Multi-Tiered Bio-Inspired Self-Healing Architectural Paradigm for Software Systems

AZWEEN ABDULLAH, RIA CANDRAWATI
& M. AGNI CATUR BHAKTI

ABSTRAK

Keupayaan sembuh-diri telah mula muncul sebagai suatu sifat yang menarik dan berkeupayaan untuk menjadi sangat berharga bagi suatu sistem perisian. Ciri-ciri sembuh-diri membolehkan sistem perisian untuk sentiasa dan secara dinamiknya memantau, mendiagnos, dan menyesuaikan dirinya selepas berlaku suatu kegagalan pada komponen-komponennya. Bagaimanapun, membina sistem perisian sembuh-diri yang sedemikian rupa adalah suatu cabaran yang signifikan. Tabiatnya memperkenalkan kepada kita konsep-konsep tak-terduga dalam bentuk sistem biologi yang mempunyai kebolehan untuk mengendalikan keadaan-keadaan tak-biasa bagi dirinya. Berdasarkan kepada pemerhatian ini, kerja penyelidikan ini mempersembahkan seni bina sembuh-diri untuk suatu sistem perisian berdasarkan kepada salah satu proses biologi yang berkebolehan untuk menyembuh secara sendiri (proses sembuh-luka). Seni bina sembuh-diri menyediakan sistem perisian dengan keupayaan mengendalikan keadaan-keadaan janggal yang diterbitkan oleh komponen-komponennya. Seni bina yang dipersembahkan telah dibahagikan kepada dua lapisan, fungsian dan penyembuhan. Untuk membuktikan kebolehfungsian sistem perisian sembuh-diri yang dipersembahkan, perihalan formal ke atas seni bina yang dicadangkan adalah dibentangkan.

Katakunci: Sembuh-diri, penyembuhan luka, spesifikasi formal, seni bina penyembuhan diri berlapis.

ABSTRACT

Self-healing capabilities have begun to emerge as an interesting and potentially valuable property of software systems. Self-healing characteristic enables software systems to continuously and dynamically monitor, diagnose, and adapt themselves after a failure has occurred in their components. However, developing such a software system is a significant challenge. Nature introduces to us exceptional concepts in terms of presenting biological systems that have the ability to handle its abnormal conditions. Based on this observation, this work presents a self-healing architecture for software system based on one of the biological processes that have the ability to heal by itself- the wound-healing process. The self-healing architecture provides software systems with the ability to handle anomalous conditions that appear among the components. The presented architecture is divided into two layers, functional layer and healing layer. To demonstrate the functionality of the proposed self-healing software system, a formal description of the architecture is presented.

Keywords: Self-healing, wound-healing, formal specification, layered self-healing architecture.

INTRODUCTION

Today, information technology organizations encounters growing challenges in the management and maintenance of large scale software systems because these systems must be active and available 24 hours a day, 7 days a week. Existing control methods and tools are able to mange and administer todays and future software systems, but software complexities are increasing by the day. While the complexity of these systems continue to grow, the need for skilled persons who install, optimize, protect, and maintain these systems becomes more important, but in scarcity.

There have been several attempts to reduce the complexity within these systems by introducing better software engineering practices. In spite of these attempts, the complexities of such systems remain the same as more and more new technologies and systems are being incorporated together. The complexity of the software systems and their environment lead to the idea of autonomic computing as given by Horn (2001) and Kephart & Chess (2003). This new area of computing aims to provide software systems that have the ability to handle their complexities by themselves. In other words, autonomic computing is a solution which proposes to reallocate many of the management responsibilities from the administrators to the system itself as claimed by Laddaga (1999). The vision of autonomic computing given by Horn (2001) is to improve the management of the complex information technology systems by introducing self-management systems for configuration, healing, optimization, and protection purposes. From this vision, Salehie & Tahvildarie (2005) claimed that the major characteristics of autonomic computing systems are self-configuration, self-healing, self-optimization, and self-protection. This work focuses on the second characteristic of autonomic systems, namely the self-healing characteristic.

A system is said to be self-healing if it can recover from failures without external intervention. In other words, the system is capable of automatically re-organizing itself to continue operating after part of it has failed. This is obviously closely related to the notion of fault-tolerance in which a system can operate normally despite experiencing failures. There are finer shades of distinguishing semantics when we relate security, fault-tolerance, survivability and self-healing. We use the term self-healing to mean a wider class of systems and degree of re-organization than is usually denoted by the term fault tolerance.

Specification logics are extensively utilized to verify necessary and inherent properties of self-healing systems. These logics can allow notion of good behavior and abnormal behavior to be formally specified and as a result permit precise reasoning about fault tolerance. We intend to look at their application in self-healing systems which can dynamically reconfigure in response to changes in their environment and allow the succinct specification of both self-healing systems and the properties that they must satisfy.

RELATED WORKS

A biologically-inspired autonomic architecture for self-healing data centers, called Symbiotic Sphere, was proposed by Champrasert & Suzuki (2006). The architecture follows certain biological principles such as decentralization, natural selection, emergence and symbiosis to design data centers (application services and middleware platforms).

On the other hand, George et al. (2002) proposed a cell-based programming model that can be used for software systems operation and healing. The model is more closely related to the biological processes. Their model supports a notion of cell division, a communication model based on chemical diffusion, and a rudimentary model of the physical forces involved.

Boonma & Suzuki (2007) proposed middleware architecture for sensor networks (a biologically-inspired middleware architecture for self-managing wireless sensor networks, BiSNET). BiSNET follows certain biological principles such as decentralization, food gathering/storage and natural selection to design MWSN application. They present some biological systems such as bees and their interaction with each other and food gathering and storage.

Adjacent to this research, biologically inspired self-governance and self-organization techniques for autonomic networks also have been proposed by Balasubramaniam et al. (2006). They propose key self-organization and self-governance techniques that are drawn from principles of molecular biology. The biological processes which are included in their works are blood glucose homeostasis, reaction diffusion, microorganism mobility using chemo taxis techniques, and hormone signaling.

Ahmed et al. (2007) proposed ETS Self-healing service as an integral part of MARKS (Middleware Adaptability for Resource Discovery, Knowledge Usability and Self-healing) middleware. The fault detection and fault recovery related issues are taken care of by this service. ETS service use states for detecting sources of failures, and the notification of failure source is universal for error detections. Park et al. (2006) proposed a self-healing system that monitors, diagnoses and heals its own internal problems using self-awareness as contextual information. The proposed system consists of multi agents that analyze the log context, error events and resource status in order to perform self-diagnosis and self-healing. Bokareva et al. (2005) proposed self-healing hybrid sensor network architecture, called SASHA. This architecture represents an immunology inspired solution to the design of a self-healing sensor network.

Although many biological autonomic computing approaches have been proposed, none has fully adopted and implemented a complete biological process. These works have adopted small parts of the biological process such as DNA, cell division, and chemical diffusion. The different between these works and our work is that, our work adopts a complete biological process modeling to introduce the self-healing software system architecture.

Mapping Wound-Healing Into Self-Healing Software System

In this section we show how the wound-healing process can be mapped into self-healing software system. We mapped each phase in the wound-healing process to the expected phases in the self-healing software system (see Figure 1).

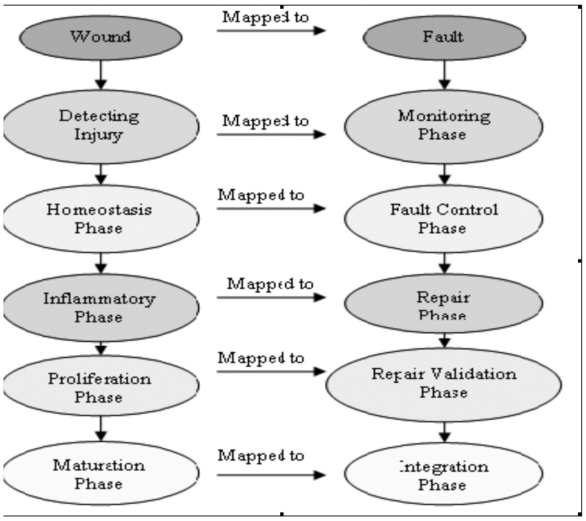


FIGURE 1. Mapping the Wound-Healing Process into Self-Healing Software System

Woundheal (2007) said that in the wound-healing, a wound is created when the anatomic integrity of the tissue is disrupted. In a software system, a fault is a structural imperfection that may lead to the system's failure. Therefore, we mapped the wound to the fault.

Diegelmann & Evans (2004) said that in the wound-healing process, when an injury happens, the body sends particular chemical signals to indicate that there is an injury at the specified area. In software systems, when a fault occurs in one of the system's components, a particular technique is needed to detect this failure.

Homeostasis Phase Into Fault Control Phase

In Keast & Orsted (2007), the body tries to stop the bleeding at the injury site. In software systems, the system needs to stop losing components that are related to the faulty component. In wound-healing, special cells are sent to the injury site either to stop bleeding or to produce substances that help in stopping the bleeding. To map this phase to self-healing software system, we need to find a technique that stops fault propagation among components.

Inflammatory Phase Into Repair Phase

In this phase, in order to start the healing process after detecting the injury and stopping the bleeding, the body starts to clean up the debris at the injury site. Also, special types of cells start to produce some substances which help on rebuilding (repairing) the injury site, as mentioned by Clark & King (2004) and Cockbill (2002). In software system, the faulty component must be isolated from the other components. This isolation process prevents the other non-faulty components from being affected by the error propagation from the faulty ones. Moreover, repairing (healing) the faulty components becomes easier when the faulty components are isolated from other non-faulty components.

Proliferation Phase Into Repair Validation phase

Two types of cells appear in this phase. The first type of cells migrates across the surface of the wound. The second type of cells proliferates in the deeper part of the wound. These two types of cells produce large amounts of substances which contribute in creating and connecting new tissues and blood vessels. By the end of this phase, new tissues and blood vessels are produced. Repairing the faulty component in a software system, results in a new healed component. Therefore, testing the new component (the healed component) is needed to make sure that the healed component is working in a predefined way.

Remodeling Phase Into Integration Phase

The role of the last phase in wound-healing process is to remodel the tissues and strengthen the scar. Also, collagen fibers are remodeled to more organized matrix. In software systems, we need to reconfigure the healed components into the running system without affecting the other components seamlessly.

Table 1 summarizes the task of each phase in the wound-healing process as well as in the self-healing software systems.

TABLE 1. The Description of the Phases in Wound-Healing and Self-Healing Software System

Wound-Healing		**Self-Healing Software**	**System**
Phase	**Description**	**Phase**	**Description**
Hemostasis	Stop bleeding	Fault Control	Stop losing other components via error propagation.
Inflammatory	Removing the debris at the injury site	Repair	Isolating and repairing the faulty component.
Proliferation	Build and fill the injury site	Repair Validation	Test the healed component.
Maturation	Remodeling the tissues and Strengthen the scar	Integration	Returning the healed component to the system.

We believe that infection of viruses should be controlled in the wound-healing process, as mentioned by Woundheal (2007), as well as in self-healing software systems during the healing time. Infection of viruses is considered as a security issue. As mentioned earlier, there is another area of autonomic computing that focuses on security issues called self-protecting, which in this paper is omitted.

BIOLOGICAL SELF-HEALING SOFTWARE SYSTEM ARCHITECTURE

Shaw & Garlan (1996) claimed that software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system. Perry & Wolf (1992) said that the abstractions involve descriptions of the elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns.

In the previous sections, we discussed how the mapping of the phases of the wound-healing process into phases of the self-healing software system is done. In this section, we introduce our self-healing software system architecture.

In the wound-healing process, particular types of cells are responsible for particular tasks. For example, in the first phase of wound-healing process, the homeostasis phase, blood vessels constrict to stop bleeding and platelets adhere and produce special substances which help to stop bleeding. Likewise, in our self-healing software system architecture, we introduce some modules in each phase. These modules play the same role of the cells in the wound-healing process. In other words, each module is responsible for a specific task.

THE PROPOSED ARCHITECTURE

Our self-healing software system architecture consists of two layers: the functional layer and the healing layer.

1. The Functional Layer

In this layer, the system executes normally without any fault. In other words, the system provides its full services in this layer. Each component in the system provides its full service and interacts with other components without any disruption. For example, in Figure 2, the system consists of four components C1, C2, C3, and C4.

2. **The Healing Layer**

If one of the components fails to provide its services during the execution of the system (receive input, process, deliver output), the component is considered as a faulty component. In this layer, we aim to return the faulty component to its normal condition (the functional layer) by applying the wound-healing phases. The healing layer composes of five phases: Monitoring phase, Fault Control Phase, Repair Phase, Repair Validation Phase and Integration Phase. Each phase consists of a set of modules. These modules interact with each other to achieve the task of their phase. The modules are numbered from 1 to 10 in Figure 2.

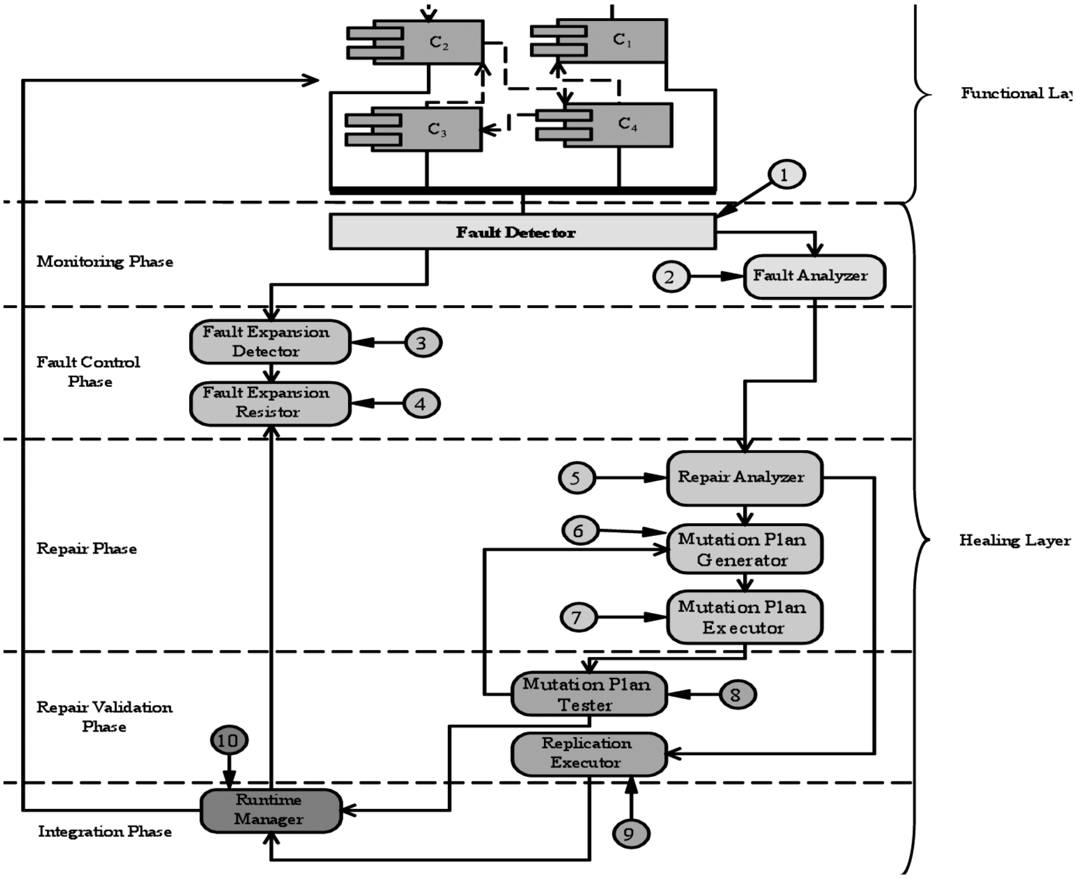


FIGURE 2. Biologically Inspired Self-Healing System Architecture

- **Monitoring Phase (Failure Detection Phase)**

  This phase consists of two elements; Fault Detector and Fault Analyzer. These two elements act like the chemical signals in wound-healing process which would be sent once the injury is discovered.

  - Module No.1: Fault Detector

    The task of this module is to observe the component's behavior by monitoring its execution. The normal execution of the component is expressed by constraints. If a fault occurs during the component's runtime execution (revealing conditions that violate correctness assumption about the execution of the component), the Fault Detector sends two messages; the first message contains the fault information (fault time and the current conditions of the component) will be sent to the Fault Analyzer, the second message will be sent to the Fault Expansion Detector (in the Fault Control Phase). This message notifies the Fault Expansion Detector that the component has failed.

- Module No.2: Fault Analyzer
  This module analyzes the root cause of the fault and whether the fault is internal or external).

- **Fault Control Phase**
  The task of the homeostasis phase in the wound-healing process is to stop bleeding after the injury is detected. In this phase we aim to stop the expansion of the fault. If one of the components of the system fails, this fault may affect the other components that are connected to the faulty component. Figure 3 shows an example.

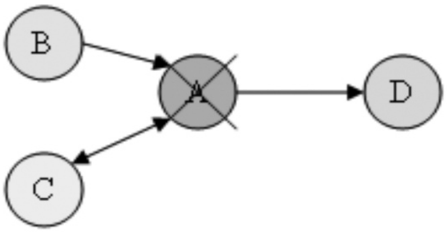


FIGURE 3. Example of Fault Expansion

In the normal execution, component A receives messages from component B and C, and sends two messages; one to component D and the other back to component C.

If component A fails to provide its service, which means component A will not be able to receive messages from other components and each output message will be a wrong output which might affect the other components by sending the wrong data or messages. This also might lead to failures in components C and D. In this case, we need to stop component A from sending and receiving messages. In other words, we need to isolate component A.

In the wound-healing process, two types of cells are responsible for stopping the bleeding: platelets and blood vessels. To achieve this in our model, we provide two elements:

- Module No. 3: Fault Expansion Detector
  The task of this element is to create two sets: the first set, called Sender Components Set (SCS), contains the components that send messages to the faulty component, for example, in Figure 3, if the faulty component is A, SCS set contains B and C. The second set, called Receiver Components Set (RCS) contains the components that receive messages from the faulty component, for example, in Figure 3, if the faulty component is A, RCS set contains C and D. In the wound-healing process, platelets produce special chemicals that aid to stop the bleeding. Likewise, Fault Expansion Detector generates the two sets (SCS, RCS), which aid in stopping fault expansion, and sends these sets to the Fault Expansion Resistor.
- Module No.4: Fault Expansion Resistor
  After receiving the two sets, this module blocks the components that are related to the faulty component from sending/receiving messages to/from the faulty component. In Figure 4, component B, C and D will be blocked from sending/receiving messages to/from the faulty component A. To stop the bleeding, the blood vessels constrict in the area of the wound area. In the same way, Fault Expansion Resistor is responsible for stopping the fault from spreading to the other connected components.

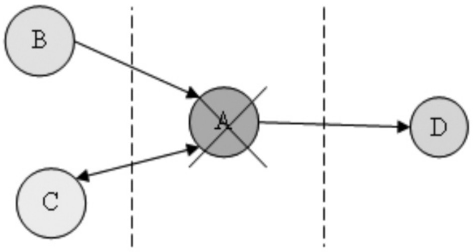


FIGURE 4. Blocking SCS and RCS

- **Repair Phase**

  Two types of cells in the inflammatory phase of the wound-healing process are responsible for repairing the injury site by producing some substances. These substances direct the next phase of the wound-healing process. Moreover, at the beginning of this phase, the body tries to remove the debris at the injury site in order to start repairing it.

  In this phase, we aim to isolate the faulty component (remove debris) in order to start repairing it. There are two ways to repair the faulty component; either to mutate the component or to replicate it. In some cases the system performs the two repair plans; mutate and replicate. This phase contains:

  - Module No.5: Repair Analyzer
    The system needs to determine what action should be taken (mutate, replicate or mutate-replicate). The Repair Analyzer then replicates, mutates or mutate-replicates the component after receiving a message from the Fault Analyzer. The Repair Analyzer sends a message to: (1) Replication Executor if the action is replicate, (2) Mutation Plan Generator if the action is mutate or (3) Mutation Plan Generator and Replication Executor if the action is mutate-replicate.
  - Module No.6: Mutation Plan Generator
    This module generates the mutation plan against the faulty components. The generated mutation plan is based on the current state of the faulty component. The Mutation Plan Generator sends the mutation plan to the Mutation Plan Executor.
  - Module No.7: Mutation Plan Executor
    The task of this module is to execute the mutation plan which has been generated by the Mutation Plan Generator.

- **Repair Validation Phase**

  In the proliferation phase of the wound-healing process, the cells start rebuilding the injury site by performing the mitotic activity. Here, we intend to make sure the chosen repair plan has been executed in a proper way.

  - Module No.8: Mutation Plan Tester
    This module tests whether the component after the mutation process works. If the component after the mutation works properly, the Mutation Plan Tester sends a message to the Runtime Manager; otherwise it sends a message to the Mutation Plan Generator to choose another configuration plan.
  - Module No.9: Replication Executor:
    The Replication Executor module replicates the component after receiving a message from the Repair Analyzer.

- **Integration Phase**

  In the wound-healing process, the last phase remodels the tissues and strengthens the scar at the injury site. In a similar manner, the integration phase task is to return the isolated component (healed component) to the system in a way that will cause no interruptions, either temporary or permanent, to the running system.

  - Module No.10: Runtime Manager

    This module returns the healed component back to the system by receiving two messages, one from the Mutation Plan Tester, which indicates that the test result is positive, and the other, from the Replication Executor which indicates that the replication has been completed. A notification message is sent to the Fault Expansion Resistor to unblock the two sets of components to reestablish interaction with the healed component.

Formal Description

DEFINITION 1

A self-healing system S can be represented by a 2-tuple element $< \rho , \mu >$:

- ρ : is the functional layer of the system S which consists of a set of components and a logical framework:

  C is a set of components,

  $C = \{\varsigma_1, \varsigma_2 \dots \varsigma_n\}$.

  Each ς_i is a 4-tuple element: <Fun$_i$, Interface$_i$, Info$_i$, Perf$_i$>, where:

  n: is the number of the components in system S,

  $1 \leq i \leq n$

  Fun$_i$: is the function of component ς_i,

  Interface$_i$: is the interface of component ς_i,

  Info$_i$: is the information of component ς_i,

  Perf$_i$: is the performance of component ς_i.

- μ : is the healing layer of the system S represented by a 5-tuple element: <Monitor, Fault Control, Repair, Repair Validation, Integration>. Each of these elements is a finite set of modules. The numbers of modules in the finite set equals to the numbers of components in the healing layer. There is one module for each component.

  - Monitor: can be represented by 2-tuple element $< A, \Sigma >$:

    - A: is a finite set of sensors modules (or Fault Detectors).

      $$\forall \varsigma_i \in C \longrightarrow \exists \alpha_i \in A.$$

      These modules analyze the state of the components at the run time (the time that the component receives input, executes process or sends output) using a set of constraints $\varkappa$ for each component at a particular time t.

      $$\forall \varsigma_i \in C \longrightarrow \exists \varkappa(\varsigma_i)_t \subseteq \varkappa(\varsigma_i).$$

      The component ς_i is considered to be in its normal condition if α_i updates the component state as follows:

      $$\forall \varsigma_i \in C, \exists t_j, t_{j+1}, \longrightarrow \varkappa(\varsigma_i)_{tj+1} \subseteq \varkappa(\varsigma_i)_{tj}, \text{ where } \varkappa(\varsigma_i)_{tj} = \varkappa(\varsigma_i).$$

      The component ς_i is considered to be in its abnormal condition if α_i updates the component state as follows:

$$\forall\ \varsigma_i \in C,\ \exists\, t_j, t_{j+1}, \longrightarrow \varkappa(\varsigma_i)_{tj+1} \not\subset \varkappa(\varsigma_i)_{tj}, \text{ where } \varkappa(\varsigma_i)_{tj} = \varkappa(\varsigma_i).$$

If the component is in its abnormal condition, α_i moves the component to the next state with this input:

$$< \varkappa\,(\varsigma_i)_{tj+1}\ ,\ \varkappa(\varsigma_i)_{t\ j} >.$$

The related components to the faulty component send/receive messages to/from the faulty component need to be blocked from sending and receiving messages. Therefore, α_i sends a blocking request to the Fault Expansion Detector.

<block>.

- Σ: is a finite set of modules (Fault Analyzers):

$$\forall\ \varsigma_i \in C \longrightarrow \exists\ \sigma_i \in \Sigma.$$

The Fault Analyzer module σ_i analyzes the constraints of the faulty components c_i at a particular time t_j to find the type of fault б_i.

$$\text{Б} = \{\text{б}_1, \text{б}_2 \ldots \text{б}_m\},$$

$$\forall\ \varsigma_i \in C \longrightarrow \exists\ \sigma_i \in \Sigma \xrightarrow{analyze} \varkappa(\varsigma_i)_{tj+1} \not\subset \varkappa(\varsigma_i)_{tj} \xrightarrow{find} \text{б}_i \in \text{Б},$$

Then, the Fault Analyzer moves the faulty component to the next state with input:

$$<\varkappa(\varsigma_i)_{tj+1}\ , \varkappa(\varsigma_i)_{tj}, \text{б}_i>.$$

- Fault Control: can be represented by a 2-tuple element <D, R>:
  - D: a finite set of modules (Fault Expansion Detectors).

$$\forall\ \varsigma_i \in C \longrightarrow \exists\ \eta_i \in D$$

The Fault Expansion Detector σ_i of a faulty component ς_i, creates two sets of components SCS, RCS:

SCS: is a set of components that send outputs to the faulty component ς_i.

$$SCS = \{\zeta_1, \zeta_2 \ldots \zeta_n\},\ \text{where } \zeta_1, \zeta_2 \ldots \zeta_n \in C.$$

RCS: is a set of components that receive inputs from the faulty component ς_i.

$$RCS = \{\xi_1, \xi_2 \ldots \xi_n\}, \text{ where } \xi_1, \xi_2 \ldots \xi_n \in C.$$

The Fault Expansion Detector moves the faulty component to the next state with input:

<SCS, RCS>

- R: a finite set of modules (Fault Expansion Resistors):

$$\forall \varsigma_i \in C \longrightarrow \exists\ \varepsilon_i \in R.$$

The Fault Expansion Resistor receives the two sets of components, SCS and RCS, and blocks the two sets from sending/receiving messages to/from the faulty component.

$$\forall \varsigma_i \in C \longrightarrow \exists\, \varepsilon_i \xrightarrow{block} SCS\,\hat{}\ \varepsilon_i \xrightarrow{block} RCS, \text{ where } \varepsilon_i \in R.$$

After receiving a notification message from the Runtime Manager indicates that the faulty component is healed, the Fault Expansion Resistor unblocks the two sets of component.

$$\forall \varsigma_i \in C \longrightarrow \exists\, \varepsilon_i \xrightarrow{unblock} SCS\,\hat{}\ \varepsilon_i \xrightarrow{unblock} RCS, \text{ where } \varepsilon_i \in R.$$

- Repair: can be represented by 3-tuple element <Θ, Ψ, Φ>:
  - Θ: a finite set of modules (Repair Analyzers):

$$\forall \varsigma_i \in C \longrightarrow \exists\, \theta_i \in \Theta.$$

The Repair Analyzer θ_i of a faulty component ς_i receives an input from the Fault Analyzer which contains the message:

$$<\varkappa(\varsigma_i)_{tj+1}\,,\ \varkappa(\varsigma_i)_{tj},\ \text{б}_i>.$$

The Repair Analyzer analyzes the state of the component and the fault type б_i to find a suitable Repair Plan γ_j, where $1 < j <3$.

Y is the set a three repair plans.
Y = $\{\gamma_1, \gamma_2, \gamma_3\}$, where
γ_1: Mutate,
γ_2: Replicate,
γ_3: Mutate-Replicate.

$$\forall\ \varkappa(\varsigma_i)_{tj+1} \not\subset \varkappa(\varsigma_i)_{tj} \longrightarrow \exists \text{б}_i \in \text{Б}\ \text{, then,}$$

$$\forall\ \text{б}_i \in \text{Б} \longrightarrow \exists\ \gamma_n \in Y, :\ 1 \leq\ n\ \leq 3$$

After determining the suitable plan, the Repair Analyzer sends a notification message to the next state containing the appropriate plan:

$$<\gamma_n>.$$

  - Ψ: a finite set of modules (Mutation Plan Generators):

$$\forall \varsigma_i \in C \longrightarrow \exists\ \psi_i \in \Psi.$$

The Mutation Plan Generator ψ_i of a faulty component ς_i uses the current constraints of the component $\varkappa(\varsigma_i)_{tj+1}$ and the type of the fault б_i to find the suitable Mutation Plan ω_i.

$$\forall\ \varkappa(\varsigma_i)_{tj+1} \not\subset \varkappa(\varsigma_i)_{tj}\,\hat{}\ \exists \text{б}_i \in \text{Б} \longrightarrow \omega_i \in \Omega.$$

Then the Mutation Plan Generator ψ_i moves the faulty component to the next state with input:

$$<\omega_i>.$$

  - Φ: a finite set of modules (Mutation Plan Executors):

$$\forall \varsigma_i \in C \longrightarrow \exists\ \phi_i \in \Phi.$$

The Mutation Plan Executor ϕ_i of a faulty component ς_i executes the Mutation Plan ω_i by performing the operation in definition 2, then notifies the Mutation Plan Tester to test the healed component:

<test>

- Repair Validation: can be represented by 2-tuple element <T, X>
    - T: a finite set of modules (Mutation Plan Testers):

      $$\forall \varsigma_i \in C \longrightarrow \exists\, \tau_i \in T.$$

      The Mutation Plan Tester τ_i tests whether the faulty component is healed. The Mutation Plan Tester τ_i achieves the test by sending a test data to the component. After receiving the output data, the τ_i checks the constraints of the component, if the constraints of the component after the test equals to the constraints of the component, then, the faulty component is healed. As a consequence, τ_i sends a notification message to the Run-Time Manager:

      <succeed>,

      Otherwise, τ_i sends a notification message back to the Mutation Plan Generator ψ_i to determine another mutation plan.

      <failed>.

    - X: a finite set of modules (Replicate Executors):

      $$\forall \varsigma_i \in C \longrightarrow \exists\ \chi_i \in X.$$

      The Replicate Executor replicates the component by performing definition 3, and then sends a notification message to the Runtime Manager.

      <replicated>

- Integration: can be represented by 1-tuple element <I>:
    - I: a finite set of modules (Runtime Managers).

      $$\forall \varsigma_i \in C \longrightarrow \exists\, \upsilon_i \in I.$$

      The Runtime Manager υ_i returns the healed component ς_i to the running system. It sends an unblocking message to the Fault Expansion Resistor to unblock the two sets of component SCS, RCS.

DEFINITION 2

An action in a self-healing system S with component ς_i is called mutate if the following conditions are satisfied:

1. $\forall \varsigma_i \in C, \exists\, \gamma_1 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{mutate} \varsigma_i'$, where
2. Fun(ς_i) = Fun(ς_i') ,
3. Interface(ς_i) = Interface(ς_i'),
4. Perf(ς_i) < Perf(ς_i').

DEFINITION 3

An action in a self-healing system S with component ς_i is called replicate if the following conditions are satisfied:

1. $\forall \varsigma_i \in C, \exists\ \gamma_2 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{\textit{replicate}} \varsigma_i'$, where
2. Fun$(\varsigma_i) \subseteq$ Fun(ς_i') ,
3. Interface$(\varsigma_i) \subseteq$ Interface(ς_i'),
4. Perf(ς_i) = Perf(ς_i').

DEFINITION 4

An action in a self-healing system S with component ς_i is called mutate-replicate action if the following conditions are satisfied:

$\forall \varsigma_i \in C, \exists\ \gamma_1 \in Y$ CurInfo $(\varsigma_i) \supseteq$ Cond $(\varsigma_i) \Rightarrow \varsigma_i \xrightarrow{\textit{mutate}} \varsigma_i'$

1. Fun$(\varsigma_i) \subseteq$ Fun(ς_i')
2. Interface$(\varsigma_i) \subseteq$ Interface(ς_i')
3. Perf$(\varsigma_i) <$ Perf(ς_i')

$\forall \varsigma_i' \in C, \exists\ \gamma_2 \in Y$ CurInfo $(\varsigma_i') \supseteq$ Cond $(\varsigma_i') \Rightarrow \varsigma_i' \xrightarrow{\textit{replicate}} \varsigma_i''$,

1. Fun(ς_i') = Fun(ς_i'') ,
2. Interface(ς_i) = Interface(ς_i''),
3. Perf(ς_i') = Perf(ς_i'').

That means:

$$\varsigma_i \xrightarrow{\textit{mutate}} \varsigma_i' \xrightarrow{\textit{replicate}} \varsigma_i''$$

The system performs the mutate operation in order to heal the faulty component. Then, the system replicates the healed component.

Associative Property

Associative property means, within an expression two or more of the same associative operators in a row, the order of operations does not matter as long as the sequence of the operands is not changed. We prove whether the associative property is satisfied in definition 2, 3, 4.

- **Proof:** Associative Property for Definition 2:
  In definition 2, the mutate operation, which is:

  $$\varsigma_i \xrightarrow{\textit{mutate}} \varsigma_i',$$

  does not satisfy the associate property because of the time dimension. In other words, the mutate operation is not reversible because of the time dimension; therefore the associate property is not satisfied.

- **Proof:** Associative Property for Definition 3
  Likewise, time dimension makes the replicate operation in definition 3 non reversible.

  $$\varsigma_i \xrightarrow{\textit{replicate}} \varsigma_i'$$

  As a consequence, the replicate operation does not satisfy the associate property.

- **Proof:** Associative Property for Definition 4
  Definition 4 which is the mutate-replicate operation consists of two parts, mutate and replicate. Here, we intend to prove the associate property of the mutate-replicate operation.
  If the sequence of the operation is mutate then replicate,

$$\varsigma_i \xrightarrow{\textit{mutate}} \varsigma_i' \xrightarrow{\textit{replicate}} \varsigma_i''$$

After the faulty component ς_i is mutated, the resulting component ς_i' will have complete basic properties (Fun, Interface, Perf). The outcome of the fist part is:

$$\begin{aligned}&\mathrm{Fun}(\varsigma_i) \subseteq \mathrm{Fun}(\varsigma_i')\\&\mathrm{Interface}(\varsigma_i) \subseteq \mathrm{Interface}(\varsigma_i')\\&\mathrm{Perf}(\varsigma_i) < \mathrm{Perf}(\varsigma_i')\end{aligned}$$

The second part of the operation is to replicate the mutated component ς_i'. The outcome of the second part is:

$$\begin{aligned}&\mathrm{Fun}(\varsigma_i') = \mathrm{Fun}(\varsigma_i'')\ ,\\&\mathrm{Interface}(\varsigma_i) = \mathrm{Interface}(\varsigma_i''),\\&\mathrm{Perf}(\varsigma_i') = \mathrm{Perf}(\varsigma_i'').\end{aligned}$$

At the end of the mutate-replicate operation, the resulting component will have the complete specifications of the system's component.

In the same way, if the sequence of the two parts of the operation has changed, the resulting component will have the complete specifications of the system's component.

$$\varsigma_i \xrightarrow{\textit{replicate}} \varsigma_i' \xrightarrow{\textit{mutate}} \varsigma_i''$$

As a result, definition 4 satisfies the associative property.

Closure Property

The closure property means, the operation on members of the set produces a member of the set. We prove whether the closure property is satisfied in definitions 2, 3, and 4.

- **Proof** : Closure Property for Definition 2
  In definition 2, the mutate operation, which is

$$\varsigma_i \xrightarrow{\textit{mutate}} \varsigma_i',$$

The result of this operation is:

$$\begin{aligned}&\mathrm{Fun}(\varsigma_i) \subseteq \mathrm{Fun}(\varsigma_i')\\&\mathrm{Interface}(\varsigma_i) \subseteq \mathrm{Interface}(\varsigma_i')\\&\mathrm{Perf}(\varsigma_i) < \mathrm{Perf}(\varsigma_i')\end{aligned}$$

After the faulty component ς_i is mutated, the resulting component ς_i' will have the complete specifications (Fun, Interface, Perf) of the system's component. Therefore, the mutate operation satisfies the closure property.

- **Proof** : Closure Property for Definition 3
  The replicate operation in definition 3 is:

$$\varsigma_i \xrightarrow{replicate} \varsigma_i'$$

  The result of this operation is:

$$Fun(\varsigma_i) = Fun(\varsigma_i')\ ,$$
$$Interface(\varsigma_i) = Interface(\varsigma_i'),$$
$$Perf(\varsigma_i) = Perf(\varsigma_i').$$

  The replicate operation makes a copy of the system's component at a certain time. As a consequence, the resulting component will have the same specifications (Fun, Interface, Perf) of the system's component.

  From definition 3, we can deduce that the replicate operation satisfies the closure property.

- **Proof**: Closure Property for Definition 4
  To prove the closure property of definition 4, we need to perform the two parts of this definition. The two part of mutate-replicate operation, definition 4, are:

1. Mutate:

$$\varsigma_i \xrightarrow{mutate} \varsigma_i',$$

   The result of this part is:

$$Fun(\varsigma_i) \subseteq Fun(\varsigma_i')$$
$$Interface(\varsigma_i) \subseteq Interface(\varsigma_i')$$
$$Perf(\varsigma_i) < Perf(\varsigma_i').$$

2. Replicate:

$$\varsigma_i' \xrightarrow{replicate} \varsigma_i''$$

   The result of this part is:

$$Fun(\varsigma_i') = Fun(\varsigma_i'')\ ,$$
$$Interface(\varsigma_i) = Interface(\varsigma_i''),$$
$$Perf(\varsigma_i') = Perf(\varsigma_i'').$$

  At the end of the mutate-replicate operation, the resulting component will have the complete specifications of the system's component. We can deduce that the mutate-replicate operation satisfies the closure property.

Graphnet Model

Beside the set-theory definition, finite state machine (FSM) is usually encountered as graphical objects. In this section, we present a finite state machine called graphnet. Graphnet is presented to prove the functionality of our architecture. The graphnet model starts by describing the states of the system's component and the transitions between them.

Single-fault

The graphnet in Figures 5, 6, and 7 presents one component at a time. However, the architecture deals with single-fault. The healing layer creates one module for the faulty component in each state. This seems practical and easy to develop.

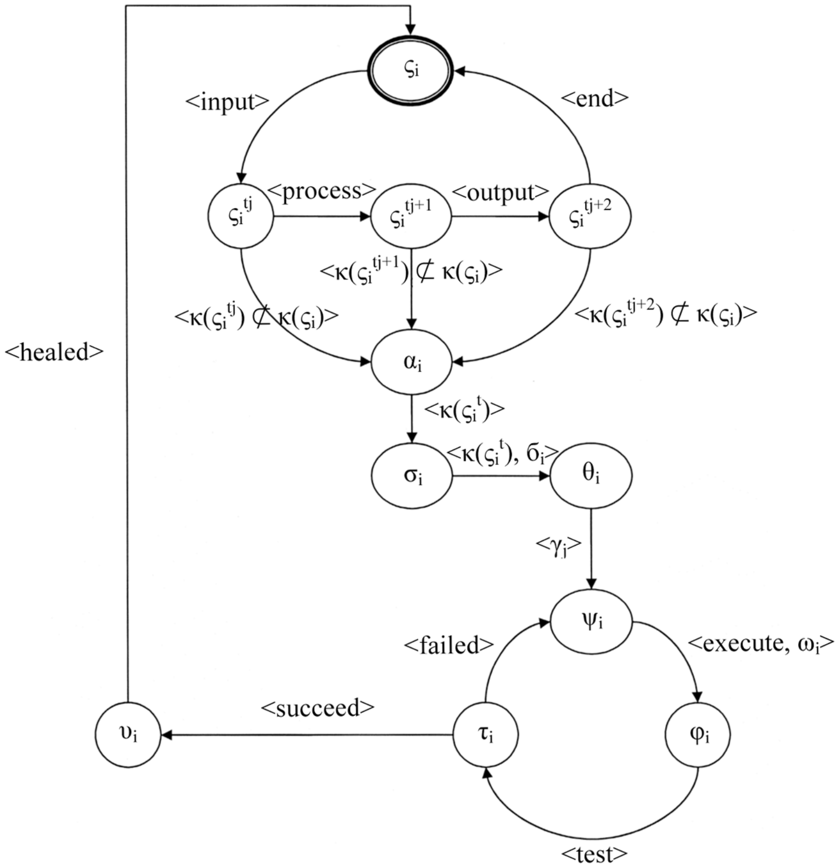

ς_i
<input>
<end>
ς_i^{tj}
<process>
ς_i^{tj+1}
<output>
ς_i^{tj+2}
$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$
<healed>
α_i
$<\kappa(\varsigma_i^{t})>$
σ_i
$<\kappa(\varsigma_i^{t}), б_i>$
θ_i
$<\gamma_i>$
ψ_i
<failed>
$<$execute, $\omega_i>$
<succeed>
υ_i
τ_i
φ_i
<test>

FIGURE 5. Mutate Plan Graphnet for Single-Fault

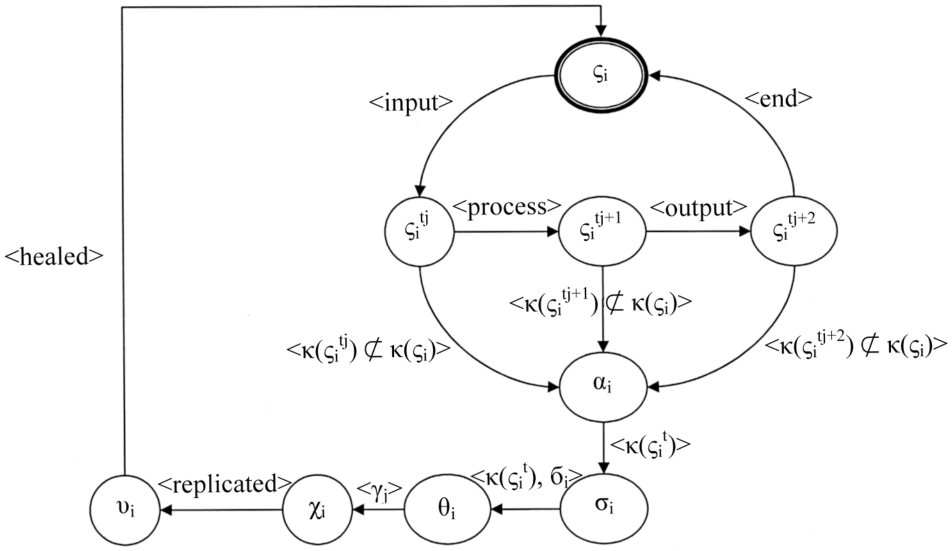

ς_i
<input>
<end>
<healed>
ς_i^{tj}
<process>
ς_i^{tj+1}
<output>
ς_i^{tj+2}
$<\kappa(\varsigma_i^{tj+1}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj}) \not\subset \kappa(\varsigma_i)>$
$<\kappa(\varsigma_i^{tj+2}) \not\subset \kappa(\varsigma_i)>$
α_i
$<\kappa(\varsigma_i^{t})>$
<replicated>
$<\gamma_i>$
$<\kappa(\varsigma_i^{t}), б_i>$
υ_i
χ_i
θ_i
σ_i

FIGURE 6. Replicate Plan Graphnet for Single-Fault

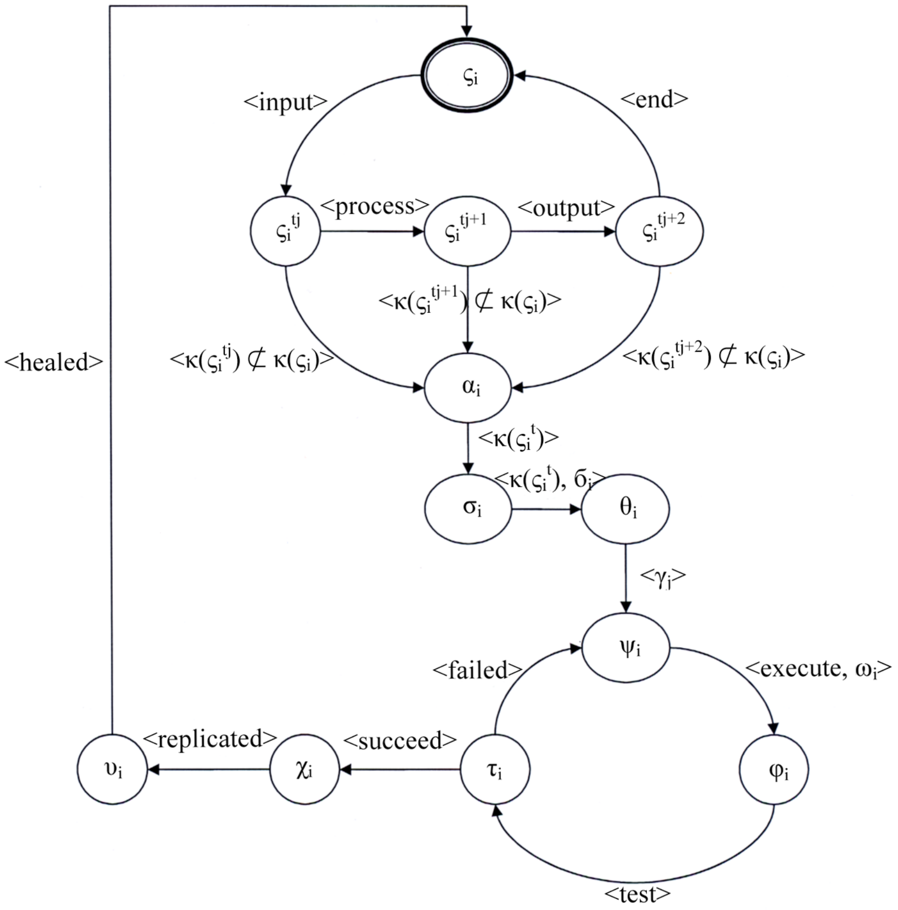


FIGURE 7. Mutate-Replicate Plan Graphnet for Single-Fault

CONCURRENT-FAULTS

The architecture can handle single-fault as well as concurrent-faults. To handle concurrent-faults, the system creates one module for each component (Figure 8, 9, and 10). In each state of the components, the system creates a set of modules in order to handle concurrent-faults. The number of modules in each state equals to the number of faulty components.

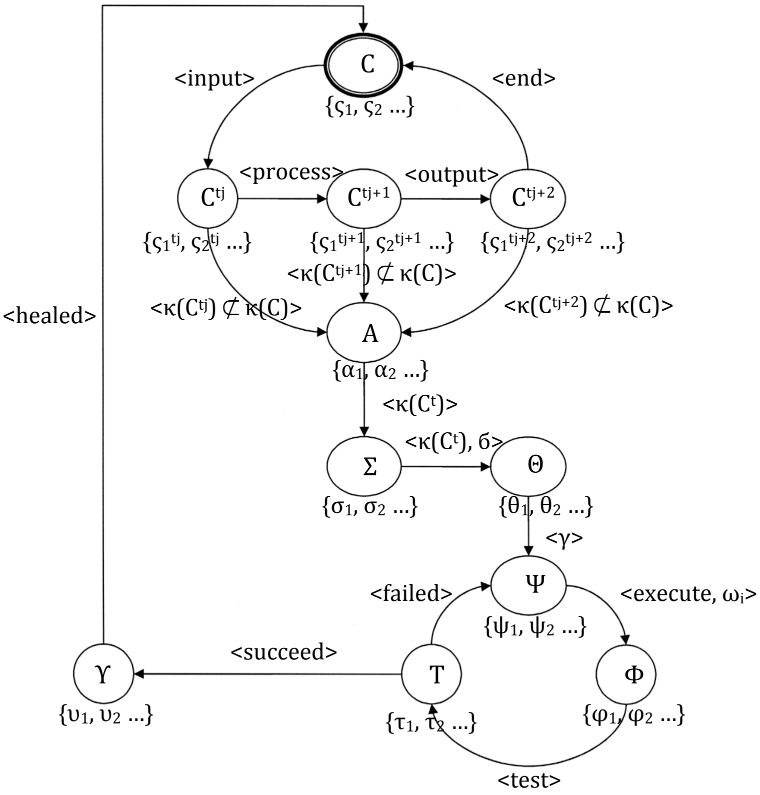


FIGURE 8. Mutate Plan Graphnet for Concurrent-Faults

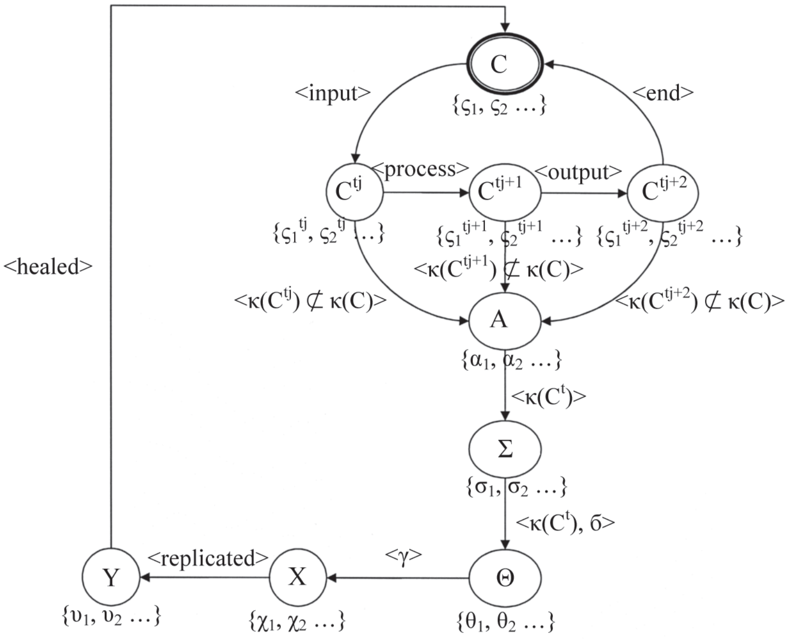

C
{ς₁, ς₂ …}
<input>
<end>
<process>
<output>
C^{tj}
C^{tj+1}
C^{tj+2}
$\{ς_1^{tj}, ς_2^{tj} …\}$
$\{ς_1^{tj+1}, ς_2^{tj+1} …\}$
$\{ς_1^{tj+2}, ς_2^{tj+2} …\}$
<healed>
$<κ(C^{tj+1}) \not\subset κ(C)>$
$<κ(C^{tj}) \not\subset κ(C)>$
$<κ(C^{tj+2}) \not\subset κ(C)>$
A
{α₁, α₂ …}
$<κ(C^t)>$
Σ
{σ₁, σ₂ …}
$<κ(C^t), б>$
<replicated>
<γ>
Y
{υ₁, υ₂ …}
X
{χ₁, χ₂ …}
Θ
{θ₁, θ₂ …}

FIGURE 9. Replicate Plan Graphnet for Concurrent-Faults

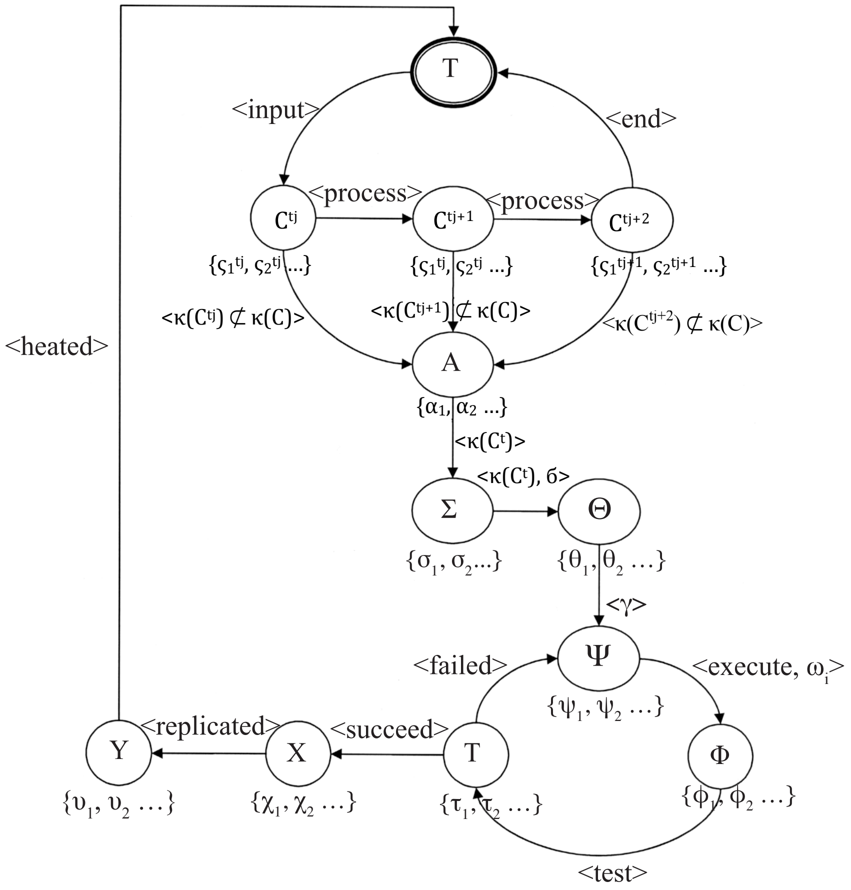


FIGURE 10. Mutate-Replicate Plan Graphnet for Concurrent-Faults

Case Study

Menasce & Kephart (2007) said that self-healing applications should be able to recover from potential faults and should continue to work smoothly without human intervention. In this section, we applied our architecture into an Automated Teller Machine (ATM) system.

De Lemos et al. (2007) defined Automated Teller Machine (ATM) as a telecommunications device that computerizes the financial transactions in a financial institution and allows the customer to access these transactions in a public space without the need for human clerk. This section introduces software system that simulates the Automated Teller Machine (ATM). We developed an ATM application using Java. The ATM application provides the basic financial transactions. The customers can check their accounts, withdraw cash, and transfer money to other customers.

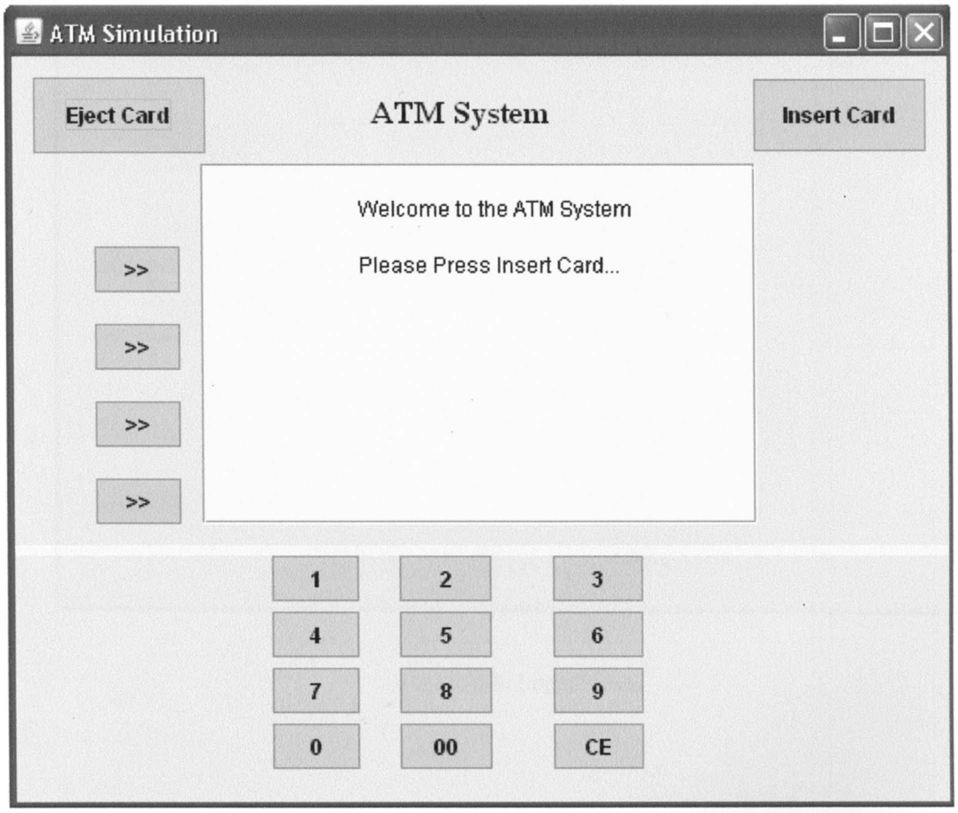
Functional Layer

The ATM system services one customer at a time. The customer needs to insert a special plastic card into an ATM card reader. After inserting the plastic card into the ATM card reader, the customer needs to enter Personal Identification Number (PIN) using a keypad. The PIN will be transmitted to the bank central system. This number prevents unauthorized persons from performing transactions. If the PIN code is correct, the customer will be able to perform one or more financial transactions. The card will be inside the card reader until the customer indicates that they desires to exit from the system.

The authorized customer must be able to:

- Check their account balance.
- Make cash withdrawal.
- Make a transfer of money to any other account liked to the bank.
- Abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

Figure 11 illustrates a screenshot of the ATM system.



Welcome to the ATM System

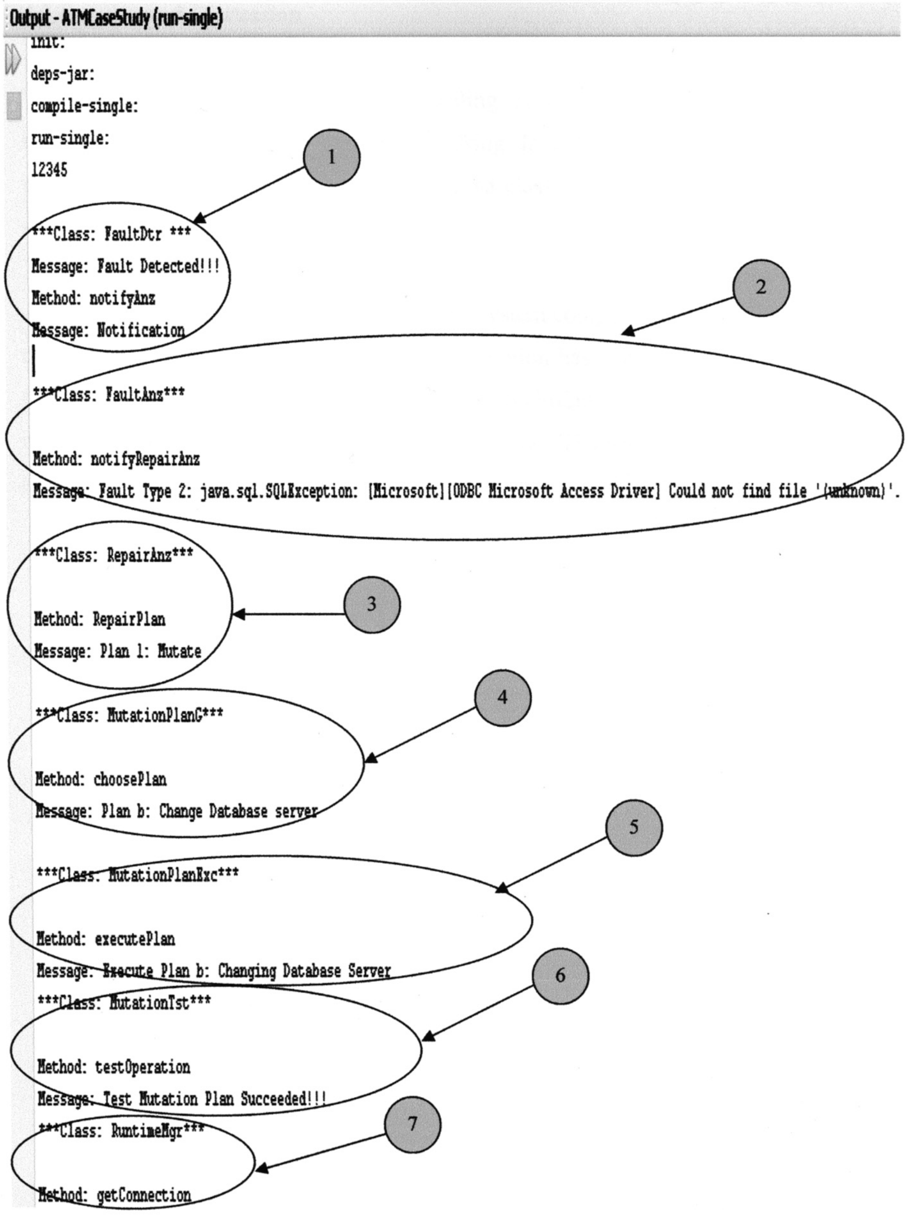
Please Press Insert Card...

FIGURE 11. Screenshot of the ATM System

HEALING LAYER

Each time the customer performs a financial transaction using the ATM, the system needs to access the bank database. The database contains all the account information of the customer. In our application, we assume that the server that contains the required information of the bank customer has encountered a failure. Fortunately, a backup from this information is located in another server. This failure might happen during the execution of any transaction.

The failure of the database server which is located in the bank site leads to a failure into the ATM system. If the system fails during a transaction operation, customers will not trust the services provided by the ATM system. The system must be able to detect the failure of the database server and must be able to access the backup server. Figure 12 illustrates the output from each module in the healing.



```
init:
deps-jar:
compile-single:
run-single:
12345

***Class: FaultDtr ***
Message: Fault Detected!!!
Method: notifyAnz
Message: Notification

***Class: FaultAnz***

Method: notifyRepairAnz
Message: Fault Type 2: java.sql.SQLException: [Microsoft][ODBC Microsoft Access Driver] Could not find file '(unknown)'.

***Class: RepairAnz***

Method: RepairPlan
Message: Plan 1: Mutate

***Class: MutationPlanG***

Method: choosePlan
Message: Plan b: Change Database server

***Class: MutationPlanExc***

Method: executePlan
Message: Execute Plan b: Changing Database Server
***Class: MutationTst***

Method: testOperation
Message: Test Mutation Plan Succeeded!!!
***Class: RuntimeMgr***

Method: getConnection
```

FIGURE 12. The Output Results of the Healing Modules in the Healing Layer

COMPARATIVE STUDY

Firstly, proactive self-healing for application maintenance in ubiquitous computing environment has 6 multi agents and 6 processes which are; monitoring agent, component agent, system agent, diagnosis agent, decision agent, and searching agent; as mentioned by Park et al. (2006). Meanwhile, Haydarlou et al. (2005) said that self-healing approach for object-oriented applications architectural approach has two feedback loops that manage general and detail functions in its processes. On the other hand, in Fuad et al. (2006) adds self-healing capabilities into legacy object oriented application architecture using sensors and hooks at the strategic point to interact with the managed code. Shin (2005) claimed that self-healing components in robust software architecture for concurrent and distributed systems architecture structured to the layered architecture with two layers; the service layer and the healing layer. And finally, Park et al. (2008) proposed architecture of self-healing mechanism for reliable computing which designed and structured with three layers: the monitoring layer, diagnosis & decision layer, and adaptation layers.

The proposed multi-tiered bio-inspired self-healing architectural paradigm for software systems is inspired by a biological process, i.e., the wound-healing process. This architecture consists of two layers: functional layer and healing layer. In the functional layer, the system components operate and interact with each other without any disruptions. The healing layer will return the faulty component to its normal condition by applying the wound-healing phases into the self-healing paradigm. Healing layer consists of five phases; monitoring, fault control, repair, repair validation, and integration phase. This architecture compliments the state-of-the-art architecture by implementing layers to the system. This architecture also has the looping structure where it returns the faulty component to its normal execution state while monitoring the system. The multi-tiered approach in all these works has some common features such as robustness, expandability distributability and concurrency.

Conclusions and Future Works

This paper presented software system architecture with self-healing characteristics. The proposed architecture is based on biological system that has the ability to heal by itself (the wound-healing process). The architecture consists of two layers; functional and healing layers. In the functional layer, the system components operate and interact with each other without any disruptions. The healing layer aims to provide the ability for the system to handle anomalous conditions. Theoretical and formal descriptions of the proposed architecture have been presented. In order to test the functionality of the proposed architecture, an ATM system has been developed and tested.

Our recommendations for further research are: a combination of self-healing and self-protecting is highly recommended in order to stop viruses' attacks during the healing process and to heal from a system failure that has occurred after an attack by virus, and developing self-healing middleware for large-scale distributed software systems based on the biological systems. This middleware will increase the reliability of the distributed application and support the interoperability among the system components in a heterogeneous environment.

REFERENCES

Ahmed, S., Sharmin, M., and Ahamed. S.I. 2007. ETS (Efficient, Transparent, and Secured) self-healing service for pervasive computing applications. International Journal of Network Security 4(3): 271–281.

Balasubramaniam, S., Botvich, D., Donnelly, W., Foghlu, M. O., and Strassner, J. 2006. Biologically inspired self-governance and self-organization for autonomic networks. Proceedings of the First International Conference on Bio Inspired Models of Network, Information and Computing Systems, Cavalese, Italy.

Bokareva, T., Bulusu, N., and Jha, S. 2005. SASHA: Toward self-healing hybrid sensor network architecture. Proceedings of the Second IEEE Workshop on Embedded Networked Sensors, 71-78. IEEE Computer Society.

Boonma, P. & Suzuki, J. 2007. BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks. Computer Networks 51(16): 4599-4616. Elsevier.

Champrasert, P. & Suzuki, J. 2006. A biologically-inspired autonomic architecture for self-healing data centers. Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06) 01: 103-112. IEEE Computer Society.

Clark, J. & King, S. 2004. The wound-healing process. Peterborough Wound Care Manual.

Cockbill, S. 2002. Wounds the healing process. Royal Pharmaceutical Society of Great Britain.

De Lemos, R., Timmis, J., Ayara, M., and Forrest, S. 2007. Immune-inspired adaptable error detection for automated teller machines. IEEE Transactions on Systems, Man, and Cybernetics-Part C: Applications and Reviews 37(5): 873 - 886.

Diegelmann, R. F. & Evans, M. C. 2004. Wound healing: an overview of acute, fibrotic and delayed healing. Frontiers in Bioscience 9: 283-289.

Fuad, M. M., Deb, D., and Oudshoorn, M. J. 2006. Adding self-healing capabilities into legacy object oriented application. Proceedings of the International Conference on Autonomic and Autonomous Systems, 51. IEEE Computer Society.

George, S., Evans, D., and Davidson, L. 2002. A biologically inspired programming model for self-healing systems. Proceedings of the First Workshop on Self-Healing Systems, Charleston, South Carolina, 102-104.

Haydarlou, A.R., Overeinder, B. J., and Brazier, F. M. T. 2005. A self-healing approach for object-oriented applications. Proceedings of the 16th International Workshop on Database and Expert Systems Applications, 191-195.

Horn, P. 2001. Autonomic computing: IBM's perspective on the state of information technology. IBM Corporation.

Keast, D. & Orsted, H. The basic principles of wound-healing. (online). http://www.pilonidal.org/pdfs/Principles-of-Wound-Healing.pdf (November 2007).

Kephart, J. O. & Chess, D. M. 2003. The vision of autonomic computing. Computer 36(1): 41-50. IEEE Computer Society.

Laddaga, R. 1999. Creating robust software through self-adaptation. IEEE Intelligent Systems 14(3): 26-29.

Menasce, D. A. & Kephart, J. O. 2007. Guest editors' introduction: Autonomic computing. IEEE Internet Computing 11(1): 18-21.

Park, J., Jung, J.,Piao, S., and Lee, E. 2008. Self-healing mechanism for reliable computing. International Journal of Multimedia and Ubiquitous Engineering 3(1).

Park, J., Yoo, G., Jeong, C., and Lee, E. 2006. Proactive self-healing system for application maintenance in ubiquitous computing environment. In Lecture Notes in Computer Science (LNCS) 3981, 430-440. Heidelberg: Springer-Verlag.

Perry, D. E. & Wolf, A. L. 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4): 40-52.

Salehie, M. & Tahvildari, L. 2005. Autonomic computing: emerging trends and open problems. ACM SIGSOFT Software Engineering Notes 30(4): 1-7.

Shaw, M. & Garlan, D. 1996. Software architecture: perspectives on an emerging discipline. Prentice-Hall.

Shin, M. E. 2005. Self-healing components in robust software architecture for concurrent and distributed systems. Science of Computer Programming 57(1): 27-44. Elsevier.

WoundHeal. (online). http://www.woundheal.com/healing/processIndex.htm (22 November 2007).

Azween Abdullah
Department of Computer & Information Sciences
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, 31750 Tronoh, Perak
azweenabdullah@petronas.com.my

Ria Candrawati
Department of Computer & Information Sciences
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, 31750 Tronoh, Perak
riacandrawati@yahoo.com

M. Agni Catur Bhakti
Department of Computer & Information Sciences
Universiti Teknologi PETRONAS
Bandar Seri Iskandar, 31750 Tronoh, Perak
m.agni.cb@gmail.com