notifies*. The |*ExceptionHandler* role has two behavioral feature roles, (|*notifyException*() and |*handleException*()) which constrain that a class playing the |*ExceptionHandler* role must have operations playing the behavioral feature roles. In this paper, the classes playing a class role are considered to be architectural components, and associations playing association roles are considered to be pairs of provided and required interfaces (The Object Management Group, 2007). The IPS specifies that when an exception occurs, the exception should be notified to an exception handler. We use the UML package notation to group related SPSs and IPSs for a tactic.

The RBML enables capturing the structural variations of tactics, which greatly influences the reusability of the tactic. A tactic is generic (i.e., application independent) and can be used in various contexts where its structure may differ to some extent. For instance, in one application only one instance of a tactic participant may be needed, while in another application two instances may be required. This can be captured by a realization multiplicity in the RBML. Such a variation frequently occurs, and tactic specifications that are capable of capturing it can be better reused. Tactic variability may also influence the quality concerned in the tactic. For example, the FIFO performance tactic, which involves producers, FIFO queues, and consumers, is concerned with scheduling requests for resources. In one case, there may be only one queue for all consumers, while in another case, multiple queues may be required, each for one consumer. In these cases, the latter has higher throughput than the former. A concrete example of the latter case is presented in Section 4.2.

3. Specifying architectural tactics

In this section, we define feature models for availability, performance and security architectural tactics and their semantics in the RBML. The feature models and RBML specifications are developed based on the work (Bachmann et al., 2002; Bass et al., 2003; Blazewicz et al., 2007; Cole et al., 2005; Ramachandran, 2002; Silberschatz et al., 2005).

3.1. Tactics for availability

Availability is the degree to which an application is available with the expected functionality. There are several tactics for availability which can be categorized into fault detection, recovery-preparation and repair, and recovery reintroduction (Bass et al., 2003; Schmidt, 2006) as shown in Fig. 3.

The *Fault Detection* tactic in Fig. 3 is concerned with detecting a fault and notifying the fault to a monitoring component or the system administrator. The *Fault Detection* tactic can be refined into *Ping/Echo*, *Heartbeat* and *Exception* tactics. The *Ping/Echo* tactic detects a fault by sending ping messages to receivers regularly. If a receiver does not respond to the sender within a certain time period, the receiver is considered to be failed. Fig. 4a defines the *Ping/Echo* tactic.

The SPS in Fig. 4a shows that the *Ping/Echo* tactic involves three concepts: senders, receivers and a monitor which are captured by the |*PingSender*, |*PingReceiver* and |*FaultMonitor* roles, respectively. The |*PingSender* role is responsible for sending a ping message every certain time and waits for an echo from a ping receiver until

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

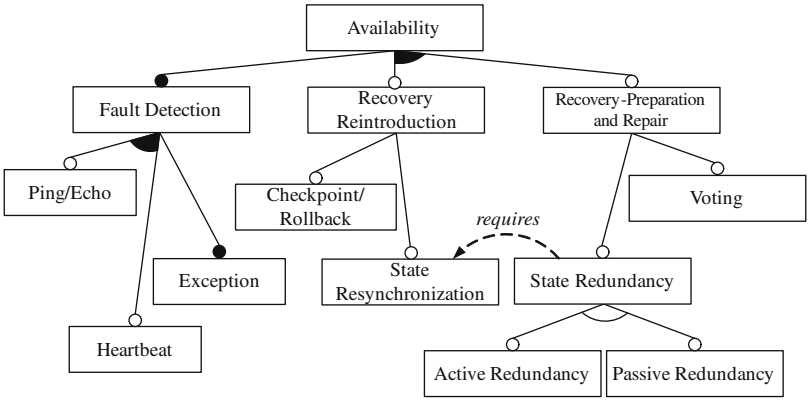
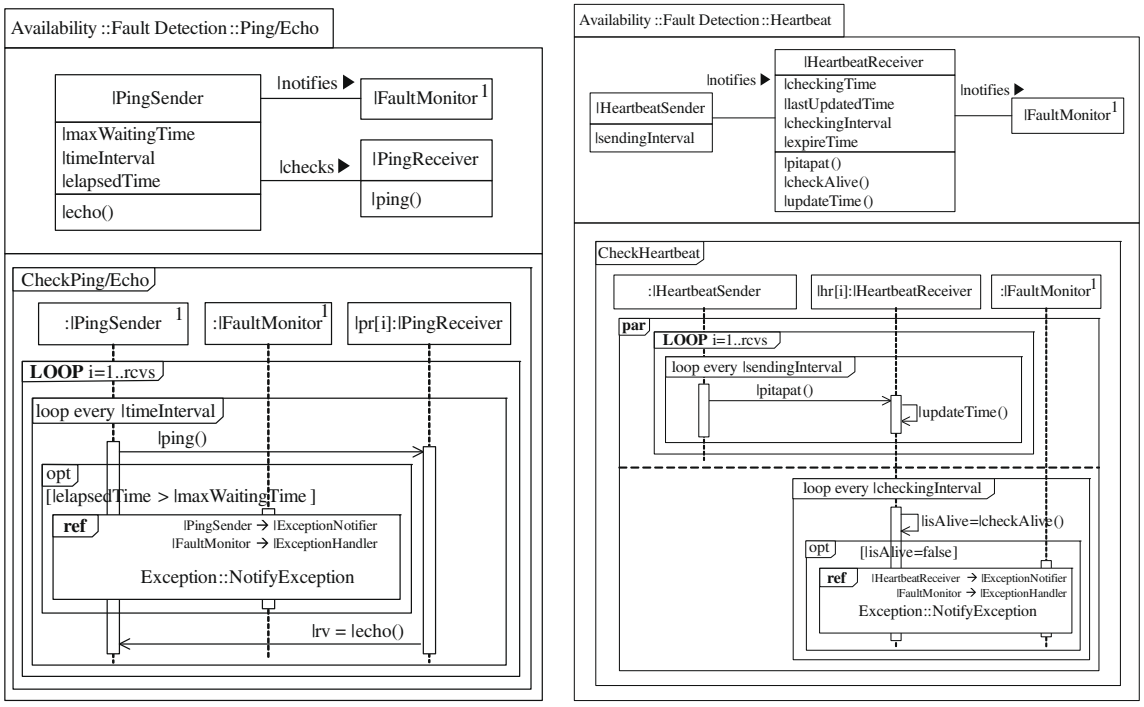


Fig. 3. Availability architectural tactic feature model.



(a) The *Ping/Echo* Tactic (b) The *Heartbeat* Tactic

Fig. 4. The *Ping/Echo* and *Heartbeat* tactics.

the maximum waiting time. If an echo is not received within the maximum waiting time, an exception occurs, and it is detected by the fault monitor. The *LOOP* fragment in the IPS specifies that a sender sends a ping message to all receivers every time interval where *rcvs* in the *LOOP* denotes a function that returns the number of receivers. It should be noted that the *LOOP* operator is an RBML operator which constrains the structure of instantiations of the IPS (Kim, 2007), not the UML *loop* operator. The inner *loop* fragment is the UML *loop* operator, describing that ping and echo messages are sent and received every specified time interval. If a receiver does not send an echo back within the maximum waiting time, the sender throws an exception to the monitor. This is captured in the *Exception* fragment which references the *Exception* tactic in Fig. 2. The mapping specified in the *Exception* fragment describes the binding information between the *Ping/Echo* and *Exception* tactics. The realization multiplicity *1* in the |*FaultMonitor* role constrains that there must be only one monitor.

The *Heartbeat* tactic detects a fault by listening to heartbeat messages from monitored components periodically. A sender sends a heartbeat message to all the receivers every specified time interval. The receivers update the current time when the message is received. If the message is not received within a set time, the monitored component is considered to be unavailable. Fig. 4b defines the *Heartbeat* tactic.

Similar to the *Ping/Echo* tactic, the *Heartbeat* tactic involves three concepts, senders, receivers and a monitor, each of which is captured by the |*HeartbeatSender*, |*HeartbeatReceiver* and |*FaultMonitor* roles. The *HeartbeatSender* role is responsible for sending a heartbeat message periodically. The sent heartbeat message is received by a heartbeat receiver, and the last received time of a heartbeat message is updated, which is captured by the |*updateTime*() role. The aliveness of heartbeat senders is checked regularly by comparing the latency time between the current time and last updated time in consideration of the max waiting time for

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

the next heartbeat message. This is captured by the |*checkAlive*() role. The IPS specifies that sending heartbeat messages and checking aliveness are executed on separate threads for concurrency.

When a fault occurs, the fault should be recognized, and an exception handler should be informed of it. This is captured in the *Exception* tactic. Typically, the component that raises an exception executes in the same process of the exception handling component. As shown above, the *Exception* tactic is usually used together with the *Ping/Echo* tactic and *Heartbeat* tactic for handling faults.

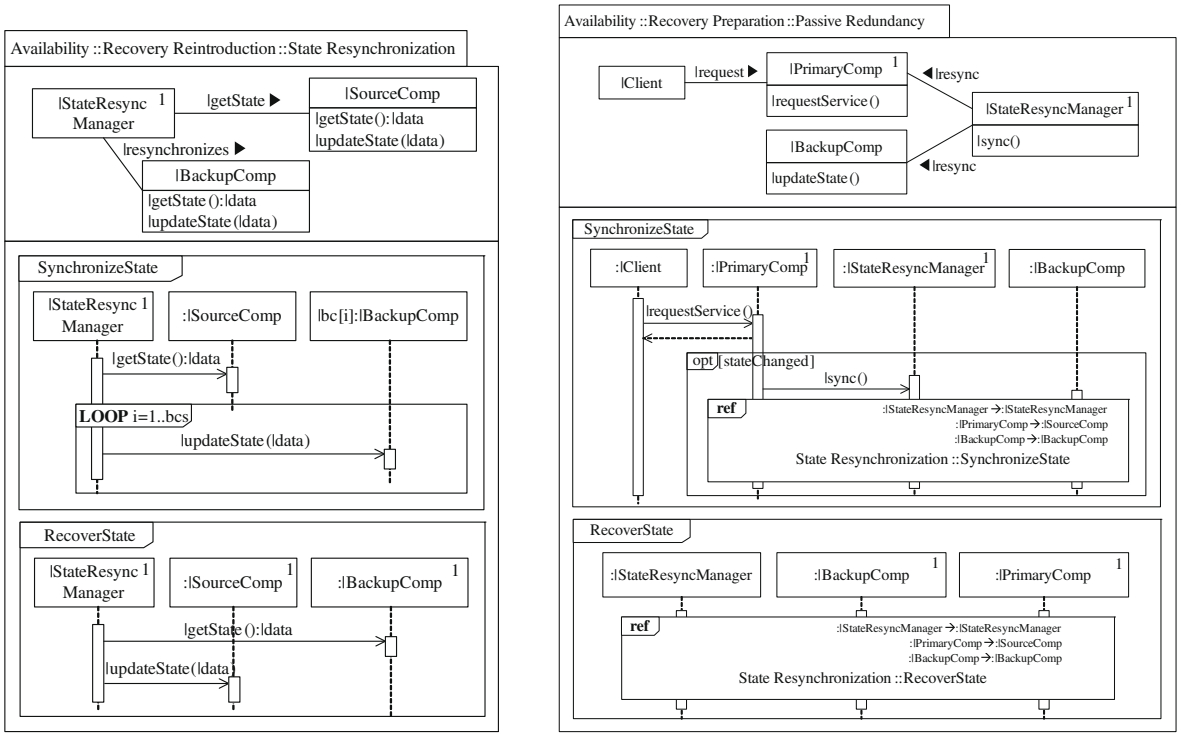
The *Recovery Reintroduction* tactic is concerned with restoring the state of a failed component. This tactic can be refined into *Checkpoint/Rollback* and *State Resynchronization* tactics. The *Checkpoint/Rollback* tactic keeps track of the state of the system either periodically or in response to a specific event. The time when the system's state is updated is called checkpoint. When a failure occurs in the system, the system is restored to the state at the most recent checkpoint before the system failed.

The *State Resynchronization* tactic restores the state of a source component through resynchronization with the state of a backup component. Fig. 5a shows the specification of the *State Resynchronization* tactic. The tactic involves concepts of a state resynchronization manager, source components and backup components. The state resynchronization manager is responsible for synchronizing the state of a source component with that of a backup component. A synchronization can occur either when the state of the source component is changed or when the source component is recovered from a failure. In the former case, the state of the backup components is synchronized with that of the source component. This is specified in the *SynchronizeState* IPS. The **LOOP** fragment specifies the state update of each backup component. On the other hand, in the latter case, the synchronization is opposite that the state of the source component is resynchronized with that of a backup component. This is specified in the *RecoverState* IPS.

The *Recovery Preparation and Repair* tactic is concerned with recovering and repairing a component from a failure. The tactic can be refined into *Voting* and *State Redundancy* tactics. The *Voting* tactic uses algorithm redundancy for recovery preparation. In the tactic, the same algorithm is run on redundant components, and the voter monitors the behavior of the components. When a component behaves differently from others, the voter fails the component. Although one component failed, the system itself still runs normal with other components.

Similar to the *Voting* tactic, the *State Redundancy* tactic uses redundant state data on multiple components for recovery preparation. There are two ways to use state redundancy. One is to select only one response from the responses that are received concurrently from redundant components for a service request. If a redundant component is failed, the tactic recovers the component by resynchronizing the state of the failed component with one of the other alive components. This is captured in the *Active Redundancy* tactic which is a subtactic of the *State Redundancy* tactic. The other way is to use the response from a specific component (primary) and inform other (backup) components to update the state of the backup components with that of the primary component. If the primary component fails, its state is resynchronized with the state of a backup component. This is captured in the *Passive Redundancy* tactic.

Fig. 5b shows the specification of the *Passive Redundancy* tactic. The tactic involves concepts of clients, a primary component, backup components and a state resynchronization manager. When the primary component receives a request from a client that causes a state change, the primary component subsequently requests the state resychronization manager for a synchronization with backup component for the change. This is specified in the *SynchronizeState* IPS. The **ref** fragment references the *SynchronizeState* IPS of the *State Resynchronization* tactic. When the primary component is recovered from a failure, the state resynchronization manager resynchronizes the primary component with that of a backup



(a) The *State Resynchronization* Tactic (b) The *Passive Redundancy* Tactic

Fig. 5. The *State Resynchronization* and *Passive Redundancy* tactics.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

component. This is specified by the *RecoverState* IPS. The **ref** fragment references the *RecoverState* IPS of the *State Resynchronize* tactic.

The RBML specifications for the *Checkpoint/Rollback*, *Voting* and *Active Redundancy* tactics are presented in Appendix A.

3.2. Tactics for performance

Performance is concerned with various aspects including processing time, response time, resource consumption, throughput and efficiency. In this work, we focus on the response time of the system for events such as interrupts, messages or user requests. Tactics for performance can be classified into resource arbitration and resource management as shown in Fig. 6.

The *Resource Arbitration* tactic is used to improve performance by scheduling requests for expensive resources (e.g., processors, networks). There are three general ways to schedule resource requests. One is to use FIFOs where requests are treated equally in the order in which they are received. This is known as the *FIFO* tactic which is defined in Fig. 7a. The *FIFO* tactic involves producers, FIFO queues and consumers which are specified by the |*Producer*, |*FIFOQueue* and |*Consumer* roles, respectively. The *Enqueue* IPS describes the behavior of enqueueing data, and the *Dequeue* IPS describes the dequeueing behavior. The *FIFO* tactic is often used for job distribution for threads and processes, especially in a network.

Another way is to assign priorities to resources, and the resource requests are scheduled based on priority. This is known to be the *Priority Scheduling* tactic. Using the *Priority Scheduling* tactic, performance can be improved by balancing the request load by priority. For example, a lower priority may be given to resources that have myriad requests. Priority can be given by assigning either a fixed priority or a dynamic priority, which is captured in the *Fixed Priority Scheduling* tactic and the *Dynamic Priority Scheduling* tactic, respectively.

The *Fixed Priority Scheduling* and *Dynamic Priority Scheduling* tactics may be complementarily used with the *FIFO* tactic to prevent the starvation issue for low priority resources. Fig. 7b shows the specification of the *Fixed Priority Scheduling* tactic. The tactic in-

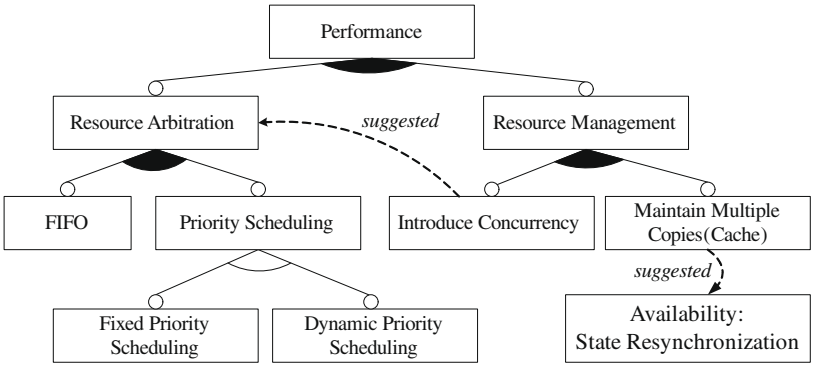


Fig. 6. Performance architectural tactic feature model.

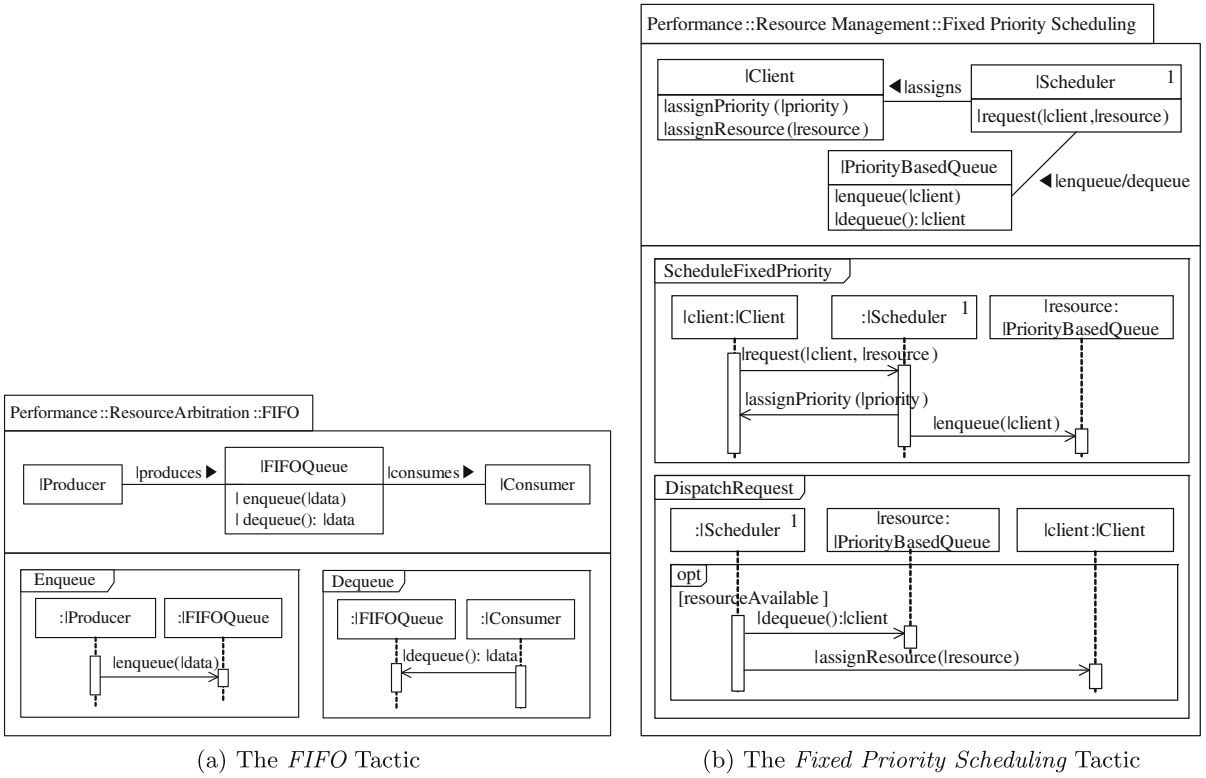


Fig. 7. The *FIFO* and *Fixed Priority Scheduling* tactic.

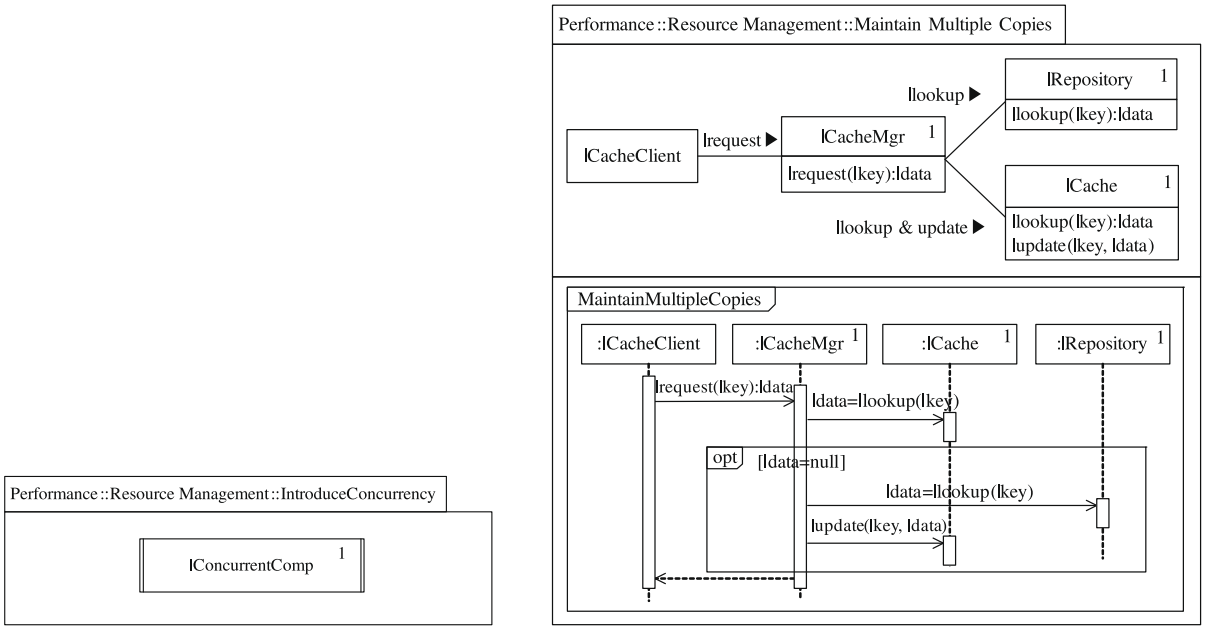
Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

volves concepts of clients, a scheduler and requests queues. When a client requests for a resource, the scheduler assigns a fixed priority to the client based on a specific strategy (e.g., semantic importance, deadline monotonic) and then enqueues the request with the priority to the queue associated with the requested resource. This is specified in the *ScheduleFixedPriority* IPS. When the requested resource is available, the scheduler dequeues a request that has the highest priority and assigns the resource to the client. This is specified in the *DispatchRequest* IPS.

In the *Dynamic Priority Scheduling* tactic, priorities are determined at runtime based on execution parameters such as upcoming deadlines or other runtime conditions. Common policies used for determining priority are round robin and earlier deadline first (Silberschatz et al., 2005). To support the dynamic assignment of priority, the *Dynamic Priority Scheduling* tactic involves the behavior of preempting a resource for a higher priority request and releasing the assigned resource from a request that has a lower priority. The specification of the *Dynamic Priority Scheduling* tactic is presented in Appendix A.

The *Resource Management* tactic improves performance by managing the resources that affect response time. One way for managing resources is to allocate threads or processes to resources for concurrent execution. In this way, waiting time in response can be significantly reduced. This is known to be the *Introduce Concurrency* tactic which is specified in Fig. 8a. The |*ConcurrentComp* role in the tactic captures concurrent components that have their own thread or process. The double lines in the components denote concurrency. The *Introduce Concurrency* tactic is often used with a resource arbitration tactic for concurrent communication among threads or processes, which is specified by the *suggested* relationship in Fig. 6.

Another way for managing resources is to keep replicas of resources on separate repositories, so that contention for resources can be reduced. This is called *Maintain Multiple Copies* tactic shown in Fig. 8b. The tactic involves concepts of clients, a cache, a cache manager and a data repository. The cache maintains copies of data that are frequently requested for faster access. When a data request is received, the cache manager first searches the cache. If the data is not found, then the cache manager looks up the repository and makes copies of the data into the cache. The *Maintain Multiple Copies* IPS in Fig. 8b shows the searching behavior. An important issue in this tactic is to maintain the consistency of



(a) The *Introduce Concurrency* Tactic (b) The *Maintain Multiple Copies(Cache)* Tactic

Fig. 8. The *Introduce Concurrency* and *Maintain Multiple Copies (Cache)* tactics.

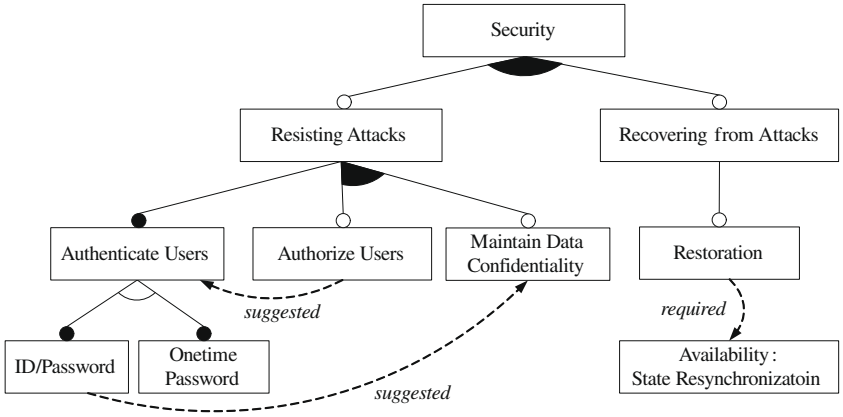


Fig. 9. Security architectural tactic feature model.

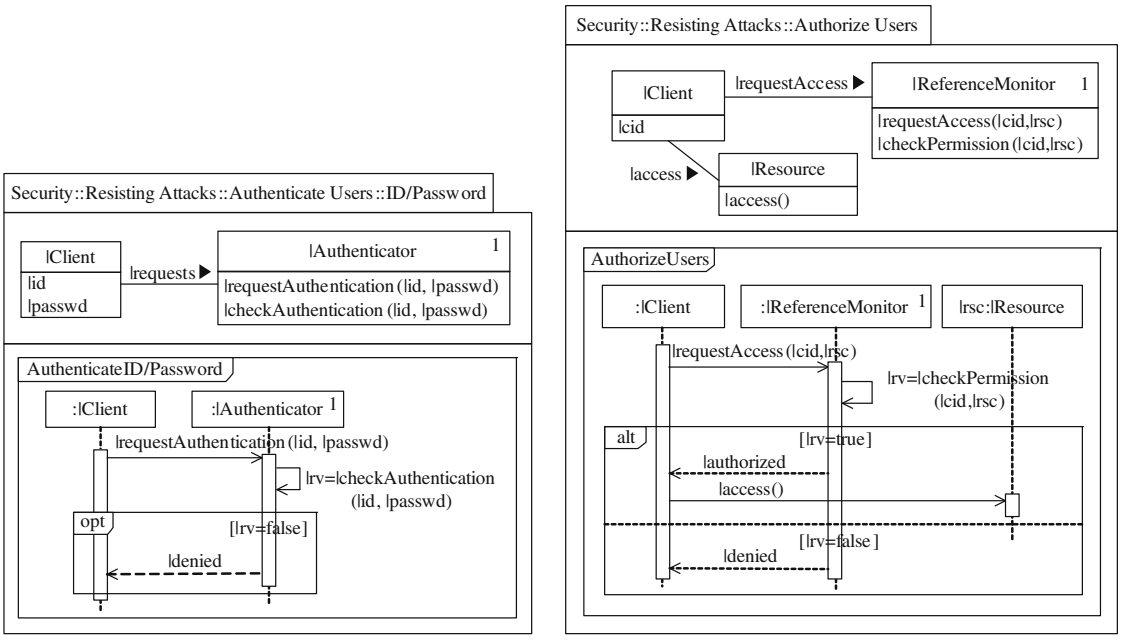
Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

the copies and keep them synchronized. This issue can be addressed by using the *State Resynchronization* tactic in Section 3.1.

3.3. Tactics for security

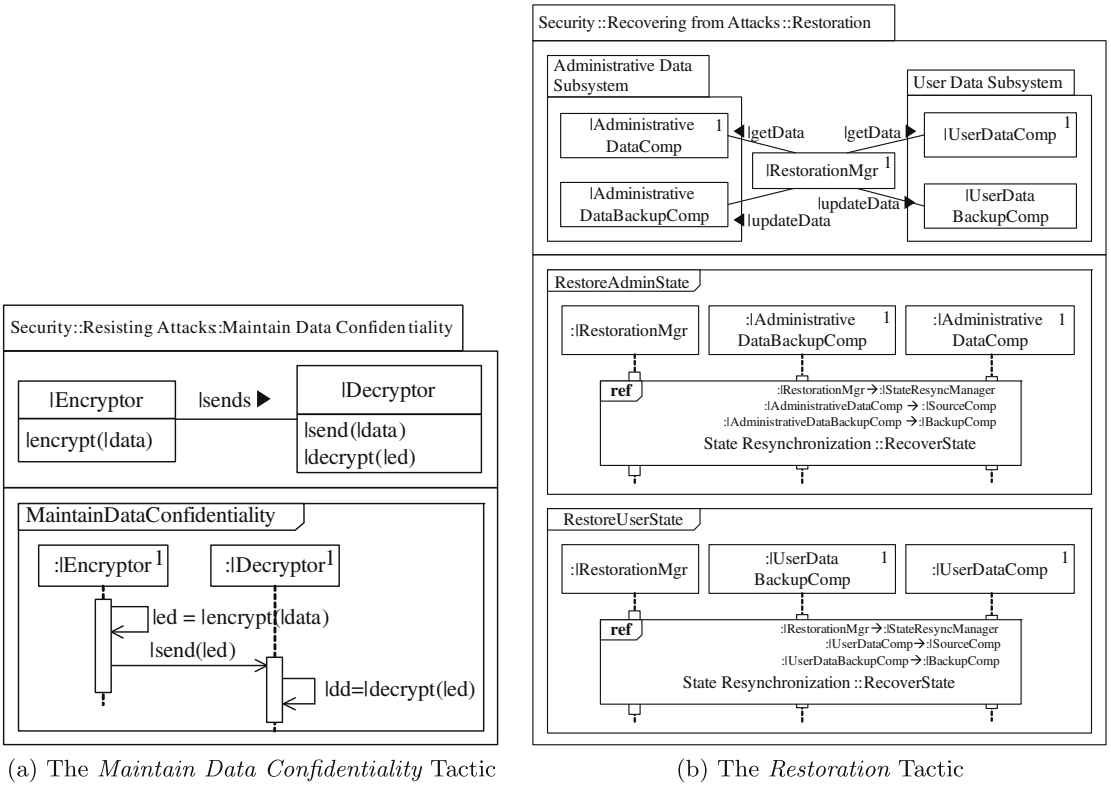
Security is concerned with preventing unauthorized usage of the system while providing its services to legitimate users. Various tactics have been introduced for security, and they can be classified into *Resisting Attacks* and *Recovering from Attacks* as shown in Fig. 9.

The *Resisting Attacks* tactic provides several ways of protecting the system from attacks. One way is to check authentication of the user using the user's credentials (i.e., user IDs, passwords) which is in general considered to be an incipient protection. User credentials can be either set by the user or generated by the system every time the user uses the system. The former is known to be the



(a) The *ID/Password* Tactic (b) The *Authorize Users* Tactic

Fig. 10. The *ID/Password* and *Authorize Users* tactics.



(a) The *Maintain Data Confidentiality* Tactic (b) The *Restoration* Tactic

Fig. 11. The *Maintain Data Confidentiality* and *Restoration* tactic.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

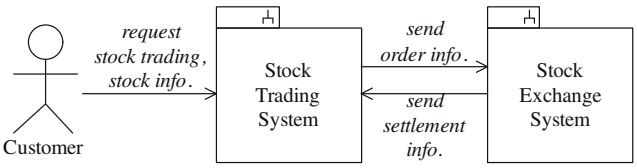


Fig. 12. A stock trading system.

ID/Password tactic, and the latter to be the *Onetime Password* tactics. Fig. 10a shows the specification of the *ID/Password* tactic. The *AuthenticateID/Password* IPS describes that an authentication request is denied if the user's credentials are not valid. In the *ID/Password* tactic, a possible security breach could occur if the userID and password are exposed by an attack such as snooping (Bass et al., 2003). To prevent this, the *Onetime Password* tactic can be used. In the tactic, the authenticator generates a secure value when

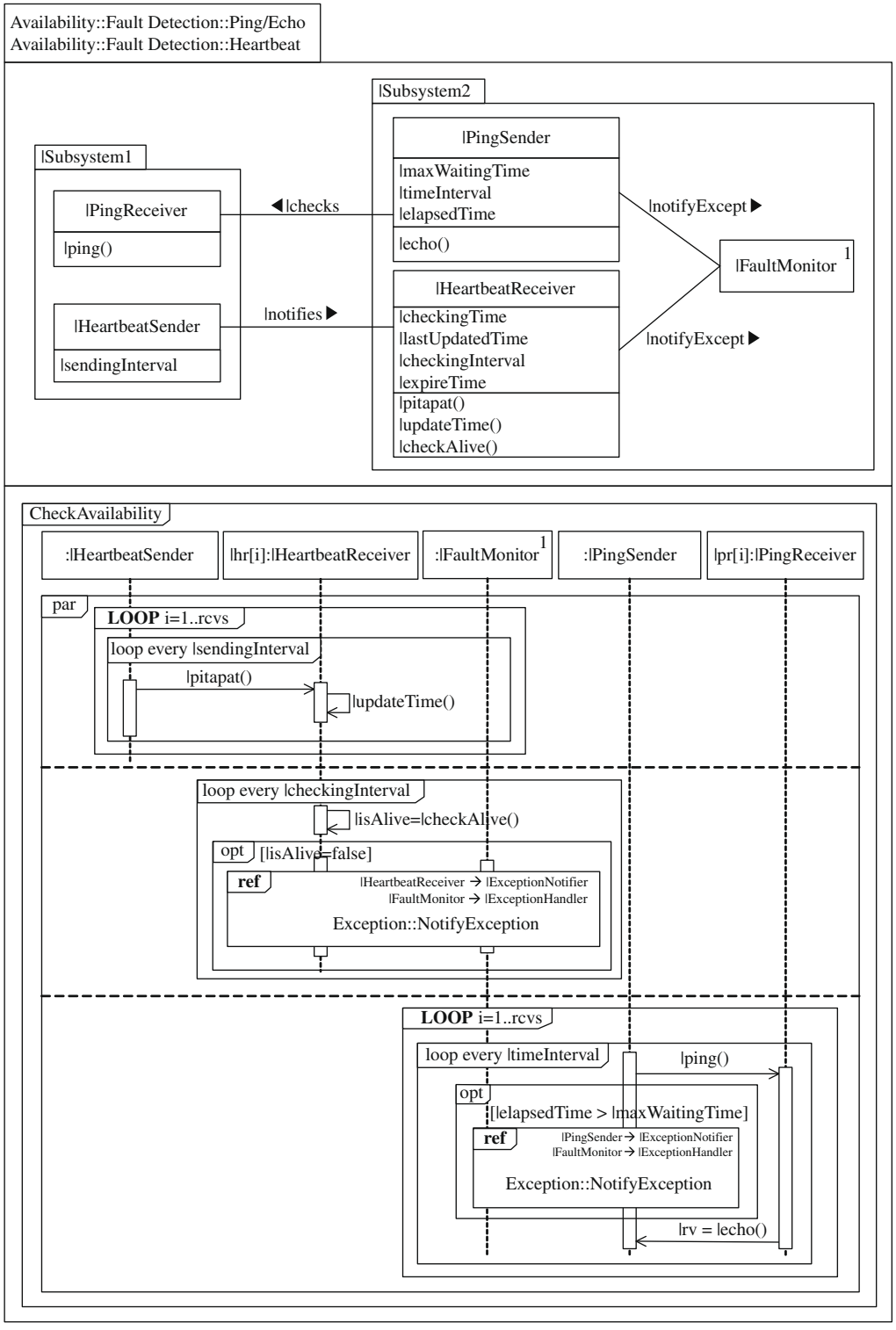


Fig. 13. The composition of the *Ping/Echo* and *Heartbeat* tactics.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

an authentication request is received, and shares the value with the client. Then, the client and authenticator use the same secure value to generate the same password (Cole et al., 2005). In this tactic, the client and authenticator are assumed to have the same algorithm for generating a password. The specification of the *One-time Password* is presented in Appendix A.

Another way of protecting a system is to restrict user access to data and services. This is known as the *Authorize Users* tactic. The tactic involves concepts of clients, resources and a reference monitor as shown in Fig. 10b. The *AuthorizeUsers* IPS in the figure describes that an access request is intercepted by the reference monitor to check permission. If a permission is found, the request is allowed, otherwise it is denied. This behavior can be more concretized for a specific access control model such as DAC, MAC or RBAC (Kim et al., 2006). In many cases, users are assumed to have already been authenticated before they request for access to resources. This means that the *Authenticate Users* tactic is often used together with the *Authorize Users* tactic, which is specified by the *suggested* relationship between the *Authenticate Users* and *Authorized Users* tactics in Fig. 9.

The *Maintain Data Confidentiality* tactic protects data from unauthorized modifications using encryption and decryption. Fig. 11a shows the specification of the *Maintain Data Confidentiality* tactic. The *MaintainDataConfidentiality* IPS in the figure describes that an encryptor encrypts data before sending, and a decryptor decrypts once the encrypted data is received. This tactic is often used together with the *ID/Password* tactic to protect the login information.

While the *Resisting Attacks* tactic is concerned with protecting a system, the *Recovering from Attacks* tactic helps to restore a system in a security regard. One way to restore a system is to maintain administrative data, which is critical for security, separately from user data and restore them separately. In this way, administrative data can be better protected, and so is the system. This is known as the *Restoration* tactic which is defined in Fig. 11b. Similar to the *State Redundancy* availability tactic in Section 3.1, the *Restoration* tactic uses state resynchronization, but separately for administrative data and user data as shown in the *RestoreAdminState* and *RestoreUserData* IPSs.

4. Case example: Stock trading system

In this section, we demonstrate how the availability, performance and security tactics presented in Section 3 can be used to embody NFRs of a stock trading system (STS) into its architecture. The STS is an online stock trading system that provides real-time services for checking the current price of stocks, placing buy or sell orders and reviewing traded stock volume. It sends orders to the stock exchange system (SES) for trading and receives the settlement information from the SES. The system can also trade options and futures. Fig. 12 shows a context diagram of the STS.

In addition to the functional requirements, the system has the following non-functional requirements for availability, performance and security:

- **NFR1.** The STS should be available during the trading time (7:30 AM–6:00 PM) from Monday through Friday. If there is no response from the SES for 30 s, the STS should notify the administrator.
- **NFR2.** The system should be able to process 300 transactions per second 400,000 transactions per day. A client may place multiple orders of different kinds (e.g., stocks, options, futures), and the orders should be sent to the SES within 1 s in the order they were placed.
- **NFR3.** The SES sends the trading information of about 600 items every second on average to the STS. Updating such a high volume of information imposes an intensive load on the STS's database, which may cause slow performance. In order to minimize the impairment of performance, updates should be the least possible.
- **NFR4.** Only authenticated users can access the system. The user credentials must not be seen to unauthorized personnel while transmitting the information to the system.

4.1. Tactic configuration

In this subsection, we describe how each of the above four NFRs can be embodied into architecture using architectural tactics. We assume that tactic selection is made by the architect based on his/her experience:

- **Configuring Tactics for NFR1.** NFR1 is concerned with high availability of the system during trading time. To support this, the *Ping/Echo* and *Heartbeat* tactics in Section 2.1 can be used together. Using the *Ping/Echo* tactic, the availability of the STS

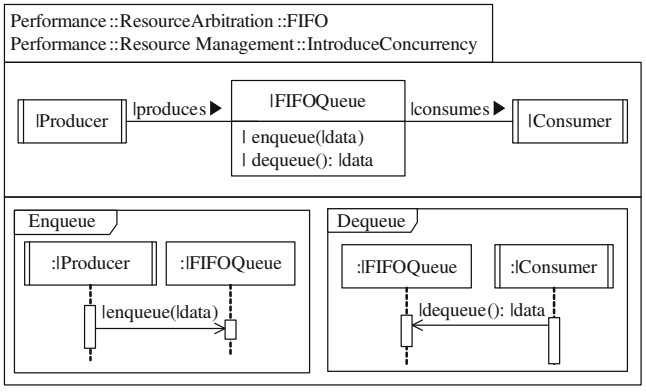


Fig. 14. The composition of the *FIFO* and *Introduce Concurrency* tactics.

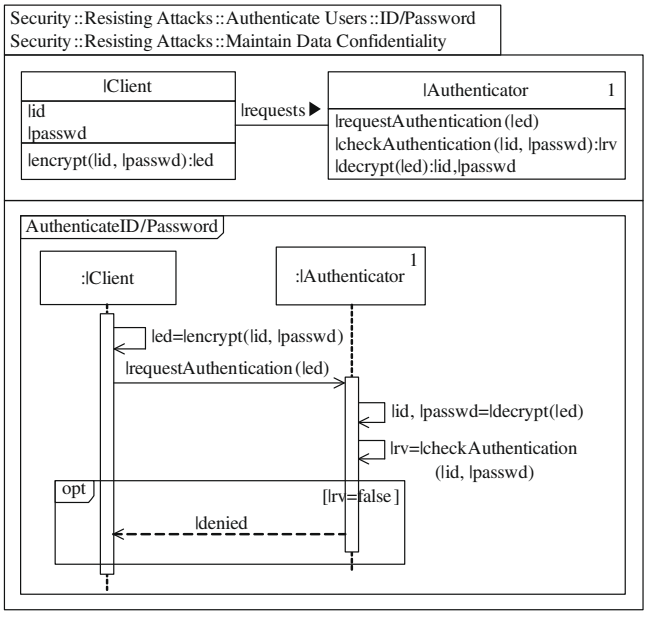


Fig. 15. The composition of the *ID/Password* and *Maintain Data Confidentiality* tactics.

can be checked by sending a ping message regularly and listening to an echo back. However, in some cases, the ping sender may receive an echo even if the STS is failed (Abawajy, 2004; Blazewicz et al., 2007; Stelling et al., 1999). To detect such a case, the *Heartbeat* tactic may be used together, which can be found in some commercial and open-source applications such as WebLogic Application Server and JBoss. Using the *Heartbeat* tactic, the failure of the STS can be detected by listening to heartbeat messages from the STS. Even if an echo is received from the STS, the STS is considered to be failed if there is no heartbeat message coming from the STS. The combined use of the two tactics also helps early detection of a typical failure by sending a ping message during the latency time between heartbeat messages.

Two tactics can be composed based on sets of binding rules and composition rules. Binding rules define the corresponding roles in the two tactics, while composition rules define changes to be made during composition. The following defines the binding rule for the composition of the *Ping/Echo* and *Heartbeat* tactics:

B1. *Ping/Echo* :: |*FaultMonitor* ↦ *Heartbeat* :: |*FaultMonitor*

This rule describes that the |*FaultMonitor* role in the *Ping/Echo* tactic corresponds to the |*FaultMonitor* role in the *Heartbeat*

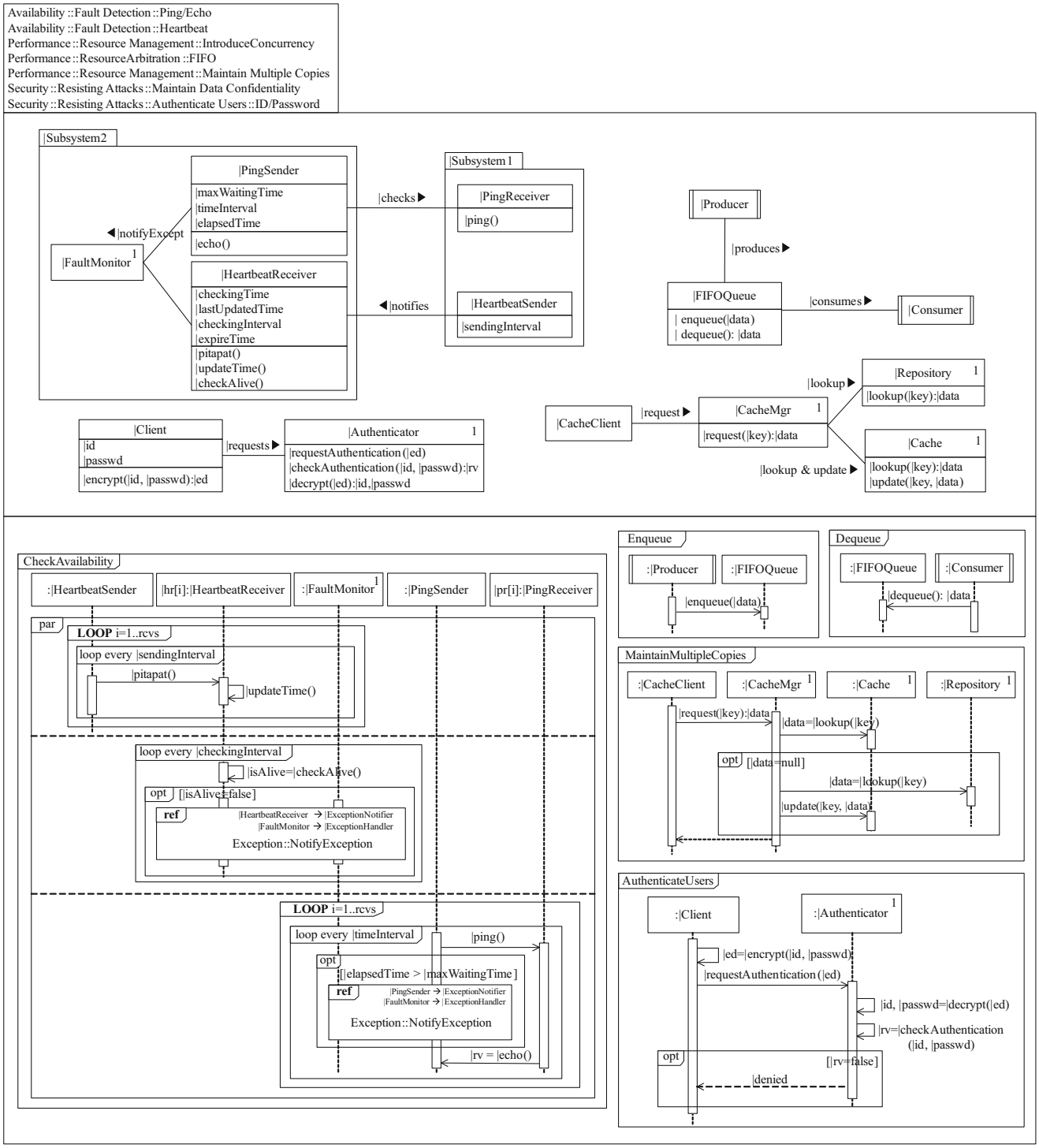


Fig. 16. The consolidated tactic for the NFRs of the STS.

tactic. Thus, the two |*FaultMonitor* roles are merged in the composed tactic. Given the binding rule, the following composition rules are defined:

SPS_C1. Put the |*PingReceiver* and |*HeartbeatSender* roles into the same package role and name the package role |*Subsystem*1.

SPS_C2. Put the |*PingSender*, |*HeartbeatReceiver* and |*FaultMonitor* roles into the same package role and name the package role |*Subsystem*2.

IPS_C3. Create a parallel fragment of three operands.

IPS_C4. Add the behavior of the two parallel operands of the *CheckHeartbeat* tactic into the first two operands of the new parallel fragment.

IPS_C5. Add the behavior of *CheckPing/Echo* tactic into the last operands of the new parallel fragment.

IPS_C6. Name the composed IPS *CheckAvailability*.

SPS_C1 and SPS_C2 defines the composition rules for SPSs. SPS_C1 specifies that ping receivers and heartbeat senders should be deployed in the same subsystem (a monitored subsystem). Similarly, SPS_C2 specifies that ping senders, heartbeat receivers and the monitor should be deployed in the same subsystem (a monitoring subsystem). This enables the monitoring subsystem to both send ping messages and receive heartbeat messages. IPS_C3 to IPS_C6 describe the composition rules for IPSs. These rules specify that the behavior of the *Ping/Echo* tactic and the *Heartbeat* tactic should be executed concurrently for

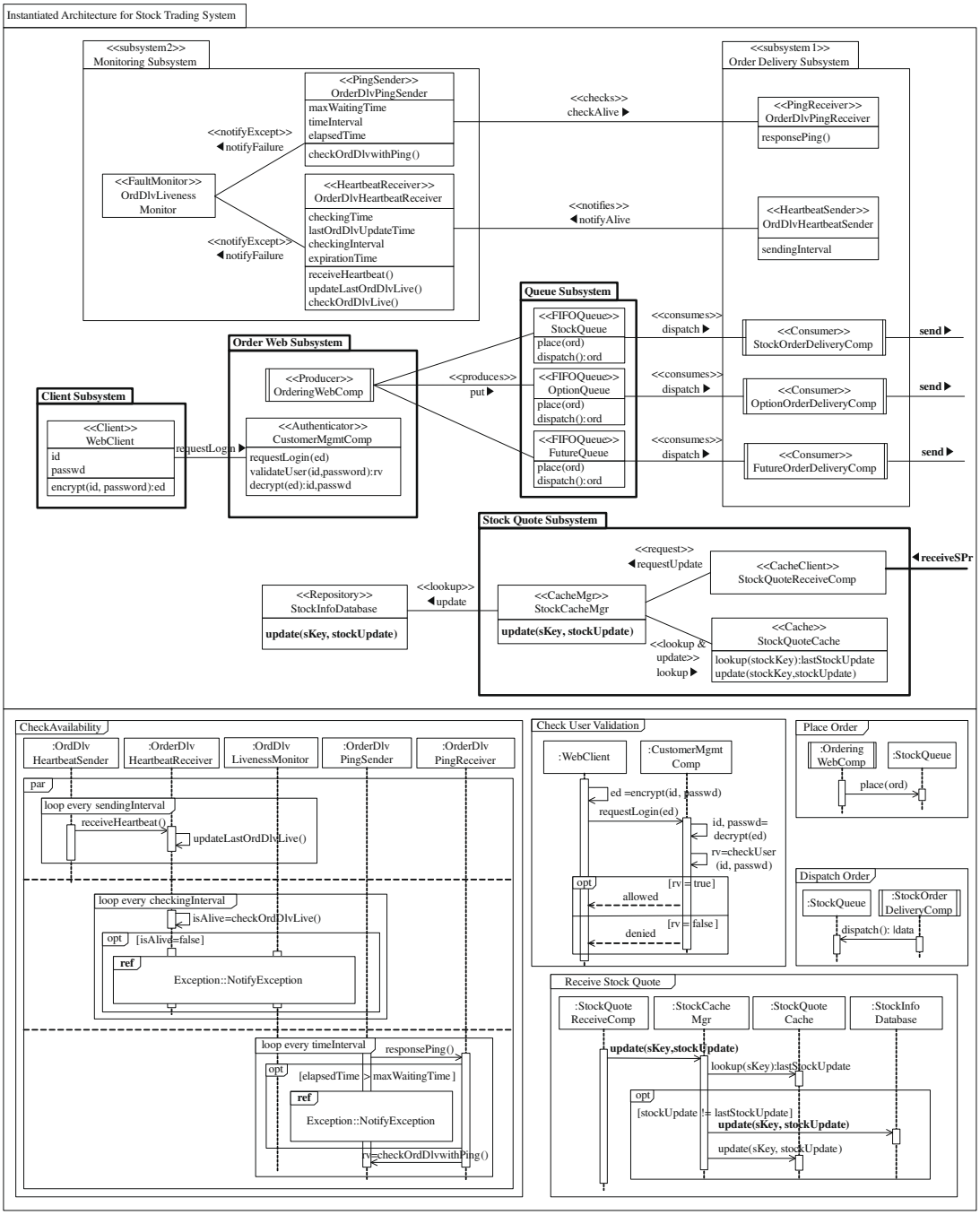


Fig. 17. Instance 1: STS architecture.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

rigorous monitoring. Fig. 13 shows the composed tactic resulting from the application of the above rules.

- **Configuring Tactics for NFR2.** NFR2 is concerned with the performance of the STS, requiring handling considerable amount of transactions by their kinds within a very short time. NFR2 can be supported by reducing the blocking time of transactions on I/O, which can be realized by combined use of the *FIFO* and *Introduce Concurrency* tactics. Using the *FIFO* tactic, each type of orders can be assigned to a dedicated FIFO queue instance for immediate processing. Use of the *Introduce Concurrency* tactic with the *FIFO* tactic can further improve the performance by adding a thread or process to the STS interface ports for order delivery to the SES. This improves the performance by concurrent dispatching of the same kind of orders. The two tactics can be composed based on the following binding rules:

  B1. *IntroduceConcurrency* :: |*ConcurrentComp* ↦ *FIFO* :: |*Producer*

  B2. *IntroduceConcurrency* :: |*ConcurrentComp* ↦ *FIFO* :: |*Consumer*

  These rules allocate a thread to each of FIFO producers and consumers, making it concurrent. By applying the binding rules, the *FIFO* and *IntroduceConcurrency* tactics are composed as shown in Fig. 14.
- **Configuring Tactics for NFR3.** NFR3 is concerned with the performance of the STS's database which may be decreased by the intensive updates from the SES. The requirement states that the STS receives about 600 items per second. In general, when updates are received at such a high frequency, some items (e.g., stocks having less trades) may not have changes in every update. Taking into account this, many systems use caches to filter out the actual items that need to be updated by comparing the received update with the previous update. To support such selective updates, the *Maintain Multiple Copies* tactic can be used, introducing a cache, a cache client and a cache manager. Using the tactic, we have the cache client receive the update from SES, instead of the database. The cache client then requests the cache manager to update the received update. The cache manager looks up the previous update in the cache and compare it with the one that is received and identify the items that have changes. Only those items that have changes are updated in the database, which reduces the update load on the database.
- **Configuring Tactics for NFR4.** NFR4 is a common security requirement for most login-based systems. To support the confidentiality of login information, it is obvious that the *ID/Password* and the *Maintain Data Confidentiality* tactics should be used. Through the *ID/Password* tactic, only authenticated users are able to login the system. The encryption/decryption mechanism of the *Maintain Data Confidentiality* tactic prevents unauthorized access to the user information during transmission. The following defines the binding rules for composing the two tactics:

  B1. *MaintainDataConfidentiality* :: |*Encryptor* ↦ *ID/Password* :: |*Client*

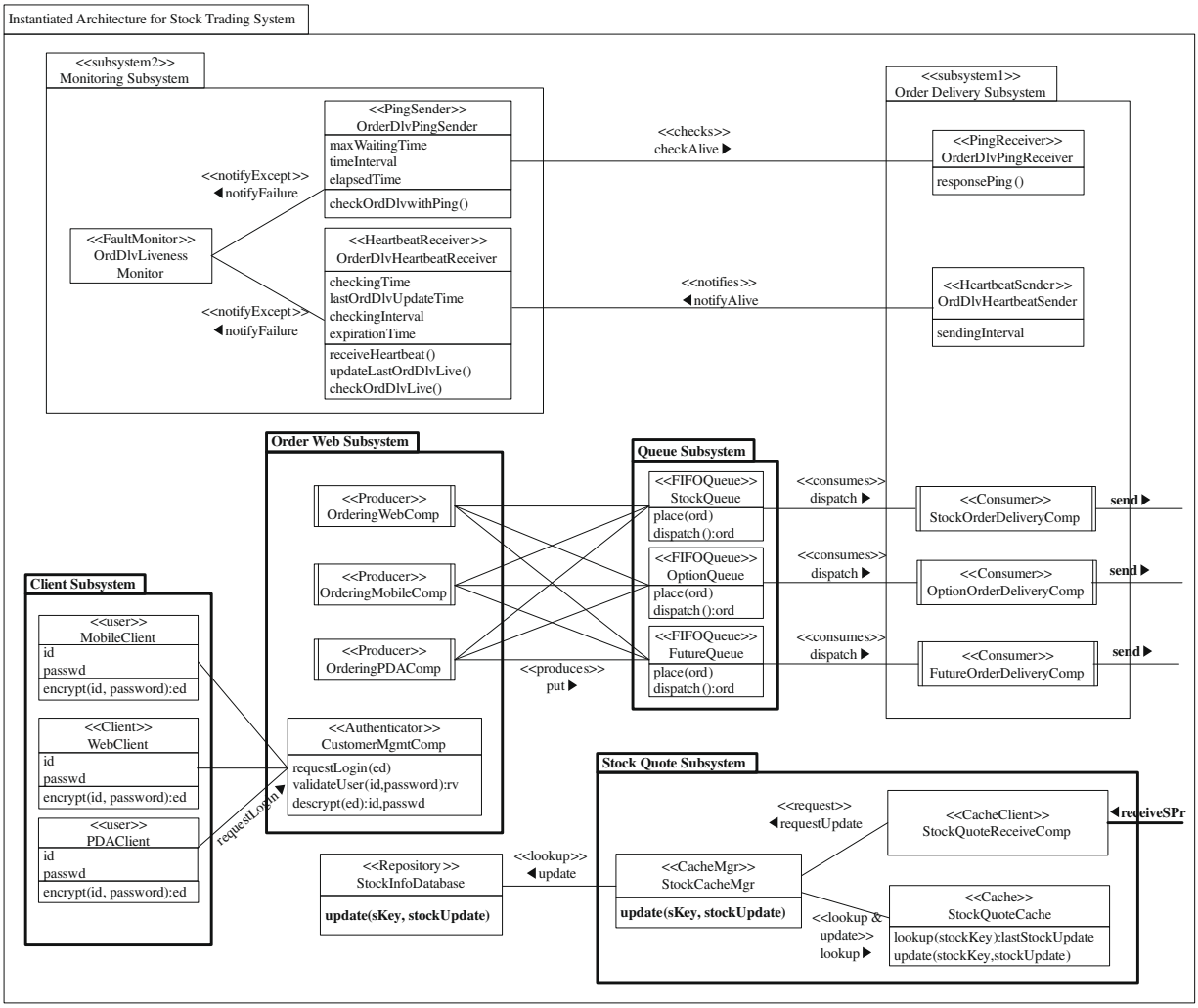


Fig. 18. Instance 2: STS architecture.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

B2. *MaintainDataConfidentiality* :: |*Decryptor* ↦ *ID*/*Password* :: |*Authenticator*

B3. *MaintainDataConfidentiality* :: |*data* ↦ *ID*/*Password* :: |*id*, |*passwd*

The binding rules specify that the client requesting for authentication is responsible for encrypting the user ID and password, and the authenticator should be able to decrypt the login information. Given the binding rules, the following composition rules are defined:

SPS_C1. Merge the |*Encryptor* role with the |*Client* role, and name the merged role *Client*.

SPS_C2. Add parameter roles |*id* and |*passwd* to the |*encrypt*() behavioral feature role in the merged |*Client* role.

SPS_C3. Merge the |*Decryptor* role with the |*Authenticator* role, and name the merged role *Authenticator*.

IPS_C4. Merge the *MaintainDataConfidentiality* IPS with the *AuthenticateID/Password* IPS, and name the merged IPS *AuthendicateID/Password*.

IPS_C5. Place the |*encrypt*() message role before the |*requestAuthentication*() role in the merged *AuthenticateID/Password* IPS.

IPS_C6. Place the |*decrypt*() message role between the receipt of the *requestAuthentication* and |*checkAuthentication*() roles in the merged *AuthenticateID/Password* IPS.

As the new parameter roles are added in SPS_C2 and SPS_C3, the same changes should be made to the corresponding message roles in the merged IPS. Fig. 15 shows the resulting tactic of composing the *Maintain Data Confidentiality* and *ID/Password* tactics.

In NFR4, the security concern is limited to secure login which has little impact on performance. However, if every order transaction is required to be encrypted, encryption may cause a non-trial overhead which may have impact on performance. This implies that security and performance may have conflicts, and a trade-off analysis of the two is necessary to minimize conflicts.

The three composed tactics in Figs. 13–15 and the *Maintain Multiple Copies* tactic addressing NFR3 are consolidated to build an initial architecture that embodies the NFRs of the STS via instantiation. Fig. 16 shows the consolidation of the composed tactics.

The composition rules for other possible tactic configurations can be found in Appendix B. The tactic configurations that are not presented in Appendix B are simple combinations (i.e., no composition is involved).

4.2. Building architectures

Given the tactic consolidation in Fig. 16, an initial architecture for the STS can be instantiated. Fig. 17 shows an instantiated architecture where the stereotypes represent the roles from which the architectural elements are instantiated. For example, the stereotype ≪ *PingSender* ≫ in the *OrderDlvPingSender* component denotes that the component is instantiated from the |*PingSender* role in Fig. 16.

Instantiating an architecture involves making certain architectural decisions such as the number of components to be instantiated or grouping instantiated components. The number of components to create is specified in the realization multiplicity of tactic roles. For instance, in Fig. 17, there are three instances (*StockQueue*, *OptionQueue*, *FutureQueue*) of the |*FIFOQueue* role, each

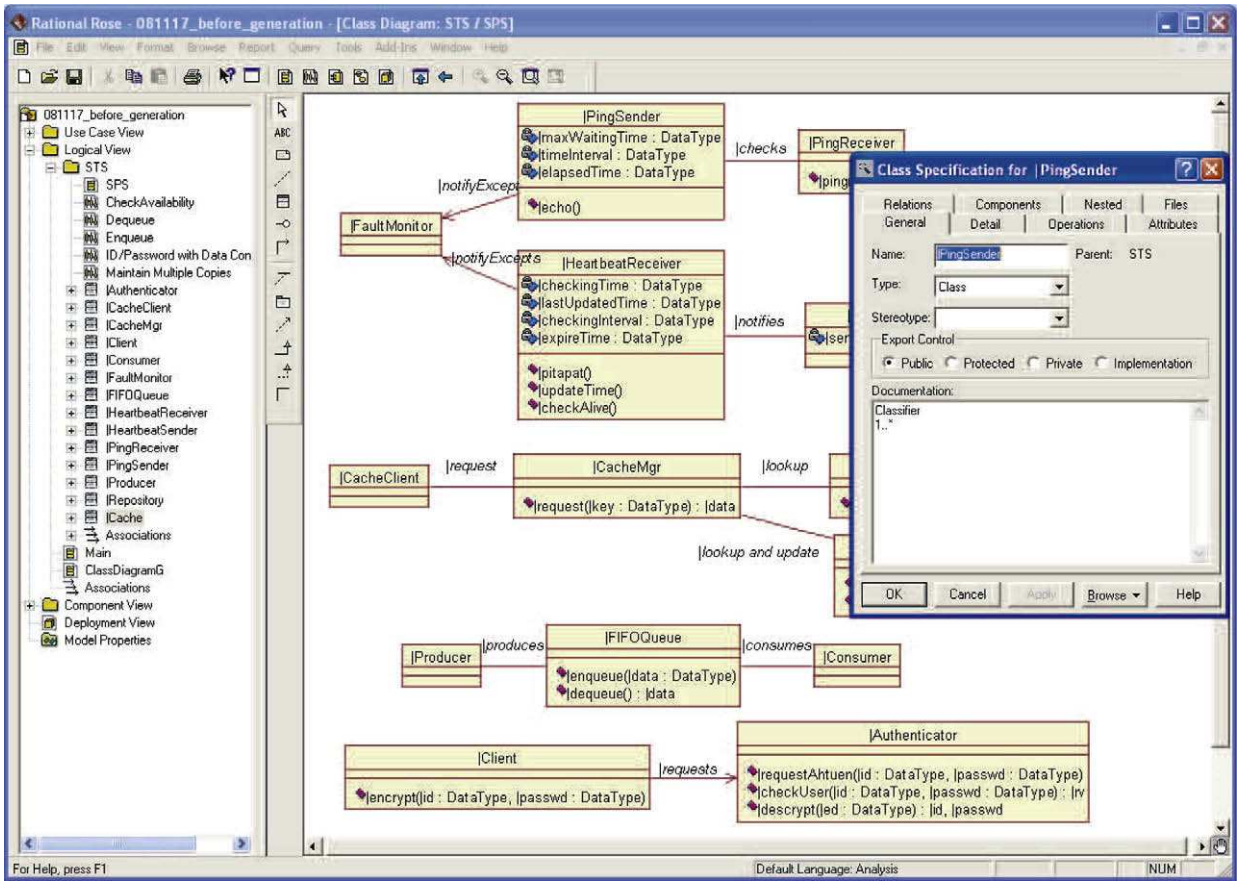


Fig. 19. The consolidated tactic modeled in Rose.

for one kind of orders. They are created by setting the lower bound of the realization multiplicity of the |*FIFOQueue* role to 3. Also, certain decisions for grouping instantiated components may be made. In Fig. 17, the packages in bold line show the grouping decisions made. For example, the *OrderingWebComp* and *CustomerMgmtComp* components are grouped to be deployed in the same subsystem named *Order Web Subsystem*. Such a decision could be strategic or due to environmental constraints. The *Client Subsystem* package having only one component implies that the subsystem is to be deployed physically separate on the network. The *StockCacheMgr*, *StockQuoteReceiveComp* and *StockQuoteCache* components (which are instances of the |*CacheMgr*, |*CacheClient* and |*Cache* roles of the *Maintain Multiple Copies* tactic) are grouped to be a cache subsystem which is responsible for receiving the update from the SES. The *update()* operation in the *StockCacheMgr* component is an instance of the |*request*() role in the |*CacheMgr* role, tailored with the received update passed as a parameter and no return. A similar tailoring is made for the *update()* operation in the *StockInfoDatabase* component. The tailored operations are denoted in bold. It should be noted that the tailoring and the architectural decision made during instantiation are specific to the STS, and thus the tactics themselves should not be considered to be underspecified for not addressing them.

In summary, (1) the *Order Delivery Subsystem* with the *Monitoring Subsystem* embodies NFR1, and (2) the *OrderingWebComp* component, the three dedicated queues (*StockQueue*, *OptionQueue*, *FutureQueue*) and the three concurrent delivery components (*StockOrderDeliveryComp*, *OptionOrderDeliveryComp*, *FutureOrderDeliveryComp*) embody NFR2, and (3) the *Stock Quote Subsystem* with the *StockInfoDatabase* component embodies NFR3, and (4) finally, the *WebClient* component together with the *CustomerMgmtComp* component embodies NFR4.

Fig. 18 shows another instantiation designed to support various types of clients such as mobile clients, web clients and PDA clients. This involves creating three instances of the |*Client* role as shown in Fig. 18. As the platforms of clients are different, so are the formats of order information. This necessitates having a separate ordering component for each type of formats, which results in creating three instances of the |*Producer* role (*OrderingWebComp*, *OrderingMobileComp*, *OrderingPDAComp*), each of which is associated with the three kinds of stock queues as shown in Fig. 18.

5. Tool support

In this section, we demonstrate tool support for automatic instantiation of architectural tactics. We use RBML Pattern Instantiator (RBML-PI) which was developed in our previous work to support instantiation of RBML specifications (Kim and Whittle, 2005). RBML-PI is developed as an add-in component to IBM Rational Rose. We use RBML-PI to demonstrate how the consolidated tactic in Fig. 16 can be automatically instantiated to generate the architecture instance given in Fig. 18.

Instantiating an architectural tactic using the RBML-PI involves the following steps:

1. Model the architectural tactic in Rose. Architectural tactics are specified by the RBML. Since the RBML is UML-based, any UML modeling tool that bears minor notational variations (e.g., the bar symbol for roles) can be used.

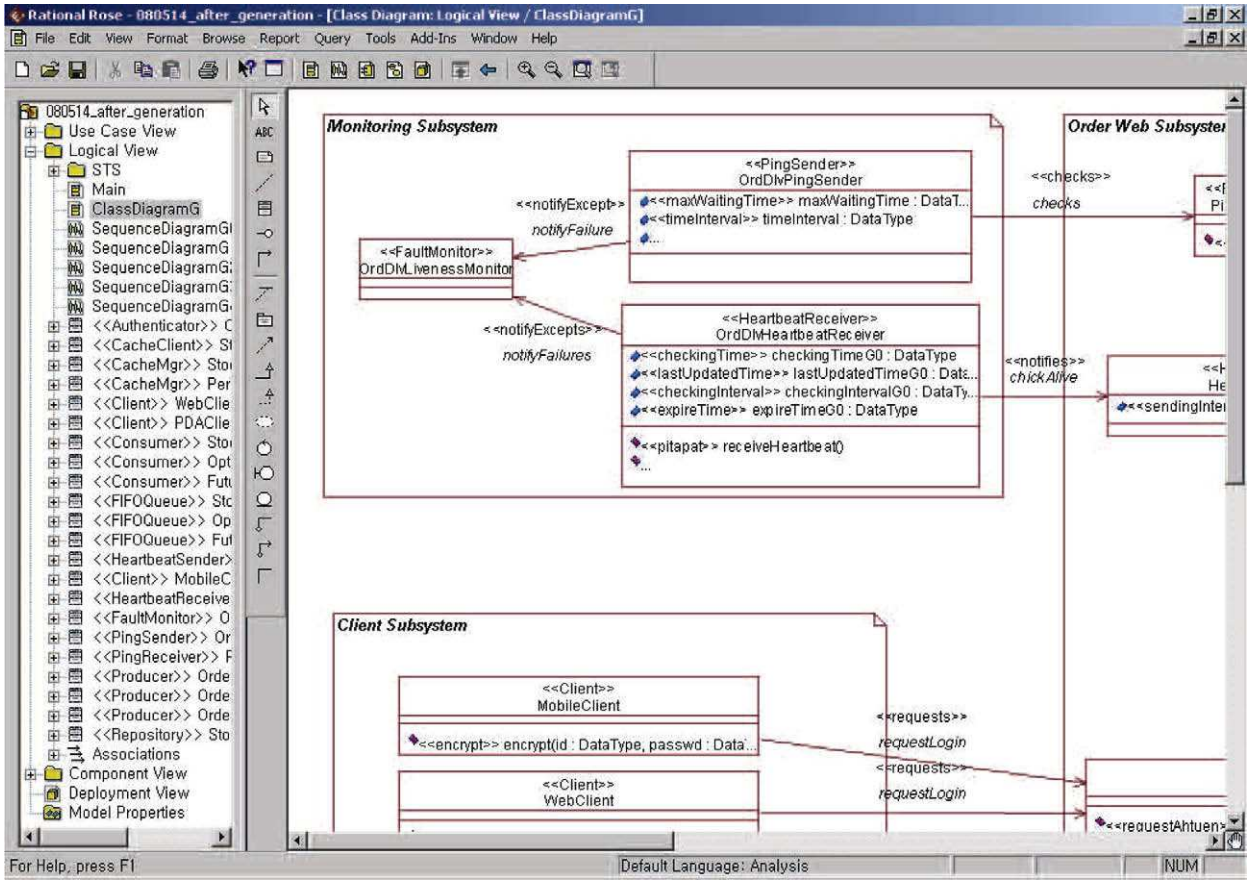
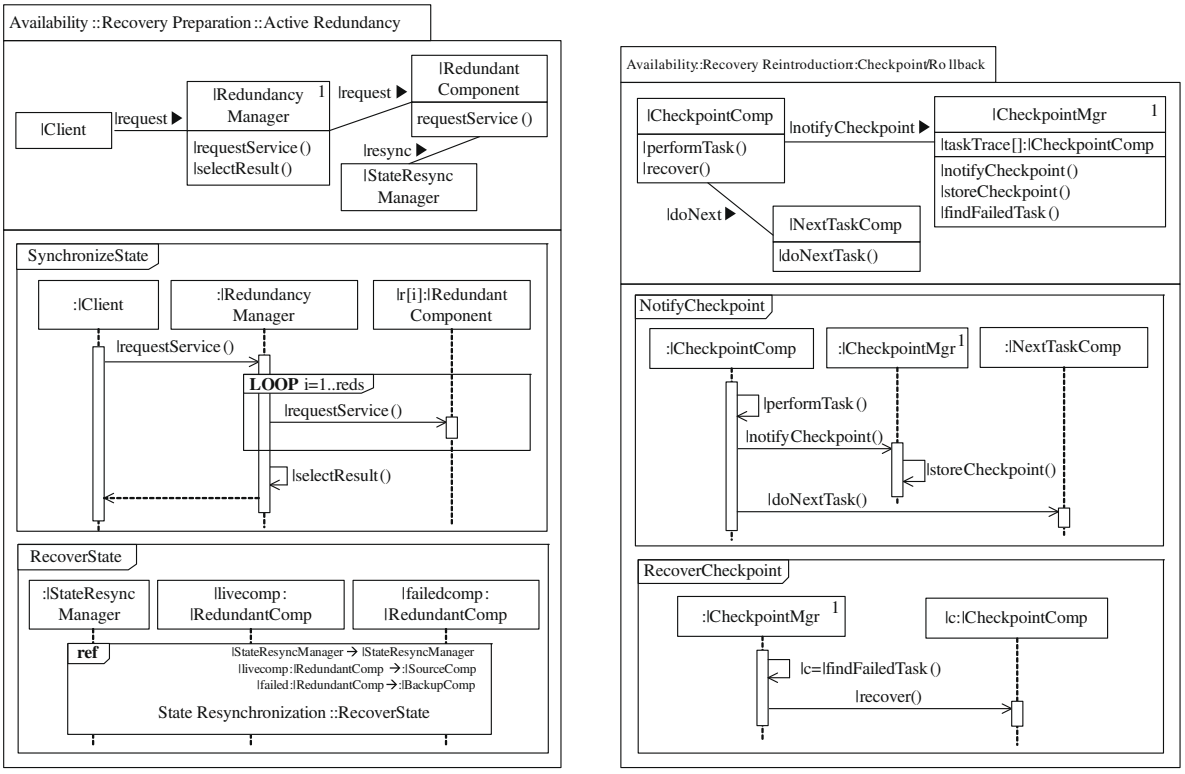


Fig. 20. The architecture in Fig. 18 instantiated using RBML-PI.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

2. Determine the number of instances to create for the roles in their the realization multiplicity.
3. Name the instantiated elements specific to the context of the application.
4. Group the instantiated components according to an architectural strategy if any.
5. Tailor the instantiated elements specific to the application as needed.

By carrying out Step 1, the consolidated tactic in Fig. 16 is drawn in Rose as shown in Fig. 19. In the figure, the tree in the left compartment shows a list of the roles involved in the consolidated tactic, and the right compartment shows the SPS of the consolidated tactic. The window appeared on the right shows an example of the properties of a role specified in Rose, which is carried out by Step 2. The window shows that the base metaclass and the realization multiplicity of the |*PingSender* role is set, respectively, to the



(a) The *Active Redundancy* Tactic

(b) The *Checkpoint/Rollback* Tactic

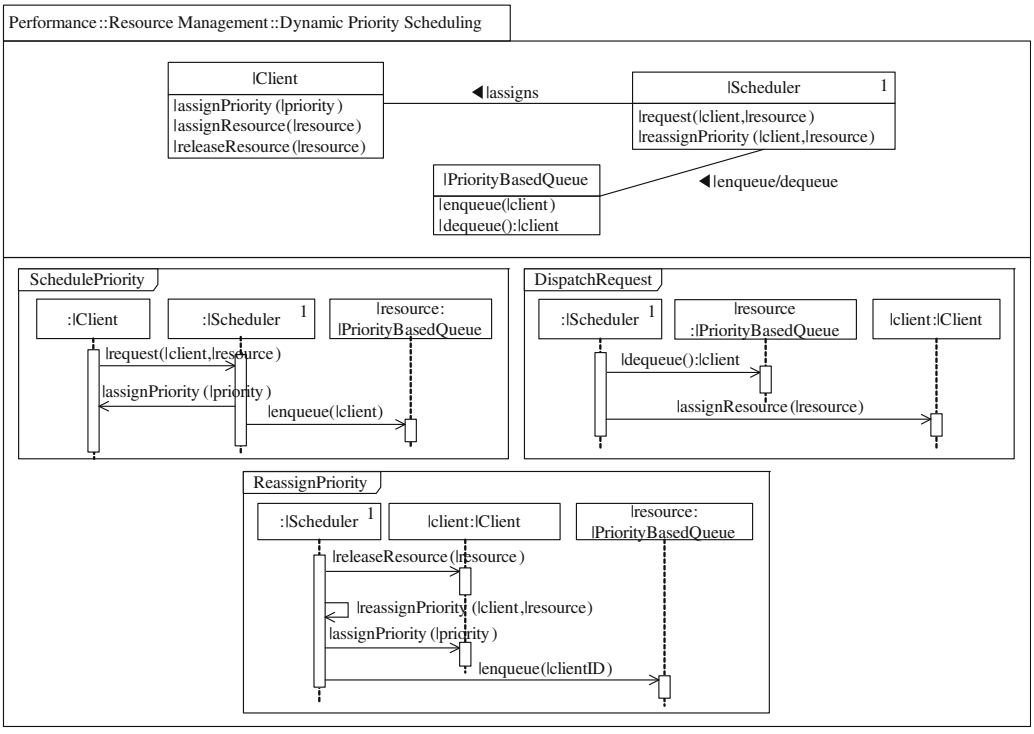
(c) The *Voting* Tactic

Fig. 21. The *Active Redundancy*, *Checkpoint/Rollback* and *Voting* tactic for availability.

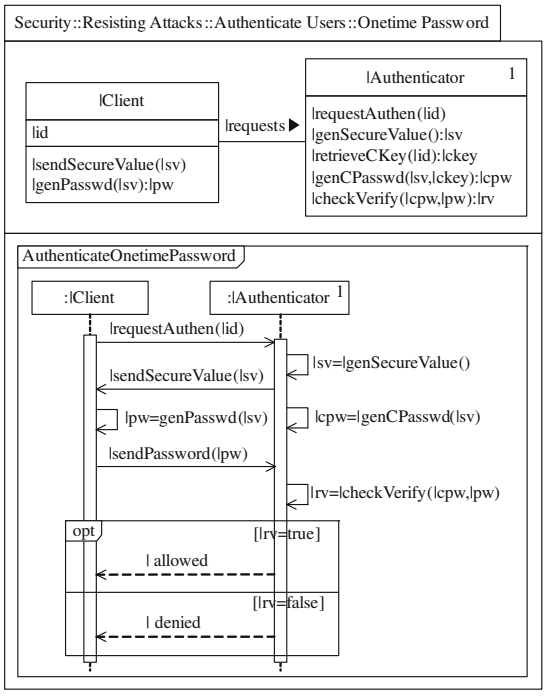
Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

Classifier metaclass and *1..**. The metaclass constrains that the type of instances of the role must be *Classifier*, and the lower bound of the multiplicity constrains that there must be at least one instance of the role. This results in creating one class for the role as its instance.

Resulting from Step 3 through 5, Fig. 20 demonstrates how the architecture given in Fig. 18 can be instantiated using RBML-PI. RBML-PI generates model elements based on the properties of roles specified in the consolidated tactic with an ad-hoc name given by the tool. The names are changed in Step 3 specific to the STS. The model elements are then grouped strategically in Step 4. We used the note notation for grouping components instead of the package notation to explicitly show the details of the grouped components since packages in Rose hide the contained elements.



(a) The *Dynamic Priority Scheduling* Tactic for Performance



(b) The *Onetime Password Tactic* Tactic for Security

Fig. 22. The *Dynamic Priority Scheduling* and *Onetime Password Tactic* tactic.

However, in actual development, packages should be used as shown in Fig. 18. In Step 5, the instances of the |*request*() role in the *CacheMgr* and *StockInfoDatabase* are tailored for updating.

6. Related work

There has been much work addressing NFRs at the architecture level. Chung et al. (1999) proposed a framework that guides the selection of architectural design alternatives for NFRs. In the framework, NFRs are expressed as goals, and complementing and conflicting relationships of goals are expressed in *and* and *or* relationships. Goals are refined into lower-level goals which are eventually concretized into solutions. While their architectural solutions are specific to an application, the architectural tactics in our work are generic, and as such reusable. More importantly, they do not define the semantics of architectural methods.

Based on Chung et al.'s work (Chung et al., 1999), Cysneiros and Leite (2004) presented an approach for eliciting NFRs and realizing them in a conceptual model using *Language Enhanced Lexicons* (LELs) which capture domain vocabularies. In their work, a quality attribute is designed as a goal graph where each node in the graph specifies conditions in terms of lexicons. When a modeling element (e.g., use cases, classes) pertaining to a quality attribute is developed, the element is checked for the lexicons involved in the quality attribute. If any of the lexicons is involved in the element, the conditions specified in the goal graph of the attribute is checked for the element. Their approach is for verifying NFRs, rather than embodying NFRs.

Some researchers (Khan et al., 2005; Franch and Botella, 1998; Rosa et al., 2000) have attempted to incorporate NFRs directly into architecture. Rosa et al. (2000, 2001) describe NFRs in the Z language (Spivey, 1988) and specify them as quality constraints in an architecture. Similar to Rosa et al., Khan et al. (2005) specify NFRs in *WRIGHT* Allen (1997) as constraints for components and connectors. Franch and Botella (1998) proposed *NoFun*, a language for describing NFRs in architecture components and connectors. Based on Franch et al.'s work, Zarate and Botella (2000) translate *NoFun* NFRs into expressions of Object Constraint Language (OCL) (Warmer and Kleppe, 2003) which are attached as notes in UML class diagrams. While these works help ensuring the satisfaction of NFRs, they do not provide a concrete solution to embody NFRs

Availability		
Ping/Echo, Active Redundancy	**Composition Rules**	**SPS_C1.** Add a new association role between the|*FaultMonitor* and |*StateResyncManager* roles, and name it |*recover*. **SPS_C2.** Add a new behavioral feature role in the|*StateResyncManager* role, and name it |*recover()*. **IPS_C3.** Add a :|*FaultMonitor* lifeline role in the *RecoverState* IPS. **IPS_C4.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* reference fragment in the *RecoverState* IPS.
Ping/Echo, Passive Redundancy	**Composition Rules**	**SPS_C1.** Add a new association role between the|*FaultMonitor* and |*StateResyncManager* role, and name it |*recover*. **SPS_C2.** Add a new behavioral feature role in the|*StateResyncManager*, and name it |*recover()*. **IPS_C3.** Add the :|*FaultMonitor* lifeline role in the *RecoverState* IPS. **IPS_C4.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* referance fragment in the *RecoverState* IPS.
Heartbeat, Active Redundancy	**Composition Rules**	**SPS_C1.** Add a new association role between the|*FaultMonitor* and |*StateResyncManager* roles, and name it |*recover*. **SPS_C2.** Add a new behavioral feature role in the|*StateResyncManager*, and name it |*recover()*. **IPS_C3.** Add the :|*FaultMonitor* lifeline role in the *RecoverState* IPS. **IPS_C5.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* reference fragment in the *RecoverState* IPS.
Heartbeat, Passive Redundancy	**Composition Rules**	**SPS_C1.** Add a new association role between the|*FaultMonitor* and |*StateResyncManager*, and name it |*recover*. **SPS_C2.** Add a new behavioral feature role in the|*StateResyncManager*, and name it |*recover()*. **IPS_C3.** Add the :|*FaultMonitor* lifeline role in the *RecoverState* IPS. **IPS_C5.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* reference fragment in the *RecoverState* IPS.
Ping/Echo, Heartbeat, Active Redundancy	**Composition Rules**	**Pre:** Performed the binding rules and composition rules of the *Ping/Echo* and *Heartbeat* tactics. **SPS_C2.** Add a new association role between the|*FaultMonitor* and the |*StateResyncManager* roles, and name it |*recover*. **SPS_C3.** Add a new behavioral feature role in the|*StateResyncManager* role ,and name it |*recover()*. **IPS_C4.** Add the :|*FaultMonitor* lifeline role in the *RecoverState* IPS. **IPS_C5.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* reference fragment in the *RecoverState* IPS.
Ping/Echo, Heartbeat, Passive Redundancy	**Composition Rules**	**Pre:** Performed the binding rules and composition rules of the *Ping/Echo* and *Heartbeat* tactics. **SPS_C2.** Add a new association role between the|*FaultMonitor* and |*StateResyncManager* role, and name it |recover. **SPS_C3.** Add a new behavioral feature role in the|*StateResyncManager* role, and name it |recover(). **IPS_C4.** Add the :|*FaultMonitor* lifeline role in *RecoverState* IPS. **IPS_C5.** Place the |*recover()* message role before the *StateResynchronization:RecoverState* reference fragment in the *RecoverState* IPS.

Fig. 23. The binding and composition rules for availability.

into architecture. Also, their notations are not standardized, which makes it difficult to adapt their approaches.

Metha and Medvidovic (2002) proposed architectural primitives which are reusable architectural building blocks that specify quality attributes. Architectural primitives are incorporated into an architectural style, and NFRs are embodied into an architecture by extending the architectural primitives in the architectural style (e.g., client–server). Architectural primitives are described in Alloy (Jackson et al., 2001). The architectural primitives in their work can be viewed as architectural tactics in our work. However, their approach assumes that architectural styles are used in an architecture design, which we view as a restriction. From an architect's point of view, use of architectural style should be optional.

Xu et al. (2006) classifies NFRs into *operationalizable* NFRs and *checkable* NFRs. Operationalizable NFRs are those that can be realized by functional components, and quality attributes of operationalizable NFRs are designed as an *Aspectual Component*. Checkable NFRs are those that can be checked or verified, and quality attributes (e.g., performance) of checkable NFRs are designed as a *Monitoring Component*. Aspectural and monitoring components are described in a programming template which is woven with a functional architecture based on a set of binding rules that are described in XML. While use of templates simplifies the weaving process, templates cannot capture structural variations of components. The nature of templates lends themselves only to stamping out the same structure in instantiation without having any variation point. Their classification can be related to the classification of execution quality and evolution quality by Matinlassi and Niemela Matinlassi and Niemela (2003) which is more commonly used. Operationalizable and checkable NFRs can be considered as subsets of execution quality which is concerned with observable qualities (e.g. performance, security, availability) at run-time. In terms of coverage, the classification by Matinlassi and Niemela is broader since it also covers evolution qualities.

Bruin and Vliet (2001) proposed a refinement-based approach for developing a software architecture that addresses quality concerns (e.g., security, performance). In their approach, quality requirements are analyzed in terms of features, and the features are mapped to architectural solutions which are based on a specific architecture style (e.g., peer-to-peer, client–server). Feature-solution (FS) graphs are used to represent quality features, architectural solutions and their mapping. Architectural solutions are represented as a tree structure where each node represents a particular architectural solution (e.g., Firewall, Encryption/Decryption) in the architectural style. The semantics of an architectural solution is defined using a use case map (UCM) (Buhr, 1998) which captures the structural and behavioral aspects of the solution with refinement points where other solutions can be plugged in. The root node of a solution structure provides a basic architectural structure for the application, and the structure is refined with other solutions mapped with quality features in the FS graph. Architectural solutions in their work can be viewed as architectural tactics in our work. However, our tactics are not dependent on any architectural style. We argue that defining tactics for a specific architectural style would significantly limit the reusability of tactics. Also, the generic nature of architectural tactics makes themselves inappropriate to be specific to an architectural style. The tactics in our work can be used to realize a specific architectural style as in their work. We would also like to pose a counter point on their arguments that activity diagrams in the UML are the closest type to UCMs, and activity diagrams are not able to capture both structural and behavioral aspects in one diagram, which lead them to use UCMs to define the semantics of architectural solutions. Firstly, we point out that communication diagrams in the UML 2.0 (which was previously called collaboration diagrams in UML 1.x) can specify both structural and behavioral aspects, and thus communication diagrams are the closest to UCMs, not activity diagrams. Secondly, showing both structural and behavioral aspects in one diagram has little benefit over showing them separately. Mixing the two different aspects in one diagram often makes it hard to understand and specify the details of each aspect. For example, inheritance structure or iterative behavior cannot be specified in UCMs.

7. Conclusion

We have presented a systematic approach for embodying NFRs into software architecture using architectural tactics. Major benefits of the approach are that (1) the RBML allows one to precisely specify tactics, (2) the rigorous tactic composition rules enable systematic composition of tactics, (3) the variations captured in tactic specifications allow various tactic intantiations, and thus improve tactic reuse, and (4) fine-grained solutions of tactics enables strategic architecture development through various configurations. It should be noted that not all tactics can be specified in the RBML. For instance, the resource demand tactics (Bass et al., 2003), which are concerned with managing resource demand, are difficult to formalize in the RBML due to the abstract nature of their solutions. For such tactics, a manual realization is required based on the experience of the architect.

Performance		
Introduce Concurrency , Fixed Priority Scheduling	**Binding Rules**	**B1.** *IntroduceConcurrency*::|*ConcurrentComp* → *FixedPriorityScheduling*::|*Client* **B2.** *IntroduceConcurrency*::|*ConcurrentComp* → *FixedPriorityScheduling*::|*Scheduler*
	Composition Rules	**SPS_C1.** Merge the *IntroduceConcurrency*::|*ConcurrentComp* role with the *FixedPriorityScheduling*::|*ResourceRequest* role and name the merged role |*Client*. **SPS_C2.** Merge the *IntroduceConcurrency*::|*ConcurrentComp* role with the *FixedPriorityScheduling*::|*Scheduler* role and name the merged role |*Scheduler*.
Introduce Concurrency , Dynamic Priority Scheduling	**Binding Rules**	**B1.** *IntroduceConcurrency*::|*ConcurrentComp* → *DynamicPriorityScheduling*::|*Client* **B2.** *IntroduceConcurrency*::|*ConcurrentComp* → *DynamicPriorityScheduling*::|*Scheduler*
	Composition Rules	**SPS_C1.** Merge the *IntroduceConcurrency*::|*ConcurrentComp* role with the *DynamicPriorityScheduling*::|*ResourceRequest* role and name the merged role |*Client*. **SPS_C2.** Merge the *IntroduceConcurrency*::|*ConcurrentComp* role with the *DynamicPriorityScheduling*::|*Scheduler* role and name the merged role |*Scheduler*.

Fig. 24. The binding and composition rules for performance.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102

Security		
ID/Password, Authorize Users	**Binding Rule**	**B1.** *ID/Password*::|*Client* → *AuthorizeUsers*::|*Client*
	Composition Rule	**SPS_C1.** Merge the *ID/Password*::|*Client* role with the *AuthorizeUsers*::|*Client* role and name the merged role |*Client*.
ID/Password, Authorize Users, Maintain Data Confidentiality	**Binding Rules**	**Pre:** Performed the binding rules of the *ID/Password* and *Maintain Data Confidentiality* **Pre:** Performed the binding rule of the *ID/Password* and *Authorize* Users.
	Composition Rules	**Pre:** Performed the composition rules of the *ID/Password* + *Maintain Data Confidentiality* **Pre:** Performed the composition rules of the *ID/Password* + *Authorize Users*.
Onetime Password, Authorize Users	**Binding Rule**	**B1.** *OnetimePassword*::|*Client* → *AuthorizeUsers*::|*Client*
	Composition Rule	**SPS_C1.** Merge the *OnetimePassword*::|*Client* role with the *AuthorizeUsers*::|*Client* role and name it |*Client*.

Fig. 25. The binding and composition rules for security.

While we assumed in this paper that tactic selection is made by the architect, it is possible to automate tactic selection. One approach which we are currently investigating is using metrics and inference systems. In the approach, features and NFRs are quantified using metrics, and the quantified NFRs are projected as constraints into the quantified features via an inference system (e.g., Prolog) to infer valid feature configurations that satisfy the quantified NFRs. The architect then can simply choose one configuration from the result.

When the presented approach is applied for other quality attributes such as modifiability and usability, a comprehensive analysis of the attribute domain should be conducted so as to find available tactics and define a complete set of composition rules for the tactics. Also, tactics in different quality attributes may be related to one another, which requires an analysis across the domains of the involved quality attributes.

Acknowledgements

This work is partly supported by the National Science Foundation under Grant Nos. CCF-0523101 and CCR-0131862 and by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement (C1090-0903-0004).

Appendix A. Tactic specifications

See Figs. 21 and 22.

Appendix B. Composition rules

See Figs. 23–25.

References

Abawajy, J.H., 2004. Fault detection service architecture for grid computing systems. In: Proceedings of the Computational Science and Its Applications-ICCSA. Lecture Notes in Computer Science, vol. 3044. Springer, Berlin, pp. 107–115.

Allen, R., 1997. A Formal Approach to Software Architecture. Ph.D. Thesis, Carnegie Mellon University.

Bachmann, F., Bass, L., Llein, M., 2002. Illuminating the Fundamental Contributors to Software Architecture Quality. Technical Report CMU/SEI-2002-TR-025, Software Engineering Institute, Carnegie Mellon University.

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice, second ed. Addison Wesley.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J., 2007. Handbook on Scheduling: From Theory to Applications. Springer.

Bruin, H., Vliet, H., 2001. Scenario-based generation and evaluation of software architectures. In: Proceedings of the Third Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany, pp. 128–139.

Buhr, R.J.A., 1998. Use Case Maps as architecture entities for complex systems. IEEE Transaction on Software Engineering 24 (12), 1131–1155.

Chung, L., Nixon, B., Yu, E., Mylopoulos, J., 1999. Non-Functional Requirements in Software Engineering. Springer.

Cole, E., Krutz, R., Conley, J., 2005. Network Security Bible. John Wiley.

Cysneiros, L., Leite, J., 2004. Nonfunctional requirements: from elecitation to conceptual models. IEEE Transaction on Software Engineering 30 (5), 328–350.

Cysneiros, L., Yu, E., Leite, J., 2003. Cataloguing non-functional requirements as softgoal networks. In: Proceedings of the Requirements Engineering for Adaptable Architectures, Monterey Bay, CA, pp. 13–20.

Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison Wesley.

France, R., Kim, D., Ghosh, S., Song, E., 2004. A UML-based pattern specification technique. IEEE Transaction on Software Engineering 30 (3), 193–206.

Franch, X., Botella, P., 1998. Putting non-functional requirements into software architecture. In: Proceedings of the Ninth International Workshop on Software Specification and Design, Ise-Shima, Japan, pp. 60–67.

Jackson, D., Shlyakhter, I., Sridharan, M., 2001. A micromodularity mechanism. In: ACM SIGSOFT Software Engineering Notes, ESEC-FSE01, Vienna, Austria, pp. 62–73.

Khan, K., Eenoo, C., Hylooz, S., 2005. Addressing non-functional properties in software architecture using adl. In: Proceedings of the Sixth Australasian Workshop on Software and System Architectures, Brisbane, Australia, pp. 6–12.

Kim, D., 2007. The role-based metamodeling language for specifying design patterns. In: Taibi, Toufik (Ed.), Design Pattern Formalization Techniques. Idea Group Inc., pp. 183–205.

Kim, D., Whittle, J., 2005. Generating UML models from pattern specifications. In: Proceedings of the Third ACIS International Conference on Software Engineering Research, Management and Applications (SERA2005), Mount Pleasant, MI, pp. 166–173.

Kim, D., France, R., Ghosh, S., Song, E., 2003. A role-based metamodeling approach to specifying design patterns. In: Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference, Dallas, TX, pp. 452–457.

Kim, D., Mehta, P., Gokhal, P., 2006. Describing access control patterns using roles. In: Proceedings of the Pattern Languages of Programming Conference (PLoP), Portland, OR.

Matinlassi, M., Niemela, E., 2003. The impact of maintainability on component-based software systems. In: Proceedings of the 29th Conference on EUROMICRO, Antalya, Turkey, pp. 25–32.

Metha, N., Medvidovic, N., 2002. Distilling Software Architecture Primitives form Architectural Styles. Technical Report USC-CSE-2002-509, University of Southern California, Los Angeles, CA.

Ramachandran, J., 2002. Designing Security Architecture Solutions. John Wiley.

Rosa, N., Justo, G., Cunha, P., 2000. Incorporating non-functional requirements into software architectures. In: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing, pp. 1009–1018.

Rosa, N., Justo, G., Chuha, P., 2001. A framework for building non-functional software architectures. In: Proceedings of the 2001 ACM Symposium on Applied Computing, Las Vegas, NV, pp. 141–147.

Schmidt, K., 2006. High Availability and Disaster Recovery: Concepts, Design, Implementation. Springer.
Silberschatz, A., Galvin, P., Gagne, G., 2005. Operating System Principles. Addison Wesley.
Spivey, J., 1988. Understanding Z: A Specification Language and its Formal Semantics. Cambridge University Press.
Stelling, P., Foster, I., Kesselman, C., Lee, C., Laszewski, G., 1999. A fault detection service for wide area distributed computations. Cluster Computing 2 (2), 117–128.
The Object Management Group (OMG), 2007. Unified Modeling Language: Superstructure. Version 2.1.2 formal/07-11-02. OMG. <http://www.omg.org>.
Warmer, J., Kleppe, A., 2003. The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison Wesley.
Xu, L., Ziv, H., Richardson, D., Liu, Z., 2005. Towards modeling non-functional requirements in software architecture. In: Proceedings of the Early Aspects: Aspect-Oriented Requirement Engineering and Architecture Design, Chicago, IL.
Xu, L., Ziv, H., Alspaugh, T., Richardson, D., 2006. An architectural pattern for non-functional dependability requirements. Journal of Systems and Software 79 (10), 1370–1378.
Zarate, G., Botella, P., 2000. Use of UML for modeling non-functional aspects. In: Proceedings of the 13th International Conference on Software and System Engineering and their Application, Paris, France.
Zou, X., Huang, J.C., Settimi, R., Solc, P., 2007. Automated classification of non-functional requirements. Requirements Engineering 12 (2), 103–120.

Suntae Kim received his B.S. degree in computer science and engineering from Chung-Ang University in 2003, and the M.S. degree in computer science and engineering from Sogang University in 2007. He is a Ph.D. candidate at Sogang University. He worked in SoftwareCraft Co. Ltd., as a senior consultant and engineer during 2002–2004. His research focuses on software architecture, design patterns and requirements engineering.

Dae-Kyoo Kim is an assistant professor of the Department of Computer Science and Engineering at Oakland University. He received the Ph.D. in computer science from Colorado State University in 2004. He worked as a senior software engineer at McHugh Software International from 1997 till 2000. He has published more than thirty papers and regularly serves as a workshop chair, program committee member of numerous conferences and journals in the area of software engineering. He is currently serving on the editorial board of the International Journal of Pattern. His research interests include design pattern formalization, model refactoring, aspect-oriented modeling, software architecture modeling, access control modeling, and formal methods. He is a member of the IEEE Computer Society.

Lunjin Lu graduated from the University of Birmingham (England) with a Ph.D. in Computer Science in 1995. From 1995 to 1999, he worked as a postdoctoral fellow at the Ecole Polytechnique in Paris, the Ben-Gurion University of the Negev, and the Univesity of Waikato. He was a lecturer at the East China Normal University from 1985 to 1990 and from 1995 to 1996 and a visiting assistant professor at the Texas State University in 1999. Dr. Lu joined the Oakland University in 2000 and is now an associate professor of engineering. Dr. Lu is interested in semantics and analysis of programs, constraint and logic programming, program debugging, programming environments, software verification.

Sooyong Park received his B.S degree in computer science from Sogang University, Seoul, in 1986, the M.S. degree in computer science from Florida State University, in 1988, and the Ph.D. degree in information technology with major in software engineering from George Mason University, in 1995. In 1994, he received an American Institute of Aeronautics and Astronautics(AIAA) Software Engineering Award. He served as a senior software engineer at TRW ISC during 1996–1998. He is currently a professor of computer science and engineering at Sogang University. His research interests include requirements engineering, embedded and adaptive software development, and software architecture.

Please cite this article in press as: Kim, S., et al. Quality-driven architecture development using architectural tactics. J. Syst. Software (2009), doi:10.1016/j.jss.2009.03.102