Configuration-Level Adaptation

Md. Jamshed Haider Siddiqi

***Abstract*—The mechanism of self-healing enables the system to continue operating properly on the event of the failure of some of its components, to determine the errors and to recover from them. In traditional mechanisms, analyzing, changing and reusing have never been an easy task. An alternative and perhaps more efficient way is allowing the system to adapt dynamically at configuration level. Configuration-level adaptation mechanism assumes that the architecture of the system is modeled as a collection of components. Each component has two-layer architecture, healing layer and service layer. Anomalous tasks are detected by means of message communication between tasks in a component. Message communication is done via connectors and plays the important role in detecting anomalous tasks, and, thus consequently, in reconfiguring components and repairing anomalous tasks.**

I. INTRODUCTION

DAY by day, computer systems are becoming more sophisticated, and high degree of reliability is demanded. With the change of system resources and environment, the system may tend to show malfunctionality.

In adaptation mechanism, it is an important issue that in case of failure of any component of the system, the system will be able to detect the errors and continue to operate with degraded functionality. Eventually the system must be able to recover from failure.

The adaptation mechanisms can be classified as internal and external. In internal, adaptation mechanisms are tightly integrated with the application itself and wired in at the code level supported by some programming languages (e.g., Java Exceptions) or usual control checking by the programmers. The major difficulty of traditional mechanisms is localizing the errors.

Configuration-level adaptation, the external mechanism, outweighs traditional mechanisms because of its global perception on the system. This perception helps to adapt the system dynamically. Moreover, the mechanism can support reuse in more efficient way, since adaptation is not wired into application. Configuration-level adaptation is based on the architecture of the self-healing system [4][8], and the architecture changes during software execution. The structure of the run-time adaptation can be shortly viewed as illustrated in Figure 1. The system behavior is monitored by components outside the running system. The component always remains vigilant to detect if any anomaly occurs. After detection, errors are analyzed and repaired.

Manuscript received March 12, 2007

The author is with the Department of Computer Science, University of Helsinki, Finland. (e-mail: jamshed.siddiqi@cs. helsinki.fi)

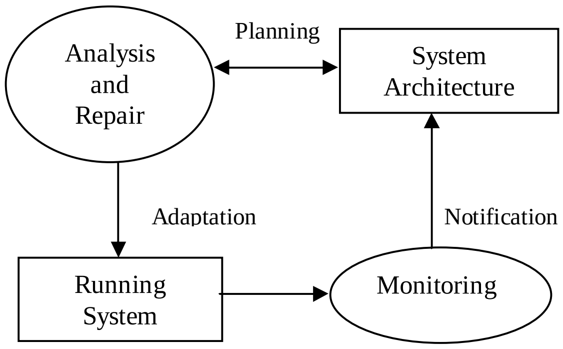

Figure 1: Configuration-Level Adaptation

Despite the advantages of configuration-level adaptation, the researchers have been facing some challenges regarding this run time adaptation. Challenges include [3]:

Monitoring: How monitoring capabilities can be achieved so that the regular functionalities of the components are not disturbed? What type of parameters should be considered for monitoring? How system components should be designed so that they can be easily monitored?

Interpretation: How it can be determined that system components are erroneous? What is the source of the system errors?

Resolution: There might be set of possible repairing techniques. How the best technique can be chosen?

Adaptation: How adaptation should be done so that the system continues to run properly? How the system components should be designed so that they can be adapted dynamically?

This paper is organized as follows. Section II describes the architecture of self-healing component. Then we describe the architecture of self-healing connector in Section III. How anomalies are detected in self-healing systems is described in Section IV. We discuss the self-healing mechanism by means of message communication in Section V. The last section, Section VI, draws some conclusions of configuration-level adaptation.

II. ARCHITECTURE FOR SELF-ADAPTATION

The self-healing system architecture consists of components. Each component has two layers, healing layer and service layer. The components in haling layer are designed to detect and heal anomalies in service layer that might emerge at run time [7]. The connectors in service layer help the healing layer in detecting anomalies by notifying the status of service layer components.

When the healing layer detects any abnormalities in a task, it initiates self-healing mechanism for anomalous task. During healing phase the service layer limits its services to handling test data only. At the very first step, the healing layer reconfigures the service layer to isolate the anomalous task, and inform the neighboring components about the abnormalities of the task. After that, the healing layer starts to repair the anomalous task. The healing layer of each component (Figure 2) consists of the following components [2]:

Component Monitor: To monitor the behavior of the service components the Component Monitor contains the state transition diagrams for each task. State transition diagrams represent the functional conditions of the components. Component Monitor has a thread for each task in the component. The thread is responsible for executing the state transition diagrams. The Component Monitor supervises the behavior of tasks, connectors and passive objects (the object that does not have its own control, i.e. invoked by only tasks).

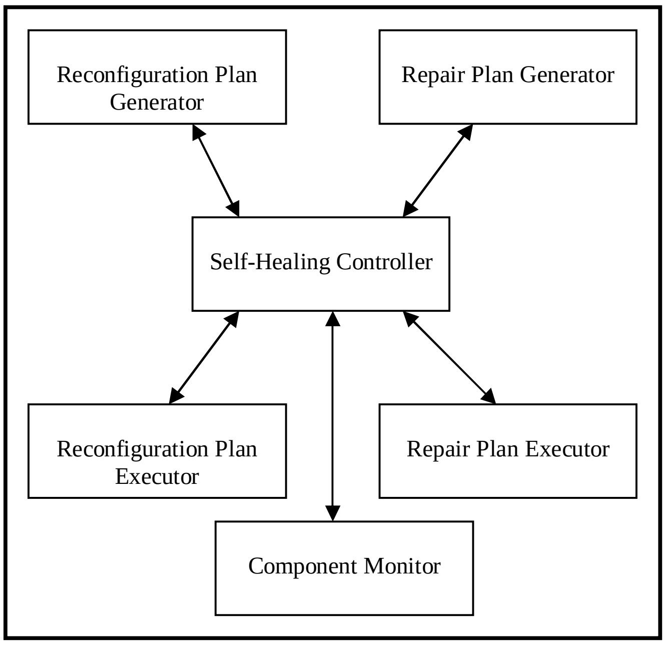

Figure 2: The Healing Layer of a Self-Healing Component

Reconfiguration Plan Generator: Once Component Monitor detects any anomaly in the component it notifies the Self-Healing Controller. The Self-Healing Controller requests Reconfiguration Plan Generator to generate plan for reconfiguring the component. Reconfiguration Plan Generator contains the information about the configuration of the tasks in the components. There may be other neighboring components in the system whose objects may be affected by the behavior of the anomalous tasks. To mitigate the impact of paralyzed tasks in other components the Reconfiguration Plan Generator maintains the information about the organization of neighboring components.

Reconfiguration Plan Executor: The Reconfiguration Plan Executor executes the plan generated by the Reconfiguration Plan Generator. The Reconfiguration Plan Executor is informed by the Self-Healing Controller and reconfigures the service layer by blocking sending and receiving connectors associated with the anomalous task. The sending and receiving connectors will remain blocked until the anomalous task has been repaired.

Repair Plan Generator: The Repair Plan Generator has knowledge of how to repair each individual task and passive objects. This component performs the operations similar to Reconfiguration Plan Generator with the exception that it plans for repairing.

Repair Plan Executor: The Repair Plan Executor executes the plan generated by Repair Plan Generator. It also works in similar fashion like Reconfiguration Plan Executor.

Self-Healing Controller: The Self-Healing Controller acts as a coordinator to conduct the self-healing mechanism. The Self-Healing Controller listens to the Component Monitor. According to the received information from the Component Monitor, the Self-Healing Controller requests plan generators for generating plans. After receiving responses it informs the executors to execute the desired plans.

III. ARCHITECTURE OF SELF-HEALING CONNECTOR

As described in the previous section the healing layer of a self-healing component detects anomalies in the service layer. To identify mulfunctional tasks in the service layer, the Component Monitor communicates with the connectors associated with the tasks. That means that detection of anomalies is performed by means of communication with the connectors and repairing is executed through the connectors. The self-healing connector can also be designed of two layers, communication layer and healing layer [1]. Figure 3 illustrates the architecture of self-healing connector.

The communication layer of the self-healing connector sends messages to and receives them from healing layer of the component. The healing layer consists of healing manager. The healing manager detects, reconfigures and repairs anomalies as directed by the self-healing mechanism. The communication layer consists of (Figure 3):

Call Routine: The *Call Routine* packs a message and sends the packed message to a intended receiver.

Return Routine: The *Return Routine* unpacks the message and sends it to the destination component

Incoming Message Queue: The *Incoming Message Queue* stores messages received from other components

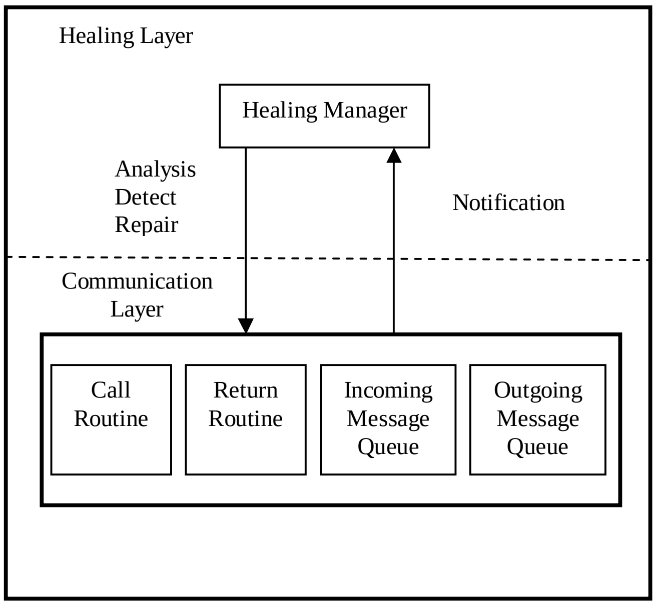

Figure 3: The Architecture of Self-Healing Connector

Outgoing Message Queue: The *Outgoing Message Queue* stores messages to be sent to the incoming queue of other components.

IV. ANOMALY DETECTION

The components in the architecture of a system are designed to perform specific and intended tasks. If any component in the system does not carry the action as it is specified, then the system is considered as anomalous. In self-healing mechanisms, the first step is to localize the anomaly. An anomaly occurs in the components or connectors between the components. A Finite State Machine (FSM) represents the behavior of the system composed of finite number of states and transition between states [6]. The specification of components and connectors between components can be described using state transition diagram. Besides several techniques, a FSM can be represented as a directed graph, where the set of vertices represents set of specified states and a directed edge represents a transition from one state to another. The FSM can detect faults if the machine provides an output different from one specified by the output function or it enters into a different state rather than it is specified by the transfer function.

Anomalies in the system are detected at the level of components and at the level of connectors between components [5]. The Component Monitor detects anomalies by observing tasks and connectors between tasks. The Component Monitor detects the errors with the assistance of connectors between tasks. Connectors provide message communication mechanism, as well as, they inform the Component Monitor about the status of message passing to ensure the robustness of anomaly detection.

When a task invokes connectors between tasks and passive objects, the connectors notify the Component Monitor. The connectors also notify the Component Monitor when the tasks and passive objects complete their operations. The notification message is used by the Component Monitor to detect anomalies of tasks, connectors between tasks and passive objects accessed by the task.

During and after the notification the Component Monitor is involved with the state transition diagram for the task. The Component Monitor examines state diagram of the task and considers the task anomalous if the task does not execute properly. In this way, the anomaly in a component can be traced, but if there is any anomaly in the connector itself the anomaly may remain undetected. To overcome this, connectors notify the Component Monitor about the status of message communication. Meanwhile, the Component Monitor uses timeout. If the Component Monitor does not hear from the connector within a time intervals, it considers the respective connectors as erroneous.

V. SELF-HEALING MECHANISM AND MESSAGE COMMUNICATION

Besides message communication, connectors also perform some extended operations to support the self-healing mechanisms [2]. These extended operations include reconfiguring the anomalous task in the service layer of a component and testing the repaired task by test data. Connectors acknowledge their status to the Component Monitor after receiving and storing data or messages in buffers or queues. They also acknowledge the status of delivering messages to other task on behalf of the associated tasks. When a passive object is accessed by other tasks in the service layer of a component and operations are completed successfully, also needs to notify the Component Monitor. The trace of a task thread within a connector and a passive object can determine the abnormalities of the connector and passive object.

Figure 4 illustrates the architecture of a self-healing component and a scenario of sequence of message communication in usual and intended case and also in case of some abnormalities. Normal services of the Task1, Connector1 and Connector2 are described in Figure 4 by message sequence M1 through M8. When Connector1 receives the message M1 on behalf of the Task1 from any external object, it notifies the Component Monitor by sending the message M2 labeled *input arrived*. The message M2 bears the meaning that Connector1 receives input from an external object. After that, Connector1 allows the Component Monitor to wait for next message *input placed*. When connector1 places the input in the queue or buffer associated with Task1 it notifies the Component Monitor *input placed* by the message M3. When Connector1 places the input it expects to

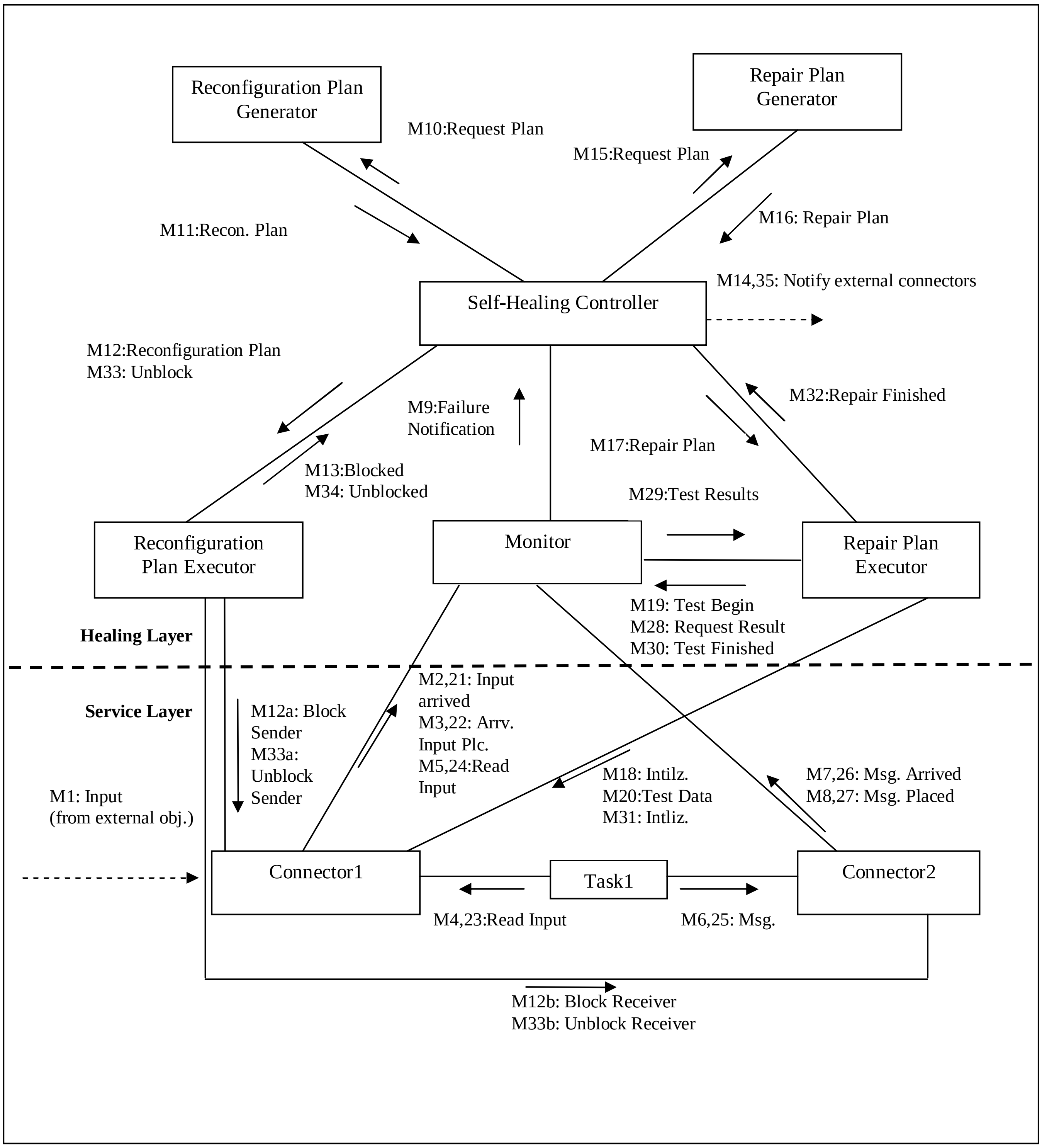

Figure 4: Self-Healing Component Architecture and Message Sequence [2]

receive an acknowledgement *read input* (message M4 in Figure 4) from Task1. After receiving the message it informs the Component Monitor via message M5. Receiving any messages by the connectors and notifying the Component Monitor go on in the same fashion. If the Component Monitor missed to be informed about the sequence of messages from the connectors it determines that the respective tasks or connectors are anomalous and immediately start to carry its action to repair the unhealthy task or connector by means of dynamic reconfiguration.

After detecting anomaly, the Component Monitor notifies the failure to the Self-Healing Controller (message M9). The Self-Healing Controller requests Reconfiguration Plan Generator to generate and send reconfiguration plan. The Reconfiguration Plan Generator honors the request and sends reconfiguration plan to the Self-Healing Controller. In order to eliminate the impact of anomalous task to other healthy objects in the component or other external objects, the task is blocked and restricted to send or receive any messages during reconfiguration and repairing phases. In Figure 4, Task1 is blocked. The Reconfiguration Executor sends *block sender* (M12a) message to the incoming connector (Connector 1) to refrain the connector from adding any new messages in the queue or buffer. In the meantime, it sends the *block receiver* message to outgoing connector (Connector2) to block receiving any message from the queue or buffer associated with the anomalous task. Reconfiguration against a sick task is performed through the message sequence M10 to M13. The Self-Healing Controller also informs neighboring components about the unhealthy state of the component via message M14.

After reconfiguring the anomalous task, the Self-Healing Controller takes initiatives to repair the task. Self-Healing Controller consults with the Repair Plan Generator and let the Repair Executor know (by message M17 in Figure 4) about the planning for repairing the anomalous task, Task1. The repairing of anomalous task generally includes re-initialization or re-installation. The Repair Executor repairs the task according to the plan received through message M17. After repairing, the task is tested. Testing begins by initializing the queue or buffer by means of message M18. The Executor informs the Component Monitor that the test begins (M19) and sends test data (M20) to the Connector1. The test data are defined when the self-healing mechanism is modeled. The remaining part of testing is performed by the usual operations of Conncetor1, Task1 and Connector2 through message sequence M21 to M27.

When the Component Monitor receives the test result by means of message M27 it sends the results to the Repair Executor via message M29. The Repair Executor informs the Self-Healing Controller that the repairing of the anomalous tasks is completed via message M32. In order to allow the repaired task (Task1) to resume its usual operation, the Self-Healing Controller requests the Reconfiguration Executor to unblock (M33) the sending and receiving connectors. When the component gets rid of its malfunctionality, the Self-Healing Controller informs neighboring components about the healthy state of the component through the message M35.

VI. CONCLUSIONS

External mechanism to detect and repair anomalies has some advantages than that of programming level mechanism. In this paper, we have described the required architecture for configuration-level adaptation and healing mechanism. The architecture is designed in such a way so that faults can be detected and repaired with the functionality of the components of the model. One of the challenges in configuration-level adaptation is to plan repairing policy and applying the appropriate plan to a certain failure. Self-healing systems must consider the fact that they change over time. The changes may come from operating mode changes, resource faults, adaptation to external environment etc. Dynamic changes of adaptation will increase the efficiency in self-healing mechanisms. There must be some intelligent mechanism in repair plan generator to allow the self-healing system to adapt dynamically. For example the history of previous failures can be stored in order to analyze those. These sorts of analysis may make the self-healing mechanism intelligent and more adaptive.

REFERENCES

[1] Michael E. Shin, and Jung Hoon An, *Self-Reconfiguration in Self-Healing Systems,* Proceedings of the Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems, March 2006, pp 89-98.

[2] Michael E. Shin, and Daniel Cooke, *Connector-Based Self-Healing Mechanism for Components of a Reliable System,* Proceedings of the 2005 workshop on Design and evolution of autonomic application software, International Conference on Software Engineering, 2005, pp 1-7.

[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, 2002, pp 27-32.

[4] 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, November2002, pp 27-32.

[5] Michael E. Shin, and Yan Xu, *Detection of Anomalies in a Software Architecture with connectors,* International Workshop on System/Software Architectures (WSSA05), Las Vegas, Nevada, USA, June 2005, Vol. 61, Issue 1, pp. 16-26

[6] Alexander Sakharov, *A hybrid state machine notation for component specification*, ACM SIGPLAN Notices, April 2000, Vol. 35, Issue 4, pp 51-56

[7] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf., *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems, June 1999, Vol. 14, Issue 3, pp. 54-62

[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, May 2003.