Connector-Based Self-Healing Mechanism for Components of a Reliable System

Michael E. Shin
Department of Computer Science
Texas Tech University
Lubbock, TX 79409-3104
(806) 742-3527
Michael.Shin@ttu.edu

Daniel Cooke
Department of Computer Science
Texas Tech University
Lubbock, TX 79409-3104
(806) 742-3527
Daniel.Cooke@ttu.edu

ABSTRACT

This paper describes the self-healing mechanism for components in reliable systems. Each component in a self-healing system is designed as a layered architecture, structured with the healing layer and the service layer. The healing layer of a self-healing component is responsible for detection of anomalous objects in the service layer, reconfiguration of the service layer, and repair of anomalous objects detected. The service layer of a self-healing component provides functionality to other components, which consists of tasks (concurrent or active objects), connectors, and passive objects accessed by tasks. A connector supports the self-healing mechanism for self-healing components as well as encapsulates the synchronization mechanism for message communication between tasks in a component. Connectors are involved in detection of anomalous objects, reconfiguration of components, and repair of anomalous objects. This paper also specifies connectors - the message queue self-healing connector, message buffer self-healing connector, and message buffer and response self-healing connector - which provide functionalities for the self-healing mechanism.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture.

General Terms

Design

Keywords

Connector, Self-Healing Mechanism, Component

1. INTRODUCTION

Each component in concurrent and distributed systems can be structured with objects, namely tasks (i.e., active or concurrent objects), passive objects accessed by tasks (e.g., entity objects), and connectors between tasks. A task has its own thread of control, initiating actions that affect other tasks and passive objects [4]. Unlike a task, a passive object has no thread of control; thus it cannot initiate any tasks. But a passive object is invoked by tasks and can invoke other passive objects. Connectors encapsulate the synchronization mechanism for message communication between tasks. On behalf of a task, connectors send messages to and receive them from other tasks.

The components in a reliable system that are structured with objects such as tasks, connectors between tasks, and passive objects accessed by tasks need to be self-managed against anomalies of the objects. More specifically, unmanned control systems such as elevator control systems or critical systems such as spacecraft navigational systems need the robustness to detect and self-heal anomalies of the system at run-time. To achieve this, reliable systems need to be composed of self-healing components, each of which encapsulates the self-healing mechanism to autonomously detect anomalous objects in a component and heal anomalies of the objects at run-time.

This paper describes an approach to the connector-based self-healing mechanism for self-healing systems [2, 3, 7, 9, 11], in which connectors between tasks in components play important roles in detecting, reconfiguring, and repairing anomalous objects in components. The self-healing mechanism encapsulated in a component is realized by means of connectors between tasks. In particular, this paper describes the specifications of connectors supporting the self-healing mechanism for reliable components, which is extended from the previous research [12] addressing the self-healing mechanism. Although the self-healing mechanism for components of reliable systems should include interaction between a sick component containing anomalous objects and its neighboring components (i.e., components requesting services from the sick component) at the level of software architecture, this paper mainly focuses on the self-healing mechanism within a component on the basis of connectors between tasks.

This paper begins with describing the self-healing component architecture for self-healing systems in section 2. Section 3 describes the connector-based self-healing mechanism encapsulated in components. Section 4 specifies connectors supporting the self-healing mechanism - the message queue self-healing connector, message buffer self-healing connector, and message buffer and response self-healing connector. Section 5 describes related work. Finally, section 6 concludes this paper.

2. SELF-HEALING COMPONENT ARCHITECTURE

A self-healing component [6, 8] is a component that is able to autonomously detect, isolate, and repair abnormalities on itself as well as perform functional services requested from other components. Each self-healing component [12] can be designed as a layered architecture, structured with two layers (Fig. 1) - the service layer and the healing layer - on a component basis. The service layer is composed of tasks (i.e., active or concurrent objects), passive objects accessed by tasks, and connectors between tasks. In the normal phase, the service layer of a component provides full functionality to service requests from other components. Connectors between tasks in the service layer notify the status of messages passed between tasks 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 these messages from the service layer, the healing layer 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, it launches the self-healing mechanism for the sick (anomalous) object. In the healing phase, the service layer may not provide services any more or provide degraded services. The healing layer reconfigures objects in the service layer of the component at run-time to isolate the sick object and, if needed, notifies the object sickness to neighboring components requiring services from the sick component to reduce impact from the sick component. It then starts repairing the sick object.

The healing layer of each component (Fig. 1) is structured with 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 sick objects in the service layer [12].

- Component Monitor. The *Component Monitor* contains statecharts for each task thread in the service layer, which model behavior of task threads. A thread of each task in a component executes a statechart in which an incoming message to or outgoing message from a task is expected to transition its pre-defined statechart from one state to another within a bounded time. 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 task threads.
- Component Reconfiguration Plan Generator. The *Component Reconfiguration Plan Generator* maintains information about the configuration of objects in the service layer, generating reconfiguration plans in response to changes to the status of objects in the service layer of a component. The *Component Reconfiguration Plan Generator* of a component also involves a list of its neighboring components, whose objects may also need to be reconfigured to minimize the impact from paralyzed objects in a sick 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 reconfiguration plans if there are changes in the configuration of objects in other components.

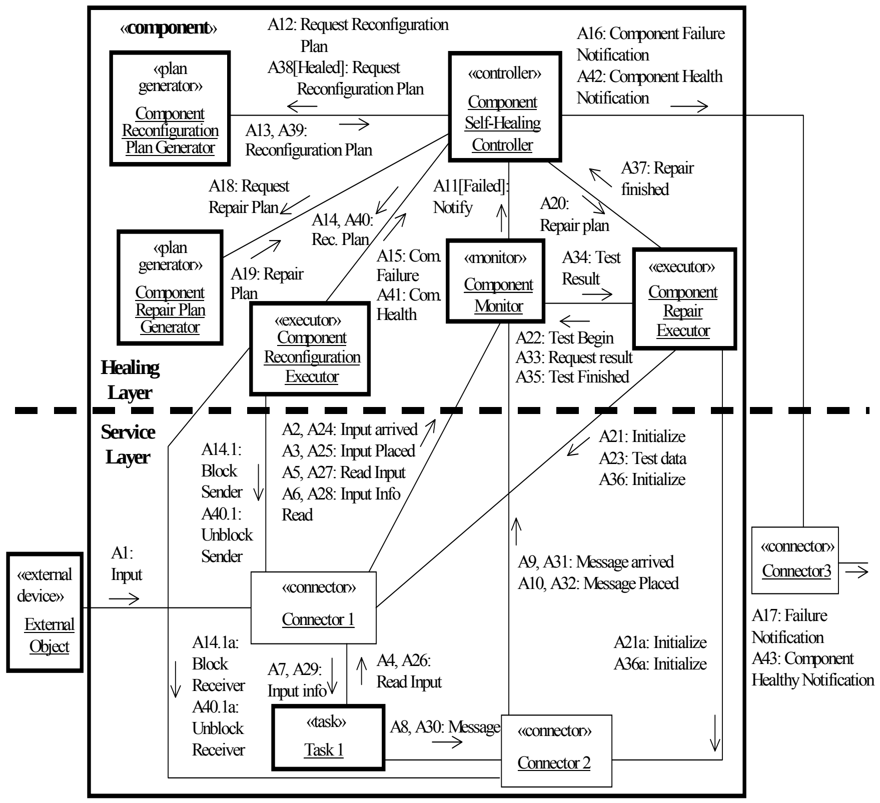


Fig.1. Self-healing component architecture and Message Sequence between Healing Layer and Service Layer

- Component Repair Plan Generator. The *Component Repair Plan Generator* maintains knowledge of repairs 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.
- Component Reconfiguration Executor. The *Component Reconfiguration Executor* substantially carries out the reconfiguration plan generated by the *Component Reconfiguration Plan Generator* to reconfigure the service layer of a component in response to sick objects in the component or other components
- Component Repair Executor. The *Component Repair Executor* performs the repair plan generated by the *Component Repair Plan Generator* to treat anomalous objects and test them after repairing in order to check whether the objects just repaired work normally.
- Component Self-Healing Controller. 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 the self-healing mechanism for sick objects in a component, cooperating with the *Component Self-Healing Controllers* of other components for reconfiguration against object anomalies

3. CONNECTOR-BASED SELF-HEALING MECHANISM

The role of connectors between tasks in a component is extended to support the self-healing mechanism for self-healing components, which includes detection of anomalous objects, reconfiguration of the objects in the service layer of a component, and testing of the repaired objects. In general, connectors between tasks in a component are designed to synchronize message communication between tasks. For detecting anomalies in components, connectors between tasks in the service layer of self-healing components notify message arrivals from tasks to the *Component Monitor* in the healing layer. Using the messages from connectors, the *Component Monitor* detects any behavioral abnormality of tasks in the component. In the meanwhile, connectors also acknowledge their status to the *Component Monitor* after storing messages in queues or buffers in the connectors, or delivering messages to other tasks on behalf of their associated tasks. A passive object (e.g., an entity object storing data) 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 connector and a passive object can determine the sickness of the connector and passive object.

A message sequence between objects in the service layer and healing layer of a component is depicted in Fig. 1 using the collaboration model of the UML [1, 10], which is simplified for this paper. 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.1), which makes the *Component Monitor* wait for the next message "Input Placed" (A3 in Fig. 1) from the Connector1. After placing a message in a queue or buffer, the Connector1 again notifies a message "Input Placed" (A3 in Fig. 1) to the *Component Monitor*. The *Component Monitor* presumes that the Connector1, Task1, or Connector2 (in Fig. 1) may be sick if the expected messages have not arrived within reasonable time intervals. In that case, the *Component Monitor* reports sickness of an object (A11 in Fig. 1) to the *Component Self-Healing Controller*, which in turn takes an appropriate action against the sick object.

The connectors in self-healing components are involved in the reconfiguration of sick objects in the components at run-time. A sick object is isolated from healthy objects in a component so that the healthy objects have minimal impact from the sick object. For the reconfiguration, the *Component Self-Healing Controller* consults with the *Component Reconfiguration Plan Generator* in the healing layer, which generates a reconfiguration plan against each sick object. Reconfiguration is carried out to block sick objects for repair. To achieve this, 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 any message from the queues or buffers in the connectors. When receiving the message, the connectors respectively freeze message communication between a sick task and healthy tasks.

Reconfiguration against a sick Task1 (Fig. 1) is depicted via the message sequence A12 through A17. The reconfiguration plan (A13 in Fig. 1) generated by the *Component Reconfiguration Plan Generator* is performed by the *Component Reconfiguration Executor* (A14 in Fig. 1). To isolate the Task1, the *Component Reconfiguration Executor* sends the Block Sender message (A14.1) to the Connector1, which blocks adding a new message from the External Object to the queue or buffer in the Connector1. In the meantime, the *Component Reconfiguration Executor* sends the Connector2 the Block Receiver message (A14.1a) in order for other tasks not to read messages from the queue or buffer in the Connector2. The reconfiguration plan may contain additionally a list of components that should be notified through messages A15 through A17.

The *Component Repair Plan Generator* generates a plan for repairing the sick object, which may include re-initialization or re-installation. The repair plan is delivered to the *Component Repair Executor,* which performs the plan as specified. In case a sick task is detected, the objects involved in the repair 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. Repairing the sick object is followed by testing of the repaired object with a set of test data.

Testing of repaired objects is also performed through connectors in the self-healing component. Testing of a repaired task begins by initializing connectors relevant to the task. The test data for each object in a component are predefined when the component is designed to be self-healed. The test data are delivered to the repaired task through the connectors associated with the task. During the testing, connectors also notify arrivals of (test data) messages from the *Component Repair Executor* and the repaired task to the *Component Monitor* in the healing layer. The testing procedure for a repaired task is similar to that of normal service, except for 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. 1 depicts repairing and testing of a sick task, Task1, by means of messages A18 through A37. Following the repair plan generated by *Component Repair Plan Generator*, the *Component Repair Executor* initializes queues or buffers in the Connector1 and Connector2 (A21 and A21a in Fig. 1) ready to receive test data. And then test data (A23 in Fig. 1) are delivered to the Connector1, which is used for testing the repaired Task1. After testing the Task1, the Connector1 and Connector2 are re-initialized (A36 and A36a in Fig. 1) to communicate between the Task1 and other tasks.

The objects and other components reconfigured for repairing a sick object are back to the original configuration via connectors as the repaired object resumes its service. The *Component Self-Healing Controller* requests a plan for reconfiguring components from the *Component Reconfiguration Plan Generator* (A38 and A39 in Fig. 1). Using the reconfiguration plan, the *Component Reconfiguration Executor* reconfigures the repaired object via connectors (A40.1 and A40.1a in Fig. 1) in the component and notifies the healthy state of the component (A41 through A43 in Fig. 1) to its neighbor components previously reconfigured through connectors between components. The neighboring components notified 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.

4. CONNECTORS SUPPORTING SEL-HEALING MECHANISM

A connector between tasks in a component encapsulates the synchronous mechanism for message communication. More specifically, a connector can be classified to a message queue connector, a message buffer connector, and a message buffer and response connector [4]. Containing a message queue, a message queue connector encapsulates the communication mechanism for loosely coupled message communication (asynchronous message communication). The size of queue is defined depending on 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.

4.1 MESSAGE QUEUE SELF-HEALING CONNECTOR

A message queue connector (MessageQueue in Fig. 2) encapsulates the synchronization mechanism for loosely coupled message communication, which provides two synchronized operations - send a message and receive a message - to other objects [4]. The operation, *send a message*, is called by a sender task, and the operation, *receive a message*, is called by a receiver task. The sender task is suspended if the queue is full (i.e., messageCount = maxCount), being reactivated when the queue has a slot to accept a message. The sender task continues to execute after adding a message to the queue. The receiver task is suspended if a message is not available in the message queue (i.e., messageCount = 0), and a new message activates the suspended receiver task. The receiver task is not suspended if a message is available in the queue.

The message queue connector (MessageQueue in Fig. 2) can be extended to the message queue self-healing connector (MessageQueueSH in Fig. 2) to support the self-healing mechanism described in section 3. The message queue self-healing connector adds new attributes, *sendStatus* and *receiveStatus*, to the message queue connector class, which indicate whether the connector provides message delivery service between sender and receiver tasks. When the connector normally delivers messages from a sender task to a receiver task, the values of attributes, *sendStatus* and *receiveStatus*, are set to the *Unblocked*, meaning that a sender task can add a message to the queue and a receiver task can read the message from it. If the connector is involved in healing an anomalous object, one of the values of the attributes in a connector is set to the *Blocked* (by messages A14.1 and A14.1a in Fig. 1), indicating that the connector does not provide message delivery service except test data. In the case where the connector is an incoming connector to an anomalous task, a sender task cannot add messages to the queue (sendStatus = Blocked). If the connector is an outgoing connector from an anomalous task, a receiver task cannot read messages from the queue in the connector (receiveStatus = Blocked).

As the message queue connector (MessageQueue in Fig. 2) is specialized to the message queue self-healing connector (MessageQueueSH in Fig. 2), the operations, *send* and *receive*, encapsulating the mechanism for synchronizing message communication between objects are modified to support the self-healing mechanism of components. The *send* and *receive* operations in the message queue connector are modified in the message queue shelf-healing connector to notify the Component Monitor (Fig. 1) of the fact that a message is arrived at and successfully handled by the connector. The *send* operation notifies the Component Monitor of the arrival of a message when it is invoked by a sender task (e.g., A1 and A2 in Fig. 1). The *send* operation is suspended if the queue is full (messageCount = 0) or the *send* operation is blocked (sendStatus = Blocked). The *send* operation is blocked when the connector receives messages (e.g., A14.1 in Fig. 1) from the Component Reconfiguration Executor so that the sick object is isolated from healthy objects in the component. Otherwise, the *send* function adds the message to the queue, and notifies the Component Monitor of completeness of storing the message in the queue (e.g., A3 in Fig. 1). This notification message shows the Component Monitor that the connector works normally up to that point. As with the *send* operation, the *receive* operation notifies the Component Monitor of the invocation of the operation by a receiver task that reads a message from the queue (e.g., A4 in Fig. 1). The Component Monitor monitors anomaly of the receiver task with the notification message from the *receive* operation (e.g., A5 in Fig. 1). The *receive* operation is suspended if the queue is full (messageCount = 0) or the *receive* operation is blocked (receiveStatus = Blocked). When an anomalous object is isolated by reconfiguration at run-time, the *receive* operation is blocked so that the receiver task cannot read the (test) data. Along with blocking the *send* operation, blocking the *receive* operation isolates objects associated with an anomalous object (i.e., sick object) from normal objects (i.e., healthy objects), which aims at localizing the anomalous object. If there is a message in the queue (messageCount > 0) and the *receive* operation is not blocked (receiveStatus = Unblocked), the receiver task can read a message from the queue and inform the Component Monitor of successful finishing of the *receive* operation by sending a notification message at the end of the operation (e.g., A6 in Fig. 1). This notification message ensures that the *receive* operation is normally working up to that point.

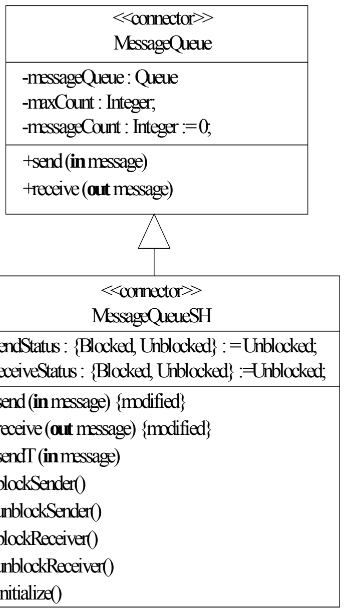

<<connector>>
MessageQueue
-messageQueue : Queue
-maxCount : Integer;
-messageCount : Integer := 0;
+send (**in** message)
+receive (**out** message)
<<connector>>
MessageQueueSH
-sendStatus : {Blocked, Unblocked} : = Unblocked;
-receiveStatus : {Blocked, Unblocked} :=Unblocked;
+send (**in** message) {modified}
+receive (**out** message) {modified}
+sendT (**in** message)
+blockSender()
+unblockSender()
+blockReceiver()
+unblockReceiver()
+initialize()

Fig. 2. Message Queue Self-Healing Connector

The message queue self-healing connector (MessageQueueSH in Fig. 2) has additional operations, *sendT(in message), blockSender(), unblockSender(), blockReceiver(), unblockReceiver() and initialize(),* which are invoked only by the Component Repair Executor and the Component Reconfiguration Executor in the healing layer of a component. The operation, *sendT(in message)*, is used to send test data to the connector (e.g., A23 in Fig. 1). This operation has the same functionality as the *send(in message)* operation except that it is invoked by the Component Repair Executor instead of a sender task. The *sendT(in message)* operation adds test data to the queue of the connector if the message queue is not full (*messageCount* < *maxCount*) and the *send* function is blocked (*sendStatus* = *Blocked*). The operation, *blockSender*(), keeps sender tasks from adding messages to the queue by setting the *sendStatus* variable to the *Blocked* (e.g., A14.1 in Fig. 1), while the operation, unblockSender(), allows sender tasks to add messages to the queue (e.g., A40.1 in Fig. 1). Similarly, the operation, *blockReceiver()*, keeps receiver tasks from reading messages from the queue by setting the *receiverStatus* variable to the *Blocked* (e.g., A14.1a in Fig. 1), whereas the operation, *unblockReceiver()*, releases the receive operation blocked (e.g., A40.1a in Fig. 1). The operation, *initialize()*, initializes the message queue and message count variable to test the repaired object (e.g., A21, A21a, A36, and A36a in Fig. 1).

4.2 MESSAGE BUFFER SELF-HEALING CONNECTOR

A message buffer connector (MessageBuffer in (a) of Fig. 3) encapsulates a message buffer and provides two synchronized operations – send a message and receive a message – for tightly coupled message communication without reply [4]. The *send* and *receive* operations are called by the sender task and the receiver task, respectively. Once the sender task has stored a message into the buffer, it is suspended until the receiver task receives the message.

The message buffer self-healing connector (MessageBufferSH in (a) of Fig. 3) is specialized from the message buffer connector (MessageBuffer in (a) of Fig. 3) so that it supports the self-healing mechanism in components. Like the message queue self-healing connector (MessageQueueSH in Fig. 2), the message buffer self-healing connector contains new attributes, *sendStatus* and *receiveStatus*, to indicate whether it is involved in normal message delivery or test data delivery. The *send* and *receive* operations in the message buffer connector are modified for supporting the notification mechanism in the message buffer shelf-healing connector. The *send* operation informs the Component Monitor both when it is invoked by a sender task and when a message has been stored in the message buffer, whereas the *receive* operation notifies the Component Monitor both when it is invoked by a receiver task and when a message has been read by a receiver task from the message buffer. Similar to the message queue self-healing connector, the message buffer self-healing connector provides additional operations, *sendT(in message), blockSender(), unblockSender(), blockReceiver(), unblockReceiver() and initialize(),* which are invoked by the Component Repair Executor and the Component Reconfiguration Executor in the healing layer.

4.3 MESSAGE BUFFER AND RESPONSE SELF-HEALING CONNECTOR

A message buffer and response connector (MessageBuffer&Response in (b) of Fig. 3) for tightly coupled message communication with reply [4] encapsulates a single message buffer and a single response buffer, and provides synchronized operations to send a message, receive a message, and send a reply. The sender task invokes the send message operation while the receiver task invokes the receive message and send reply operations. Once the sender task has stored a message in the buffer, it is suspended until the response is received from the receiver task.

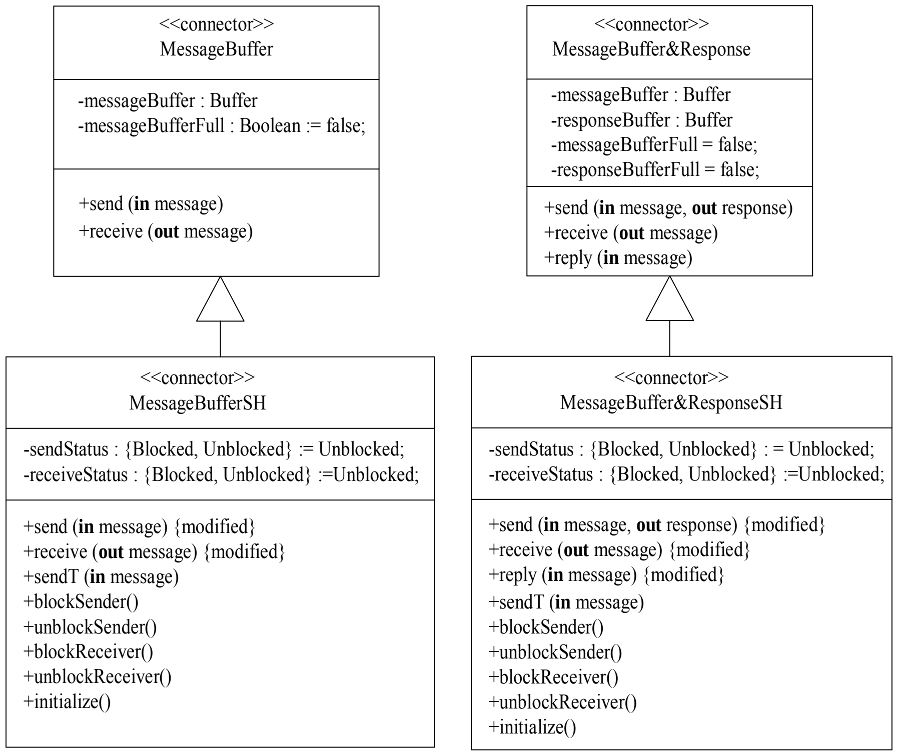
Similar to the message queue self-healing connector (MessageQueueSH in Fig. 2) and the message buffer self-healing connector (MessageBufferSH in (a) of Fig. 3), the message buffer and response self-healing connector (MessageBuffer&ResponseSH in (b) of Fig. 3) is obtained by specializing the message buffer and response connector (MessageBuffer&Response in (b) of Fig. 3). The message buffer and response self-healing connector involves new attributes, *sendStatus* and *receiveStatus*, like other self-healing connectors described above. Similarly, the *send*, *receive*, and *reply* operations in the message buffer and response connector are modified to support the self-healing mechanism in the message buffer and response shelf-healing connector. In addition, the message buffer and response self-healing connector provides operations, *sendT(in message), blockSender(), unblockSender(), blockReceiver(), unblockReceiver() and initialize()*, for reconfiguration, repair, and test, which are performed by the healing layer.

5. RELATED WORK

[8] 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 [6], 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 the approach in this paper is concerned with self-healing of a component in terms of detection, reconfiguration, repair, and test of a component. [2] 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 [2], the approach in this paper concentrates on a self-healing component architecture encapsulating a healing mechanism on each component basis. [3] 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, [11] describes an approach to devising reconfiguration mechanism for Enterprise JavaBeans (EJB). 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.

6. CONCLUSIONS

This paper has described the connector-based self-healing mechanism for self-healing components of systems. The role of connectors encapsulating the synchronization mechanism for message communication between tasks in a component is extended to the self-healing mechanism for reliable components. In the self-healing mechanism, connectors in self-healing components notify message arrivals from tasks to the healing layer to detect anomalies in the component. Once detecting an anomalous object, the component autonomously reconfigures objects and repairs the sick object through connectors. This paper also has defined the specifications for the message queue self-healing connector, message buffer self-healing connector, and message buffer and response self-healing connector, which support the self-healing mechanism.


<<connector>>
MessageBuffer
-messageBuffer : Buffer
-messageBufferFull : Boolean := false;
+send (**in** message)
+receive (**out** message)
<<connector>>
MessageBufferSH
-sendStatus : {Blocked, Unblocked} := Unblocked;
-receiveStatus : {Blocked, Unblocked} :=Unblocked;
+send (**in** message) {modified}
+receive (**out** message) {modified}
+sendT (**in** message)
+blockSender()
+unblockSender()
+blockReceiver()
+unblockReceiver()
+initialize()
<<connector>>
MessageBuffer&Response
-messageBuffer : Buffer
-responseBuffer : Buffer
-messageBufferFull = false;
-responseBufferFull = false;
+send (**in** message, **out** response)
+receive (**out** message)
+reply (**in** message)
<<connector>>
MessageBuffer&ResponseSH
-sendStatus : {Blocked, Unblocked} : = Unblocked;
-receiveStatus : {Blocked, Unblocked} :=Unblocked;
+send (**in** message, **out** response) {modified}
+receive (**out** message) {modified}
+reply (**in** message) {modified}
+sendT (**in** message)
+blockSender()
+unblockSender()
+blockReceiver()
+unblockReceiver()
+initialize()

a) Message Buffer Self-Healing Connector **b) Message Buffer & Response Self-Healing Connector**

Fig. 3. Message Buffer Self-Healing Connector and Message Buffer & Response Self-Healing Connector

The approach to the connector-based self-healing mechanism suggested in this paper has several strong points. The first advantage is that the self-healing concern and the functional service concern modeled in their separate layers of a component are composed into a self-healing component by means of connectors. To achieve this, connectors are involved in the self-healing mechanism - detection, reconfiguration and repair of anomalous objects - as well as the synchronization mechanism. Another benefit is that the connectors between tasks in a self-healing component help isolate objects under repair. Connectors separate anomalous tasks from healthy ones by controlling input to and output from the anomalous tasks. However, the connector-based self-healing mechanism can cause relatively low performance to a self-healing component due to communications between connectors in the service layer and the monitor in the healing layer.

This approach retains several research issues associated with connectors in self-healing components. The connector-based self-healing mechanism described in this paper should be extended to include inter-components interaction at the level of software architecture of systems. The connectors between self-healing components [5] may provide different functionality compared to those between tasks within a self-healing component. The status of objects (i.e., sickness or health of objects) in a component can be delivered to its neighboring components through the connectors. In addition, the detection mechanism of the Component Monitor in the healing layer should be refined to reduce the rate of error in determining whether an object in a component is anomalous. To achieve this, the Component Monitor may consider the status of thread of neighbouring tasks around a specific object before it makes a decision on sickness of the object.

6. REFERENCES

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

[2] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor, “Towards Architecture-based Self-Healing Systems,” Workshop on Self-healing systems, Proceedings of the first workshop on Self-healing systems, Charleston, SC, November18-19, 2002.

[3] David Garlan and Bradley Schmerl, “Model-based Adaptation for Self-Healing Systems,” Workshop on Self-healing systems, Proceedings of the first workshop on Self-healing systems, Charleston, SC, November18-19, 2002.

[4] Hassan Gomaa, “Designing Concurrent, Distributed, and Real-Time Applications with UML,” Addison-Wesley, 2000.

[5] Hassan Gomaa, Daniel A. Menasce, and Michael E. Shin, “Reusable Component Patterns for Distributed Software Architectures,” 2001 Symposium on Software Reusability (SSR’01 Sponsored by ACM/SIGSOFT), Toronto, Ontario, Canada, May 18-20, 2001.

[6] Paulo Asterio de C. Guerra and Rogerio de Lemos, “An Idealized Fault-Tolerant Architectural Component,” Workshop on Architecting Dependable Systems, ICSE’02 International Conference on Software Engineering, Orlando, FL, May 25, 2002.

[7] IBM, “An architectural blueprint for autonomic computing,” IBM and autonomic computing, April 2003.

[8] Philip Koopman, “Elements of the Self-Healing System Problem Space,” Workshop on Software Architectures for Dependable Systems (WADS2003), ICSE’03 International Conference on Software Engineering, Portland, Oregon, May 3-11, 2003.

[9] Oriezy, P., Gorlick, M.M., Taylor, R.N., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., and Wolf, A., “An Architecture-Based Approach to Self-Adaptive Software,” In *IEEE Intelligent Systems* 14(3):54-62, May/June, 1999.

[10] J. Rumbaugh, G. Booch, I. Jacobson, “The Unified Modeling Language Reference Manual,” Addison Wesley, Reading MA, 1999.

[11] M. J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and 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.

[12] Michael E. Shin, “Self-Healing Component in Robust Software Architecture for Concurrent and Distributed Systems,” 2005 (Accepted for publication in the Science of Computer Programming).