Interactive Self-Healing for Black-Box Components in Distributed Embedded Environments

Michael Dinkel[1], Slawi Stesny[1], and Uwe Baumgarten[2]

[1] BMW Group Forschung und Technik, Hanauerstr. 46, 80992 München
[2] Institut für Informatik, Technische Universität München, Germany

Abstract. As self-management features of computer systems depend on self-knowledge we have to cope with the issue that most large systems are not entirely known to the self-management software. Especially when system parts are supplied by external companies which want to preserve their intellectual property we have to cope with black-box components. This paper presents a novel approach for self-healing in distributed embedded systems containing black-box application software. The interactive self-healing process is based on well defined system knowledge and enables the construction of a functional dependency graph which serves as basis for rule-based root cause analysis and self-healing.

1 Introduction

Intelligent devices pervade more and more our everyday life. But new functionalities and intelligent behavior also cause overhead for the management of these systems with their hardware and software parts. Especially large systems with versatile functionalities like vehicles, aircrafts or trains consist of highly complex software systems executed by multiple, often heterogeneously interconnected hardware nodes. The management of these systems has become one of the major tasks, as life-cycles are up to several decades for the whole product, while all of its parts, especially software, have distinctively shorter life-cycles. The intelligent management of faults in such complex systems has gained a very high value for the manufacturers who would definitely benefit from every step research gets nearer to autonomous self-healing of IT systems.

In the majority of cases the software parts of such distributed and embedded systems are implemented by supplier companies which do not want to reveal their application-specific intellectual property. As a result, the OEMs[1] who integrate the different parts into the final product have to deal with software-hardware bundles which exhibit black-box properties. The internals of these modules are unknown to the integrator and only a specified interface is known. As hiding of implementation knowledge is essential for supplier companies, a self-healing approach has to consider these special conditions.

[1] OEM - Original Equipment Manufacturer

In this paper we present a novel architecture for self-healing systems in distributed embedded environments, which regards the black-box properties of application components. We underpin our concepts with examples from the automotive domain. The paper is structured as follows. Section two presents our target scenario and introduces the self-management architecture with the self-knowledge that provides the foundation for the interactive self-healing process. While the third section describes our approach of failure classification, we address the topic of interactive self-healing with it's different stages in section four. We briefly present our prototype in section five and point out related work in six, whereas section seven concludes and outlines future work.

2 Target Scenario

As not only functionality but also cost is a driving force in automotive engineering, current vehicle systems have been recognized to be much too error-prone and complex due to their heterogeneity. Hence effort is directed towards a reduction of the number of different nodes and networks in the same system. Instead of ECUs[2] in their current, very specialized design there will be two different types of nodes in future vehicle systems. On one side there will be light-weight sensors and actuators that might even be combined with mechanical units like electronic dampers. On the other side there will be multiple nodes with higher capacity and computing resources. These nodes are no longer used exclusively for one specific purpose. Instead, they become platforms able to execute different applications concurrently.

In our target environment application software is separated from hardware by an abstraction layer. A small set of powerful *platforms* executes *software components* which encapsulate intellectual property and hence have black-box properties. In short we use the term *components* in the following. The platforms are nodes which execute infrastructure software and are connected via a broadband network. Platforms may provide different *capabilities* to the software components running on them. This includes different computing resources and in particular the type and number of special dedicated devices. Especially these dedicated devices like sensors and actuators are characteristic for embedded, vehicle control systems. *Applications* are defined according to [5] as a sets of communicating components which provide coherent, user-perceivable functionality. Software components can be installed, removed or updated separately or as a part of an application.

The aim of the *interactive self-healing* is to provide the best achievable functionality with a system-wide view in the face of faults and failures in both hardware and software.

[2] ECU - Electronic Control Unit

2.1 Self-management Architecture

From an abstract point of view embedded control systems consist of platforms which execute software components. The management of such systems can be done either decentralized as an emergent effect from autonomous components as done in [10] or centralized with one manager that controls the whole system. While a decentralized system definitely has the benefit of avoiding single points of failure by principle, it causes all components to be intelligent and hence costly as they all have to include management functionality. Another difficulty with the decentralized approach is that the outcome of distributed decision making is hardly predictable. So for our approach we choose a centralized self-healing architecture with "light-weight" black-box components which is based on a central *System Manager* and so-called *Platform Managers* which carry out instructions and act as a local representatives, see Figure 1. The system manager provides functionality for typical live-cycle operations like installing, starting, stopping or removing components and applications. Additionally the system manager provides access to a component repository where all component archives are stored for later installation and to the knowledge base which stores the systems self-knowledge, see Section 2.3.

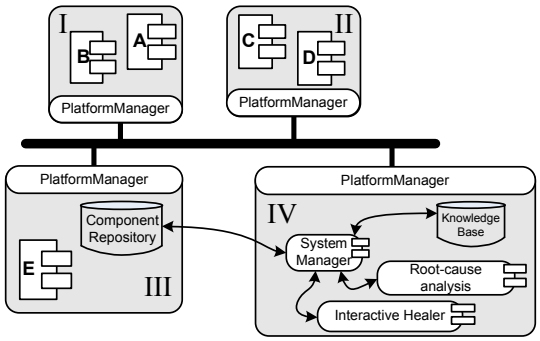

Fig. 1. Architecture for centralized self-healing

Application components are labeled A-E in Figure 1. As explained before we have to cope with black-box properties of application software. So the insides of components are unknown, but we prescribe a *management interface* which has to be implemented by every component. The structure and details about the communication system are of lesser importance in the context of this paper, but can be found in [6]. A publish/subscribe middleware provides for loose component coupling and abstraction from the underlying network technology.

2.2 Requirements and Capabilities

In embedded control systems generally all installed applications are executed concurrently. Hence it is very important to not hamper already running applications from doing their job when installing new ones. In order to be able to judge such situations at runtime, we follow the path of describing both applications and platforms. Applications are composed of software components. Each software component is in need of certain 'things' in order to fulfill the task it has been programmed for. This includes functional requirements (FRs) and non-functional requirements (NFRs) from a software component point of view [5]. These component requirements are also called constraints or dependencies. The only dependencies between components are given by functional requirements.

Functional Requirements in this context, describe a components communication relations with other components. They are expressed as *Ports* of components. Ports are connected to channels of a publish/subscribe messaging system [6] and have a defined direction which makes them either InPorts for receiving messages or OutPorts for sending messages to the connected channel. The publish/subscribe paradigm with its channels provides for indirection and location transparency of communication [7].

Non-functional Requirements in this context, describe all further entities and conditions that are necessary to enable a software component to meet its service goals. This includes the required resources like processor architecture, the amount of free and persistent memory, but also what kinds of sensor and actuator devices are needed. Also other requirements with non-functional nature are described as NFRs, like the need for encrypted communication or the type and version of infrastructural software needed as prerequisite on the executing platform.

Capabilities provide the possibility to describe the features offered by platforms. Capabilities are the counterpart of non-functional requirements. So the question of whether or not a software component can be installed on a certain platform can be answered by matching the NFRs of the component against the capabilities of the platform in question as described in detail in [5].

2.3 Self-knowledge

Self-knowledge is the basis for self-management in any way. Only with a thorough understanding of what a system looks like, a management software can come to reasonable decisions. Therefore the system manager employs the concepts of functional requirements, non-functional requirements, as well as capabilities in order to build up a model of the system. All the entities in the system like software components and platforms are required to contain their own descriptions and have to make them available to the knowledge base. The knowledge base represents and stores the systems static *self-knowledge*. It collects the information about applications and components on their installation and removes the descriptions after uninstallation. Analogously the knowledge base gathers the information about the available platforms at the time they are first attached

to the rest of the system and removes their descriptions when platforms are no longer available. This behavior enables the evolution of the complete system including dynamic change in the system's self-knowledge. The interactive healing process presented in Section 4 we will mainly focus on the functional dependencies between software components.

3 Failure classification

Self-healing systems should be able to cope with unforeseen problems. In order to come near the goal of system completeness [12] we defined a classification of different failures which governs both the localization and the healing of occurring failures. The basis for our fault model have been foundation papers like [3] but also domain-specific information like error data-bases for typical automotive errors.

A *failure* is defined as a "deviation of the actual output from the required output by more than specified tolerances" according to [15]. Failures are caused by *faults* or *defects*; failures may *propagate* and cause additional failures of a different kind. A *symptom* is a failure which has been recognized by an observer; so everything which is true for a failure is also valid for a symptom. In the following the term symptom is used to emphasize that the failure has been recognized.

We have defined a set of failure classes $fc \in FC$ where each failure class is defined as a triple $fc := (O, PF, R)$ with its possible occurrence situations O and a set of propagated failure classes $PF = \{fc_1 ... fc_p\}$. R represents a boolean value which indicates whether fc is a root failure class or not. Non-root failure classes are called propagated failure classes which only occur due to propagation. The function $isRoot(x) : FC \longmapsto \{0, 1\}$ returns 1 for a root failure class and 0 for a propagated class. The function $pf(x) : FC \longmapsto \mathcal{P}(FC)$ returns the set of propagated failure classes for a given failure class with $\forall c \in PF \mid isRoot(c) = 0$. There is only one failure per component at the same time and we are working on failure classes to be open for unforeseen occurrences. So what is interesting about a symptom is its class and the component where it was detected.

From a total of 15 different failure classes we will present only two: the timing failure as an example for a propagated failure class and the device failure as an example for a root failure class.

Timing Failure A timing failure is featured by late arrival of at least one incoming message. The occurrence O_T is hence restricted to software components which provide InPorts and depend on inputs from other components, see Section 4.2. A timing failure may propagate along its OutPorts and may cause further failures (PF_T) of the classes *timing*, *omission* and *operability*. Timing failure is a non-root failure class, $R_T = 0$.

Device Failure A device failure is given if an I/O device like a sensor or actuator does not work properly. The occurrence O_D is restricted to components which make direct use of devices. Device failures are root failures $R_D = 1$ and

may propagate (PF_D) and manifest in the failure classes *timing*, *omission*, *response* and *operability*.

4 Interactive Self-Healing

As there are many different factors that influence the procedure of self-healing, especially the root cause analysis becomes very difficult. From the many possible approaches as presented in [17] we chose a rule-based approach. *Interactive self-healing* means that we do not strictly separate the phases the self-healing control cycle, instead there may be healing-actions mixed up with the root cause locating process. In this vein our approach mimics a human expert who would change the system and would plan is next steps depending on the results of the first.

4.1 Health Monitoring

The first step in a self-healing control-loop like in [11] is the monitoring of the system at runtime. In our architecture we gather information for the self-healing process at three different layers from which we will focus on application specific monitoring in this paper.

Network monitoring keeps track of the platforms, networks and the state of connections between them. It provides information to the self-healing system by sending alarm messages in case of malfunctions.

Platform monitoring is responsible for monitoring the availability of resources on each platform and sends the information to the healing system.

Application specific monitoring How to monitor software components if you don't know much about them (black-boxes)? Our answer is the following: Application specific knowledge is needed to recognize application specific failures. Hence, application components have to include specific *observers* as this knowledge is not available at infrastructure level, see Figure 2. Components are able to detect their own failures and can additionally use their communication relations to mutually monitor themselves and report alarms to the management system. Symptoms of a timing failure for example can be easily recognized by the receivers of late messages, additionally receivers can perform plausibility checks on the contents of messages and can detect the loss of cyclic messages. In this sense every component of the system becomes a monitor for different kinds of anomalies, both self-made and caused by others.

Figure 2 depicts the concept of mutual component monitoring while a centralized system healer is used to analyze the monitoring data and perform the planning and execution steps of our self-healing control loop.

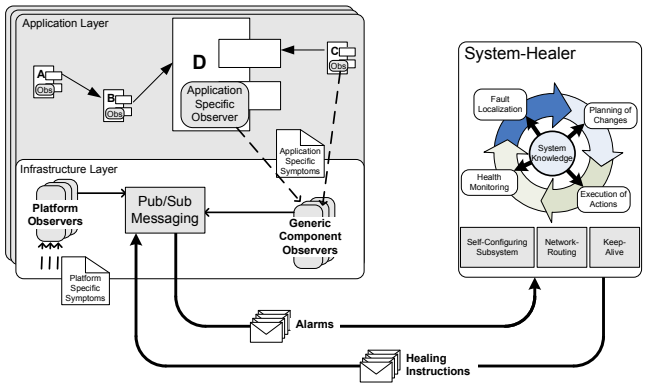

Fig. 2. Monitoring and healing concept

4.2 Functional Dependency Graph

In order to be generic, all rules of the rule-base do not depend on a specific deployment of application components, but are based on a *functional dependency graph* which represents the communication relations between software components because these links are the main path of failure propagation. The graph is extracted from the self-knowledge of the system.

There are two different types of nodes and arcs in a functional dependency graph $FDG := (SC, CC, P_i, P_o)$. There is a set of software components SC and a set of communication channels CC. Two sets of arcs connect components and channels in a directed way, representing the components ports. (1) InPorts: $\forall p = (x, y) \in P_i | x \in CC \wedge y \in SC$ and (2) OutPorts: $\forall p = (x, y) \in P_o | x \in SC \wedge y \in CC$. This results in a directed graph where the successor of each component is a channel and the successor of each channel is a component again, see Figure 3 for an example. The function $dep(x) : SC \longmapsto \mathcal{P}(SC)$ returns for each component the set of components on which it functionally depends, this means for components $x, y \in SC$ that y depends on x if $\exists a = (x, c_j) \in P_o \wedge \exists b = (c_i, y) \in P_i$ with $i = j$.

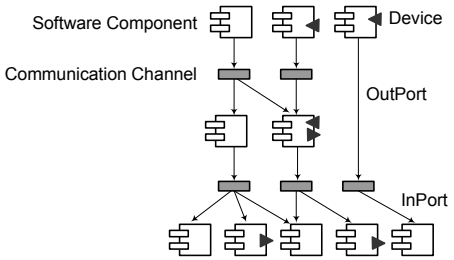

Fig. 3. Functional Dependency Graph

4.3 Propagated Symptom Elimination

As a fault in an intermeshed system is likely to cause multiple failures and symptoms which in turn may cause further symptoms and so on, we try to avoid overhead by removing unnecessary symptoms. One hint is given by the the $isRoot(x)$ function of symptoms, another hint is given by the failure propagation defined for each failure class which can be used for sorting out symptoms due to propagation. The propagation can only occur along the edges of the FDG. For a set of symptoms S we define the *distance* function $d(x, y) : S \times S \longmapsto \mathbb{N}$ which returns the distance for the two symptoms x and y as the number of channels between the two components where the symptoms occurred.

Consider a set of symptoms S_{curr} which collects the symptoms in the system manager for a certain timespan. We have to check for every newly recognized symptom if it is propagated from a different symptom which is already in S_{curr} or whether it could be the cause for symptoms in S_{curr} due to propagation. For every symptom $x \in S_{curr}$ the *explanation* function $ex(x) : S_{curr} \longmapsto \{0, 1\}$ indicates whether it could be *explained* as propagation of another one symptom. A symptom is an essential input for the healing process, if it cannot be explained by other symptoms. The function $ex(x)$ is defined as follows for a given *propagation distance* $pd \in \mathbb{N}$, which is the maximum number of component levels where a symptom of a functional dependent component is considered to be a possible propagation.

$$ex(x) = \begin{cases} 1, \exists\, s \in S_{curr} : pd > d(x, s) \wedge x \in pf(s) \\ 0, isRoot(x) = 1 \\ 0, \neg\exists\, s \in S_{curr} : d > pd(x, s) \\ 0, else \end{cases} \tag{1}$$

The function 1 defines when a symptom is not needed to be treated by the self-healing process. These symptoms however are not removed from the set S_{curr} as function 1 may not be evaluated properly for further symptoms if the important knowledge of propagated symptoms is lost. By the means of the propagation distance we are able to deal with cycles in a functional dependency graph.

4.4 Planning and Executing Actions

The result of the propagated symptom elimination phase is a set of significant symptoms, which becomes the input for a rule-based planning and healing process. The rule-based approach has been chosen in order to provide a flexible and easily tunable system. The rules for our system have been defined based on the functional dependency graph, on the global system knowledge and on our failure classification. Hence even specific system configurations can be treated in a generic way.

The rules may cause tests like component self-tests, platform tests or network tests, which in turn may cause further rules to be activated. Additionally, rules may initiate reinitialization of devices, migrate or update components and

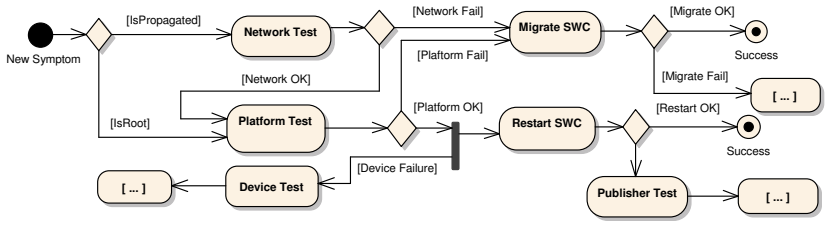

Fig. 4. Snipped of the Activity Diagram for Self-Healing

remove complete applications in case of severe problems which results in several escalation steps for healing a fault. The overall goal of the rules is to find an instance of one of the root cause failure classes and heal it. A failure is considered to be healed if the dependent components can work on without encountering new symptoms.

To get an idea of what a typical healing process looks like, we depict it's beginning in an activity diagram in Figure 4. The first distinction is to be made depending on whether the current symptom is element of a root failure class or not. Generally we first test the network and the symptomatic component's platform in order to avoid unnecessary reasoning. Depending on the outcome of a *Platform Test* we decide whether the affected component has to be restarted. If a component restart does not heal the failure we use the functional dependency graph in order to perform component self-tests on the publishers of the symptomatic component (*Publisher Test*). This may result in new alarm messages with new symptoms which are again treated by the rule engine. Rules for specialized tests and healing actions are defined according to the different failure classes as can be seen in the *Device Test* in Figure 4.

5 Prototype and Evaluation

For the evaluation we have implemented a prototype system based on four different nodes connected via two different Ethernets and a CAN[3] network. The nodes run Linux and a Sun JVM 1.5 which executes a Knopflerfish OSGi framework. All functionality has been implemented as OSGi bundles and the platform manager (see Figure 1) enables the life-cycle management of application software, while a publish/subscribe middleware enables location transparent communication. The rule-base self-healing process has been implemented by the means of the Jess rule engine[4].

The web ontology language (OWL) [1] has been chosen as a suitable format for encoding the the system knowledge with its descriptions of platforms and components. From the three sub-languages of OWL we used OWL DL (Description Logic) since it supports the maximum expressiveness while retaining

[3] CAN - Controller Area Network
[4] Jess - http://www.jessrules.com/jess/index.shtml

computational completeness [13]. OWL is based on the Resource Description Framework (RDF) which provides XML-formated data representation. For the knowledge representation, we adopted the N-triple notation of RDF, details on the description language can be found in [9].

So far we could validate our approach with two example scenarios where we could successfully locate and treat timing and device failures in our prototype.

6 Related Work

Model-based adaptation as in [8] builds an architectural model and uses descriptions of architectural styles to recognize violations of constraints which have to cause repair operations. Like the architectural self-healing in [8] and [14] we also use an *external* healing process with external knowledge. We include however internal aspects with the use of application specific observers. Self-management in enterprise networks and the Internet is addressed in [4] with the help of local rule-based expert systems which have been identified to be well suited to automate management tasks as they are extensible, understandable, reusable and interoperable. The ABLE tool-kit provides rule-based planning and execution support for domain independent autonomic computing [16]. It is strongly coupled with Java and defines a specific rule language which allows to specify different kinds of rules.

The escalation steps of many of our healing actions are designed in the style of "recursive restartability" as described in [2] which also tries to fix a problem locally and only extend to a broader context if the local restart did not succeed. However most of these approaches do not explicitly address the problem of self-healing in the face of black-box components.

7 Conclusions and future work

In this paper we have presented a novel approach for dealing with black-box software components in a self-healing system. The basis for the self-healing process is given by the systems self-knowledge from which we construct the functional dependency graph. The DFG provides a suitable basis for symptom elimination and rule-based reasoning for the actual interactive root cause analysis. Rules are based on generic assets and on expert knowledge.

However testing with scenarios does not allow general statements about the correctness and quality of our rule-base. As most of the rules rely on the cooperation of software components it is difficult to tell by scenarios how the system would behave in the face of false positives and other incorrect answers. Therefore we are currently implementing a simulation environment which mimics the behavior of a complete system, including multiple symptoms and the possibility of incorrect component self-testing.

References

1. S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language - reference. Technical report, World Wide Web Consortium, Februar 2004.
2. G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.
3. F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, May 1993.
4. W. de Bruijn, H. Bos, and H. Bal. Robust Distributed Systems Achieving Self-Management through Inference. In *WOWMOM '05: Proceedings of the First International IEEE WoWMoM Workshop on Autonomic Communications and Computing (ACC'05)*, pages 542–546, Washington, DC, USA, 2005. IEEE Computer Society.
5. M. Dinkel and U. Baumgarten. Modeling nonfunctional Requirements: a Basis for dynamic Systems Management. In *SEAS '05: Proceedings of the Second International Workshop on Software Engineering for Automotive Systems*, pages 1–8, New York, NY, USA, 2005. ACM Press.
6. M. Dinkel and D. Fengler. Unified Communication in Heterogeneous Automotive Control Systems. In *Proceedings of the 3rd International Workshop on Intelligent Transportation*, pages 21–26, Hamburg, Germany,, March 2006. Hamburg University of Technology.
7. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. In *ACM Computing Surveys (CSUR)*, volume 35, pages 114 – 131. ACM Press, New York, NY, USA, 2003.
8. D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.
9. T. Gotovac. Design und Implementierung eines selbstbeschreibenden Komponentenmodells für die Verwaltung von verteilten selbstorganisierenden Komponentensystemen. Master's thesis, Technische Universität Darmstadt, March 2006.
10. P. E. H. Hofmann and S. F. Leboch. Evolutionaere elektronikarchitektur für kraftfahrzeuge (evolutionary electronic systems for automobiles). *it - Information Technology*, 47(4):212–219, April 2005.
11. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer Magazine*, 36:41–50, January 2003.
12. P. Koopman. Elements of the self-healing system problem space. In *Workshop on Architecting Dependable Systems (WADS03)*, May 2003.
13. D. L. McGuinness and F. van Harmelen. OWL web ontology language - overview. Technical report, World Wide Web Consortium, February 2004.
14. Y. Qun, Y. Xian-Chun, and X. Man-Wu. A framework for dynamic software architecture-based self-healing. *SIGSOFT Softw. Eng. Notes*, 30(4):1–4, July 2005.
15. F. Saglietti. *Design and Assessment of Fault-Tolerant Software.* PhD thesis, Technische Universität München, Mai 1996.
16. B. Srivastava, J. P. Bigus, and D. A. Schlosnagle. Bringing Planning to Autonomic Applications with ABLE. In *International Conference on Autonomic Computing*, pages 154–161, May 2004.
17. M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming, Special Edition on Topics in System Administration*, 53:165–194, November 2004.