DOI 10.12694/scpe.v17i4.1206
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 4, pp. 351–368. http://www.scpe.org

A SELF-HEALING ARCHITECTURE BASED ON RAINBOW FOR INDUSTRIAL USAGE

ALI FARAHANI*, ESLAM NAZEMI* AND GIACOMO CABRI†

Abstract. Over recent decades computer and software systems become more and more complex because of the applications' and user's requirements. The complexity makes the software systems more vulnerable to the error and bugs. Also, environmental situations affect software systems which do not react to the environmental activities. Self-healing architectures have been proposed in order to make systems defeat these problems and to make systems capable of reacting to the environmental activity. Hence, these architectures help system to become dynamic and more robust, but finding a proper architecture which can support and cover system's requirements is an issue. This is particularly true in industrial environments, which consist of some known and some unknown parameters.

This paper presents an architecture that can be used in some industrial environment to facilitate the process of adapting the system to unpredicted situations. This architecture has been developed over the base of RAINBOW infrastructure and it is compliant to the MAPE control loop (*Autonomic Computing* control loop). The paper reports also about the practical experience of implementing this architecture for a painter robot in an automotive factory, which deals with problems in painted part by itself. The proposed architecture uses rule-based reasoning and it actualizes the method of environmental modeling by using a rule-based system as the model extractor. The results of the implementation shows huge benefits in reusability and even in the quality of painting process.

Key words: Self-healing, RAINBOW infrastructure, Rule-Based

AMS subject classifications. 68M14, 68N19

1. Introduction. Computer systems play an unavoidable role in every field. In this paper, we focus on industrial environments, where they are widely exploited. Industrial environments have a unique property which makes it different from other environments: they have so many parameters and the value of some of them could not even be defined. Dealing with this kind of situations needs a software system that can manage the unpredicted situations. The main goal of the research experience presented in this paper is to propose an architecture and its implementation for industrial applications, which is able to adapt to environmental unpredicted situations by means of the self-healing property.

As mentioned in [1], to face the size and complexity of software systems, design has become more important than algorithms and data structures. In this context, defining an architecture is necessary for developing software systems. Another issue which should be considered is addressing failures [2]. Fault management can be considered and categorized with following reasons and cases:

- The scope of the systems could change during the process and also changes in system's specification will occur because of early system's requirements analyzing.
- Because of the system's environment characteristics, the system could face changes when executing in a non deterministic environment.
- Failures in system's part could occur.

Systems having unknown, incorrect and/or improvable behavior are not used by any developer and user of computer systems, because of their unreliability in covering the users' needs [2].

Starting from these considerations, our aim is to propose an architecture to react to failures in an industrial system. Previous study in fault management considered fault management as gathering and analyzing alerts and failures in a service [3]. According to the explanation in [3], "service" means *tasks* and *functions* presented by an industrial IT system, although this definition can be considered for the usage of any other software systems as well.

The main steps in fault management process are described as below [4]:

- Detect
- Diagnose
- Decide

*Computer Science and Engineering Faculty, Shahid Beheshti University, Tehran, Iran ({a_farahani,nazemi}@sbu.ac.ir)
†Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia Modena, Italy (giacomo.cabri@unimore.it)

- Respond

Beside this viewpoint for remediating the systems from fault state, there are several solutions for fault management and responding to failure in the software systems context. Similar to [3], we are looking for the capability of adapting the system to the environment. For a long time, self-adaptation property's implementations were used in computer systems and it was blended with other system functionalities. In fact, a part of the system is usually responsible for responding to events (e.g. failure) but not separately from the part responsible for the main system functions. As also proposed in software engineering, it is better to have separation of concerns. For supporting this separation, it has to be applied in both system design and implementation. Separation is mentioned as one of the views in [6]. If the change and fault management functionality domain remains at the code level (high coupling), the domain of changes detection and responding to changes will be small and inadequate, introducing another weakness.

The solution and approach that can be conceived to face these problems, also emphasized in [7], is to address the response to changes and failures at the architecture level. In another word, the architecture of the system should be designed for reacting against failures and changes.

In the case of lack of an architecture, the developed system will no longer carry the benefits of an architectural approach to systems development. Some of these benefits are [8]:

- *reusability*: reusing developed modules;
- *easy improvement*: continuous and simple improvement of each section;
- *extendibility*: simple extension capability;
- *changeability*: ability to make simple changes to different parts of the system.

Various architectures have been presented for supporting the reacting to changes. One of these architectures is derived from *Autonomic Computing* (AC), which is described in [9, 10]. Also, architectural solutions for fault-management are used for confronting and reacting against failures. Examples of these researches are mentioned in [11]. The main feature of the architecture in [11] is that it supports responding to changes at the architecture level. Also, *self-healing* is one of the properties and capabilities supported by the architecture presented in [11]. In [11] the task for self-healing is to recover the system from errors and responding to failures. Self-healing is defined as "To discover, diagnose and react to disruptions" in [10]. As it is clear, the viewpoint of this feature is aligned with the *fault-management* approach and both seek to react to events that take the system out of a correct state. By having in mind the researches mentioned in this section, it could be an option to consider self-healing and fault-management as two features that have similar goals and they are trying to reach their goals with different views and paths.

For achieving a system capable of managing and responding to changes and failures, self-healing should be considered as an architectural aspect is systems architecture [5]. In order to implement an autonomic architecture for self-healing a *model of failure* must exist [7]. So there will be a knowledge about states in which failures can happen in the system. Also, there should be the information about which state requires responding and which does not. Due to the differences between architecture and implementation abstraction level, a solution is needed for realizing architecture level views in the implementation. Thus, an architecture should be implemented which supports the AC at the architectural level.

Another research field in AC which can have benefit in fault management is policy-based autonomic systems, which are mentioned as a solution for making systems react to the changes [12]. For implementing a policy-based autonomic system, there is an approach that uses a rule-based viewpoint to make a rule-based engine for systems' reaction task [13]. The rule-based approach is mentioned as a type of policy-based system implementation and also rule-based reasoning as a solution for defining policies [14]. Humans are also reasoning in the way rule-based systems reason. This could facilitate the process of using the self-healing systems (rule-based self-healing systems) for humans (system's user).

The contribution of this paper is to present a *self-healing architecture* in the industrial environment, which takes inspiration from the fault management view and based on Autonomic Computing. In this architecture, the reason of using rule-based approaches is to achieve a more specific architecture definition and also to facilitate the process of human dealing with systems specification and implementation.

There are some general proposed architectures that can be found in self-healing software, self-healing systems and autonomic systems research fields. The proposed architectures have been used to facilitate the adaptation

of a robot's software system to its new task. The architecture presented in this paper does not rely on any specific platforms or implementation's language. This architecture can be used in a wide range of software systems.

Another contribution of this research is that the prosed architecture is integrated for all phases in the software development (from designing a general architecture to its implementation). Namely, it could cover the process from the high-level architecture development step (like AC architecture) to deciding about the implementation solution (like policy-based system) and implementation algorithm (like rule-based implementation).

By presenting the literature and previous work in Section 2, we become familiar with achievements in this field and we can define our architecture. Section 3 introduces a real-life case study that has been used for explaining and also examining the architecture implementation. In Section 4 the proposed architecture is presented. In Section 5 the results of implementation and examination is reported and Section 6 discusses the conclusions and further work.

2. Related Work. The previous researches which are related to architectures of reacting systems in the industrial usage (based on the knowledge from software engineering, Autonomic Computing and fault-tolerant) will be presented in different three categories:

- Autonomic Computing and self-healing systems;
- Fault-management with self-healing viewpoint;
- RAINBOW architecture [1] as a base for proposed architecture.

According to the literature, there are papers including the combination of more than one of these categories which will be discussed in detail in the following.

2.1. Autonomic Computing and Self-healing Systems. In [21], entitled *Towards architecture-based self-healing systems*, the goal is presenting an architecture with a self-healing capability in order to execute `repair system` in running time and without human interference. In order to reach this goal, different concepts and tools are used which are mentioned in the following:

- Elements of this architecture are formed by components and connectors between them; this makes performing fault management changes in the architecture very flexible, because of the loosely coupled connection point feature.
- Connections between these components are established through "independent messages or events" instead of "shared memory between components". Moreover, benefiting from the events will provide the possibility to separate components from each other and eventually it will provide the system with the ability to remove, add or replace the components easily in run time without changing the code.
- In order to obtain a vast range of applications supporting specific types of repairs, domains, or implementation platforms, middleware, and languages, it is necessary to use a general architecture description language (such as xADL 2.0 in this work).

The software architecture description is the basis of the system implementation, in other words, the components are located and loaded (also generating of links and connector) based on implementation knowledge contained in the architecture description. In the situation that a fault happened and there is a need to perform some reactions, there is an engine that takes the description of two xADL 2.0 architecture described (the first is the current architecture, and the second is the architecture proposed as a remediation) as input and tries to extract the difference between them and gives a new architecture description as an output. In another part, this new architecture, which is derived from the differences, will be analyzed. After deciding that is a good solution for the system, it can be executed.

From these explanations, it is obvious that infrastructures that do not support adding or removing components in a software system are unsuitable for this approach. Also, we know that although this policy is not sufficient for all types of repair, it is suitable for many applications, like those with no strict timing constraints. But considering software architecture descriptions as integral parts of the deployed software system described by them is an incomparable aspect of the approach given here.

As it is obvious, discussed research in [21] is carried out at the architecture level and it tries to support healing by changing the architecture (without interfering the change ability inside the system implementation). Also, the detail level remains at the architecture level. As mentioned, this solution is not applicable in a wide range of systems.

Being the change in management part outside the system implementation (except for changes in relationships and architectural order), the ability to change the implementation will be taken away from the system. This point questions the self-healing capability.

In [22], easing the self-healing in software based on software control principle is discussed. It focuses on remediation of failures in software based on reacting to the failure with the help of non-internal component of the system. It prepares Final State Machine for software and inserts remediation states into the software execution flow. This research adds some states in order to control the failure in the system. This research does not interfere in the main system architecture and just tries to remediate failure with adding some remediation state. By the way, interfering the architecture could improve the result of remediation but it has more difficulties.

Another research brings architectural patterns to support self-adaptation in architectural level (SimSOTA) [23]. **SimSOTA** is an integrated Eclipse plugin that brings self-adaptation into the system based on a feedback loop. It uses model-driven viewpoint and implemented by a case study in the cooperative electric vehicle. It does not deal with software architecture in requirement analysis or design phase. Need for a reference architecture for easier implementation of the self-adaptive system in industrial usage is not covered by this research.

2.2. Fault-management with Self-healing. As mentioned in [16], autonomic fault management can be done with the usage of IBM's reference architecture for AC. This architecture is mentioned in the previous section. This article tries to show this that in order to create an automatic fault management system it is necessary that the knowledge capable of reasoning exists in all steps from detection to response to the fault.

This research in [16] provides an engineering process in service level for fault management. But this technique only applies to the service oriented systems. Also, there is no detail on the system architecture and how components are placed in the system and the internal system architecture in [16]. In another research [20], an investigation is done for solutions of self-healing in software systems. Here it is indicated that one of the solutions in this field is obtaining knowledge from the environment based on the model. Meaning that, the method presented in this paper for identifying and understanding the environment is carried out by creating and processing the model using the environment and its conditions. But in this research details and specifications about the system architecture and also the implementation and the solution to fulfill it, are not given. As it will be mentioned in future papers of this research, a possible alternative could be investigating first for more intelligent repair policy mechanisms. It means what the structure basics and building blocks of the self-healing capability implementation are.

Also in a fault management viewpoint, a structure for fault-management systems is presented in a mentioned architecture and commercial used in [20]. This structure is used for fault management systems in network management systems. It includes a few elements provided generally in all systems; `Modeling event/alerts`, `Parsing`, `Correlation`, `Validation rules & Filters`, `Fault DB`.

These sections' function provides input to enter the `parse` procedure. In the `parse` and its internals information is extracted (for example information are placed in a structure like XML and need parsing for recovery). Now this information is correlated and handed over to `Modeling event/alert` in order to draw the current system condition model. This model is checked considering all rules and filters and its required decision is extracted and the decision is executed by `Action`. This isolation that which models require a response and which do not be stored in fault database (DB). Information about conditions which must be assumed as a fault by the system is stored in this database.

The architecture in [20] was presented with enough information about the details of implementation of each components and internal modules. However, it does not mention how the knowledge is maintained and checked the models based on knowledge. Also, this architecture is designed for a specific area of systems (network management). This architecture requires implementing conditions and environment policies.

2.3. RAINBOW Architecture. In [15], RAINBOW as a framework/infrastructure is presented including a structure capable of being reused and also solutions for applying it. RAINBOW infrastructure gives the executor the capability of explaining necessary motivations and performing the suitable instructions. On the other hand, this presented structure in [15] generates the ability for reusing solutions and methods (codes and implemented system) which are prepared in the first place.

This generation is formed by two main modules, *adaptation infrastructure* and *system layer*. The system layer indicates a running system (without change capability) to which self-adaptive property must be added.

This task which will be inserted into the system will be done by adaptation infrastructure. RAINBOW framework also includes translation and architecture parts in infrastructure module. The translation module is used for transforming the structure of the information that comes from the environment into the information which can be understood by another part of self-healing architecture. This transformation of information is the reason of RAINBOW framework reusability feature. In architecture module, there are four main elements; Model manager, constraint evaluator, adaptation engine and adaptation executor. The required knowledge for changing the system is known as system specific knowledge; Using this knowledge and also defining mappings, types and properties, rules, strategies and tactics and operators will make the system capable of responding to the case[15]. This information and data are necessary to this system and could be considered as knowledge. In RAINBOW architecture, after detecting information and conditions according to resources by the probe, they are reported to the architecture layer by knowledge and resource discoveries and in between; transforming this information into comprehensible information for the architecture layer is carried out using existing rules in mapping module. Then, information is aggregated by gauges and is transformed to the environment model by the model manager. Next, limitations are checked by the analyzer and if necessary, it will send an adaptation request to the adaptation manager and after explaining the adaptation strategy, adaptation manager will provide the strategy to the change executer. The applied result to the system layer will be translated into the system level by the translator and then it will be sent.

Clearly, in RAINBOW framework few features are emphasized:

- Reusability: having the ability to use some implemented part in future;
- Specified architecture: defining an architecture and roadmap for creating a system;
- Based on a reference architecture: being based on a well-known control loop (MAPE loop);
- General propose: not being domain specific.

But besides these features, there could be different ways to implement the knowledge detection method and also knowledge levels, system details and controlling details and the system implementation. These kinds of information about implementation of RAINBOW architecture/infrastructure have been introduced in [17, 18].

The architecture presented in [17, 18], considers AC and self-adaptation generally, and could be specialized for special-purpose functions and attributes, like self-healing in order to gain more performance and reusability in that domains.

Similar to RAINBOW, there is a research [19] that also takes decisions on the base of known probability of a failure. It provides failure avoidance based on changing systems component (services) based on scenarios and situations. The cost of each change in system and failure rate of each service is calculated for each scenario and adaptation plan.

According to these researches and our aim of having a self-healing architecture for industrial usage, the following issues must be considered in our work:

- The RAINBOW framework includes reusability and also other capabilities for having an architecture for a system and we can guarantee these abilities by underlying on it. Furthermore, this framework is based on the AC presented by IBM and this point is an acknowledgment to the performance and the verification of this framework and adaptive architectures based on this framework.
- In order to obtain a usable architecture as mentioned in [15] the solution for implementing the architecture must be considered in the architecture itself.
- As given in [9], including knowledge about the environment and the system also knowledge on adaptation in architecture is necessary. In addition, as in [16], reasoning from the knowledge base on the situation should be considered, and also how the knowledge is implemented must be provided as other elements of the architecture.

Considering these aspects, in the next section we introduce a real case study and then we will present our architecture and prove this architecture by referring to previous works.

3. Case Study. To explain our approaches we introduce a real case study, which is taken from an industrial project about using a robot for painting automotive parts. This robot had been programmed to paint a specific automotive part. The software consists of a program P that was written in C# language. The program P can run a software code G that was written in G-code; the execution happens on the robot through an API. G-code is also known as RS-274, which is the common name for numerical code (NC) programming language. The

main usage of the G-code is in the industries that use computer-aided robots. An example of a G-code for our robot and its description from starting the painting process is the following (code descriptions will follow '#'):

```
# version
3
# title
first side buck
# Start Delay
# 36.5
60
# VerticalOffset
0
#iteration delay
6
#GUN Mult
0.75
#GEAR Mult
1.0

COMMAND
# delay action [data]
# delay can be a double number represent an absolute delay passed from start delay
# delay can be as this format: $index,offset == index*iterationDelay + offset
# examples :
# 0.1 HOMINGHEAD
# $1,0.5 MOVE 1 0.5 0.5
#
# delay MOVE vertical gear gun
# 0.2   MOVE 10.3     0.5  0.5
#
# delay COLOR vertical gear gun sprayStartDelay sprayDuration
# 0.5   COLOR 20.5     0.5  0.5 0.1             1.8
#
# delay COLORPOS vertical gear gun

-2.5 MOVE  -3  0.15 0.55

$0,0 COLOR  120   0.15     0.40      0.200 1.600
&0.1 COLOR  -3    0.16    0.63      0.050 1.700
&3.8 COLOR  120   0.15     0.30      0.000 1.300
&0.5 HOMINGHEAD
```

Fig. 3.1 shows the robot. The robot consists of 5 main parts:

1. the *body* which the hand is installed on;
2. the *hand* which carries the wrist;
3. the *wrist* which controls the nozzle;
4. the *nozzle*;
5. the *camera* which can capture the state of the painting position on the automotive part.

The body can lift up and put down the hand which carries the wrist. The wrist is the most important part. It has 2-dimensional freedom degree for changing the position of the nozzle (this grants the robot 3 type of different movements) and of the camera, which takes the picture from the painting position on the automotive part.

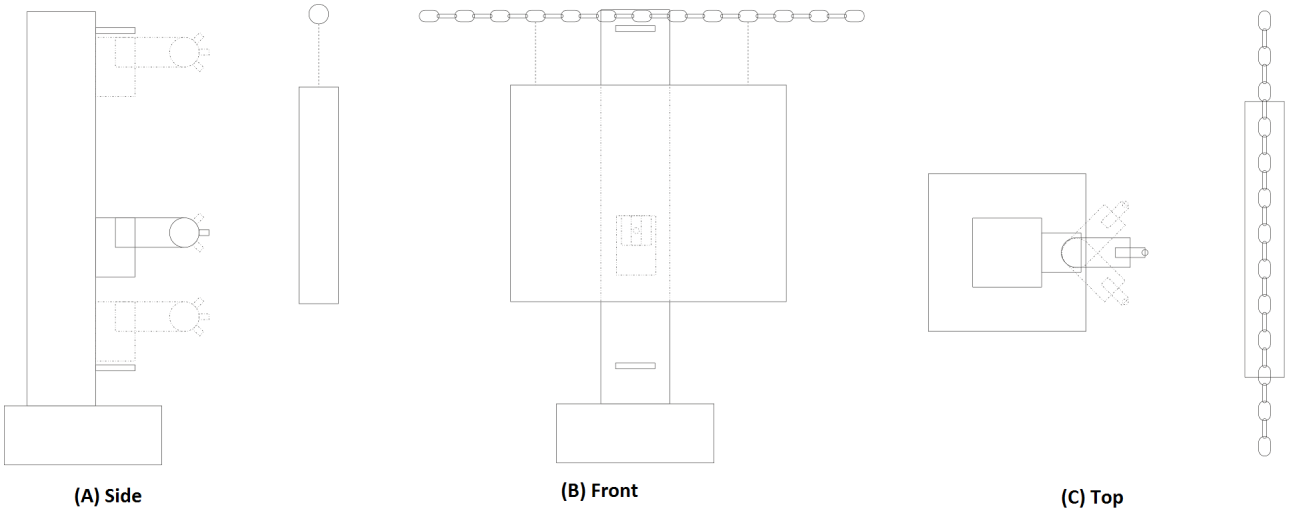

FIG. 3.1. *Painter Robot*

The automotive part to be painted is chained into the conveyor and comes from one side and goes to another side. The speed of the conveyors could be considered slow. The actual speed in the factory is 20 cm per minute. The robot control code (G) was written based on a best practice, which states that the painting is better to be in not all time slots. This means that, for example, it is better to start painting in t=0s and in t=10s the painting stop. After 30 seconds of pause, another 10-second painting slot is started.

Stops between each painting slot make the robot capable of remedying the painting process that has had faults: the corrections are provided in between two painting slots. To remedy the painting processes there must be a feedback from the painting process so the remediation program for those slots is going to be prepared. This process needs some heuristics from humans that are familiar with G-codes and know the object to be painted and the environment and also know about constraints in the painting process. For this purpose, the painting of an object ran so many times and the images had been processed by computer for colorless spots. The experts introduce some remediation in the code G for each situation that shows that the system did not do its job well enough after analyzing the situation (for example a problem on the color density of some automotive parts).

The main problem is that changing this G-codes for programming and remediation is a heavyweight process and task for software developers and will cost so much resource for the project. Any kind of enhancement in this area could be considered as a huge benefit for the company.

This real case study is going to be used to present the proposed architecture.

4. Proposed Architecture. In this section, a self-healing architecture is proposed to address fault management introduced in the previous sections, by means of self-healing. This architecture passed through two phases of improvement. The first phase is going to reduce the time for changing the software for different situations and tasks; the second one is to even make it easier to change and also more general to use.

We did not start our work from scratch, instead we take inspiration from the RAINBOW architecture previously presented. However, some issues drove us to propose a *new* architecture based on RAINBOW; the main needs can be summarized in the following reasons:

- No need for online feedback: differently from the RAINBOW architecture which supports online feedback, in our architecture we do not need online feedback because of the environment and problems nature. In industrial usage, not all the factors that are going to influence the process are known and this will make the online feedback not only unnecessary but also misleading for the context. Knowing the cause of the fault in nondeterministic environment (such as industrial usage) is impossible (even implicit cause) and grabbing the result, analyzing the data and importing it into the knowledge could be a bad idea.

- Nondeterministic environment: there will be so many unknown situations and parameters in industrial usage. The example in this paper is one of them. In this kind of environment, presenting the probability function for each action and also knowing the result set o the environmental variable is not doable. So, simplifying the architecture will help the modeling of the problem.

The following architecture will address the industrial problem easier with less complexity and in an enough complete way rather than the RAINBOW architectural-based solution.

Starting from the existing implementation, we report the improvement we applied to it in two phases, detailed in the next subsections.

4.1. Phase 1. As mentioned before, we aim at proposing an architecture for software systems that can heal themselves against faults and keep achieving their tasks. Considering the case study, an example of the fault is when a small portion of automotive parts has some areas which have oil drop on it so there will be the need for repainting those areas. The robot should recognize these situations and perform a sequence of tasks in order to repaint the unpainted areas. This remediation is going to be the self-healing part for the first phase of the proposed architecture. Besides the functional requirements as the example mentioned above, below we mention some non-functional requirements that have led our work:

- Being general-purpose. Meaning that it is not developed for a specific application.
- It must be developed based on reference architectures or best practice; this will help the architecture and so the system to have the benefits of those best practice or reference architecture in them.
- User of this proposed architecture must be supported in all phases of system development, from strategy to design. Meaning that when creating a macro design, detailed designing, and implementation methods must be considered.
- Bringing the benefits of the architecture for the software system (for instance, reusability).

The first idea that comes to mind about proposing the architecture is to go with best practice and well-known architecture. There are so many works that adopt the feedback loop, but the general purpose and the most well-known reference architecture is the Monitor-Analyze-Plan-Execute (MAPE-K) loop which was introduced by IBM [9]. This control loop is known thanks to its properties: it is a general purpose architecture for supporting all the AC properties (like self-adaptation, self-configuration and so on), it also supports self-healing as one of the four main self-managing properties.

There are some works that introduce an architecture or extension of architecture over the MAPE-K loop, the most well-known is the RAINBOW architecture [1]. It actualizes the MAPE-K loop with a model-based viewpoint.

As presented in the Section 2, the RAINBOW architecture can support the capabilities and advantages of using an architectural approach for the developed system. Also, this architecture is based on the AC architecture reference model (MAPE-K loop) presented by IBM in [9]. Hence, it can guarantee the accuracy and solidity of the proposed architecture based on this framework. Our architecture also inherits the mentioned characteristics. Our architecture is based on the RAINBOW framework and adopts the four main phases in the mentioned framework. We remark that the fact that our architecture is based on RAINBOW can grant reusability to the elements of the developed system based on our architecture. In addition, our architecture is based on the AC reference architecture.

Fig. 4.1 shows the proposed conceptual architecture. In this architecture, we have four main components (*Model Manager*, *Model Analyzer*, *Self-healing Engine* and *Executor*) similar to the RAINBOW and IBM's MAPE loop. A knowledge-base component supports these four components and all the data that are supposed to be transferred in or out of the system should pass through the *Environment Adaptor*, which enables the interactions between the environment and the self-adaptation parts of the system.

An aspect of the RAINBOW framework that must be considered is the needed knowledge and information for reacting to the environment. In the RAINBOW framework, this information is seen as part of the architecture puzzle which is added to each part when needed. It means that each module of the architecture somehow maintains and uses its specific knowledge and attributes. This technique in RAINBOW is different from the MAPE-K IBM reference architecture, where an integrated database is used for maintaining knowledge of all modules. In our architecture, we spent an effort in committing to the information and knowledge described in each RAINBOW framework module, but in order to simplify the implementation further and also to use a

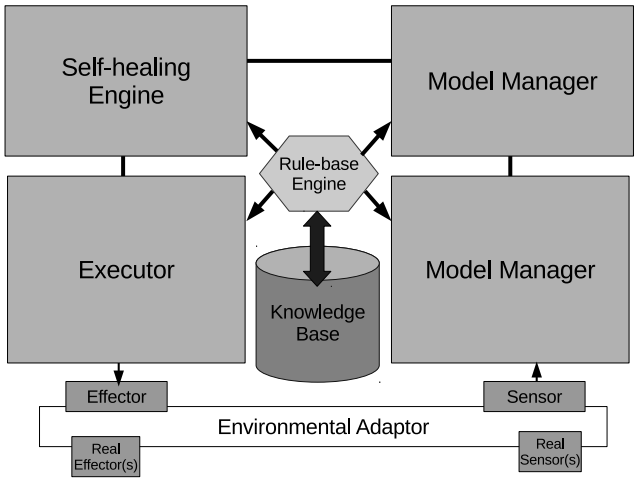

FIG. 4.1. *Self-healing architecture (Abstract view)*

knowledge structure for the all of the system's component, we use a centralized database in our architecture. Using the same knowledge structure throughout the whole architecture can help us in implementing the system easier and it will not carry on different and sometimes inconsistent methods for storing, recovering and using knowledge.

The mentioned knowledge component includes a reasoning capability for managing the knowledge at the run time. Decision making based on this knowledge needs knowledge management capabilities. Hence, the mentioned knowledge component must be able to inference. The existing method for storing, recovering and using the knowledge also having the ability of inference is called "rule-based knowledge" and the "rule-based knowledge base" in [24]. So, in the proposed architecture the knowledge is maintained in a centralized manner in a rule-based type knowledge base and inferences are done based on this type of knowledge.

After the definition of this conceptual architecture, the program P for running the painter robot went through a refactoring process and the new project can count on six main packages:

- `ModelExtractor`: represents Model Manager in the self-healing architecture in the refactored code.
- `ModelAnalyzer`: represents Model Analyzer in the self-healing architecture in the refactored code.
- `SelfHealingEngine`: represents Self-healing Engine in the self-healing architecture in the refactored code.
- `Executor`: represents Self-healing Engine in the self-healing architecture in the refactored code.
- `Knowledge`: represents Knowledge Engine in the self-healing architecture in the refactored code.
- `Main System`: represents Environment which self-healing Architecture deals with.

As mentioned in the packages' presentation, these packages are derived from the component in the proposed architecture (Figure 4.1). Also, having `Model Extractor`, `Environmental Adapter`, and `Self-healing Engine` in the proposed architecture are similar to `monitoring`, `interpretation`, `resolution` and `adaptation` which are mentioned in [25]. This similarity could be considered as compliance of the proposed architecture with previous works.

In the following, we are going to explain the classes and their relationships.

4.1.1. Model Extractor. This package extracts some models from the environmental situations and reports them to next packages. This package consists of the following classes:

- `PainterSensor`: This class is an implementation of a `Sensor` interfaces. It senses the environment (through the `Main System`) and extracts data from robot position, robot occupancy, and automotive part position. Also, it uses `CameraImageProcessing` for knowing about the unpainted areas of automotive parts.
- `CameraImageProcessing`: This class processes the images from the area of painting and find out about

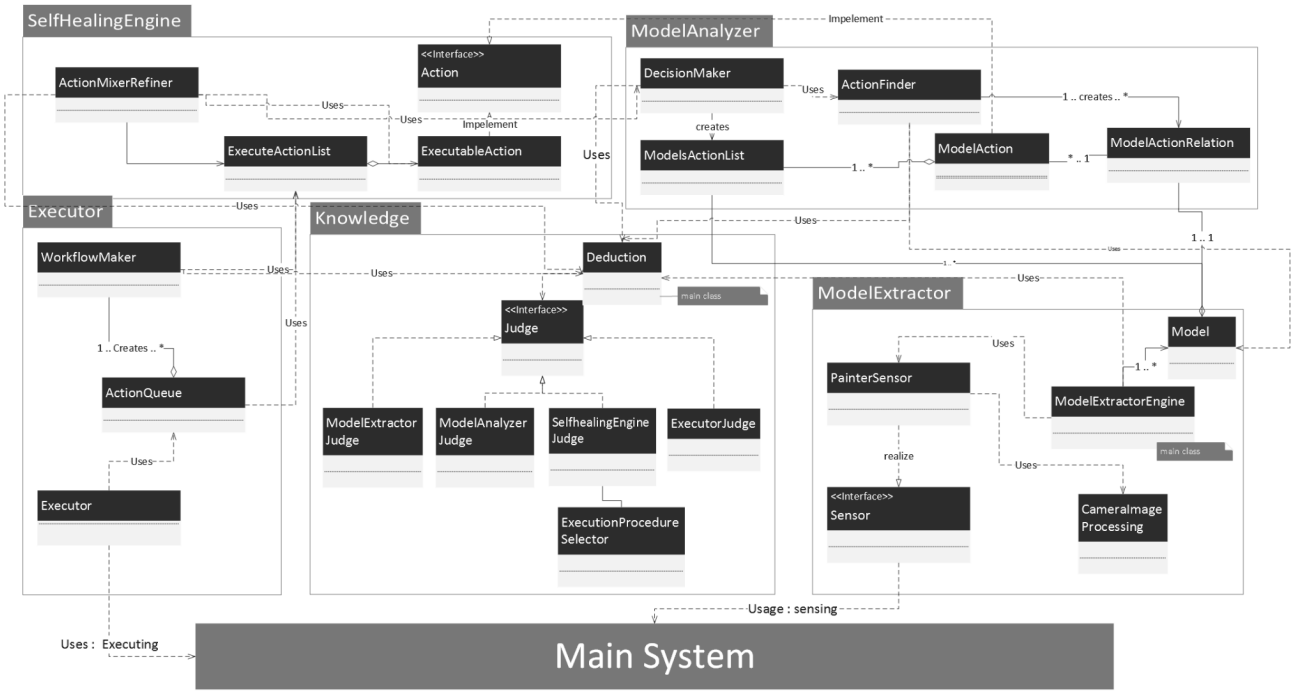

FIG. 4.2. *Class diagram of Phase1's system implementation*

the unpainted areas in the automotive parts.

- `Model`: This class represents the object model in the system. It contains information about the situation which is detected. For example, recognizing the spot with bad painting could be a model.
- `ModelExtractorEngine`: This class based on the information from a sensor and also knowledge from `Knowledge` package, concludes about the models that can fully or even partly represent the current environment and robot states.

4.1.2. Model Analyzer. This package analyzes the situations of the environment and the robot. Suitable actions for each model are the results of the model analyzing the process. The classes in this package are:

- `ModelActionList`: This class represents the list of actions that are concluded for the extracted models or situations.
- `ModelAction`: It represents the action which could be mapped into a model. For example spraying color from above can be an action.
- `ModelActionRelation`: This class represents the relationship between `ModelAction` and `Model`.
- `ActionFinder`: This class creates instances of `ModelActionRelation` between each `ModelAction`'s instances and `Model`'s instances.
- `DecisionMaker`: This class decides about what action should be performed for each model. This class uses the knowledge from the `Knowledge` package. For example, `DecisionMaker` takes decision about repainting the area changing the nozzle angle.

4.1.3. Self-healing Engine. This package will analyze and combine (if needed) the actions for reacting to models. This package will consider the impact and the influence of each action on the robot and environment for combining and scheduling the actions. Also, refinement for actions is applied for action to prepare the applicable and understandable actions for the robot.

- `ExecutableAction`: This class implements the `Action` interface. It represents the action to be executed. For example, changing the nozzle angle in X axle of +22 degrees is an executable action.
- `ExecutableActionList`: This class represents the list of `ExtecutableAction`. For example, a list

consist of these 3 actions are a ExecutableAction list: 1) change the nozzle angle in X axle to +22 degree, 2) spray color for two seconds and 3) change the nozzle angle to 0 degree.

- `ActionMixerRefiner`: Some actions are going to be mixed based on their impact and influence and these actions also should be refined into some fine-grain actions. These tasks are implemented by an `ActionMixerRefiner` class's instance. For example, changing the nozzle direction in X axle of +22 degrees and after that reversing it and after that changing the nozzle direction in X axle for +10 degrees and reversing it could be optimized: first changing direction to +10 degrees and after that +12 degrees more and returning the nozzle to +0 in X-axle.

4.1.4. Executor. This package provides a doable action list and workflow and sends it to the executor for applying the actions to the environment (`MainSystem`). It contains the following classes:

- `Executor`: Instance of this class will execute all the actions (from the `ActionQueue`'s instance) on the environment (`Main System`). This class contains the business logic for executing the actions.
- `WorkflowMaker`: Maybe some actions and some tasks need some prerequisite or maybe it is better to perform them in a specific order. This class will take care of ordering and timing of actions. This class uses the `Knowledge` package and also the `DecisionMaker` class to carry out its tasks. Making a set of changing of robot's position with better efficiency is a goal for this class.
- `ActionQueue`: This queue contains the tasks and actions that can be enacted in the environment.

4.1.5. Knowledge. This package responds to the request (questions and queries) of other packages (actually of their classes). It responds to the queries based on the knowledge that is hard-coded in it. The classes are:

- `Judge`: This interface is implemented by each of the four main components of the control loop (Model Extractor, Model Analyzer, Self-healing Engine and Executor). These classes will answer to the `Deduction`'s instance in order to make it able of deducting about situations and models.
- `Deduction`: This class will answer to the query about situations and models which come from any of four main components of the control loop.

4.1.6. Main System. This package contains the rest of system, which works and paint automotive parts based on its simple work-chain. The other packages see this package as `Environment` and `Main system`.

The improvements applied in this first phase reduced the time for preparing the robot for another automotive part painting. Results are discussed in *Implementation and Experimentation* section.

4.2. Phase 2. The proposed architecture is abstract and conceptual; in order to make it more concrete, each of the architecture elements will be discussed with more details. This explanation should be based on four main RAINBOW framework phases and the IBM reference architecture. These internal elements should be general purpose and not only applicable to one solution. So these elements differ from the packages and classes in the 4.1.

Because the implemented system (which is implemented based on presented abstract architecture in phase 1) and the detailed architecture (which is presented in phase 2) derive from one source, the implemented system in Section 4.1 should be able to fit into the detailed architecture. Based on the related work (RAINBOW, self-healing architecture, fault-tolerant system architecture, etc.) elements in the proposed architecture will be explained with more detailed. Because of that, our architecture must tend to a self-healing architecture based on fault managing. Hence we have used pure fault management architecture, derived elements and details are adapted to our problem's context.

According to these descriptions, we detail our architecture elements. For a better explanation, we are going to continue using the example as a tool for description and clarification of the research.

The *detailed architecture* could be found in Fig. 4.3. The description and specification of the components are presented in the following subsections , referring to the case study.

4.3. Model Manager. In this phase, input information from the environment is transformed into a model of the current environment situation using the related knowledge about the system state and input data. It means that the raw input data is separated, verified and aggregated. The information is given to the *Rule-Based Situation Monitor* module based on the rules. Situations resulted from the rules are transformed into a model

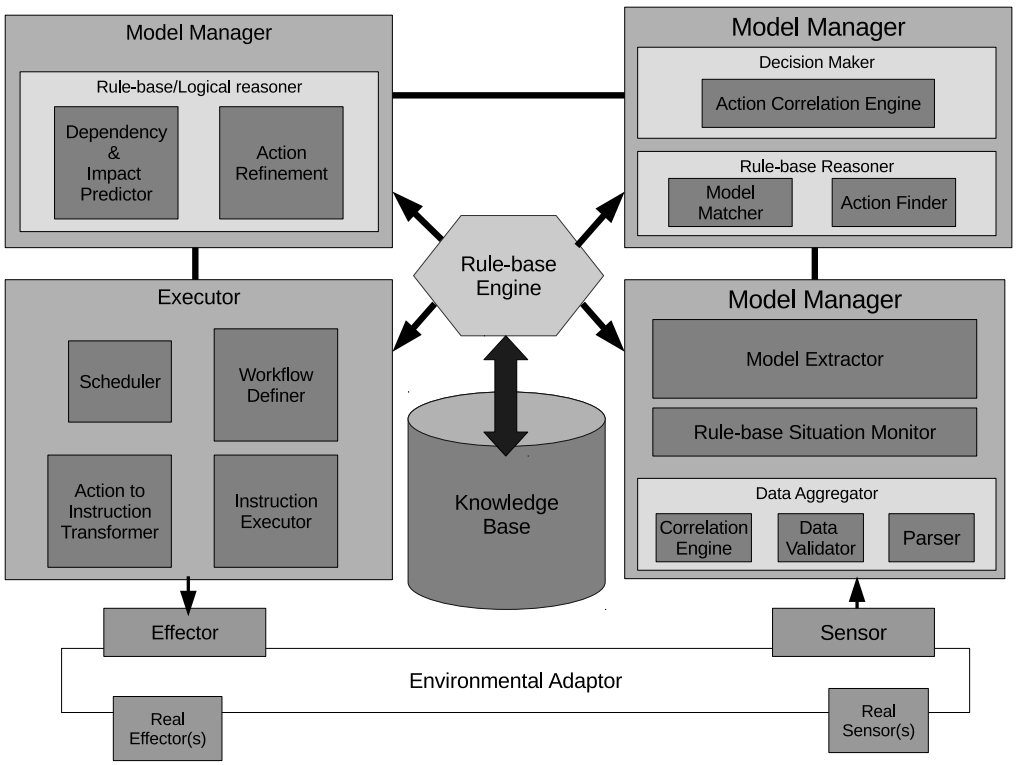

FIG. 4.3. *Self-healing Architecture (Detailed View)*

by the *Model Extractor*. The management of input data is in charge of the *Data Aggregator*, which is composed of the following parts:

- *Parser*: input data from the environment are different depending on context and data structures. Using a parser (which can be inside the *Sensor*), data within these data structures are extracted and changed into a predefined data structure. The *Parser* is implemented by "regular expression matcher" in the case study.
- *Validation Engine*: in case input data are inconsistent with the primary assumptions, the *Validation Engine* will make decisions about these situations (dismissing the data, some changes into the data). For example, in the painting problem, if the position of the area which was not painted correctly is calculated outside the painting area, this information will be ignored. This module examines primary assumptions. This module is implemented as a code execution engine which runs a predefined code on the data in order to validate the data.
- *Correlation Engine*: data received from different elements are aggregated and packed based on their relationship with each other. Data will be correlated with each other in order to construct a more abstracted data (which is usable for understanding the situation) and will cover the details which are not important for the problem. In the example, correlating the four different data about the locations of the body, hand, wrist and the nozzle into a single data structure (which is called position).

Others components of the *Model Manager* are:

- *Rule-based Situation Monitor*: This module is responsible for checking if the information can trigger some rules existing in the database. These rules determine the state according to the input data. In the case study, this module is implemented as a procedure that queries the knowledge base. Query consist of data such as success position (robot and automotive's part position) and failure positions in the painting process.

- *Model Extractor*: Extracted states are examined and the environment model is extracted based on these states. This step integrates and combines extracted situations from the Rule-based module into a set of models. In the case study, failure in past cycle painting (position 0,0), robot's position (50,10,10) and automotive part's position (-90) produce the model (0,0,50,10,10, need_to_reposition(true,true,true)) The model is translated into the following call:
  `method signature:`
  `model (failure's position X, failure's position X, robot's hand position, robot writ's angle in X direction, robot writ's angle in Y direction need_to_reposition( hand's position (true/false), wrist's position (true/false), nozzle's position (true/false)))`

4.4. Model Analyzer. In Model Analyzer subsystem, first models are handed over to the system as inputs. It must be determined whether, based on these models that describe states of the system, any event has occurred which requires a reaction. Hence, the existing model in the *KB* (Knowledge Base) is searched for and checked using inference methods so similar models to the current one are extracted.

In the case study, extracted situations and models will be examined and the suitable actions for these situations and model will be extracted for responding to situations and models. Also, if it is possible, these actions will be merged together.

The modules are the following:

- *Model Matcher* In this module models and situations which are reported by *Model Manager* are examined with the knowledge in the KB and if there is an exact or even similar situation reported in the KB would be found. In the example examining the `(0,0,50,10,10,need_to_reposition(true,true, true))` will match `bad_painting_near_position`.
- *Action Finder*: This component will use the knowledge in KB and also the model in order to find appropriate set of actions for the situation. This set of actions could be extracted from different knowledge in the KB. In the Example for previous situation based on knowledge (because the area is in the center not in the corner, maybe the area is not flat and needs to be painted from above and beyond) will result this action sets.
  `paint (repaint, from_above, 0,0,-10), paint (repaint, from_beyond, 0,0, -10)`
  the description of paint procedure is as follows:
  `paint (repaint/first_time_paint/paint_over_paint, direction of painting (from_above/ from_beyond/front/back), position X, position Y, position in Z)`

These two modules together are called *Rule-Based Reasoner*.

- *Action Correlation Engine*: This module, if there is an option available for correlating some actions with each other, it will reduce the actions set. In the case study, if action set is like this: `paint (repaint, from_above, 0,0,-10), paint (repaint, from_above, 0,-10,-10)` it will change into `paint (repaint, from_above, 0,0,-10,-10)` because these two procedures have so many similar moves in themselves. This mixed data will be refined in the next component.

After preparing the action sets, these action sets can be called "*decisions*". These decisions are entered the Self-healing Engine module.

4.5. Self-healing Engine. Responding to the changes by performing doable decisions is the goal of the architecture presented in this paper. In order to prepare doable decisions from the system's knowledge, it is necessary to first determine the relationships and dependencies between the decisions and the required resources for the decisions using an inference engine and existing knowledge in *action detailer*. After refining actions into detailed actions (i.e., a set of detailed actions could be extracted from an action) and extracting the relation between decisions and required resources for each detailed actions, these data will be used in the process of dependency and impact checking. Afterward, the list of doable actions will take places as the result of this module.

- *Action Refinement*: In this module, overall decisions of the previous phase are transformed to refined and precise decisions applicable to environment using the existing knowledge in the Knowledge Base and logical rules. In the case study, the action set `paint (repaint, from_above, 0, 0, -10, -10)` is translated into this set of actions:

```
1. go(0,-10,0)
2. shoot(1000)
3. go(0,-10,-5)
4. shoot(1200)
```

- *Dependency and Impact Predictor*: In this module the impact and influence of each step are gathered and, if there is a conflict, this module will take decision about doing one of these three options for those actions: 1) stop *both* of them, 2) stop the *second* one, 3) run both of them accepting the *risk*.
  The work of this module is similar to that in RAINBOW that defines a utility function for knowing about the future impact of each decision on the adaptation goal [17]. In the case study, if the time slot is free for 3 seconds and if action 1+2+3+4 are going to take more than 3 seconds, the action 3 and 4 will be eliminated. This decision making is stored in the knowledge base in a simple rule form.

These resulting decisions are sent to the *Executor*.

4.6. Executor. In order to be executed, decisions must include a priority planning and a scheduling, and then they can be executed. The responsibility for scheduling the priority of these decisions and with which precedence they are executed is of the *Workflow Definer* module. Also, planning and determining schedules for these decisions is done by the *Scheduler* module. After scheduling, based on the knowledge, the outputs are transformed from these actions to a series of instructions in system level. This task is carried out by the *Action to Instruction Transformer* module. These instructions are executed in a row by the *Instruction Executer* and the help of the *Effector*.

As happens in the RAINBOW framework, for separating internal system knowledge from the system which adaptive section affects, there is an *Environment Adaptor* with knowledge about translating events and inputs. This module translates the instruction to applicable instructions and also as a part for importing data to the presented adaptive loop, it will translate the information and inputs into comprehensible information into the understandable information for *Model Manager*. This task gives the possibility to reuse the developed system in different applications. For example, if there is another production line with different products, different robots with different tasks (for example welding instead of painting), the *Self-healing Engine* module could be informed of the changes by making a change in the *Knowledge*. The self-healing section of the system will start to act properly in a new situation. This is possible by translating the environment state into an understandable language for the self-healer section of the system. For example, if robot's location is (-10,10,0) and the automotive part's location is -90 so the distance will be translated from these data as -A ($70 \leq A \leq 80$).

The modules in this subsystem are:

- *Scheduler*: This module will do the scheduling based on the knowledge in the KB. In the case study, painting the location which is far from the nozzle sooner than another location is an example of the knowledge in the KB. Results will be like the following.
  ```
  1. go(0,-10,-5)
  2. shoot(1200)
  3. go(0,-10,0)
  4. shoot(1000)
  ```
- *Workflow Definer*: This component will decide about the order and timing of the tasks that are going to be done based on the knowledge in the KB. In the case study, the following tasks will be the results.
  ```
  1''   -> go(0,-10,-5)
  5.5'' -> shoot(1200)
  7''   -> go(0,-10,0)
  10''  -> shoot(1000)
  ```
- *Action to Instruction Translator*: Translations of the instructions from high-level instructions into the understandable instructions for robot will be done in this module. After this translation, the instructions are handed over to the *Effector* and will be execute. In the case study, the previous four tasks are translated into the following G-code, based on the knowledge in the KB.

  ```
  COMMAND
  1.0    MOVE   0  -10  0
  ```

```
#(absolute delay) (command) (x)  (y)  (z) (pre_delay) (task_time)
5.5    COLOR  0  -10  0   0.000   1.200
#(absolute delay) (command) (x)  (y)  (z)
7.0    MOVE   0  -10 -5
#(absolute delay) (command) (x)  (y)  (z) (pre_delay) (task_time)
10.0   COLOR  0  -10 -5   0.000   1.000
END COMMAND
```

- *Instruction Executer*: This module will take responsibility for running prepared instruction on the environment.

4.7. Environmental Adaptor. In this module, if there are any changes (for better understanding or unification of data models) needed, these modifications will be applied. For example, if the idle situation of the robot is called (-10,0,0) in the system and we know that the robot needs floating point number for input (-10.00, 0.00, 0.00) output data will be change into floating point type. Input data from the environment are also comprehended by the *Sensor* and after applying necessary changes will be sensed by a virtual sensor and they enter the *Model Manager* module.

In the case study, the previous four tasks are translated into the following G-code, based on the knowledge in the KB.

Given these explanations and getting familiar with the presented architecture, in the next section, the results of the implementation of the case study with the proposed architecture will be discussed.

5. Implementation and Experimentation (Evaluation and Measurement). In this section all the results from the implementation of the painter robot system will be discussed in two phases, corresponding to the two main improvements we made starting from the original system. Also, for each phase, some implemented module will be discussed. We report the results of the experiments in Fig. 5.1, 5.3 and 5.2; in all figures, the first three items are from the phase 0 (system without improvement), the second three items are from phase 1 and the other items (the last four items) are from phase 2.

5.1. Phase 1. For phase 1 the code had been refactored in order to obey the architecture of a self-adaptive system. The architecture has derived from the RAINBOW architecture. Refactoring procedure took 14 days and after that period refactored code become capable of being adapted easily to the changes and able to support new automotive part easily.

Fig. 5.1 reports the time for preparing the code for each automotive part which this robot supports. Fig. 5.1 shows that the preparation time for changing the code in order to support a new automotive part has been significantly reduced from phase 0 to phase 1, from 1 person for a month to 6 days of a person's time. The reduction in preparation time also will be discussed in next section for phase 2.

Hence, the reduction in preparation time was the main improvement project goal, other improvements also came up from the code refactoring. As it is shown in the Fig. 5.2, the quality of painting after refactoring has seen an improvement. It is because that remediation of known problems in painting becomes easier by just calling some predefined procedures. For example, by running the robot for a new automotive part and finding the area that needs more painting, the predefined procedure for repainting the area (in robot's free time slot) could be exploited easily so the robot paints a larger area and the quality of the painting improves.

As it could be guessed, these improvements will consume some other resources. As it is obvious from Fig. 5.3, the portion of the time slot in which the robot is moving and painting increased after the code refactoring (phase 1). Having predefined procedure for remediation of the painting problems makes preparing of the remediation code much easier and this will lead into more remediation per time slot. Also, this predefined procedure will increase the time of the remediation (even the same remediation) because of eliminating the ability to optimize the path and painting of the robot by integrating two or more predefined procedures in one set of actions.

5.2. Phase 2. In phase 2 the refactored code had faced a significant change. The code's architecture changed into a more mature architecture. The deduction and conclusion which before were taken on the base of if-then-else code changed into a set of knowledge which is implemented in the packages related to the deduction. Also, data and information that traverse through the system were unified into predefined information. These

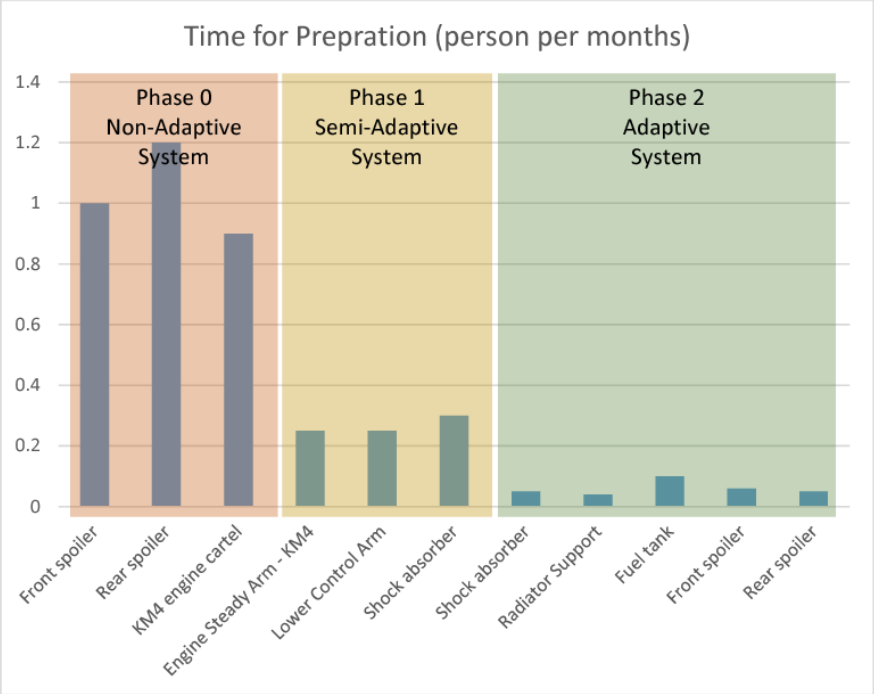

FIG. 5.1. *Preparation time for each automotive part*

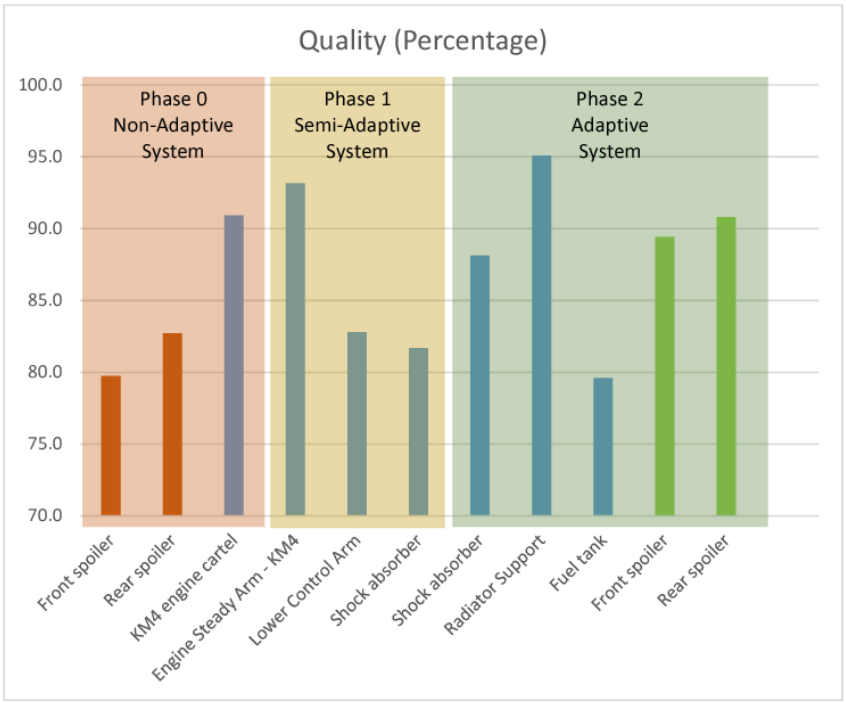

FIG. 5.2. *Quality of painting*

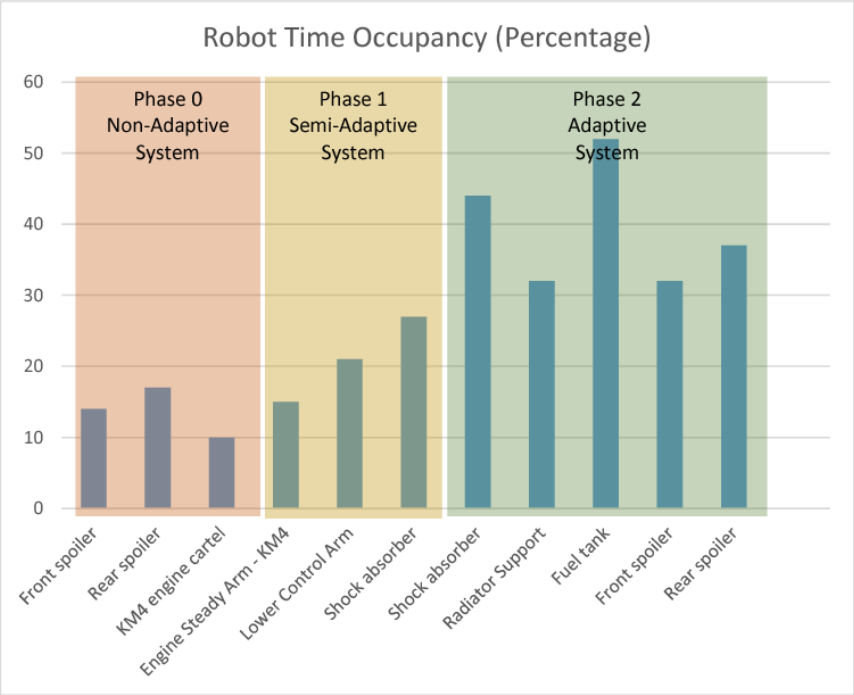

FIG. 5.3. *Robot's time occupancy (percentage of time)*

changes influence the time that is needed for preparing the system for new automotive parts, making it adaptable in a faster way. The time percentage of four automotive parts that had been painted in phase 2 can be seen in the figures. Times for these four parts painting had a significant reduction even compared to phase 1. For example, the first automotive parts (front and rear spoiler) took 1 and 1.2 person months in phase 0, while after the refactoring and implementation in phase 2 they took only 2 days. Also, as it is reported in Fig. 5.2 the quality of painting of these two parts were improved from around 80% to around 90%.

As mentioned before, these improvements imply some costs. The amount of occupied time slots of the robot increased from phase 0 to phase 2. In phase 0 it was around 10-15% and phase 1 has increased to around 15%, while in phase 2 it was increased to around 35%.

6. Conclusion. Computer systems are required to face new and often unpredicted situations during their execution. To react to the faults or to the changes in the environment they must exhibit the self-healing property, introduced in the context of AC, which enables the system to recognize incorrect results and to provide a remediation, by modifying its behaviour. This is true in particular in industrial environments, where the physical part of the systems are subject to faults and variation in the behaviour.

In this paper, we have reported an experience in the field of automotive, in particular, a system to paint parts in an automatic way. We have presented the previous version of the system, which was quite rigid and required a significant effort to repair faults and to be adapted to new parts. We have proposed a self-healing architecture to overcome these limitations.

A case study shows the applicability of the proposed architecture. This case study has been exploited to make some experiments in order to present the improvements in painting quality besides the reduction in time of preparation of the system for a new task. The cost of the improvements is more robot's time occupancy.

In the future works, this architecture could be checked for conflict and error by formal methods and also by software architecture evaluation methods such as ATAM [26] and CBAM [27]. In addition, the proposed architecture will be exploited in other case studies, to verify its generality.

REFERENCES

[1] D. Garlan and M. Shaw, *An Introduction to Software Architecture* ,Addison-Wesley, 1994.
[2] R. Isermann, *Fault-diagnosis systems: an introduction from fault detection to fault tolerance.*, Springer, 2005.
[3] S. Armando and G. Betancur, *Fault management in TELCO platforms using Autonomic Computing and Mobile Agents*, 2011.
[4] N. Technical, *Fault management handbook*, 2012.
[5] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, Proceedings of the first workshop on Self-healing systems - WOSS 02, p. 27, 2002.
[6] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. D. Palma, V. Quema, and J. Stefani,*Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters*, pp. 120.
[7] D. Garlan and B. Schmerl, *Model-based adaptation for self-healing systems*, Proceedings of the first workshop on Self-healing systems - WOSS 02, p. 27, 2002.
[8] P. Eeles and P. Cripps, *The process of software architecting*, Pearson Education, 2009.
[9] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, Computer, vol. 36, no. 1, pp. 4150, 2003.
[10] A. Computing, *An architectural blueprint for autonomic computing*, IBM White Paper, no. April, 2003.
[11] D. W. Cheun and S. D. Kim, *An Engineering Process for Autonomous Fault Management in Service-Oriented Systems*, 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, pp. 901 906, Aug. 2010.
[12] R. Sterritt, *Autonomic computing*, Innovations in Systems and Software Engineering, vol. 1, no. 1, pp. 7988, Mar. 2005.
[13] Y. Qun, Y. X. XU, and X. Man-wu, *A Framework for Dynamic Software Architecture-based Selfhealing*.
[14] G. Antoniou, M. Baldoni, W. Nejdl, and D. Olmedilla, *Chapter 1 RULE-BASED POLICY SPECIFICATION*,
[15] D. Garlan, S. Cheng, and A. Huang, *Rainbow: Architecture-based selfadaptation with reusable infrastructure*, Computer, pp. 4654, 2004.
[16] D. W. Cheun and S. D. Kim, *An Engineering Process for Autonomous Fault Management in ServiceOriented Systems*, 2010 IEEE/ACIS 9th International Conference on Computer and Information Science, pp. 901 906, Aug. 2010.
[17] B. Schmerl, J. Cámara, J. Gennari, D. Garlan, P. Casanova,G. A. Moreno, J. M. Barnes, *Architecture-based self-protection: Composing and reasoning about denial-of-service mitigations*, ACM International Conference Proceeding Series, 2014
[18] J. Camara, P. Correia, D. Lemos, D. Garlan, P. Gomes, B. Schmerl, R. Ventura, *Evolving an Adaptive Industrial Software System to Use Architecture-based Self-Adaptation*, Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on, 1322
[19] R. Mirandola, P. Potena, *A QOS-based Framework for the Adaptation of Service-based Systems*, SCPE, 2011
[20] E. Wegner,*WebNMS Framework: A Complete EMS Framework*,White Paper, 2012, www.webnms.com.
[21] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, *Towards architecture-based self-healing systems*, Proceedings of the first workshop on Selfhealing systems - WOSS 02, p. 21, 2002.
[22] B. Gaudin, M. H. Hinchey, E. Vassev, J. Garcia, W. Coelho Maalej, *FASTFIX: A Control Theoretic View of Self-healing for Automatic Corrective Software Maintenance*, SCPE, 2012
[23] D. B. Abeywickrama, N. Hoch, F. Zambonelli, *Engineering and Implementing Software Architectural Patterns Based on Feddback Loops*, SCPE, 2014
[24] S. Myat, M. Soe, M. Paing, P. Zaw, , *Design and Implementation of Rule-based Expert System for Fault Management*, World Academy of Science, Engineering and Technology, 3439, 2008
[25] D. Garlan, B. Schmerl, *Using Architectural Models at Runtime: Research Challenges*, EWSA Workshop, 2004
[26] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, K. Lipson, J. Carriere, *The architecture tradeoff analysis method*, Sei - Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Institute, 1998
[27] R. L. Nord, M. Barbacci, P. Clements, R. Kazman, M. Klein, L. O'Brien, J. E. Tomayko, *Integrating the Architecture Tradeoff Analysis Method (ATAM) with the cost benefit analysis method (CBAM)*, Sei - Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Institute, 2003

Edited by: Dana Petcu
Received: July 13, 2016
Accepted: August 18, 2016