A Tactic-Based Approach to Embodying Non-functional Requirements into Software Architectures

Conference Paper · October 2008

DOI: 10.1109/EDOC.2008.18 · Source: IEEE Xplore

4 authors, including:

Suntae Kim
Kangwon National University
38 PUBLICATIONS **457** CITATIONS

SEE PROFILE

Dae-Kyoo Kim
Oakland University
101 PUBLICATIONS **1,563** CITATIONS

SEE PROFILE

Lunjin Lu
Oakland University
78 PUBLICATIONS **419** CITATIONS

SEE PROFILE

A Tactic-Based Approach to Embodying Non-functional Requirements into Software Architectures

Suntae Kim[1,2], Dae-Kyoo Kim[1], Lunjin Lu[1], Soo-Yong Park[2]
[1]Oakland University, Rochester, MI 48309, USA
[2]Sogang University, Seoul, South Korea
E-mail: {jipsin08,sypark}@sogang.ac.kr, {kim2,l2lu}@oakland.edu

Abstract

This paper presents an approach for embodying non-functional requirements (NFRs) into software architecture using architectural tactics. Architectural tactics are reusable architectural building blocks, providing general architectural solutions for commonly occurring issues related 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. The composed tactic is then used to instantiate an initial architecture for the application where the NFRs are embodied. A stock trading system is used to demonstrate the approach.

1 Introduction

Software development is initiated with application requirements which consist of functional requirements (FRs) and non-functional requirements (NFRs). Functional requirements describe what the software should do, while non-functional requirements impose constraints on the solution how the system should accomplish the functionality of the system, 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 considered only in the late phase of the development, while FRs are well exercised throughout the development. This makes it difficult to satisfy NFRs and often causes changes to the previously developed artifacts such as architectures and designs. 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 [3, 4, 5, 11, 15, 16]. There are two major streams. One is to specify NFRs as architectural constraints in the functional architecture of the application. In the 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 in that domain. However, their building blocks are not general enough to be used in other domains, which reduces their reusability. Also, the granularity of building blocks is coarse (e.g., one architecture building block per quality attribute), which requires significant tailoring work 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 [1]. An architectural tactic is a reusable architectural building block that provides a generic solution to address issues related to quality attributes. In our approach, architectural tactics are selected based on a given set of NFRs, and the selected tactics are composed to produce a tactic that combines the solutions of the selected tactics. The composed tactic is then instantiated to create an initial architecture of the application where the NFRs are embodied. We use feature modeling [6] to represent tactics and their relationships. 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 [7, 9]. We present architectural tactics for availability and performance, and demonstrate how the tactics can be used to embody the NFRs for the availability and performance of a stock trading system into its architecture.

The major contributions of this paper are 1) the analysis of the relationships of the architectural tactics for availability and performance, 2) the semantic specifications of availability and performance tactics, and 3) the composition mechanism of tactics to build a high quality architecture for a specific application that embodies the NFRs of the appli-

1541-7719/08 $25.00 © 2008 IEEE
DOI 10.1109/EDOC.2008.18

cation.

The remainder of this paper is structured as follows. Section 2 gives an overview of related work. Section 3 gives a background on architectural tactics, feature modeling and the RBML. Section 4 presents feature models for availability and performance tactics and the RBML specifications of the tactics. Section 5 demonstrates how availability and performance tactics can be used to build an architecture that satisfies NFRs of a stock trading system. Section 6 concludes the paper.

2 Related Work

There has been much work on addressing NFRs at the architecture level. Chung *et al.* [4] proposed a framework that guides the selection of architectural design alternatives for NFRs. In the framework, NFRs are expressed as goals, and the 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 specific 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 [4], Crysneiros and Leite [5] 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) related 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 evaluated for the element. However, their approach is for verification of NFRs, not for NFRs embodiment.

Some researchers [3, 8, 12] have attempted to incorporate NFRs directly into architecture. Rosa *et al.* [12] describe NFRs in the Z language and specify them as quality constraints in an architecture. Similar to Rose *et al.*, Khan *et al.* [3] specify NFRs in *WRIGHT*, an architectural description language, as constraints for components and connectors. Franch *et al.* [8] proposed *NoFun*, a language for describing NFRs in architectural components and connectors. Based on Franch *et al.*'s work, Botella *et al.* [16] translate *NoFun* NFRs into OCL expressions attached as notes in UML class diagrams. While these works help ensuring the satisfaction of NFRs, they do not provide a solution to embody NFRs into an architecture. Also, their notations are not standardized, which makes it difficult to adapt their approaches.

Mehta and Medvidovic [11] 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, a formal specification language. 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 the architecture design, which we view as a restriction. From an architect's point of view, use of architectural style should be optional.

Xu [15] 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 weaving process, templates cannot capture structural variations of components.

3 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.

3.1 Architectural Tactics

An architectural tactic is a fine-grained reusable architectural building block that provides an architectural solution built from experience for achieving a quality attribute. There have been many architectural tactics proposed for various quality attributes including 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 using inheritance while raising 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. Optional tactics provide the architect maneuverability in building an architecture according to an architectural strategy. Tactics may be related to one another. A tactic may require another complementary tactic or exclude a conflicting tactic. For instance, 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 achieve the quality attribute of the tactic. In some cases, use of tactics for one quality attribute may affect another quality attribute. For example, use of both the *ID/Password* and *Maintain Data Confidentiality* tactics for security usually decreases performance due to encryption of login information.

3.2 Feature Modeling

Based on our analysis in Subsection 3.1, we found feature modeling [6] suitable for representing the relationships of 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 constrains that only one feature can be selected from the group, while an *inclusive-or* group specifies 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 relationships, *suggested* to specify complementary relationships for a synergistic combination in tactic selection. 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 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, 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, 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 use of a touch screen suggests a large size battery due to more power consumption.

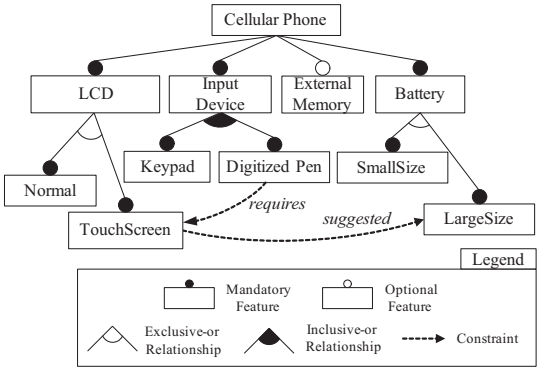

Figure 1. A Feature Model

3.3 Role-Based Metamodeling Language

We use the Role-Based Metamodeling Language (RBML), which is a UML-based pattern specification language developed in our previous work [7, 9], to define the semantics of an architectural tactic. The RBML specifies tactic semantics in terms of roles which are played by UML model elements (e.g., classes, messages). A role is based on a UML metaclass 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. Major benefits of using the RBML for defining architectural tactics are 1) the RBML facilitates the use of tactics at the model-level, and 2) the RBML is capable of capturing both the generic structure of a tactic and its variations through realization multiplicities, which increases the reusability 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, respectively, the *Classifier* and *Relationship* metaclasses 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 by 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. IPSs are dependent on SPSs. 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.

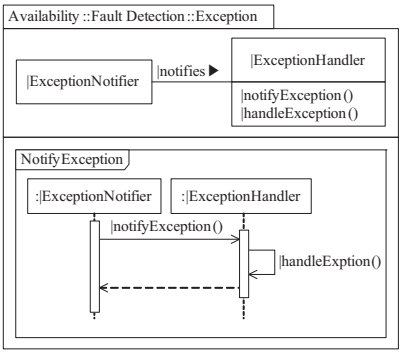

Figure 2. An RBML Specification for the *Exception* Tactic

Fig. 2 shows an RBML specification for the *Exception* tactic for availability. In the figure, the upper compartment shows an SPS, and the lower compartment shows an IPS. The "|" symbol in the diagrams denotes roles. The SPS has two class roles, |*ExceptionNotifier* and |*ExceptionHandler*, and one 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. The IPS specifies that when an exception occurs, it is notified to an exception handler. The UML package notation is used to group related SPSs and IPSs for a tactic.

4 Specifying Architectural Tactics

In this section, we define feature models for availability and performance tactics and the semantics of the tactics using the RBML. The feature models and RBML specifications are developed based on the work [1, 2, 14].

4.1 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 [1, 13] as shown in Fig. 3.

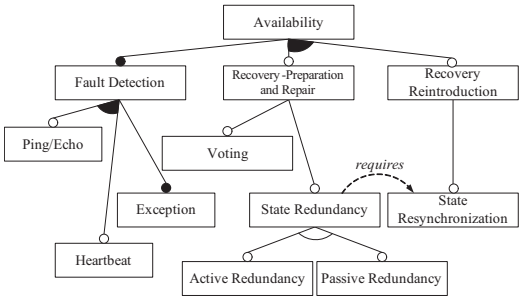

Figure 3. Availability Architectural Tactic Feature Model

The *Fault Detection* tactic is concerned with detecting a fault and notifying 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 the sender does not receive an echo message back from a receiver within a certain time period, the receiver is considered to have failed. Fig. 4 defines the *Ping/Echo* tactic.

The SPS shows that the *Ping/Echo* tactic involves three concepts of senders, receivers and a monitor which are captured by the |*PingSender*, |*PingReceiver* and |*FaultMonitor* roles, respectively. The |*maxWaitingTime* feature role in the |*PingSender* role specifies the maximum waiting time for an echo after sending a ping message. The |*timeInterval* feature role specifies the time interval to send ping messages. The *LOOP* fragment in the IPS specifies that a sender sends a ping message to all receivers every certain time. *rcvs* in the *LOOP* is a function that returns the number of receivers. Note that the *LOOP* operator is not the UML *loop* operator, but an RBML operator that constrains the structure of instantiations of the IPS [9]. The inner *loop* fragment describes the behavior of sending a ping message and receiving an echo back 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 specified by the *NotifyException* fragment in the *Exception* tactic (see Fig. 2 in Section 3). The mapping in the *Exception* fragment specifies the binding information between the *Ping/Echo* and *Exception* tactics. The realization multiplicity *1* in the |*FaultMonitor* role constrains that there must be

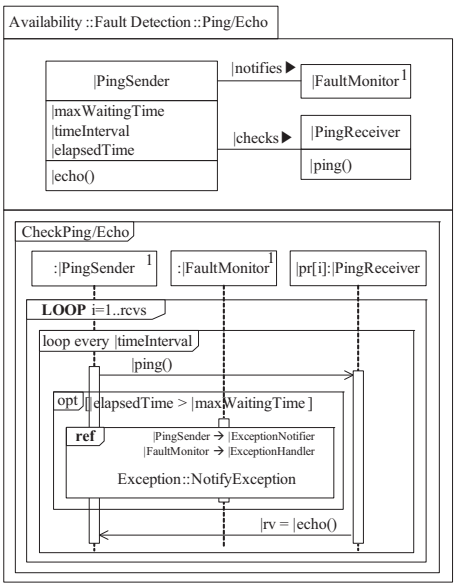

Figure 4. The *Ping/Echo* Tactic

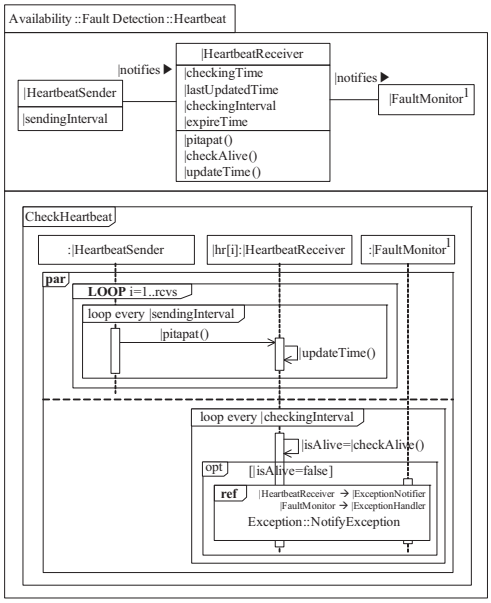

Figure 5. The *Heartbeat* Tactic

only one monitor.

The *Heartbeat* tactic detects a fault by listening to the 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. 5 defines the *Heartbeat* tactic.

Similar to the *Ping/Echo* tactic, the *Heartbeat* tactic involves three concepts of senders, receivers and a monitor which are captured by the |*HeartbeatSender*, |*HeartbeatReceiver* and |*FaultMonitor* roles. The |*sendingInterval* role in the *HeartbeatSender* role specifies the time when a heartbeat message is sent periodically. The |*checkingTime* and |*lastUpdatedTime* roles in the *HeartbeatReceiver* role specify the last checking time and the time when the last heartbeat message was received, respectively. The |*checkingInterval* role denotes a time to check regularly the aliveness of heartbeat senders by comparing the latency time between the current time and last updated time with the expire time. The |*expireTime* role specifies the max waiting time for the next heartbeat message. The IPS specifies that sending heartbeat messages and checking aliveness are executed on separate threads for concurrency. Once a fault occurs, the fault should be recognized and informed to an exception handler. This is captured by the *NotifyException* fragment in Fig. 5. Typically, the component that raises an exception executes in the same process of the exception handling component. As shown above, the *Exception* tactic is generally used together with the *Ping/Echo* tactic and *Heartbeat* tactic for handling faults.

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 this tactic, the same algorithm is run on redundant components, and the voter monitors their behavior. When the behavior of one component is different from the rest, the voter fails the component. Even with the failure of one component, the system still runs normally with rest of the components. 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 received concurrently from redundant components for a service request. If a redundant component fails, it recovers the component by resynchronizing the state with one of the alive components. This is called the *Active Redundancy* tactic. The other way is to use the response from a specific component (primary) and inform other components to update their state with that of the primary component. If the primary component fails, its state is resynchronized with the state of a redundant component. This is captured by the *Passive Redundancy* tactic. Fig. 6 shows the *Active Redundancy* tactic.

The SPS in Fig. 6 describes that the *Active Redundancy* tactic involves clients, redundant components, a redundancy manager and a state resynchronization manager. The *Pro-*

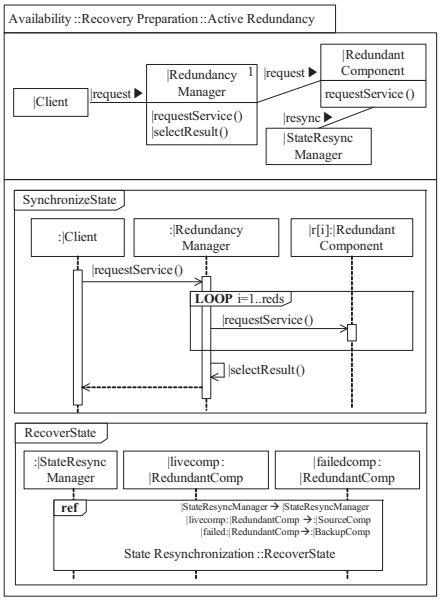

Figure 6. The *Active Redundancy* Tactic

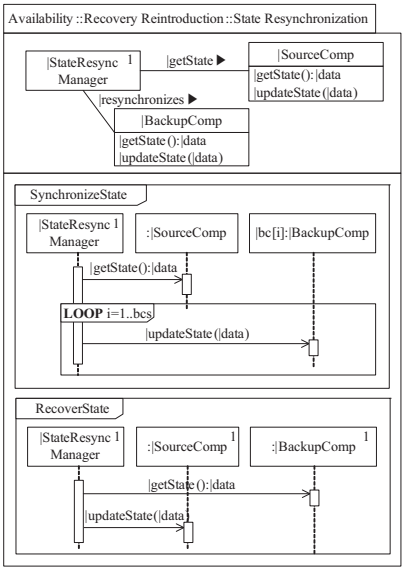

Figure 7. The *State Resynchronization* Tactic

cessRequest IPS specifies that when there is a request for service, the redundancy manager broadcasts the request to redundant components which respond in parallel. The redundancy mananger selects one of the responses and returns to the client. The *Recovery* IPS specifies that a failed component is recovered by resynchronizing its state with that of a live component. The *RecoverState* fragment in the IPS references the *State Resynchronization* tactic which is described in the following. The RBML specification of the *Passive Redundancy* tactic is not presented due to space limitation.

The *Recovery Reintroduction* tactic is concerned with restoring the state of a failed component. One way to restore is by resynchronizing the state of the failed component with that of a live component. This is called the *State Resynchronization* tactic which is defined in Fig. 7.

The *State Resynchronization* tactic involves concepts of a state resynchronization manager, source components and backup components. The state resynchronization manager is responsible for synchronizing the state between a source component and a backup component. A synchronization can occur when either the state of a source component is changed, or the source component is recovered from a failure. In the former case, the state of backup components is synchronized with that of the source component. This is captured by the *SynchronizeState* IPS where the **LOOP** fragment specifies updating the state of each backup component. In the latter case, the synchronization is opposite where the state of the source component is resychronized with that of a backup component. This is captured by the *RecoverState* IPS.

4.2 Tactics for Performance

Performance is concerned with 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. 8.

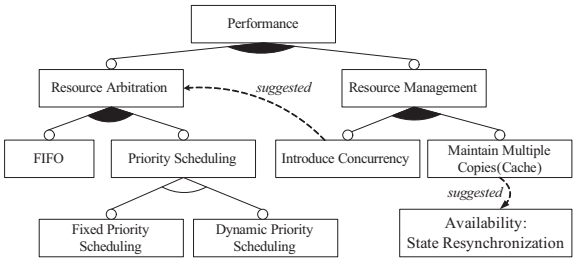

Figure 8. Performance Architectural Tactic Feature Model

The *Resource Arbitration* tactic improves performance by scheduling requests for 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 they are received. This is called the *FIFO* tactic which is defined in Fig. 9.

The *FIFO* tactic involves producers, FIFO queues and consumers. The *Enqueue* IPS describes the behavior of en-

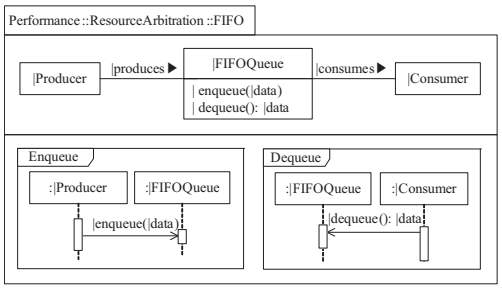

Figure 9. The *FIFO* Tactic

queueing data, and the *Dequeue* IPS describes the dequeueing behavior. The *FIFO* tactic is often used for job distribution for threads and processes, especially in network.

Another way is to assign a priority to resources, and based on the priority, the resource requests are scheduled. This is called 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 high requests. Priority can be given by assigning either a fixed priority or a dynamic priority, which is captured by the *Fixed Priority Scheduling* tactic and the *Dynamic Priority Scheduling* tactic, respectively. In the *Dynamic Priority Scheduling* tactic, priorities are determined at runtime based on execution parameters such as upcoming deadlines or other runtime conditions. The *Fixed Priority Scheduling* and *Dynamic Priority Scheduling* tactics may be used with the *FIFO* tactic to prevent the starvation issue for low priority resources.

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, blocked time in response can be significantly reduced. This is called the *Introduce Concurrency* tactic which is defined in Fig. 10.

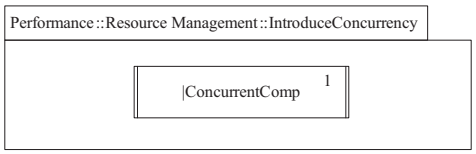

Figure 10. The *Introduce Concurrency* Tactic

The |*ConcurrentComp* role in the tactic captures concurrent components that have their own thread (or process) which is denoted by the double lines at the ends of the box. The *Introduce Concurrency* tactic is often used with a resource arbitration tactic for concurrent communication among threads. This is specified by the *suggested* relationship in Fig. 8.

Another way for managing resources is to keep replicas of resources on separate repositories, so that contention for resources is reduced. This is called the *Maintain Multiple Copies* tactic. Caching is used to replicate resources. An important issue in this tactic is to maintain consistency among the copies and keep them synchronized. To address this issue, the *State Resynchronization* tactic in Subsection 4.1 can be used together.

The RBML specifications of the *Fixed Priority Scheduling*, *Dynamic Priority Scheduling* and *Maintain Multiple Copies* tactics are not presented.

5 Case Study: Stock Trading System

In this section, we demonstrate how the performance and availability tactics presented in Section 4 can be used to embody NFRs of a stock trading system (STS) into the architecture of the system. The STS is an online stock trading system that provides real-time services for checking the current price of stocks, placing buy and 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. 11 shows a context diagram of the STS.

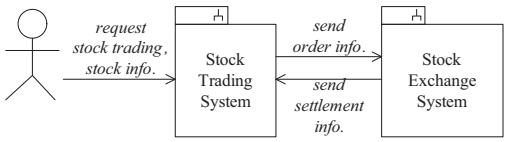

Figure 11. Stock Trading System

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

- **NFR1.** The STS should be available during the trading time (8:00 AM – 4:00 PM) from Monday through Friday. If there is no response from the SES for 30 seconds, the STS should notify the administrator.
- **NFR2.** The system should be able to process 5000 transactions per second and 200,000 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 second in the order they were placed.

5.1 Configuring Tactics for NFR1

NFR1 requires high availability of the system during the trading time. To support this, the *Ping/Echo* and *Heart-*

beat tactics in Section 3.1 can be used together. Using the *Ping/Echo* tactic, the availability of the STS can be checked by sending a ping message regularly and listening to an echo. However, in some cases, the ping sender may receive an echo even if the STS has failed [2]. To detect such a case, the *Heartbeat* tactic can be used together. Using the *Heartbeat* tactic, the failure of the STS can be detected by listening to heartbeat messages from the STS. Furthermore, even in a typical failure, the combined use of the two tactics can detect the failure earlier by sending a ping message during the latency time between heartbeat messages. Tactic selection, however, really depends on the architect based on his experience and an architectural strategy. In our approach, feature models provide some guidelines for tactic selection, denoted by *suggested* relationships.

Two tactics are composed based on sets of binding rules and composition rules. Binding rules define the corresponding roles in the two tactics, while composition rules define operational steps for changes to be made in the composed tactic. The following defines the binding rule for 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 *Hearttbeat* tactic. Thus, the two |*FaultMonitor* roles are merged into one 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 |*Subsystem1*.

SPS_C2. Put the |*PingSender*, |*HeartbeatReceiver* and |*FaultMonitor* roles into the same package role and name the package role |*Subsystem2*.

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 are the composition rules for SPSs. SPS_C1 specifies that ping receivers and heartbeat senders should be deployed in the same subsystem. Similarly, SPS_C2 specifies that ping senders, heartbeat receivers and the monitor together should be deployed in the same subsystem (monitoring subsystem). The subsystem of SPS_C1 should be deployed physically separate from that of SPS_C2. These rules enable the monitoring subsystem to both send ping messages and receive heartbeat messages. IPS_C3 to IPS_C6 describe the composition rules for IPSs. The rules specify that the behavior of the *Ping/Echo* tactic and the behavior of the *Heartbeat* tactic should be executed concurrently for rigorous monitoring.

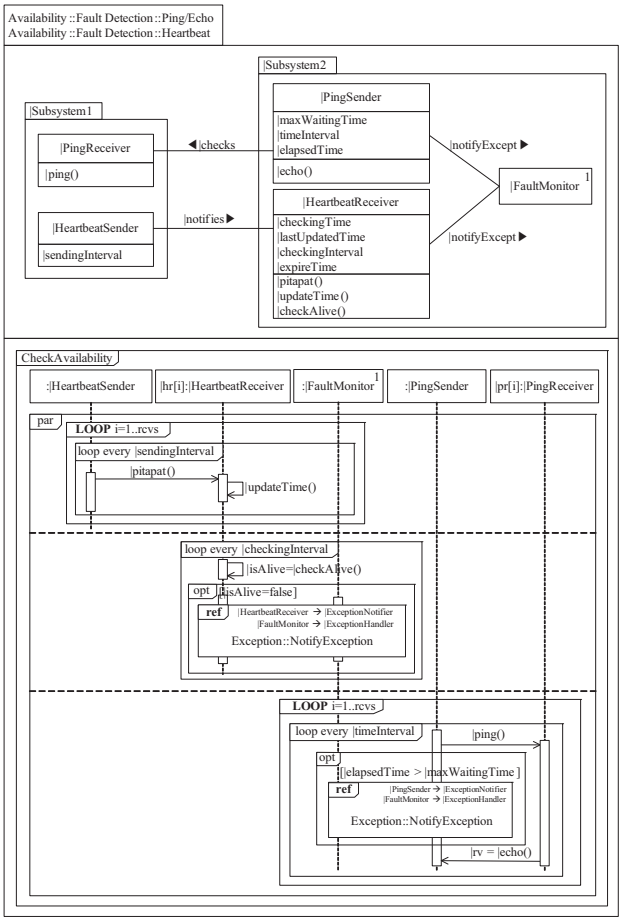

Figure 12. Composition of the *Ping/Echo* and *Heartbeat* Tactics

By applying the binding rule and composition rules, the *Ping/Echo* and *Heartbeat* tactics are composed as shown in 12. In the composed tactic, high availability is supported by checking both ping/echo and heartbeat messages concurrently. Given the composed tactic, an initial architecture can be instantiated in consideration of the functional requirements of the STS. Fig. 13 shows an instantiated architecture.

In the figure, the stereotypes represent the roles from which the architectural elements are instantiated. For example, the *OrdDlvPingReceiver* component is an instance of the |*PingReceiver* role in Fig. 12. The architecture designs that the *Order Delivery* subsystem is responsible for sending orders and receiving settlement information from the SES, and the subsystem is monitored by the *Monitoring*

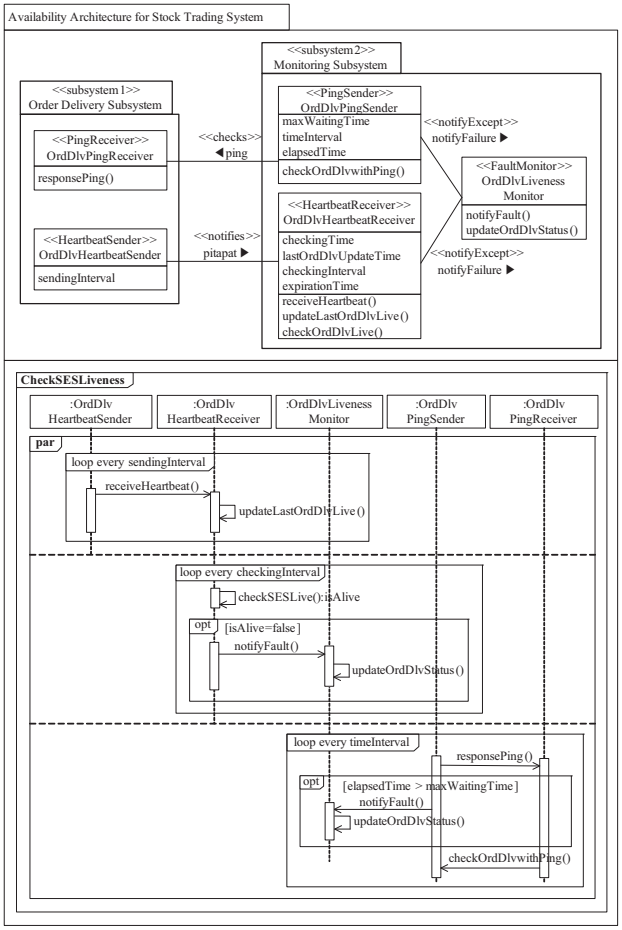

Figure 13. An Instantiated Architecture for the STS Embodying NFR1

subsystem using ping/echo and heart messages. The *Monitoring* subsystem is deployed separately on the network. In this example, the structure of the instantiation architecture is same as the composed tactic. However, it is not always the case as will be shown in the following subsection for performance architecture.

5.2 Configuring Tactics for NFR2

NFR2 is concerned with the performance of the STS, requiring handling considerable amount of transactions by their kinds in very short time. To embody NFR2, the *FIFO* and *IntroduceConcurrency* tactics can be used together. Using the *FIFO* tactic, each type of orders can be assigned to a dedicated FIFO queue instance for immediate process. Use of the *IntroduceConcurrency* tactic with the *FIFO* tactic can further improve the performance by adding a thread to the STS interface ports for order delivery to the SES. This improves the performance by dispatching the same kind of orders concurrently. 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 them concurrent. By applying the binding rules, the *FIFO* and *IntroduceConcurrency* tactics are composed as shown in Fig. 14.

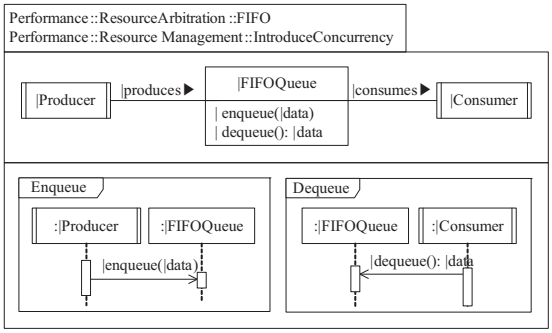

Figure 14. Composition of the *FIFO* and *Introduce Concurrency* Tactics

Using the composed tactic, an initial architecture for the performance of the STS can be instantiated as shown in Fig. 15. In the figure, the order placing component (*OrderingWebComp*) places the same kind of orders into a dedicated queue instance. Also, each queue has a dedicated delivery component that has its own thread for dispatching the orders. The concurrent streamline of order processing greatly improves the performance of the STS by reducing the blocked time of orders. Various structures of architecture can be instantiated from the composed tactic via the variation points captured in the roles of the composed tactic.

6 Conclusion

In this paper, we have presented an approach for systematically embodying NFRs into software architecture using architectural tactics. The feature modeling of tactics provides maneuverability for configuring tactics for a given set of NFRs. The RBML specifications of tactics with the rigorous composition rules facilitate the mechanical composition of tactics. The presented approach is not intended to give an implementation solution that directly addresses NFRs, but aims at providing an architectural solution that

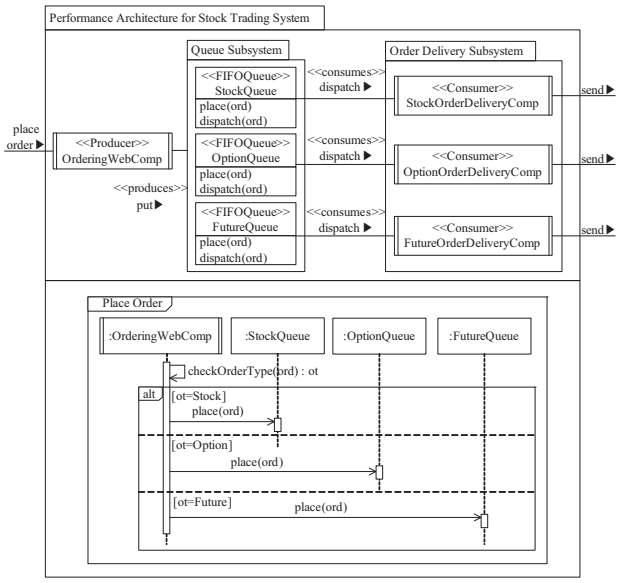

Figure 15. An Instantiated Architecture for the STS Embodying NFR2

helps to realize NFRs. We found that some tactics (e.g., resource demand tactics [1]) are hard to formalize due to the abstract nature of their solutions. Such tactics should be realized by the architect. A composed tactic can be automatically instantiated using RBML-PI, a tool developed in our previous work for instantiating RBML models [10], to generate an initial architecture. Using the tool, various architectural structures can be generated. We are currently developing tool support for composing tactics.

Acknowledgements.

This work is supported in part by the National Science Foundation under Grant No. CCF-0523101 and CCR-0131862.

7 References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture In Practice 2nd Edition*. Addison Wesley, 2003.

[2] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Handbook On Scheduling : From Theory to Applications*. Springer, 2007.

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

[4] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 1999.

[5] L. Cysneiros and J. Leite. Nonfunctional Requirements:From Elecitation to Conceptual Models. *IEEE Transaction on Software Engineering*, 30(5):328–350, 2004.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] R. France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004.

[8] X. Franch and P. Botella. Putting Non-Functional Requirements into Software Architecture. In *Proceedings of the 9th International Workshop on Software Specification and Design*, pages 60–67, Ise-Shima, Japan, 1998.

[9] D. Kim. The Role-Based Metamodeling Language for Specifying Design Patterns. In Toufik Taibi, editor, *Design Pattern Formalization Techniques*, pages 183–205. Idea Group Inc., 2007.

[10] D. Kim and J. Whittle. Generating UML Models from Pattern Specifications. In *Proceedings of 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005)*, pages 166–173, Mount Pleasant, MI, 2005.

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

[12] N. Rosa, G. Justo, and P. Cunha. Incorporating Non-functional Requirements into Software Architectures. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1009 – 1018, 2000.

[13] K. Schmidt. *High Availability and Disaster Recovery : Concepts, Design, Implementation*. Springer, 2006.

[14] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Principles*. Addison Wesley, 2005.

[15] L. Xu, H. Ziv, T. Alspaugh, and D. Richardson. An Architectural Pattern for Non-functional Dependability Requirements. *Journal of Systems and Software*, 79(10):1370–1378, 2006.

[16] G. Zarate and P. Botella. Use of UML for Modeling Non-functional Aspects. In *Proceedings of the 13th International Conference on Software & System Engineering and their Application*, Paris, France, 2000.