Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 57 (2005) 27–44

Science of Computer Programming

www.elsevier.com/locate/scico

Self-healing components in robust software architecture for concurrent and distributed systems

Michael E. Shin

Department of Computer Science, Texas Tech University, Lubbock, TX 79409, United States

Received 15 March 2004; received in revised form 20 September 2004; accepted 19 October 2004
Available online 13 January 2005

Abstract

This paper describes an approach to designing self-healing components for robust, concurrent and distributed software architecture. A self-healing component is able to detect object anomalies inside of the component, reconfigure inter-component and intra-components before and after repairing the sick object, repair it, and then test the healed object. For this, each self-healing component is structured to the layered architecture with two layers, the service layer and the healing layer, which are designed separately from each other. The service layer of a self-healing component provides functional services to other components, whereas the healing layer encapsulates the self-healing mechanism for monitoring objects in the service layer and repairing the sick objects detected. The process of component self-healing includes detection, reconfiguration before and after repairing, repair, and testing. To illustrate this approach, the elevator system is considered.
© 2004 Elsevier B.V. All rights reserved.

1. Introduction

Although verification and validation technologies for software components in concurrent and distributed systems have been enhanced, components of the systems may still hide design faults resulting in system failures or come across deadlock freezing the system. More specifically, unmanned control systems such as elevator control systems or critical systems such as spacecraft navigational systems need the robustness to detect

E-mail address: Michael.Shin@coe.ttu.edu.

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.10.003

and self-heal anomalies of the systems at runtime. Each component in these systems is either tested with test data under the operational profile of the software system after implementation, or transformed into its executable component before implementation [18] so that dynamic behavior of the system can be validated. However, a running system structured with components can reach a state that may not be expected in the design phase, or validation and verification phase of software development. To avoid this, robust concurrent and distributed systems need to be structured with self-healing components, each of which encapsulates a self-healing mechanism to autonomously detect anomalous objects in the component and heal anomalies of the objects at runtime.

Many efforts have been made to make running systems with anomalies more reliable. The traditional approach [5,11,13,19] is to insert fault tolerant mechanisms into systems where the mechanisms are performed when faults or errors are trapped. Faults are specified for any fault tolerant system so that the system can be evaluated as regards whether it is actually fault tolerant [12]. Other approaches [3,6,10,14] to handling anomalies of systems have systems retain abilities to heal themselves when they are sick. Such an approach has a self-healing mechanism to autonomously heal anomalies of the system [12].

This paper describes the design of components that are able to autonomously repair anomalies of objects constituting each component, which are the basis of robust software architecture for concurrent and distributed systems. Other approaches to self-healing systems have been focused on either the system level [3,6] or the component level [4,5,16] in which most approaches support part of the process of the self-healing mechanism, but not the whole process including anomalous object detection, dynamic reconfiguration before and after object self-healing, repair of sick objects, and testing of repaired objects. Thus, this paper concentrates on the whole process of the self-healing mechanism contained in each self-healing component.

This paper describes the architecture of self-healing components, each of which is designed with two layers: the service layer and the healing layer. The service layer of a self-healing component is structured to objects such as tasks (active objects), connectors, and passive objects (e.g., entity objects) accessed by tasks, providing functionality to service requests from other components. The service layer notifies the status of messages passing between objects to the healing layer where objects in the service layer are monitored using the notification messages from the service layer to detect sick objects in the component.

In this paper, the role of communication synchronization of connectors between tasks in the service layer of a component is extended to support the self-healing mechanism. A connector acts on behalf of tasks in terms of communication between tasks, encapsulating the details of inter-task communication such as synchronous or asynchronous message communications. Moreover, for a component to be enabled to heal itself, connectors between tasks in the component notify messages arrived from tasks to the healing layer in which the notification messages are used to detect anomalies of task threads. Once the healing layer makes a decision that an object in the service layer of the component becomes sick, the healing process is launched via connectors. To illustrate this approach, an elevator system is considered, which is described in [7].

The approach in this paper may be more applicable for soft real-time, concurrent and distributed component-based systems, rather than hard real-time systems that require a tight deadline for a specific service from components. The time required for repairing sick objects in a component may go beyond the deadline in hard real-time systems.

This paper begins by describing the architecture of self-healing components in concurrent and distributed systems in Section 2. Section 3 describes self-healing component-based software architecture for concurrent and distributed systems. Section 4 describes the process of the self-healing mechanism including detection, reconfiguration, repair, and test of anomalous objects. Section 5 describes the self-healing elevator system, used to apply the approach in this paper. Section 6 describes related work. Finally, Section 7 concludes this paper.

2. Self-healing component architecture

A concurrent and distributed system can be described by means of distributed components referred to as subsystems, their connectors, and the configuration. Components are designed as distributed component types [7], each of which defines functionality that is relatively independent of those provided by other components. A component is self-contained, that is, it can be compiled, instantiated, and linked separately into a distributed system [7]. A connector acts on behalf of components in terms of communication between components, encapsulating the details of inter-component communication. Once distributed components and their connectors are designed, a distributed system is configured on a real operating environment.

A component in the concurrent and distributed system can be further decomposed into objects, namely active objects (tasks), connectors between active objects, and passive objects (e.g., entity objects) accessed by tasks. An active object (a concurrent object or task) has its own thread of control, initiating actions that affect other active and passive objects [7]. Unlike an active object, a passive object has no thread of control; thus it cannot initiate any active objects. But a passive object is invoked by active objects and can invoke other passive objects. Because a passive object does not have its own thread, it performs its operations using the thread of the task that invoked the object. Tasks in a component can communicate with each other through connectors. On behalf of a task, connectors send messages to and receive them from other tasks. Like a connector between components, a connector between tasks within a component encapsulates a synchronous mechanism for message communication. More specifically, a connector can be classified to a message queue, a message buffer, and a message buffer and response connectors [7]. Containing a message queue, a message queue connector encapsulates the communication mechanism for loosely coupled message communication (asynchronous message communication), which provides two synchronized operations: send a message and receive a message. The size of queue is defined depending on the performance of components involved in communication. A message buffer connector is used to encapsulate the mechanism for tightly coupled message communication without reply, while a message buffer and response connector is designed for tightly coupled message communication with reply. Both the message buffer connector, and a message buffer and response connector are synchronous message communication where the size of buffer is one.

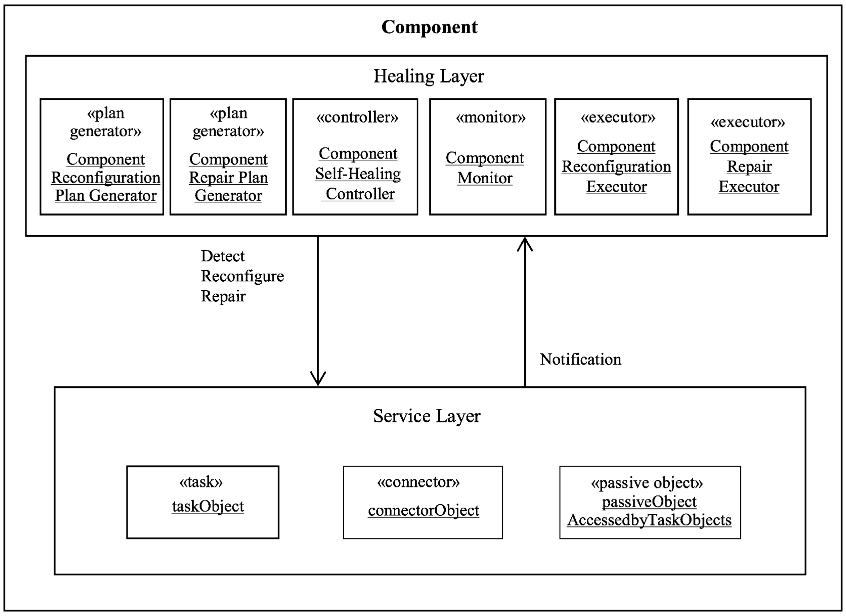
Component
Healing Layer
«plan generator» Component Reconfiguration Plan Generator
«plan generator» Component Repair Plan Generator
«controller» Component Self-Healing Controller
«monitor» Component Monitor
«executor» Component Reconfiguration Executor
«executor» Component Repair Executor
Detect
Reconfigure
Repair
Notification
Service Layer
«task» taskObject
«connector» connectorObject
«passive object» passiveObject AccessedbyTaskObjects

Fig. 1. Layered software architecture for a self-healing component.

A self-healing component is a component that is able to autonomously detect and repair abnormalities on itself as well as perform functional services requested from other components. A thread of each task in a component executes a statechart in which an incoming message to or outgoing message from the task is expected to transition its pre-defined statechart from one state to another within a specified time. The allowable time internal that the thread of a task can stay at a specific state may be specified depending on system properties such as hard real-time or soft real-time. For example, an Automatic Teller Machine (ATM) system in a bank may specify about 3 s for reading a customer debit card, while a poisonous gas detection system in a chemical factory may specify less than 0.1 s for sensing the leak of gas. Violating the expected transition of the task thread's statechart within a bounded time can be interpreted as the task faults that may lead to system failures. The task faults may be caused by software errors such as design errors, or hardware errors such as external I/O devices (e.g., sensors). Self-healing in this research focuses on software errors, not hardware errors.

Each self-healing component is designed as a layered architecture, structured with two layers (Fig. 1) – the service layer and the healing layer – on a component basis. In the normal phase, the service layer of a component provides full functionality to service requests from other components and notifies the status of messages passing between objects in that layer to the healing layer. The service layer of a component communicates with service layers of other components through connectors to provide functions to and request them from other components. The service layer is composed of tasks (active objects), passive objects accessed by tasks, and connectors between tasks. A connector not only encapsulates a mechanism synchronizing message communication between tasks, but also notifies messages passing it to the healing layer. Passive objects accessed by tasks (e.g., entity objects) also notify messages arrived from tasks to invoke their operations to the healing layer. With the message notification from the service layer, the healing layer of the component monitors objects in the service layer to detect any anomalous behavior of objects. Once the healing layer detects an anomalous object in the service layer and presumes that the object needs to be treated, the component changes its mode from the normal phase to the healing phase. In the healing phase, the service layer may not provide services any more or provide partial services by means of remaining healthy objects, whereas the healing layer involves repairing the sick object in the service layer. The healing layer reconfigures objects in the service layer of the component and, if needed, notifies the object sickness to other components requiring services from the sick component to minimize the impact on the components from the sick component. It then starts repairing the sick object.

The healing layer of each component depicted in Fig. 1 is composed of the *Component Reconfiguration Plan Generator, Component Repair Plan Generator, Component Self-Healing Controller, Component Monitor, Component Reconfiguration Executor*, and *Component Repair Executor*, which are responsible for detection, reconfiguration and repair of objects in the service layer. The *Component Monitor* contains statecharts for each task thread in the service layer, which model dynamic behavior of the task thread. With messages notified by both connectors between tasks and passive objects accessed by tasks in the service layer, the *Component Monitor* supervises the behavior of tasks, connectors and passive objects accessed by tasks using statecharts for the task threads. The *Component Reconfiguration Plan Generator* maintains configuration information of objects in the service layer, generating a reconfiguration plan in response to changes to the status of objects in the service layer within the component. The *Component Reconfiguration Plan Generator* also involves a list of other components, whose objects may need to be reconfigured to minimize the impact from the paralyzed object in the neighboring component and to provide services continuously without stopping any more than necessary. To achieve this, the *Component Reconfiguration Plan Generator* of a component maintains information on the interconnection with other components in the system, and generates a reconfiguration plan if there are changes in the configuration of objects in other components. The *Component Repair Plan Generator* maintains knowledge of repair plans specific to each object such as a task, connector, and passive object accessed by tasks in the service layer of a component, and generates repair plans for repairing sick objects. The *Component Reconfiguration Executor* substantially carries out the reconfiguration plan generated by the *Component Reconfiguration Plan Generator* to reorganize the objects in the component in response to sick objects in the component or different components. The *Component Repair Executor* performs the repair plan generated by the *Component Repair Plan Generator* to treat abnormal objects and test them after repairing in order to check whether the objects just repaired work normally. The *Component Self-Healing Controller* of a component coordinates the *Component Reconfiguration Plan Generator, Component Repair Plan Generator, Component Monitor, Component Reconfiguration Executor*, and *Component Repair Executor* to conduct self-healing processes for the sick

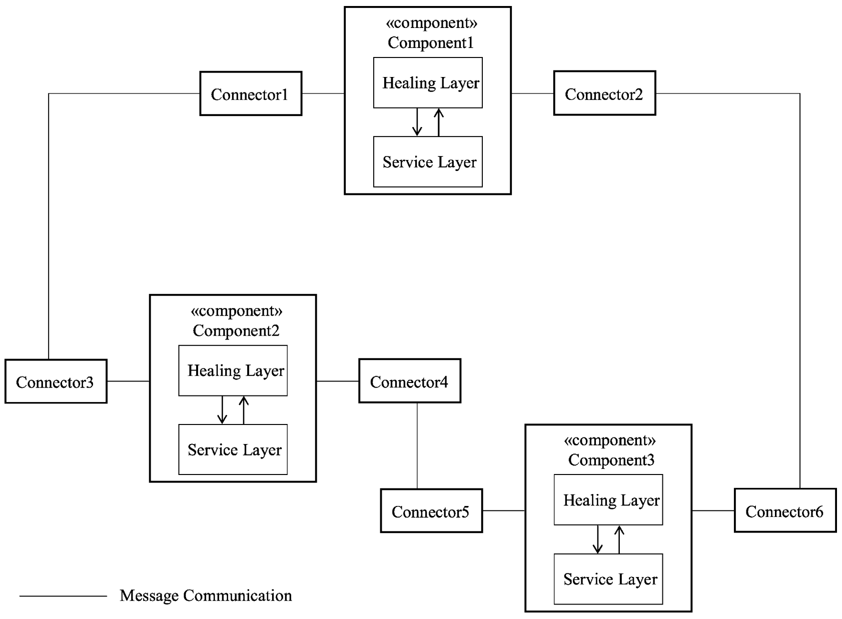

Fig. 2. Self-healing component-based software architecture for concurrent and distributed systems.

objects, cooperating with the *Component Self-Healing Controllers* of other components for reconfiguration against the object anomalies.

3. Self-healing component-based software architecture

Software architecture [2,8,17] for a robust concurrent and distributed system can be designed by means of self-healing components and their connectors. Self-healing components communicate with each other via connectors between them. In the normal phase, the service layer of a component may request services from the service layers of other components, or provide services to them through connectors. When detecting an object in the service layer of a component to be sick, the healing layer of the component reconfigures objects in the service layer using a reconfiguration plan generated by its *Component Reconfiguration Plan Generator*, and notifies its neighboring components of the object sickness through connectors. The notification messages are delivered to the healing layers of neighboring components, which reconfigure objects in the service layers of the components using their reconfiguration plans generated by their *Component Reconfiguration Plan Generators* to minimize the impact from the neighbor's sickness.

Fig. 2 depicts self-healing component-based software architecture for resilient concurrent and distributed systems. If the healing layer of the Component1 (in Fig. 2) detects a sick object in the service layer, it generates a reconfiguration plan responding to the sick object. On the basis of the reconfiguration plan, the healing layer reconfigures objects in the service layer and then may notify the healing layers of the Component2 and the Component3 via Connector1, Connector3, Connector2, and Connector6 (in Fig. 2),

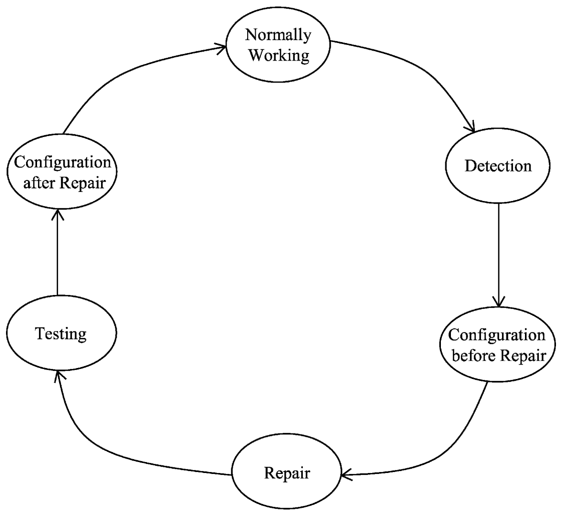

Fig. 3. Component self-healing process.

which reconfigure their objects in the service layers using their reconfiguration plans. After repairing the sick object, software architecture is also dynamically reconfigured as the sick object is returned to normal.

4. Self-healing mechanism

Self-healing of components is carried out in a sequence: (1) detection of the object anomaly, (2) reconfiguration before repairing the anomalous object, (3) repairing of the sick object, (4) testing of the repaired object, and finally (5) reconfiguration after repairing of the sick object. Fig. 3 depicts the component self-healing process against anomalous objects in components. Once a self-healing component detects anomalous behavior of a task thread within the component, it launches the self-healing procedure for fixing it. The component is reconfigured at runtime to isolate the abnormal object for repairing from the healthy objects, and then starts repairing the sick object. Repair goes through, based on the repair plan for the specific object. After repairing the sick object, the component tests the repaired object using test data that are pre-generated for the sick object. As the sick object becomes healthy, the component is dynamically reconfigured back into the normal service.

4.1. Detection

Connectors between tasks in the service layer of each component play an important role in a self-healing component architecture. They not only synchronize message communication between two tasks, but also notify message arrivals from tasks to the *Component Monitor* in the healing layer. Connectors also acknowledge their status to the *Component Monitor* after they store messages in queues or buffers in the connectors, or deliver messages to other tasks on behalf of their associated tasks. Using the messages

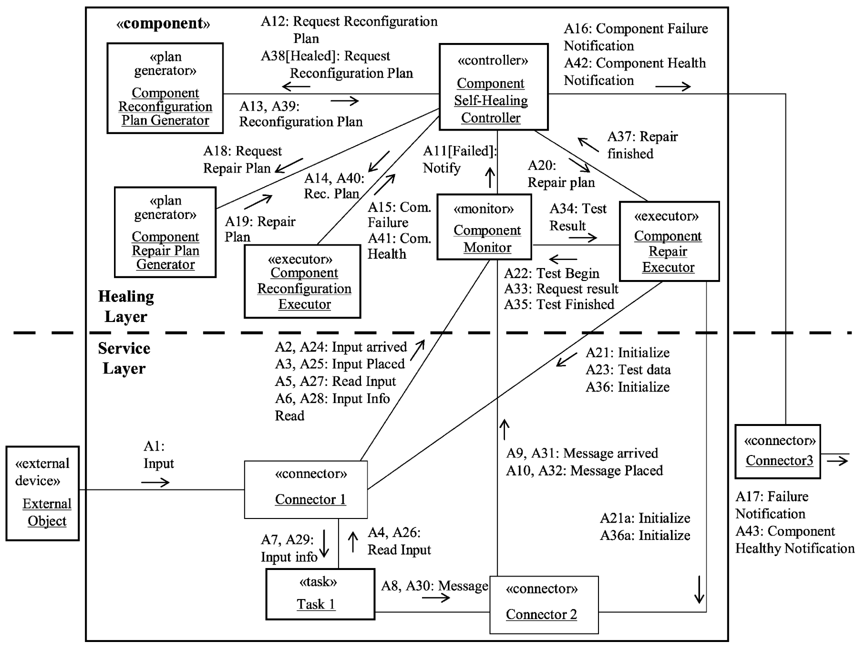

Fig. 4. Message sequence between healing layer and service layer of a component.

from connectors, the *Component Monitor* detects any behavioral abnormality of tasks and connectors in the component.

A passive object accessed by several tasks in the service layer of a component also needs to notify the *Component Monitor* when its operation is invoked by a task and when the operation has been executed successfully. The trace of a task thread within a passive object can determine the sickness of the passive object. It is analogous to a connector that notifies the trace of a task thread within a connector to the *Component Monitor*.

Abnormality of tasks, connectors, and passive objects accessed by tasks in the service layer of a component is detected using statecharts, which are executed by message events from connectors and passive objects accessed by tasks. The *Component Monitor* (in Fig. 1) contains the statecharts for threads of tasks in the service layer of a component, monitoring the behavior of tasks, connectors, and passive objects accessed by tasks by tracing expected state transitions of each thread. Statecharts in the *Component Monitor* describe the behavior of connectors associated with a task and passive objects accessed by a task, as well as a task itself. An incoming message to the *Component Monitor*, which is notified by a connector or a passive object accessed by tasks, causes a transition in the statechart for the task thread. The statechart includes states and transitions corresponding to tasks, connectors neighboring the task and passive objects accessed by the task.

Using the collaboration model of the UML [1,15], a message sequence between objects in the service layer and healing layer of a component is depicted in Fig. 4, which is a simplified message communication for self-healing of a component. The message sequence A1 through A10 describes the normal service of Task1, Connector1, and Connector2.

When the Connector1 receives a message A1, it notifies the *Component Monitor* of the message arrival (A2 in Fig. 4), which makes the *Component Monitor* wait for the next message “Input Placed” (A3 in Fig. 4) from the Connector1. After placing a message in a queue or buffer, the Connector1 again notifies a message “Input Placed” (A3 in Fig. 4) to the *Component Monitor*. The Task1 reads the input from the queue or buffer (A4 and A7 in Fig. 4) and the Connector1 notifies the fact to the *Component Monitor* through messages — Read Input (A5) and Input Info Read (A6). This message notification results in a transition in the statechart for the Task1 thread, making the *Component Monitor* expect the next message “Message Arrived” (A9 in Fig. 4) from the Connector2. The Task1 processes the arrived message and sends a message “Message” (A8 in Fig. 4) to the Connector2, which places the message in the queue or buffer that is read by other tasks. Just after the Connector2 receives the message A8, it notifies the “Message Arrived” (A9 in Fig. 4) to the *Component Monitor*. Similarly, after the Connector2 places the message in the queue or buffer, it informs the *Component Monitor* of the message placed (A10 in Fig. 4).

The *Component Monitor* presumes that an object such as the Connector1, Task1, or Connector2 (in Fig. 4) may be sick if the expected messages have not arrived within reasonable time intervals. In that case, the *Component Monitor* reports sickness of the object (A11 in Fig. 4) to the *Component Self-Healing Controller*, which in turn takes an appropriate action against the sick object.

4.2. Reconfiguration before repair

With the notification of sickness of an object by the *Component Monitor*, the *Component Self-Healing Controller* reconfigures other objects affected by the sick object in the component and in other components that request functional services from the component containing the sick object. The reconfiguration isolates the sick object from healthy objects at runtime so that the healthy objects have minimal impact from repair of the sick object. Under partial or no services from the sick component, the components requiring the services from the sick component reconfigure themselves to continue to provide their functional services to other components. For reconfiguration, the *Component Self-Healing Controller* consults with the *Component Reconfiguration Plan Generator* in the healing layer.

The *Component Reconfiguration Plan Generator* in the healing layer generates a reconfiguration plan against each sick object, based upon the current configuration of the component. Using the reconfiguration plan for a sick object, the *Component Self-Healing Controller* has *Component Reconfiguration Executor* reconfigure the service layer of the component and notify the sick symptom of the object to the other components. The sickness notification from a neighbor is delivered to the *Component Self-Healing Controllers* in the healing layers of the other components, which also consult with their own *Component Reconfiguration Plan Generators* to obtain reconfiguration plans against the symptom of the sick object. Following the reconfiguration plans, the *Component Reconfiguration Executors* undertake the reconfiguration of their own objects in the service layers.

Before repair and testing of the sick object, reconfiguration is carried out to block objects related to the repair and testing. The *Component Reconfiguration Executor* sends the (incoming) connectors (to the sick task) a message requesting blocking adding a new message in the queues or buffers, while it sends the (outgoing) connectors (from the sick task) a message requesting blocking reading test data from the queues or buffers in the connectors. When receiving the message, the connectors respectively freeze message communication with sender and receiver tasks outside of repairing and testing. The objects involved in the repair and testing are confined to the sick task, (incoming) connectors from which the sick task reads messages, (outgoing) connectors to which the sick task adds messages for delivery, and passive objects accessed by the sick task.

Reconfiguration before repairing a sick object is depicted in Fig. 4, the message sequence A12 through A17. The *Component Monitor* informs the *Component Self-Healing Controller* of the existence of a sick object in the service layer (A11 in Fig. 4) and the *Component Self-Healing Controller* requests a reconfiguration plan for objects in the component from the *Component Reconfiguration Plan Generator* (A12 in Fig. 4). The reconfiguration plan (A13 in Fig. 4) generated by the *Component Reconfiguration Plan Generator* is performed by the *Component Reconfiguration Executor* (A14 in Fig. 4). The reconfiguration plan includes an object list involved in the reconfiguration and directives of reconfiguration. Suppose that the Task1 in Fig. 4 is out of service. The configuration plan contains the Connector1, which should not allow the Task1 to read messages any more. The message delivery to the Task1 is meaningless because the Task1 is not working normally. The reconfiguration plan may contain additionally the other component list that should be notified through messages A15 through A17.

4.3. Repairing and testing

After reconfiguration of the objects and components associated with the sick object, the repairing process starts with requesting a repair plan from the *Component Repair Plan Generator* by the *Component Self-Healing Controller* in the healing layer, which contains repair plans for sick objects in a component. The *Component Repair Plan Generator* generates a plan for repairing the sick object that may indicate re-initialization or re-installation depending on the object type in the service layer such as task, connector, or passive object accessed by tasks. The plan may contain a directive for a virus scan for detecting virus infection to tasks. The repair plan is delivered to the *Component Repair Executor* through the *Component Self-Healing Controller*, which performs the plan as specified in order to make the component resilient to its failure. To avoid the same design errors in the implementation of the sick object, the *Component Repair Executor* may re-install a variant [9] of the original version of the object that implements the same functionality. Repairing the sick object is followed by testing of the repaired object with a set of test data.

Testing of the repaired object varies with the task, connector, and passive object accessed by tasks. Testing of a repaired task may begin by initializing connectors relevant to a repaired task. The testing procedure for a repaired task is very similar to that of normal service, except that the input to the connectors communicating with the repaired task is provided by the *Component Repair Executor* in the healing layer instead of tasks in the service layer of a component.

Fig. 4 depicts repairing and testing of a sick task, Task1, messages A18 through A37. The *Component Repair Plan Generator* generates a repair plan for the sick Task1 (A19 in Fig. 4) in response to the request from the *Component Self-Healing Controller* (A18 in Fig. 4), which is delivered to the *Component Repair Executor* (A20 in Fig. 4). Following the repair plan, the *Component Repair Executor* fixes the sick Task1 and then starts testing. Prior to sending test data to the connectors, the *Component Repair Executor* requests the connector queues or buffers to be initialized ready to receive test data (A21 and A21a in Fig. 4). The *Component Repair Executor* then initializes statecharts in the *Component Monitor*, which are associated with the repaired task in order to execute them on the basis of the messages coming from the connectors or passive objects accessed by tasks for testing (A22 in Fig. 4). At this time, the *Component Monitor* changes the mode of the statecharts from the normal state to the testing state. After finishing the testing procedure (A23 through A32 in Fig. 4), the *Component Repair Executor* requests the final state of the statecharts for the repaired task from the *Component Monitor* (A33 in Fig. 4). With the state (A34 in Fig. 4), the *Component Repair Executor* makes a decision on whether the sick task has been treated correctly and wraps up the testing procedure by re-initializing the *Component Monitor* (A35 in Fig. 4) and associated connectors (A36 and A36a in Fig. 4), and notifying the result to the *Component Self-Healing Controller* (A37 in Fig. 4). On the basis of the testing result, the *Component Self-Healing Controller* reconfigures the associated objects in the component and the other components.

4.4. Reconfiguration after repairing

The objects and components reconfigured for repairing the sick object are back to the original configuration as the repaired object resumes its service. The *Component Reconfiguration Executor* takes steps for reconfiguration of the component containing the repaired object and other components affected by the sick component, on the basis of the repairing results. If the repair is successfully completed, the *Component Self-Healing Controller* requests a plan for reconfiguring components from the *Component Reconfiguration Plan Generator* (A38 and A39 in Fig. 4). Using the reconfiguration plan generated by the *Component Reconfiguration Plan Generator* (A40 in Fig. 4), the *Component Reconfiguration Executor* reconfigures the component containing the repaired object and notifies the healthy state of the component (A41 through A43 in Fig. 4) to other components previously reconfigured to reduce the impact from the paralyzed component. The components that are notified of their neighbor's health reconfigure their objects blocked temporarily in communication with the recovered neighbor. Otherwise, if the repairing is not handled successfully, the *Component Self-Healing Controller* may need to notify the unhealed objects to the supervisor (i.e., human) who may monitor the system.

5. Self-healing elevator system

The approach to self-healing is applied to the elevator system [7], which is structured into three components (subsystems) such as elevator, floor and scheduler. The elevator component is a control component that provides overall coordination of devices such as elevator lamp, motor, door, arrival sensor, and elevator button. This component may be

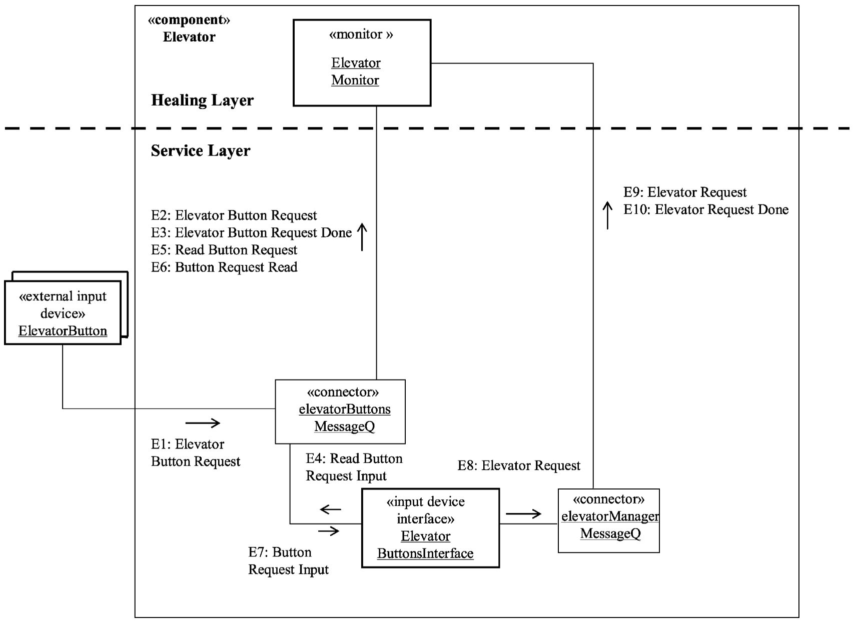
«component» Elevator
«monitor» Elevator Monitor
Healing Layer
Service Layer
E2: Elevator Button Request
E3: Elevator Button Request Done
E5: Read Button Request
E6: Button Request Read
E9: Elevator Request
E10: Elevator Request Done
«external input device» ElevatorButton
«connector» elevatorButtons MessageQ
E1: Elevator Button Request
E4: Read Button Request Input
E8: Elevator Request
«input device interface» Elevator ButtonsInterface
«connector» elevatorManager MessageQ
E7: Button Request Input

Fig. 5. Part of self-healing elevator component.

configured into several instances corresponding to the number of elevators in a building. Each instance of the elevator component type consists of a passive object such as local elevator status and plan, and four active objects (tasks) such as elevator controller, arrival sensors interface, elevator manager, and elevator buttons interface. The floor component collects floor button requests from elevator users and requests elevators to be sent to the floors from the scheduler component. The scheduler component is a coordinator component, which schedules all elevators in a building replying to the elevator requests from floor components.

Detection

The part of the self-healing Elevator Component in the elevator system is depicted in Fig. 5, which has the Elevator Monitor in the healing layer, and the Elevator Buttons Message Queue connector, the Elevator Buttons Interface task and the Elevator Manager Message Queue connector in the service layer. Message communication among objects in the self-healing Elevator Component is modeled using the collaboration model in UML [1,7,15]. An elevator user's button request for a destination floor is sent to the Elevator Buttons Message Queue connector. When the connector receives the Elevator Request message (E1 in Fig. 5), it first notifies the arrival of the message to the Elevator Monitor (E2 in Fig. 5), and then adds the message to the message queue, and again notifies to the Elevator Monitor regarding the successful placement of the message on the queue (E3 in Fig. 5). When the Elevator Monitor receives the messages (E2 and E3 in Fig. 5) from the Elevator Buttons Message Queue connector, the statechart (on the left below in Fig. 6)

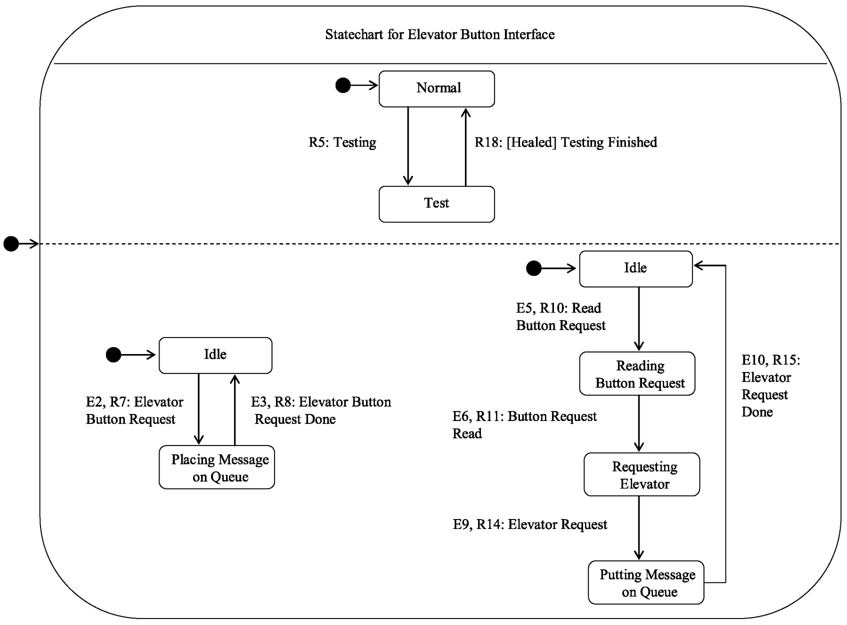
Statechart for Elevator Button Interface
Normal
R5: Testing
R18: [Healed] Testing Finished
Test
Idle
E2, R7: Elevator Button Request
E3, R8: Elevator Button Request Done
Placing Message on Queue
Idle
E5, R10: Read Button Request
Reading Button Request
E6, R11: Button Request Read
Requesting Elevator
E9, R14: Elevator Request
Putting Message on Queue
E10, R15: Elevator Request Done

Fig. 6. Statechart for elevator buttons interface.

encapsulated in the Elevator Monitor transitions from the Idle state to the *Placing Message on Queue* state with respect to the message E2, and then transitions to the Idle state in response to the message E3. When the statechart returns to the Idle state, the Elevator Monitor expects the Elevator Buttons Interface task to read a button request input from the queue in the Elevator Buttons Message Queue connector. The Read Button Request message (E5 in Fig. 5) acknowledges arrival of the Read Button Request Input message (E4) at the Elevator Buttons Message Queue connector to the Elevator Monitor. The message E5 executes the statechart (on the right below in Fig. 6) to transition from the Idle state to the *Reading Button Request* state. When the Elevator Buttons Interface task finishes reading a button request input (E7), the Elevator Button Message Queue connector notifies the Button Request Read message (E6) to the Elevator Monitor, which makes the statechart transition from the *Reading Button Request* state to the *Requesting Elevator* state. Similarly, the messages, the Elevator Request (E9) and the Elevator Request Done (E10), notified by the Elevator Manager Queue connector to the Elevator Monitor makes the statechart reach the Idle state through the *Putting Message on Queue* state. When the statechart reaches the Idle state, the Elevator Monitor is convinced that the Elevator Buttons Interface task, the Elevator Buttons Message Queue connector, and the Elevator Manager Message Queue connector work correctly. The Elevator Monitor may need to maintain both the number of buttons pressed by the user, and the number that the Elevator Buttons Interface task has read from the inputs for the destination of the elevator. This is because the processing speed of the Elevator Buttons Interface task may be different from that of the Elevator Button input device.

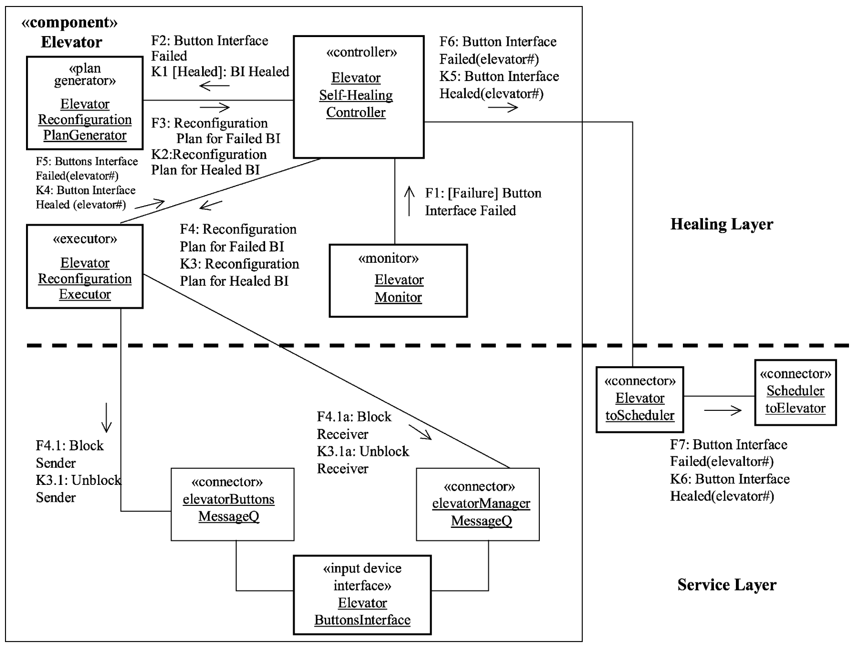
«component» Elevator
«plan generator» Elevator Reconfiguration PlanGenerator
«controller» Elevator Self-Healing Controller
«executor» Elevator Reconfiguration Executor
«monitor» Elevator Monitor
«connector» elevatorButtons MessageQ
«connector» elevatorManager MessageQ
«input device interface» Elevator ButtonsInterface
«connector» Elevator toScheduler
«connector» Scheduler toElevator
F1: [Failure] Button Interface Failed
F2: Button Interface Failed
K1 [Healed]: BI Healed
F3: Reconfiguration Plan for Failed BI
K2:Reconfiguration Plan for Healed BI
F4: Reconfiguration Plan for Failed BI
K3: Reconfiguration Plan for Healed BI
F4.1: Block Sender
K3.1: Unblock Sender
F4.1a: Block Receiver
K3.1a: Unblock Receiver
F5: Buttons Interface Failed(elevator#)
K4: Button Interface Healed (elevator#)
F6: Button Interface Failed(elevator#)
K5: Button Interface Healed(elevator#)
F7: Button Interface Failed(elevaltor#)
K6: Button Interface Healed(elevator#)
Healing Layer
Service Layer

Fig. 7. Elevator reconfiguration for sick and healthy elevator buttons interface.

While monitoring objects (tasks, connectors, and passive objects accessed by tasks) in the service layer of each component using statecharts, the Elevator Monitor in the healing layer may detect a sick object based on comparing the real execution of the statecharts to its expectation. For example, when the statechart (on the left below in Fig. 6) transitions from the *Placing Message on Queue* state to the Idle state, which means an elevator button request message has arrived at the queue, the Elevator Monitor expects that the Elevator Buttons Interface task should read it. If the task does not read it within a bounded time, it is suspected by the Elevator Monitor to be sick. In this case, the *Elevator Monitor* reports the sick object to the *Elevator Self-Healing Controller* in the healing layer, which launches the self-healing mechanism against the sick Elevator Buttons Interface task.

Reconfiguration before repair

Fig. 7 depicts the reconfiguration of the Elevator component against the sick Elevator Buttons Interface task. When the Elevator Self-Healing Controller is notified of the anomaly of the Elevator Buttons Interface task from the Elevator Monitor (F1 in Fig. 7), it requests from the Elevator Reconfiguration Plan Generator (F2 in Fig. 7) a reconfiguration plan for blocking objects directly associated with the sick task and minimizing the impact on other components from the sick Elevator Buttons Interface task. On the basis of the plan (F3 in Fig. 7), the Elevator Self-Healing Controller requests the Elevator Reconfiguration Executor to isolate the Elevator Buttons Interface task (F4 in Fig. 7). To achieve this, the Elevator Reconfiguration Executor sends the Block Sender message (F4.1 in Fig. 7) to the Elevator Buttons Message Queue connector, which blocks adding a new button request from the external Elevator Button device to the queue in the Elevator Buttons Message Queue connector. In the meantime, the Elevator Reconfiguration Executor sends the Elevator Manager Message Queue connector the Block Receiver message (F4.1a in Fig. 7) in order for other tasks not to read messages from the queue in the Elevator Manager Message Queue connector. The Elevator Reconfiguration Executor then notifies the sickness of Elevator Buttons Interface task to the Scheduler component through both the *Elevator to Scheduler* connector and the *Scheduler to Elevator* connector (F5 through F7 in Fig. 7).

Repair and testing

After reconfiguration for repairing the sick Elevator Buttons Interface task, the *Elevator Self-Healing Controller* requests a repair plan for the sick *Elevator Buttons Interface* task from the *Elevator Repair Plan Generator* (R1 in Fig. 8) that maintains repair plans for tasks, connectors, and passive objects accessed by the tasks in the Elevator Component. The repair plan is forwarded to the *Elevator Repair Executor*, which treats the sick task as planned (R2 and R3 in Fig. 8). On the basis of the repair plan, the *Elevator Repair Executor* may re-install the Elevator Buttons Interface task, and takes steps for testing it by initializing message queues in the connectors – the Elevator Buttons Message Queue and the Elevator Manager Message Queue connectors – to accommodate the test data (R4 and R4a in Fig. 8). In the meantime, the Elevator Monitor in the healing layer re-initializes the statecharts relevant to the Elevator Buttons Interface task and changes the mode from the normal state to the testing state (R5 in Fig. 8 and upper statechart in Fig. 6).

When finishing preparation of testing, the *Elevator Repair Executor* starts testing the repaired Elevator Buttons Interface task by sending a message for testing (R6 in Fig. 8) to the Elevator Buttons Message Queue connector. The messages arrived at the connectors – the Elevator Buttons Message Queue and the Elevator Manager Message Queue connectors – are notified to the Elevator Monitor for overseeing the transitions of the statecharts for the Elevator Buttons Interface task (R7, R8, R10, R11, R14 and R15 in Fig. 8). Each message notified from the connectors to the Elevator Monitor results in a transition in the statechart (on the lower right in Fig. 6) from one state to another. When all the testing procedure is finished, the *Elevator Repair Executor* requests the testing results of the repaired Elevator Buttons Interface task from the Elevator Monitor (R16 and R17 in Fig. 8). If the sick task is healed successfully, the *Elevator Repair Executor* requests the Elevator Monitor to re-initialize the statecharts relevant to the repaired task (R18 in Fig. 8) and change the mode from the testing state to the normal state (on the upper statechart in Fig. 6), re-initializing the queues and variables in the connectors – the Elevator Buttons Message Queue and the Elevator Manager Message Queue connectors (R19 and R19a in Fig. 8). And then the *Elevator Repair Executor* acknowledges the results of the repairing and testing to the *Elevator Self-Healing Controller* that takes measures based upon the results (R20 in Fig. 8).

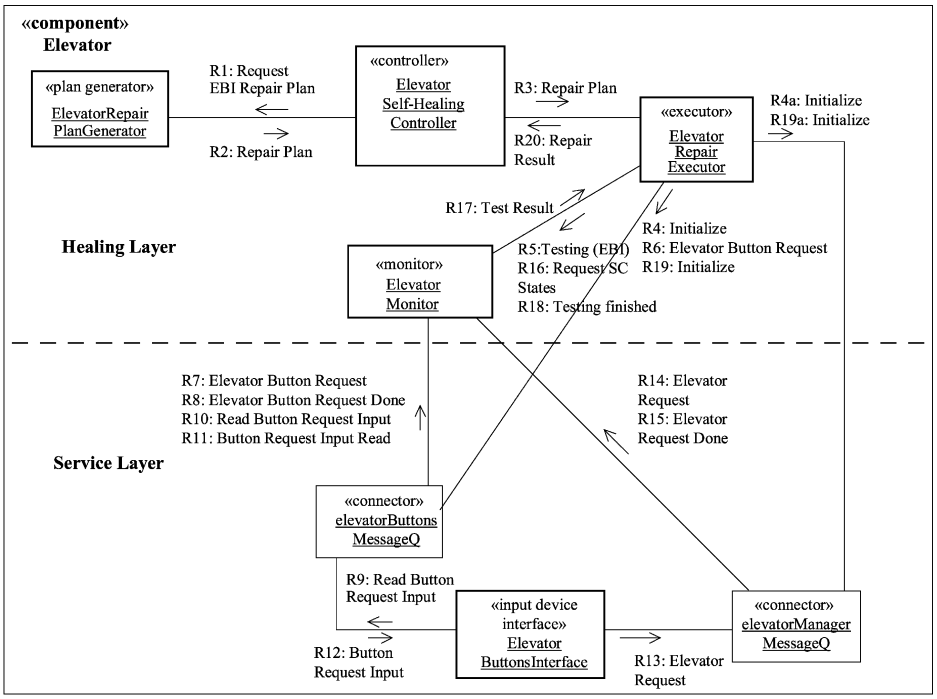
«component» Elevator
«plan generator» ElevatorRepair PlanGenerator
R1: Request EBI Repair Plan
R2: Repair Plan
«controller» Elevator Self-Healing Controller
R3: Repair Plan
R20: Repair Result
«executor» Elevator Repair Executor
R4a: Initialize
R19a: Initialize
R17: Test Result
R4: Initialize
R6: Elevator Button Request
R19: Initialize
Healing Layer
«monitor» Elevator Monitor
R5:Testing (EBI)
R16: Request SC States
R18: Testing finished
R7: Elevator Button Request
R8: Elevator Button Request Done
R10: Read Button Request Input
R11: Button Request Input Read
R14: Elevator Request
R15: Elevator Request Done
Service Layer
«connector» elevatorButtons MessageQ
R9: Read Button Request Input
R12: Button Request Input
«input device interface» Elevator ButtonsInterface
R13: Elevator Request
«connector» elevatorManager MessageQ

Fig. 8. Repair and testing of elevator buttons interface task.

Reconfiguration after repairing

The message sequence for reconfiguration of the Elevator component after repairing the Elevator Buttons Interface task is depicted in Fig. 7 (K1 through K6), which is the reverse of reconfiguration before repairing the Elevator Buttons Interface task. The Scheduler component also needs to be reconfigured as the Elevator component is completely back to work.

6. Related work

[12] has proposed a taxonomy for describing potential research issues in self-healing systems. In this taxonomy, self-healing system approaches are characterized with fault models, system responses, system completeness, and design context. In [4], the authors have presented the structure of a fault-tolerant component based on the C2 architectural style. The iC2C (Idealized C2 Component) is structured to two parts: one for maintaining the normal behavior of the component and detecting errors, and the other for being responsible for error recovery. These parts communicate with each other through connectors specialized from C2 connectors in the C2 architecture style. This approach focuses on the fault detection using pre- and post-condition, and invariants of operations, while our approach is concerned with self-healing of a component in terms of detection,

reconfiguration, repair, and test of a component. [3] has presented tools and methods for implementing architecture-based self-healing systems. The changes to a running software system are handled at the architectural level. In particular, this approach focuses on flexible architectural changes after the system was deployed. Rather than being concerned with a software architectural repair in [3], our approach concentrates on a self-healing architecture encapsulating a healing mechanism on a component basis. [6] uses architectural models as the basis for monitoring, problem detection, and repair for self-healing systems. The architectural models can be specialized to the particular style of the system such as performance, reliability or security. For reconfiguration of components, [16] describes an approach to devising a reconfiguration mechanism for Enterprise JavaBeans (EJB) as a component model. This approach is supported by the BARK reconfiguration tool, which is designed to facilitate the management and automation of the deployment life cycle for EJB.

7. Conclusions

This paper has described an approach to designing self-healing components in software architecture for robust, concurrent and distributed software systems. Each self-healing component is structured with the service layer and the healing layer. At each component, the service layer that consists of objects such as tasks (active objects), connectors, and passive objects accessed by tasks provides functionality to service requests from other components, while the healing layer encapsulating the self-healing mechanism for objects in the service layer launches the mechanism when it detects anomalies of objects in the service layer. In this architectural approach, connectors between tasks provide the notification mechanism for self-healing as well as the synchronization mechanism for concurrent message communication. Connectors between tasks have formed the basis of detection, reconfiguration, repair, and testing of anomalous objects.

In addition, this paper has described the self-healing mechanism encapsulated in the healing layer of each component, which is the dynamic perspective of self-healing component architecture. The life cycle of the self-healing mechanism is described by means of detection, reconfiguration before and after repairing, repair, and testing. The mechanism has been illustrated with the elevator self-healing component.

The approach in this paper retains several research issues associated with self-healing. In this paper, the plans for reconfiguration and repair are considered as black boxes, which should be specified in detail. In addition, an advanced decision mechanism would be necessary to reduce the rate of error in the detection of object anomalies in the healing layer. The decision mechanism is likely to need to consider total statuses of all objects in a component to reach its final decision on an object anomaly. Moreover, this approach may be extended to handling gradual degradation of performance [6], which requires an additional constituent in the healing layer to measure the fluctuation of performance.

References

[1] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1999.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, Pattern Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996.

[3] E.M. Dashofy, A. van der Hoek, R.N. Taylor, Towards architecture-based self-healing systems, in: Workshop on Self-Healing Systems, Proceedings of the First Workshop on Self-Healing Systems, November 18–19, Charleston, SC, 2002.
[4] P.A. de C. Guerra, R. de Lemos, An idealized fault-tolerant architectural component, in: Workshop on Architecting Dependable Systems, ICSE'02 International Conference on Software Engineering, May 25, Orlando, FL, 2002.
[5] P.A. de C. Guerra, R. de Lemos, An idealized fault-tolerant architectural component, in: Workshop on Architecting Dependable Systems, ICSE'02 International Conference on Software Engineering, May 25, Orlando, FL, 2002.
[6] D. Garlan, B. Schmerl, Model-based adaptation for self-healing systems, in: Workshop on Self-Healing Systems, Proceedings of the First Workshop on Self-Healing Systems, November 18–19, Charleston, SC, 2002.
[7] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
[8] H. Gomaa, D.A. Menasce, M.E. Shin, Reusable component patterns for distributed software architectures, in: 2001 Symposium on Software Reusability, SSR'01 Sponsored by ACM/SIGSOFT, May 18–20, Toronto, Ontario, Canada, 2001.
[9] H. Gomaa, M.E. Shin, Multiple-view meta-modeling of software product lines, in: The Eighth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2002, Maryland, December 2002.
[10] IBM, An architectural blueprint for autonomic computing, IBM and Autonomic Computing, 2003.
[11] K.H. Kim, C. Subbaraman, Fault-tolerant real-time objects, Communications of the ACM 40 (1) (1997).
[12] P. Koopman, Elements of the self-healing system problem space, in: Workshop on Software Architectures for Dependable Systems, WADS2003, ICSE'03 International Conference on Software Engineering, May 3–11, Portland, Oregon, 2003.
[13] H. Kopetz, G. Grundteidl, TTP — a protocol for fault-tolerant real-time systems, IEEE Computer (1994) 14–23.
[14] P. Oriezy, M.M. Gorlick, R.N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, A. Wolf, An architecture-based approach to self-adaptive software, IEEE Intelligent Systems 14 (3) (1999) 54–62.
[15] J. Rumbaugh, G. Booch, I. Jacobson, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, MA, 1999.
[16] M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, A.L. Wolf, Reconfiguration in the enterprise JavaBean component model, in: Proceedings of the IFIP/ACM Working Conference on Component Deployment, Berlin, 2002, pp. 67–81.
[17] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.
[18] M.E. Shin, A. Levis, L. Wagenhals, Mapping of UML-based system model to design/CPN model for system model evaluation, in: Workshop on Compositional Verification of UML'03, October 22, San Francisco, 2003.
[19] J. Xu, B. Randell, A. Romanovsky, C.M.F. Rubira, R.J. Stroud, Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery, in: Proc. 25th Int. Symp. on Fault-Tolerant Computing, Pasadena, June 1995, pp. 499–508.