A Self-Healing Technique based on Encapsulated Operation Knowledge

Teruyoshi Zenmyo, Hideki Yoshida, Tetsuro Kimura
Corporate Research & Development Center, Toshiba Corporation
1, Komukai-Toshiba-cho, Saiwai-ku, Kawasaki, 212-8582, Japan
{zenmyo,hideki,kimura}@isl.rdc.toshiba.co.jp

Abstract—Although autonomic computing reduces traditional operational cost, it introduces another cost factor related to operation knowledge. This paper focuses on self-healing functionality and proposes a technique which improves reusability of component-level operation knowledge for self-healing systems. Operation knowledge which attains reusability becomes independent of a specific system structure, and therefore it can be reused across organizations and adapted to changes. To achieve reusability, operation knowledge is encapsulated in components whereas system specific information is excluded. To cope with the remaining problem of dependency among components, a dependency injection mechanism is introduced. The dependency injection mechanism works out needed recovery actions by relating component system-specific information. Furthermore, this paper describes an implemented prototype together with an application example.

I. INTRODUCTION

Rising operational costs of large computer systems is spurring demand for self-healing functionality[1]. Operation knowledge that describes ways to detect faults and to specify recovery actions is essential for realizing the self-healing functionality. While self-healing functionality is expected to reduce costs dramatically, additional tasks to describe and maintain operation knowledge become another cost factor. These tasks require extensive experience of system operation and a full understanding of the target system and its components. If the operation knowledge is insufficient, complementary manual operations will be required. To improve quality of operation knowledge, operation knowledge should be revised to incorporate those complementary operations. But, in the revising process, ad hoc modifications or system-specific descriptions tend to be incorporated, and operation knowledge becomes strongly dependent on the target system’s configuration at that time. Maintenance cost of operation knowledge will be increased in this way.

Operation knowledge as a whole is suited to its target system and difficult to share or reuse. But a component of a system is often used in many other systems, and moreover, since operation knowledge for that component is essentially same, that operation knowledge can be shared by some means. We believe that realizing the self-healing functionality by building up component-level operation knowledge will improve quality and maintainability. Key points for realizing the idea are system-independent description of operation knowledge and a mechanism to resolve system-specific dependency among components.

In this paper, a self-healing technique which enables construction of self-healing systems by aggregating component-level operation knowledge is proposed. To improve reusability of operation knowledge, dependency between components is expressed as a system-independent constraint of the component which depends on other components. Such constraint and other operation knowledge which represents operations on a component are encapsulated in each component. For resolving system specific dependency, dependency injection mechanism is introduced, which injects dependency into self-healing systems with the component-specific constraint and system specific information. By the encapsulation and the dependency injection, each component’s operation knowledge becomes independent of that of every other component. Since modification of component-level operation knowledge does not affect other components’ self-healing behavior, maintainability is improved. Moreover, such a modified component-level operation knowledge, which is more sophisticated than unmodified component-level operation knowledge, can be reused across environments and organizations and can contribute to improved quality of self-healing systems since the component-level operation knowledge is independent of system structures.

Furthermore, an implemented prototype is presented together with a simple example. In the example, self-healing functionality is added by weaving an aspect[2]. Aspect-oriented approach allows construction of self-healing systems without modification of original components.

The rest of this paper is organized as follows. Section II formulates problems related to tasks concerning operation knowledge. In Section III, the concept and architecture of the proposed self-healing technique are described. Section IV presents an application example together with an implemented prototype. Section V discusses the proposed self-healing system. Related works are summarized in Section VI and Section VII concludes this paper.

II. PROBLEM FORMULATION

This section describes problems in performing tasks concerning operation knowledge.

One of the problems is difficulty in describing correct and exhaustive operation knowledge. A goal of self-healing systems is to prevent failure and maintain correct service
delivery. An error, which is a state leading to failures[3], is the
defect that should be removed by a self-healing functionality.
For recoveries from errors, operation knowledge in which
ways to identify errors and to recover from errors are described
is necessary. A variety of errors may occur in the target system
since the errors are caused by many kinds of faults (e.g. the
fault may originate from human mistakes or may originate
from bugs in software). Operation knowledge is required to
describe information on the variety of errors exhaustively,
otherwise operation knowledge restricts the scope of self-
healing. Additionally, correctness of operation knowledge is
required to make self-healing behavior correct.

To describe operation knowledge, developers or admin-
istrators have to search necessary information from many
sources (e.g. web pages) and have to formalize it so that
it can be understood by a self-healing system. The above-
mentioned information has to be organized to suit the target
system. Although the organizing tasks depend on the
target system configuration, component level is common
across organizations and environments for most of the other
tasks since there are many components which are used in many
places (e.g. open source software).

Repeating such common tasks independently for each sys-
tem and each organization is costly since it requires either
experienced engineers or searching many sources. Moreover,
it makes the describing process error-prone since the tasks
become dependent on the skill of a person who describes
operation knowledge. It is necessary to repeat common tasks
due to a lack of reusability of component-level operation
knowledge.

Maintainability of operation knowledge deployed in self-
healing systems is the other problem. Deployed operation
knowledge has to be modified in the case that a system
component is changed (e.g. a new server is added or an
application software is replaced) or a new error is detected.
These maintenance tasks are important in view of changing
user demand and the difficulty of forecasting potential errors.

Since an error can propagate to other components due to
dependency among the components, the error in a certain
component can cause a failure which is detected at exter-
nal component’s interface[3]. Error propagation makes self-
healing behavior complex since it scatters existence of a
failure cause and it requires dependency information among
components’ operation knowledge.

Dependency is system-specific information since it is de-
termined by components comprising the system and the rel-
ationship among the components. Dependency in a certain
system is one of the features which dictate a correct state of
certain component deployed in the system. On the other
hand, ways to control the component in the correct state are
component-specific information. These two types of informa-
tion should be explicitly divided. Otherwise, system-specific
and component-specific information are blended together and
operation knowledge becomes complex and hard to maintain.

III. CONCEPT AND ARCHITECTURE

This section describes the concept and architecture to cope
with the problems mentioned in Section II.

A. Basic Concept

As mentioned in Section II, repeating the tasks concern-
ing operation knowledge makes the tasks costly and error-
prone. Therefore, enabling operation knowledge to be reused
and accumulated improves the describing process. Operation
knowledge which is deployed in a self-healing system and
accumulated in a reusable form can be useful for construction
of another self-healing system.

Achieving reusability of operation knowledge enables sharing
it across organizations and environments. Shared operation
knowledge can be enriched with management experience of
various systems, and consequently, quality of self-healing
systems can be improved.

To achieve reusability, extracting reusable units from which
system-specific information is excluded is critical. The list of
the components of which the system exists is system specific
information. Operation knowledge about a certain component
should exclude other component information. Therefore, we
adopt a component as a reusable unit of operation knowledge.

A problem in adopting a component as a reusable unit is
how dependencies are treated. For a component which depends
on other components, it is impossible to exclude information
about other components completely although dependency con-
tains system-specific information (e.g. location where a related
component is deployed).

To cope with this problem, constraint description and de-
pendency injection mechanism are introduced. The constraint
description is a declarative dependency expression without
system-specific information. To satisfy constraint in a certain
system, the dependency injection mechanism dynamically de-
termines associated external components and needed actions.

1) Encapsulating Operation Knowledge in Components:
Operation knowledge of a component can be expressed in the
following two types of descriptions.

- constraint description
  expresses constraint concerning environments in which
  the component’s service can be correctly delivered.
- operation description
  expresses ways to monitor the component, to determine
  its state and to change its state.

The constraint description and the operation description are
encapsulated in each component.

We call components having the constraint description de-
pender components. A depender component depends on other
components. In contrast, components on which depender com-
ponents depend are called dependee components. It is possible
for a certain component to be both a depender component and
a dependee component.

a) Constraint Description: To perform a certain function,
a depender component expects that a related dependee com-
ponent is in a state in which the dependee component is ready
to provide a service required by the dependee component. In the case of the occurrence of an error that is an unexpected state of the dependee component, the error has to be identified and then recovery actions have to be planned and executed. Although these tasks which are identification, planning and execution relate to the correct service delivery of the dependee component, knowledge about such tasks should not be included in the dependee component’s operation knowledge. The reason is the adverse effect on reusability since the knowledge about the above-mentioned tasks originates with the dependee component.

Moreover, direct indications of dependee components are not preferable from the perspective of reusability. A location in which a dependee component is deployed differs depending on the system. For some dependee components, a dependee component may be selectable from some alternatives. For example, there are various implementations of HTTP servers and DBMSs.

The constraint description is expressed as follows.

\[ (F_{src}, L_{dst}, C_{dst}, S_{dst}) \]

where

\[ F_{src} : \] A function of the dependee component which depends on the other components’ function.

\[ L_{dst} : \] Information which relates to a location where the dependee component is deployed. This information is required to determine a host where a dependee component is deployed. This information should not be system specific (e.g., a reference to a configuration file’s parameter).

\[ C_{dst} : \] A dependee component which has a function used by the dependee component. This information should not be system specific. For example, if a dependee component is selectable from some alternatives by a parameter, \( C_{dst} \) should be a reference to the parameter.

\[ S_{dst} : \] A dependee component’s states expected by the dependee component

b) Operation Description: A component which is in an error state has to be operated to trigger transitions to a preferred state. Performance of such operations involves certain tasks which are 1) identifying a target component’s state, 2) planning needed operations and 3) executing the planned operations.

Operation description expresses information to automate the above-mentioned tasks. Firstly, we mention information for identifying a component’s state. To identify a component’s state, attribute information which characterizes the state (e.g., state of a daemon process) is needed. Besides, relations between attributes’ values and a component’s state are also necessary.

Next, we explain information for planning and executing operations. To determine operations which trigger transitions from a current state to a preferred state, relations between the operations and its pre / post states are necessary.

The operation description is summarized as follows.

\[ (Attr, State, Trans) \]

- \( Attr \): is the attribute information which is a set of tuples,

\( State : Exp \): is the state information which is a set of equations, attribute name=value in the state

\( Trans \): is the transition information which is a set of tuples, (operation, pre-state, post-state)

2) Injecting Dependency between Operation Knowledge: The constraint description presented in Section III-A.1 does not contain ways to satisfy a constraint. To satisfy a constraint, firstly a related dependee component has to be specified. For this specification, not only constraint description but also system-specific information is required. Next, a state of the specified dependee component has to be recognized and, if necessary, the dependee component has to be operated to be in a preferred state. Therefore, a mechanism which deals with the above-mentioned tasks is necessary.

The mechanism introduced is dependency injection. The dependency injection mechanism transforms dependency expression from a system-independent constraint description to a system-specific Constraint Satisfaction Request (CSR). Based on a CSR and operation description, a state of a dependee component is controlled.

The steps of the dependency injection are as follows.

1) A constraint description is transformed to a CSR with system-specific information (e.g., configuration file’s context, directory server, etc)
2) A dependee component sends the CSR to a host specified in the CSR.
3) The host receiving the CSR refers to operation description of a dependee component specified by the CSR.
4) With the operation knowledge, the host determines a dependee component’s state, plans needed operations to satisfy the CSR and executes planned operations.

In the fourth step, the host or the dependee component can send another CSR, if necessary. This mechanism enables propagation of a constraint to a root error cause even if it exists in a dependee component which is not known by the original dependee component.

Furthermore, since operation knowledge is referred to only within each host, component-specific information is hidden from external dependee components.

B. System Architecture

Figure 1 presents the architecture of the self-healing systems based on the concepts described in Section III-A.

Constraint description is packed with a dependee component. The constraint description is deployed to a host on installing the dependee component. The host in which the dependee component is installed has a CSR sender. A CSR sender is used to send a CSR to a host where a dependee component is deployed.

Similarly, operation description is packed with a dependee component and is deployed on installing the dependee component. A host in which the dependee component is deployed has a Components Manager and configuration information.
The role of the Components Manager is to satisfy constraint received from CSR senders. More concretely, a Components Manager determines a state of a component, plans needed operations and executes the planned operations. Configuration information is referred to by a Components Manager to decide which operation description should be referred to. Mappings from a component’s name to operation description are stored in the configuration information.

A conceptual model of self-healing behavior in the architecture is presented in Figure 2. In Figure 2, a CSR sender and a Components Manager exist at the boundary between a system-specific area and a component-specific area. They collaborate to mediate between depender component’s constraint description and dependee component’s operation description which are not related in a component-specific area. A constraint description is transformed to a CSR by a CSR sender. System-specific information is referred to in order to achieve this transformation. The CSR is received by a Components Manager and then the Components Manager acts to satisfy the constraint with operation description.

IV. IMPLEMENTING THE PROPOSED TECHNIQUE

This section explains a prototype implementation together with an application example. Figure 3 presents an example system used in this section. This system which consists of three hosts provides a service by which end-users can search patent data stored in a DB (MySQL\textsuperscript{TM}[4]) deployed to the server C. A servlet and an EJB\textsuperscript{TM} component are installed in the server A and the server B, respectively. The EJB in the server B searches patent data from the DB on the server C. The servlet is a client of the EJB on the server B. JBoss\textsuperscript{TM}[5] and Apache Tomcat [6] are used for an EJB container and a servlet container, respectively.

A. Describing Operation Description

The state chart diagram shown in Figure 4 represents operation description of MySQL. A rounded rectangle illustrates attribute information. In Figure 4, attribute daemon is defined as a boolean value which is true if mysqld daemon is active. This value is evaluated by an existence of a process which has process id specified by a file “/var/run/mysqld/mysqld.pid”. Rectangles attached to states represent state information. For instance, in the case that daemon attribute is true, the state of MySQL is identified with in an active state. Texts attached to the transitions represent commands which trigger related transitions.

In the implemented prototype, operation description is written in XML so that a Components Manager can understand it. Additionally, some class libraries to evaluate attribute value are provided. Based on operation knowledge written in XML, a
Components Manager gets attribute value with provided class libraries, plans needed commands and executes commands.

**B. Weaving a Self-Healing Aspect for Constraint Description**

Self-healing is not functionality originally required to components. Maintenance of self-healing functionality should be separated from maintenance of original components. Components in which self-healing functionality is embedded can be complex, and therefore, maintaining components (e.g. bug fix) can become difficult.

In the implemented prototype, constraint description is weaved into depender components as an aspect[2]. We call the aspect which expresses constraint description self-healing aspect. Weaving a self-healing aspect enables adding of self-healing capability without any modification of the original component. Furthermore, a self-healing aspect has a role of a trigger to self-healing behavior.

Figure 5 presents how to generate a self-healing aspect from constraint description and a configuration file. In Figure 5, the self-healing aspect is written in AspectJ [7].

The patent search EJB component (implemented PatentSearchBean class) in Figure 3 requires that a MySQL daemon on the server C works correctly. More specifically, the searchPatent method of the PatentSearchBean class accesses the MySQL daemon on the server C. Therefore, the patent search EJB component has following constraint description.

\[ F_{src} : \text{searchPatent method} \]

\[ L_{dst} : \text{a reference to a configuration file’s parameter which specifies an address of database} \]

\[ C_{dst} : \text{MySQL} \]

\[ S_{dst} : \text{active} \]

In above constraint description, \( L_{dst} \) does not specify concrete location. To achieve self-healing behavior, a concrete value should be substituted for the reference. In this patent search application example, an address specified in a configuration file should be substituted for the reference to the configuration file’s parameter.

Generating a self-healing aspect is a routine work which is comprised of following tasks.

1) Specify \( F_{src} \) as a pointcut (line 2-5 in Figure 5).

2) Find concrete values for \( L_{dst} \) and \( C_{dst} \) from system-specific information.

3) Set the concrete values for \( L_{dst} \) and \( C_{dst} \) to arguments of a CSR sender’s method (line 7-11 in Figure 5).

In these tasks, no logical code is implemented but only parameters are set to the self-healing aspect. The parameters to be specified are a pointcut and arguments of the requestState method of the StateRequester class where the StateRequester class is an implementation of a CSR sender.

Although Figure 5 illustrated the case that the substitution is statically performed, a dynamic substitution for \( L_{dst} \) and/or \( C_{dst} \) can be realized by implementing a self-healing aspect that parses a configuration file and retrieves required parameter value.

Description of a self-healing aspect is a component specific matter. Therefore, the approach that is weaving a self-healing aspect for constraint description can adapt to a variety of configuration files.

**C. Self-Healing Behavior**

Figure 6 illustrates self-healing behavior in the prototype. Assuming that the initial state of MySQL on the server C is stopped, the self-healing behavior should activate MySQL on the server C.
The self-healing behavior is triggered before executing the searchPatent method since the self-healing aspect defines a before-advice (line 6 in Figure 5). A CSR is sent to the server C by the aspect shown in Figure 5. Operation description of MySQL is determined by a Components Manager that receives the CSR at the server C. The Components Manager on the server C firstly identifies MySQL’s state with stopped based on the determined operation description. Next, the Components Manager plans to start the MySQL server and executes “/etc/init.d/mysql start” command with the operation knowledge. Finally, the PatentSearchBean a response from the Components Manager through the self-healing aspect and executes an original searchPatent method.

To assure a success of the self-healing behavior, a state expression in constraint description must be identical with one of operation description. However, this identity is required to only correct states. Administrators can set expressions which are used as correct states’ expressions in their system. Identity is not required for other states’ expressions since they are hidden from external components.

A servlet on server A can also send a CSR to start JBoss on the server B (not shown in Figure 6). From the servlet, actions which are necessary to recover from errors in the server B and the server C are not visible (e.g. a CSR from the EJB to the server C is not visible). It means that error causes in dependee components are transparent for depender components.

D. Adapting to the Changes

Suppose a new application and a new server are added to the system shown in Figure 3. Figure 7 shows the system in which the new server and the new application are added. In the proposed self-healing system, only a self-healing aspect (constraint description) is required to expand the scope of self-healing to the newly added server and application. No modification of other operation knowledge is necessary since it does not describe depender components.

Next, we explain how the proposed self-healing system adapts to the modification of operation description. A socket file is created in a directory specified by a configuration file while MySQL server is activated. The directory (assuming /var/run/mysql) has to be writable for user mysql otherwise MySQL server cannot start since the socket file cannot be created. The state in which an owner of the directory is not mysql is one of the error states. When an owner of the directory is changed by, for instance, the administrator’s mistake, the directory owner should be set to mysql to start MySQL server.

Figure 8 shows operation description which is modified to treat the above-mentioned error. A new attribute, state and transition are defined in Figure 8 to identify the error state and to recover from the error. This modification does not affect self-healing behavior of other components since dependee components are unaware of dependee components’ state from the beginning. Although the above-mentioned error may rarely occur, it can bother administrators since the rarity of error is linked to the rarity of recovery actions.

This modified operation description can be reused in other environments since any system-specific information is not contained. Reusing the modified operation description, a person who does not know or forecast the above-mentioned owner error can construct self-healing systems which can recover from owner error.

V. DISCUSSION

A. Future Work

Although this paper has illustrated a simple example in which the proposed technique has been applied, a number of future works are remaining to be considered.

The proposed technique focuses on component-level errors, therefore self-healing systems based on the proposed technique
cannot handle host-level failure (e.g. halted hosts). To cope with host-level failure, an integration with other technologies (e.g. HA clusters) are necessary. Such an integration is one of the future works.

Another future work is about issues which should be formulated through larger and more complicated case studies. Scalability issue is a kind of this issue. The structure and components shown in Section IV is too simple to discuss the scalability issues. Furthermore, it may be possible that problems related to dead-locks and loops persist when the proposed technique is applied to larger and more complicated systems. We have to examine and formulate problems through case-studies in which a lot of complicated components are utilized.

B. Changing Self-Healing Behavior

The self-healing aspect shown in Figure 5 uses a before advice. Requesting constraint satisfaction before executing an original function realizes proactive self-healing, which can improve reliability. However, the proactive healing has some problems. The first problem is synchronization with the completion of the constraint satisfaction. To get the benefit of the proactive self-healing, a Components Manager has to monitor whether triggered transitions have completed and a self-healing aspect has to await a reply from the Components Manager. The next problem is a performance issue since a self-healing aspect with before advice sends a CSR even if constraint is already satisfied.

Using other advice types resolves the above-mentioned problems although reliability is not achieved. For example, using a handler advice in AspectJ which is called when a specified exception is thrown realizes sending of a CSR only if errors are detected.

Advantages and disadvantages differ depending on advice types. Therefore, it is important to select an advice type appropriate for the situation. In dynamically changing situations, it seems difficult to decide an appropriate advice type. The dynamic aspect oriented programming [8], [9] helps to improve the proposed technique in dynamically changing situations.

C. Localizing Unanticipated Problems

Autonomous recovery from unanticipated problems which are not solved by the deployed knowledge is impossible for the proposed self-healing system. However, the proposed technique can support manual recovery by localizing problems.

For instance, the system shown in Figure 3 with the operation description presented in Figure 4 can recover from only stop state. The system cannot recover from the directory owner error which is resolved by the operation description shown in Figure 8. However, the system can identify which CSR is not satisfied. In the above example, a CSR to the server C is not satisfied, which indicates that some problem exists in starting MySQL server on the server C.

VI. RELATED WORKS

The AC Toolkit [10] provides self-healing capability based on a standardized event format. In self-healing systems based on the AC Toolkit, the AME (Autonomic Management Engine) monitors pre-specified resources and performs recovery actions according to occurred events. An uncatchable event or one-to-many relations between an event and errors can cause unrecoverable problems due to the event-based approach. However, in the proposed system, error causes are transparent, since the error is determined with operation description and constraint propagation.

Microreboot architecture and crash-only software [11][12] improve reliability while also realizing low-cost reboot. To build microrebootable systems, components have to be developed with crash-only properties. In contrast, our approach allows use of components developed without autonomic behavior since the proposed self-healing system can be built by attaching self-healing aspects and operation description to the components externally.

Several techniques for determining problems and constraints are proposed. In [13], effectiveness of call stack matching for finding known software problems is presented. In [14], a technique to discover constraints based on analysis of Windows registry is presented. These works are complementary to the proposed technique since we do not mention how the self-healing aspect is described.

Several researchers have focused on Dynamic Aspect Oriented Programming with the aim of making autonomic systems adaptive [15][16]. Whereas their aim is dynamic adaptation in changing situations, our aim is to support task concerning operation knowledge for self-healing systems.

VII. CONCLUSION

This paper has proposed a technique for developing self-healing systems with reusable component-level operation knowledge. The proposed technique makes operation knowledge independent of a specific system, and therefore, operation knowledge becomes reusable across organizations and becomes adaptive to changes.

To achieve reusability, system specific information is pruned off from component-level operation knowledge and it is encapsulated in components. Dependency information is expressed as component specific constraint description. To relate constraint description and needed recovery actions, dependency injection mechanism is introduced. Constraint description is transformed to a system specific Constraint Satisfaction Request (CSR), then the CSR is interpreted to determine needed recovery actions.

In self-healing systems based on the proposed techniques, unanticipated problems are localized since constraint propagates in a direction opposite to error propagation. This problem localization will help complementary manual operation.

Furthermore, an implementation approach with Aspect Oriented Programming (AOP) has been presented. AOP enables adding of self-healing capability without any modification of original components. To realize self-healing systems, the proposed technique does not require any component-embedded self-healing functionality, but requires adding operation knowledge (e.g. a self-healing aspect) externally.
Although some benefit of self-healing systems based on component-level operation knowledge has been illustrated, the simple example presented in this paper is not enough to clarify applicability and scalability for larger systems and more complex component (e.g., middle-ware such as a J2EE container). We are planning further case studies for evaluation and improvement of our prototype. Support for describing operation knowledge when errors or dependencies are newly detected is another subject for future work.

REFERENCES