

Towards automation of IT systems repairs

Janusz Górski¹ · Marek Kamiński¹

Published online: 6 September 2016

© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract Monitoring and repair are two sides of the on-the-fly maintenance of IT systems. Monitoring is well supported by automatic tools. In contrast, repairs involve much higher human intervention which negatively affects reliability and efficiency. The paper introduces a method of automating repairs of IT systems which can be integrated with any of the existing monitoring mechanisms. The method is described as a collection of models and algorithms. The monitored system is modelled as a multi-hierarchy of components, and predicates are used to detect problems and to trigger the corresponding repairs. Repair procedures are represented using flow diagrams. Graphical notations are used to specify the architecture of the runnable component of the proposed method and the corresponding repair API. Formal notation (the Z language) is used to provide for precision and unambiguity of the specifications. The formal specification facilitates reuse of the method in different target environments. For validation purposes, the method has been instantiated (using the PERL language) in an industrial helpdesk supporting the operation of several hundred IT systems. The validation data demonstrate the significant positive impact of the proposed method on the efficiency of the performed repairs.

Keywords IT systems monitoring and repair · Repair automation · System helpdesk · Maintenance of IT systems

1 Introduction

IT systems are often expected to run without interruption which requires that problems occurring during their execution are being resolved on the fly. Such systems are subjected to monitoring which essentially involves repeated assessments of the state of a system and

✉ Janusz Górski
janusz.gorski@eti.pg.gda.pl

¹ Gdańsk University of Technology, ul. Narutowicza 11/12, 80-233 Gdańsk, Poland

applying some correctness criteria to detect problems (faulty states). If the number of monitored objects is small, monitoring can be conducted manually; otherwise, support by automated tools is needed. Numerous monitoring tools are currently available (see e.g. (Radić and Škiljan 2004) for an overview). In general, such monitoring tools serve the following objectives (Albrand et al. 2005):

- Providing up-to-date information about the state of the monitored system,
- Assessing whether the monitored system works according to its functional specification and towards its business objectives,
- Assessing whether the quality of service requirements (reliability, availability, performance and others) are satisfied,
- Detecting anomalies in system functioning or even predicting the future problems.

Solving an already detected problem is called a repair. It involves corrective steps aiming at removing symptoms and causes of the problem.

Numerous monitoring ideas, methods and systems have been developed, some of them focusing on monitoring of specific functional aspects of networks, hosts and applications and others pretending to be general purpose monitoring systems (see e.g. Sakellariou and Zanicolas 2005). In general, monitoring can be defined as a combination of four activities (Sellens 2000):

- Data acquisition—collecting or generating the data,
- Data storage—logging the data,
- Data processing—analysis, comparison, evaluation,
- Data presentation—reporting and exception alerting.

A wide variety of monitoring mechanisms and systems is presently in use (Cottrell 2000).

Centralized systems collect (active monitoring) or receive (passive monitoring) the results in a designated central place for further processing and analysis. Examples of such systems include: Nagios, Opsview, Zenoss, Zabbix, Pandora, Big Brother, Big Sister, HP OpenView, NeDi (Aslett 2006; Jones 2004; CACHED et al. 2007; Barth 2008). Periodic reports provide information on the state of monitored objects, and their administrators are informed about the detected problems (through a dedicated web page) or are notified (for instance, via SMS) and then undertake the necessary repair actions (Barth 2008). Some systems analyse the historical data to detect trends of the monitored parameters (Savage 1999) which provides for undertaking preventive action before a problem actually occurs.

Distributed monitoring aims to monitor grid or cluster structures (Ceccanti and Panzieri 2004). Such systems are equipped with specialized software to provide for resource sharing to balance usage of resources, to coordinate parallel work of computational nodes and for saving the results of processing in a shared storage. Examples of distributed monitoring systems include: Ganglia, Hawkeye, MonALISA, NWS (Network Weather Service), RGMA (Relational Grid Monitoring Architecture), GridRM, Mercurry (Sakellariou and Zanicolas 2005).

At present, monitoring is highly automated and its scope corresponds to the industrial needs. Automatic repairs are less developed. Repairs are more difficult to automate and often involve manual and time-consuming intervention.

The need for monitoring and the high cost of the related repairs resulted in establishing a market of services which are offered by so-called *helpdesk* and cover remote monitoring and technical support and assistance in maintaining and running IT systems, 24 h a day and 365 days a year. For instance, the technical helpdesk of a company providing services

supporting monitoring and repair of IT systems (supporting flight planning) is characterized as follows:

- About 60 customers in over 40 countries and on 5 continents,
- Some 3800 services on over 270 monitored systems,
- The number of problems per day (based on the data recorded in 2014) is 13,
- Varying repair effort, depending on several factors, like the severity and kind of the problem, number of affected environments, competence of the employee involved in the repair and so on.

To summarize, the current industrial situation can be characterized as follows:

- The systems needing monitoring and repair are already in place,
- Majority of them do not have healing-oriented architecture,
- Their administrators have at their disposal a limited set of proven tools and solutions which do not cover the full scope of the needs:
 - The monitoring task has already been automated to a satisfactory level,
 - Complex repairs are still conducted by manual execution of simple commands which is hard to implement and highly error prone.

1.1 The research problem

The research problem addressed in this paper can be expressed as follows:

To provide a generic support for automation of repairs of IT systems with the following assumptions:

- IT system is subjected to continuous monitoring with the help of existing monitoring systems,
- The proposed solution is independent of the monitoring mechanisms used to detect problems,
- Repairs are conducted on the fly and possibly from a remote site,
- Priority is given to automation of the most typical repairs; however, the proposed solution is general enough to be applicable to a broad range of repair problems encountered in practice,
- The proposed solution is reusable in different target environments.

The main business objective for the proposed solution is increasing efficiency of the repairs.

1.2 Contribution of this paper

The paper presents a solution to the above problem, the Automated Repair Management (ARM) method. The method provides for integrating the monitoring and repair processes and for repair automation.

The ARM method includes the following innovative elements:

- The monitoring model which provides for interfacing different monitoring tools,
- The formally specified Repair Management System which provides for automatic invocation and execution of repair procedures,



- The repair API which provides for extending the scope of automated repair procedures.

The above elements of ARM were specified formally using the Z formal notation, and the formal specifications were subjected to verification using the available tools. Such specifications form an excellent basis for implementation of ARM in different target contexts.

For validation purposes ARM formal specifications were used to implement the method in the context of a real industrial helpdesk (the language PERL has been used in this implementation). This provided the basis for experimental validation of ARM. The experiments covered the period of 1.5 year. During this time, ARM was initially used in the testing environment and then has been moved to the production environment. The experiments demonstrated a significant improvement of repairs efficiency.

1.3 Related works

Repairs of selected classes of problems have been already automated with the help of dedicated mechanisms such as: exception handling (e.g. java exceptions), limiting time of operations (e.g. *RPC* timeouts), redundancy (e.g. hard discs mirroring or backup elements) or retrying failed actions a limited number of times. Another example are event handlers where an event is an undesired monitoring result (a problem occurrence) and the associated handler provides an adequate repair action, often by executing remote commands (Barth 2008; Josephsen 2007; Turnbull 2006; Zabbix 2008). Some solutions (Bauer and Campi 2009; Ansible 2015; Puppet 2015) focus on automation of administrative tasks and do not integrate with monitoring directly. Other approaches (Gerlan et al. 2003) focus on describing the architecture to facilitate automated repair. Efforts were made to integrate NAGIOS monitoring system (Barth 2008; Josephsen 2007; Turnbull 2006) with CFEngine system (in particular, regarding network problems (Retkowski 2006)) and to integrate Opsview (a NAGIOS-based monitoring system) with an IT automation software module (Opsview 2015). Some works explore directions in which the monitoring and healing (repairing) can go in a longer perspective (Pervilä 2007).

At a very high level of abstraction ARM can be considered as being similar to exception handling mechanism. A distinguishing feature of ARM however is that it is designed to work with enterprise-level applications where exceptions cross the process and machine boundaries also in situations where a process or a machine has failed to the point where it cannot be handled by the software portion of the process or its host.

The proposed ARM differs from the existing approaches in the following:

- The monitoring model of ARM provides for interfacing the ARM Repair Management System to any of the existing and practically used monitoring systems;
- The concept of problem detecting predicate provides for automatic detection of violations of the correctness criteria related to multiple machines subjected to monitoring;
- The repair API of ARM offers technical means for conducting repairs on distributed machines and in an interactive way, if necessary;
- Formal specification of the monitoring and repair functionalities of ARM provides the base for implementing ARM in different target environments;
- The repair steps of ARM are automatically deployed on the machines subjected to a given repair (there is no need to install any ‘agents’ or similar software components on these machines).



1.4 Introduction to formal method Z

In the definition of ARM we used formal specifications to free the ARM definition of the implementation details, to provide for precision and to avoid ambiguities. Z is a well-documented and matured method of specifying software formally (Davies and Woodcock 1996). It is based on set theory and mathematical logic. Tools supporting various analyses of specifications in Z are also available.

Specification in Z defines a state machine. A specification is decomposed using the concept of *schema*. Schema is represented as a named box. Schemas may contain other schemas—it is known as schema inclusion and the including schema inherits all elements of each included schema. A schema is composed of two (possibly empty) sections: *declarations* and *constraints*. If both exist, they are separated by a short horizontal line inside the schema box.

Declarations introduce named data objects of the schema along with their types. The names of the included schemas can also be listed in this section. Constraints are logical expressions imposing restrictions on the objects introduced in the declarations section of the schema.

Changes of states are defined by using unprimed (e.g. x) and primed (e.g. x') names, where the former refers to the past state (before the change) and the latter refers to the future state (after the change). Priming is used to model dynamic aspects. Each constraint is an invariant, and it must be satisfied by all states. A constraint binding value of an unprimed and primed object of the same name imposes an invariant relationship between the states of this object. Such invariant binds the future states of a system with its past states.

In formal specifications presented in the subsequent sections we assume the following notational conventions: \exists denotes existential quantifier, \forall denotes universal quantifier, \times denotes Cartesian product of two sets, \subseteq denotes set inclusion, \in denotes element inclusion, $1st(p)$ denotes the first element of an ordered pair p , $\mathbb{F}X$ denotes an incomplete power set (a set of all finite subsets) of X , $dom(R)$ denotes a set being the domain of a relation or a function R , $X \rightarrow Y$ denotes a partial function (a function that assigns elements of its codomain to elements of its domain, but not necessarily to all of them), *iseq* is an injective sequence (a sequence storing each of its elements only once), and $\#$ is a (unary) operator returning the number of elements of a set being its argument.

1.5 Structure of the paper

The paper is structured as follows. In Sect. 2 we introduce the notion of a repair problem, give examples of four typical problems occurring in IT systems and explain how these problems are dealt with. In Sect. 3 we explain how repairs are invoked by repair problem detectors based on the assessment of the monitoring data coming from an arbitrary monitoring system. In Sect. 4 we give more details on how the repairs initiated by problem detectors are being executed. In Sect. 5 we introduce the ARM method as a sequence of steps leading to automation of the repairs. In Sect. 6 we introduce the architecture of Repair Management System (RMS), the runnable component of ARM. In Sect. 7 we provide more details on repair API, the main component of RMS. In Sect. 8 we give details of the implementation of ARM in a production environment and present the results of ARM validation during a one and half year period. In conclusion we summarize the contribution of this paper.

2 Repair problems and their solutions

Repair problem of an IT system is a violation of the correctness criteria assumed for this system. We assume that repair problems of systems of interest are detected based on the data collected during monitoring of these systems. In most cases such problems reappear, and therefore, it is convenient to represent the corresponding repair actions as *repair procedures* that can be initiated if a problem occurs. A repair procedure may represent a solution to a class of similar repair problems. A *repair process (repair)* is an application of the procedure.

Repair procedure can interact with the outer world (input/output), call internal routines (local procedures or functions) and call external routines (procedures or functions outside the scope of the specified procedure). External routines represent corrective steps aiming at fixing or changing the components of the monitored system. These routines are usually executed on remote machines possessing the components being subjected to the repair. Further in this text such repair routines are called *repair steps*.

Let us consider an example of two file systems residing on different machines, A and B. To solve a particular repair problem we may need to enlarge the capacity of each file system. That is, we need two actions: an action enlarging the file system on machine A and another action doing the same on machine B. These actions of repair are, therefore, naturally distributed. They can be implemented as external routines called by the corresponding repair procedure. The body of the repair procedure is location independent—it can be executed on any machine (including A and B). This observation forms the cornerstone of the method presented in this paper: a repair procedure can run on an arbitrary machine, connecting to the remote machines only when necessary.

2.1 Example repair problems

In this section we present four typical problems encountered in industrial IT systems. Each problem is an abstraction of a class of similar situations that may occur in a system subjected to monitoring.

2.1.1 Problem 1: Corrupted scheduler table

Many operating systems offer scheduler programs which execute tasks at specified times (for instance, at night or during weekends). Such tasks are commonly used to support system maintenance and administration, but they can also be used for other purposes, for instance for connecting to the Internet and downloading email. A well-known and popular scheduler program in many UNIX distributions is called CRON. Definitions of the tasks and exact times of their planned executions are stored as entries in so-called scheduler table (in case of CRON, it is CRONTAB).

The problem occurs if, for a particular scheduler table deployed on multiple machines, some of its entries are defective and need correction.

2.1.2 Problem 2: Backup failure prevents database management system (DBMS) from working

Integrity and recoverability of databases are supported by using a special file called *redo log file* [Oracle DBMS (Whalen 2006)] or *journal file* [Ingres DBMS (Ingres Corporation

2006)]. While processing transactions, *DBMS* writes to these files the flow of changes and in case of failure, forward recovery is being applied to recover from the last backup copy of the entire database. Consequently, two kinds of backups are maintained by the OLTP (Online Transaction Processing) system: *full backup* (also called checkpoint) and *incremental backup* recording each subsequent change since the last full backup. Recovery of the full database requires access to both the last full backup and the related incremental backup.

The files keeping checkpoints and the related journal files usually reside on dedicated file systems. As each such file system has limited capacity, the files are being archived, if no longer needed. Usually, the older backups are removed, after the new ones have been created.

The problem occurs when the storage resource runs out of free space. A common scenario leading to the problem is as follows:

- The procedure performing routine checkpoint of a production database terminates abnormally,
- In consequence, journal files are not removed which increases the probability of overflowing the journal-dedicated file system,
- The journal-dedicated file system runs out of free space which prevents *DBMS* from creating new journal records,
- The archiver process creating new journal records terminates and consequently *DBMS* stops, preventing the users and the batch processes from working.

The symptom indicating the problem can be defined as a conjunction of two unwanted conditions: the journal-dedicated file system is 100 % full, and the archiver process is down.

2.1.3 Problem 3: Failures while optimizing physical structures of a production database

Due to continuous changes, OLTP database systems need regular maintenance to provide for acceptable response time and other quality parameters. This is usually performed in the following way:

- The scheduler table includes a task for optimizing physical structure of the database,
- This task is executed regularly and produces a log file indicating the database structures that have been successfully optimized.

The following indicators characterize the condition of a database concerning its performance:

- Characteristics (for instance, the level of defragmentation) of database storage structures (B-tree, heap) underlying the tables stored in the database,
- Presence or absence of indexes created for these tables,
- Freshness of data-related statistics (such statistics influence the behaviour of Query Optimizer Facilities and help to choose the best query execution plan).

The process of optimizing the performance of the whole database focuses on improving the above indicators, for each database table (CA technologies 2006; The PostgreSQL Global Development Group 2013).

The problem occurs when the log file collecting the indicators contains information about failures that occurred during optimization.



2.1.4 Problem 4: Unacceptable delay in replication of time-critical data

Some data are time dependent. Examples include data related to stock markets, flight planning, flight safety, medical treatment and so on.

In contemporary systems, time-critical data usually follow the following scenario:

- Insertion of time-critical data to a centralized or distributed database is initiated,
- The data are transported over the network to its destination locations using selected replication techniques,
- The recipient of the data detects its availability and starts using the data.

The problem occurs if the replication process fails causing that data cannot be timely delivered to data recipients.

2.2 Problem solutions: repair procedures

Below we present solutions to the problems described in Sect. 2.1. The repair procedures are specified using flow diagram notation. To reduce complexity and to avoid non-relevant details, in the presented diagrams we use a block representing a non-deterministic choice (a grey rectangle with one input and multiple outputs). Its meaning is that one of its outputs is selected based on the details which are not represented in the diagram (in particular, this choice may be based on the input received from a human operator).

2.2.1 Solution of Problem 1

The solution of Problem 1 is very simple (see Fig. 1).

Essentially, the repair involves introducing corrections to each copy of the scheduler table and making them accessible to the scheduler programs again.

2.2.2 Solution of Problem 2

The procedure of bringing DBMS to the working status is more complex (see Fig. 2). The initial steps prepare for solving the problem (finding a solution). If such solution cannot be found automatically, the procedure may stop or may ask for the operator's input. The operator's input may also be needed to support the already found solution. If the next steps cannot be successfully completed, the procedure rolls the performed changes back and stops.

2.2.3 Solution of Problem 3

The solution of this problem is shown in Fig. 3. In particular, the procedure needs some auxiliary actions to prevent the problem from reoccurring, such as suspending/un-suspending the related replications.

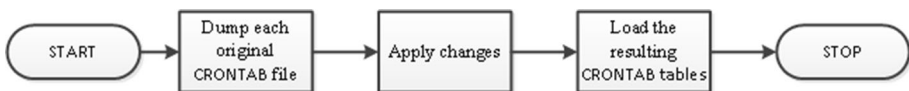


Fig. 1 Repairing a faulty scheduler table



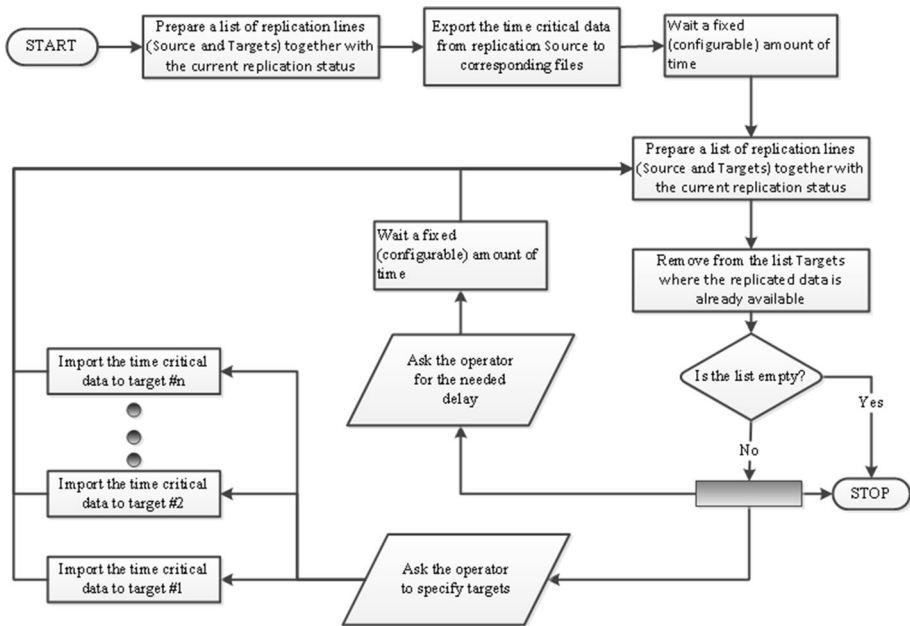


Fig. 4 Forcing emergency replication of time-critical data

2.2.4 Solution of Problem 4

The solution of this problem is shown in Fig. 4. It involves monitoring the status of a replication. After time-critical data have been inserted to the database, the repair procedure is being started as a supervising process (commonly called ‘watchdog’) and its role is to intervene whether the time-critical data are not available at its designated targets.

3 Monitoring model

We interpret monitoring as a repetitive process of examining subsequent states of the monitored objects against the defined correctness criteria and informing about the detected anomalies. This definition abstracts from the physical nature of the monitored objects. Such object may be a remote machine (computer or network device) running an operating system, a service or any other identifiable resource. States of the monitored objects assume values from specific domains, and based on these values *repair problems* are detected by applying the monitoring criteria.

The ‘monitored world’ is often structured hierarchically. *Monitoring model* is a multi-hierarchy (a forest of trees) of *containers*, and the monitored parameters are the leaves of the trees (*leaf containers*). Top-level containers are called *root containers*.

To distinguish between the monitored parameters we need a naming system. We represent such parameters as variables where a value of the variable represents the current state of the monitored object (we assume availability of a mechanism which binds variables to the real objects subjected to monitoring) and the name of the variable indicates the position of the monitored parameter within the hierarchy of containers.

3.1 Problem detectors

With the above assumptions monitoring becomes a process of continuous checking the values of multiple variables against the specified set of monitoring criteria (problem detectors).

The monitored variables represent the paths in the monitoring model hierarchy (for instance, the parameters of hosts storing replication hubs, replication channels, replication lines and so on). To refer to these variables we use their path names, for example:

- `eclipse.oracle.TESTDB.fs.usage`—represents the usage of the main filesystem of machine `eclipse` belonging to *DBMS* `oracle` and identified by *SID* `TESTDB`,
- `eclipse.CPU.temperature`—represents current temperature of the main processor of the same machine,
- `eclipse.CPU.usage`—represents usage of the main processor.

Monitoring involves applying the monitoring criteria. These criteria can be formalized as *problem detecting predicates*. Simple examples of such predicates are shown below:

- `eclipse.oracle.{$anyDATABASE}.fs.usage > 95 %`
this predicate becomes true if an `oracle` database denoted `{$anyDATABASE}` of the machine `eclipse` has used more than 95 % of space of its main file system,
- `($hostX).CPU.temperature > 70 °C` AND `($hostX).CPU.usage = 100 %`
this predicate becomes true if the main processor of a machine designated as `{$hostX}` is overheated and is 100 % busy.

In the above examples, the notation: `{$...}` represents a part of the name of the variable that may be substituted by any valid string.

With the above model detecting that a given predicate becomes true tells us about a repair problem occurrence.

The above method of representing monitored objects and examining their states abstracts from how the measurements are taken. It focuses on violation of the correctness criteria (given by the problem detecting predicates) and is applicable to virtually any hierarchically organized structure of monitored objects.

An example system subjected to monitoring is shown in Fig. 5.

The system in Fig. 5 is a single machine with a hierarchy of containers (a tree with two branches). The path names of the two monitored variables are:

`eclipse.ingres.II.processes.dmfacp`—this variable informs about status of the archiver process.

`eclipse.ingres.II.filesystems.jnl`—this variable indicates level of occupation of the journal file system.

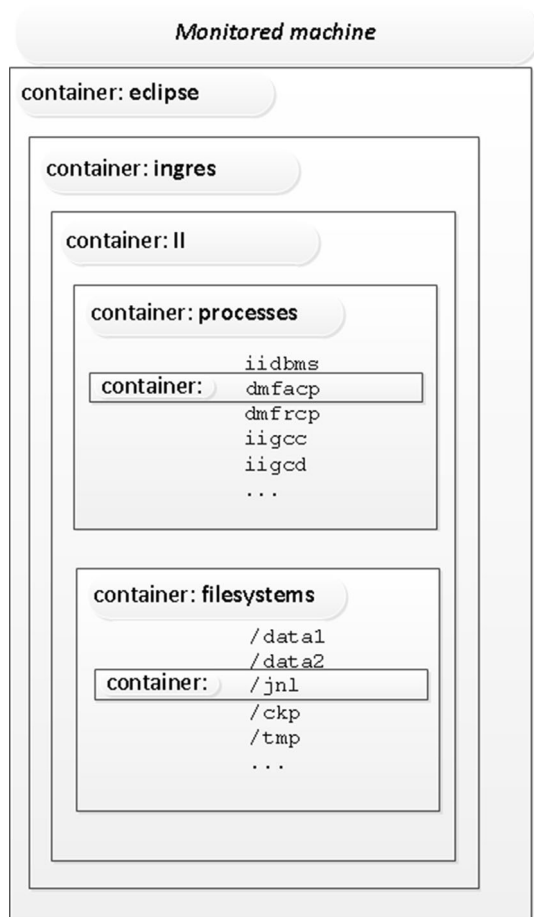
The values of variables representing the monitored parameters change in time. The name of a variable, together with its current value and a timestamp, represents a single *measurement* informing where and when the value has been collected. The measurements assume the following structure:

$\langle \text{monitored_variable} = \text{value}, \text{time-of-measurement} \rangle$.

We introduce a constant *time_interval* and assume that for two measurements $m_1 = \langle v_1 = \text{value}_1, t_1 \rangle$ and $m_2 = \langle v_2 = \text{value}_2, t_2 \rangle$ to be considered together in a given problem detecting predicate, the following condition must hold: $|t_1 - t_2| \leq \text{time_interval}$.



Fig. 5 An example system under monitoring



This means that the times of measurements used to evaluate the predicate cannot be too far one from another. In practical situations, the value of *time_interval* depends on the frequency of measurement results delivered by a given monitoring system. For instance, the system referred to in our experiments monitors some 300 production IT systems using Nagios (Barth 2008) where the minimal time distance between two measurements of the same parameter is 1 min, but for practical reasons this time distance was set to 5 min. In general, if the problem detecting predicate involves measurements of multiple parameters, the value assigned to *time_interval* needs to be large enough to provide for collecting all the measurements referred to by the predicate.

Let us assume that we are interested in detecting Problem 2 described in Sect. 2.1.2 and the problem detecting predicate is as follows.

3.1.1 Problem 2: Detecting predicate

```

${HOST}.ingres.${INSTALLATION}.processes.dmfacp='down' AND
${HOST}.ingres.${INSTALLATION}.filesystems.jnl = 100 %

```

To apply this predicate to the system shown in Fig. 5 we make the following substitutions:

'\${HOST}' is substituted by 'eclipse',
 '\${INSTALLATION}' is substituted by 'II',

which results in the following expression to be evaluated based on the measurements collected during monitoring of this system:

```
eclipse.ingres.II.processes.dmfacp='down' AND
eclipse.ingres.II.filesystems.jnl = 100 %
```

A set of measurements which together make a problem detecting predicate true is a *scope* of this predicate. Each scope contains information on when the problem occurred and which objects were contributing to this occurrence.

Let us assume that the following scope occurs in the monitoring results of the machine of Fig. 5 (we assume that *time_interval* = 1 h):

```
{<eclipse.ingres.II.filesystems.jnl = 100 %,      12:30:40    GMT,
01.jan.2014>,
<eclipse.ingres.II.processes.dmfacp='down',    12:34:56    GMT,
01.jan.2014>}
```

This scope makes the *Problem 2 detecting predicate* true which means that a problem needing repair has been detected on 1 January 2014, between 12:00 and 13:00 GMT, and the corresponding repair procedure needs to be invoked.

3.2 Monitoring subsystem

Referring to the above ideas of the monitoring model, the monitoring subsystem is specified (using the Z notation) as follows:

<p style="text-align: center;"><i>MonitoringSubsystem</i></p> <hr/> <p><i>CommonConfiguration</i></p> <p><i>leavesMeasurement</i>: CONTAINER × TIME → STATE</p> <hr/> <p>∃function: CONTAINER × TIME → STATE • <i>leavesMeasurement</i> = function</p> <p>∀p: dom(<i>leavesMeasurement</i>) • (Ist(p)) ∈ <i>monitoredLeaves</i></p>
--

The meaning of this schema is as follows.

CommonConfiguration is a schema containing a set of declarations shared by other schemas. These declarations include in particular definitions of *STATE*, *TIME*, *CONTAINER*, *leavesMeasurement* and *monitoredLeaves*.

STATE represents a domain of values assumed by the monitored objects. *TIME* represents the time domain. *CONTAINER* is defined as

CONTAINER ::= *nothing* | *contains* <<F CONTAINER>>.

leavesMeasurement is a function representing the measurements taken during monitoring. Every element of the domain of this function represents a pair where the first element is a leaf container and the second element is a timestamp. This function interfaces the monitoring subsystem to the real world where real measurements are being taken.

The invariant given in the second part of the schema asserts about the existence of the *leavesMeasurement* function and asserts that it is defined only for objects belonging to *monitoredLeaves* (the set of objects subjected to monitoring). Therefore, by referring to the values of the *leavesMeasurement* we have access to the measurements taken on the monitored objects in the specific moments of time.

The formal definition of *MonitoringSubsystem* asserts about existence of the *leavesMeasurement* which, for each time moment and each monitored object, delivers a value representing the current state of this object. This way the measurements become available and can be subjected to further processing.

4 Repair model

The repair model describes how the results of monitoring are being used to initiate repair actions. The scenario of solving a problem is as follows:

- The problem is detected by its detecting predicate,
- The corresponding repair procedure is initiated,
- From the scope of the problem, the procedure identifies the objects needing repair, and the repair steps are distributed to the corresponding remote hosts,
- The repair process invokes the repair steps as external routines and brings the remote hosts to correct states,
- The repair terminates.

4.1 Repair process environment

Each initiated repair assumes the following states:

- *Pending* the repair is already triggered but not yet running,
- *Running* the repair is in progress,
- *Suspended* the repair needs additional information, e.g. human intervention,
- *Finished* the repair has terminated.

Below we give a formal specification of the *Repair* schema which defines an initial environment of a repair process.

<i>Repair</i>	
(1)	<i>RepairSubsystem</i>
(2)	<i>Problem_detector</i> : <i>PROBLEM_DETECTOR</i>
(3)	<i>Procedure</i> : <i>PROCEDURE</i>
(4)	<i>Scope</i> : <i>SCOPE</i>
(5)	<i>scopeHosts</i> : \mathbb{F} <i>HOST</i>
(6)	<i>scopeContainers</i> : \mathbb{F} <i>CONTAINER</i>
(7)	<i>currUserName</i> , <i>currHostName</i> : <i>NAME</i>
(8)	<i>currConn</i> : <i>CONN</i>
(9)	<i>inputList</i> , <i>outputList</i> : <i>LINES</i>
(10)	<i>state</i> : <i>QUEUE</i>
(11)	<i>globalArgs</i> : <i>ARGUMENTS</i>
(12)	<i>result</i> : <i>RESULT</i>
(13)	$Problem_detector \in Problem_detectors$
(14)	$Procedure \in Procedures$
(15)	$(Problem_detector, Procedure) \in Triggers$
(16)	$\forall i: 1 \dots \#(Scope) \cdot (I^{st}(Scope(i))) \in monitoredLeaves$
(17)	$scopeHosts = \{ h: HOST \mid \forall i: 1 \dots \#(Scope) \cdot (I^{st}(Scope(i))) \in tree(h) \}$
(18)	$scopeContainers = \cup \{ c: \mathbb{F} CONTAINER \mid \forall h: scopeHosts \cdot c = tree(h) \}$

Line (1) represents the *RepairSubsystem* schema given in Sect. 4.2. Lines (2), (3), (4), (5), (6) represent a repair problem to be dealt with. A repair is triggered if *Problem_detector* becomes true, after substituting the *Scope* for its arguments. This repair uses *Procedure* to solve the problem. During execution the value of *Scope* remains accessible to the repair, and *scopeHosts* and *scopeContainers* represent information derived from *Scope* about hosts and containers affected by this repair.

The value of *currConn* (line (8)) indicates the current *USER* \times *HOST* pair, and the repair is interacting with during execution. The names identifying the current user and the current host are stored in the *currUserName* and the *currHostName* objects (line (7)).

Auxiliary variables *outputList*, *inputList*, *state* and *result* [lines (9), (10) and (12)] store intermediate values during repair execution or collect user input when the repair remains *suspended*. Global arguments are stored in *globalArgs* field (line 11). They are initially empty.

Constraints (13), (14) and (15) assert that the repair is ‘legal’ in a sense that it corresponds to the actual problems and repair procedures. Constraint (16) asserts that the measurements contained in *Scope* indicate the actual objects subjected to monitoring. Constraints (17) and (18) explain how *scopeHosts* and *scopeContainers* are derived from *Scope*.

4.2 Repair subsystem

RepairSubsystem referred to in the *Repair* specification is specified by the following schema.

*RepairSubsystem**CommonConfiguration**RemoteOperations**AutomatedProblemsSolving**Problem_detectors*: \mathbb{F} *PROBLEM_DETECTOR**Triggers*: \mathbb{F} (*PROBLEM_DETECTOR* \times *PROCEDURE*) $\forall c: \text{Problem_detectors} \cdot \forall d: \text{dom}(c) \cdot \forall i: 1 \dots \#(d) \cdot (I^{st}(d(i))) \in \text{monitoredLeaves}$ $\text{Triggers} \subseteq \text{Problem_detectors} \times \text{Procedures}$

The meaning of this schema is as follows.

CommonConfiguration, *RemoteOperations* and *AutomatedProblemSolving* are other schemas whose formal specifications are not included here. In particular, they contain definitions of *PROBLEM_DETECTOR*, *PROCEDURE*, *Procedures* and *monitoredLeaves*.

PROBLEM_DETECTOR denotes problem detecting predicates represented by their scopes (each scope is a collection of measurements that make the predicate true).

PROCEDURE is a schema defining a repair procedure, and *Procedures* is the actual set of repair procedures following this schema.

RemoteOperations represents definitions of the mechanism of remote execution of repair steps.

AutomatedProblemSolving represents definitions and schemas defining how the repair procedures triggered by problem detectors are executed.

Triggers is a set of pairs linking each problem detector to the corresponding repair procedure.

The second part of the schema specifies two invariants. The first asserts that every problem detector refers to actually monitored objects (leaf containers). The second invariant asserts that *Triggers* establishes relations between actual problem detectors and actual repair procedures.

5 Automated Repair Management (ARM) method

In this section we introduce the Automated Repair Management (ARM) method that supports automation of repeatable repairs of IT systems. The method assumes that a monitoring process is already in place.

At runtime, ARM is embodied by the Repair Management System (RMS). The main tasks of RMS are the following:

- Collecting the monitoring-related data from the remote objects (hosts),
- Based on these data, detecting problems and initiating the related repair procedures,
- Executing the initiated repair procedures.

ARM proposes the following steps leading to automation of problem repairs.

- Step 1: Unification of the monitoring results.* The monitoring results are translated in accordance with the monitoring model of Sect. 3. This requires an appropriate adaptor for each actually used monitoring system. After instantiation of such adaptors, the monitoring results become consistent with the monitoring model and can be used by RMS
- Step 2: Identification of common repair problems and their solutions.* An inventory of typical (and repeatable) problems occurring in the monitored systems is built together with the proposed solutions to these problems. Based on this, problem detectors are identified and represented with the reference to the elements of the monitoring model
- Step 3: Abstract representation of problem solutions.* Solutions to the problems are specified semi-formally using graphics and structured natural language with reference to the monitoring model elements. Then each solution is presented as a complete algorithm using a convenient notation (e.g. flow charts). From these, formal specifications may be developed to increase precision and to avoid ambiguities
- Step 4: Identification and creation of repair step interfaces.* Based on the specifications of problem solutions, the repair steps are identified (as external routines to be performed on remote hosts) and the interfaces to communicate with them are added
- Step 5: Populating the repair steps repository.* The repair steps are implemented and placed in the repair steps repository. From there, during operation of RMS, the steps are automatically distributed to their destination hosts
- Step 6: Creation of repair procedures.* Each abstract specification of problem solution is implemented as a complete repair procedure expressed in the selected programming language. The implementations are placed in the related repair procedures repository
- Step 7: Integrating with the monitoring infrastructure.* The data from the monitoring adaptors (see Step 1) are interfaced to RMS
- Step 8: Running RMS.* RMS starts execution and periodically polls for the monitoring data. Using problem detectors it detects problems and initiates the corresponding repair procedures. The resulting repairs are executed performing the necessary repair steps on defected machines, and the Automatic Repair Management (ARM) process is in place

The scope of ARM within the broader context of maintenance activities (Goettsche 2005) is illustrated in Fig. 6.

6 Architecture of RMS

The architecture of RMS is shown in Fig. 7.

The architecture distinguishes two parts: *Central* and *The Rest* interconnected through the *Network*. *The Rest* includes the machines subjected to monitoring and repair. It is assumed that in each of these machines a monitoring function is already executing (it can be any of the available monitoring systems, for instance NAGIOS). The data resulting from monitoring become available to *Central* through *Network* (the mechanism of data delivery can be arbitrary, including email and manual input). In addition to transferring the



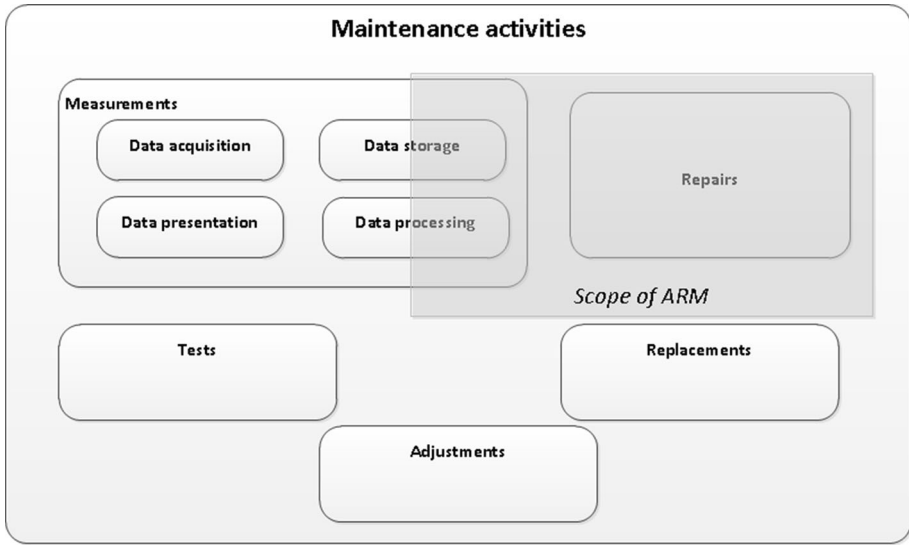
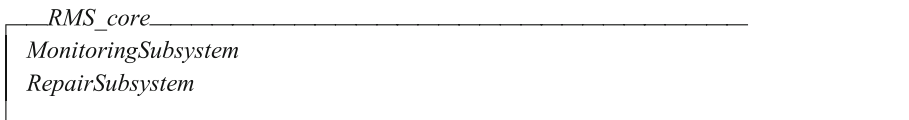


Fig. 6 Scope of ARM (the grey area) within the context of maintenance activities

monitoring data, *Network* is also used to distribute remote operations to the monitored machines and to transmit the results of their execution.

Central represents a machine where RMS has been deployed. It includes operator's interface, *Data unifiers* which convert the monitoring data in accordance with the monitoring model (see Sect. 3), *RMS Core* which is the main program of RMS and the *Repair API* which contains repair procedures and their 'building blocks' (external routines).

The abstract specification of *RMS Core* is given below as a Z schema:



The specification says that *RMS_core* is further divided into two components: *MonitoringSubsystem* and *RepairSubsystem*. The monitoring subsystem is responsible for periodically reading the results of monitoring from the remote hosts and unifying them in accordance with the ARM monitoring model. The repair subsystem is responsible for checking the monitoring results against the problem detecting predicates, invoking the corresponding repair procedures and executing the repairs to bring the defected objects back to their correct states. To fulfil their tasks, the monitoring and the repair subsystems use the *Repair API*.

Before a repair procedure is executed, the repair steps called by the procedure have to be distributed to their destinations (the remote hosts where the steps are to be executed). This distribution is performed as a prologue to the repair procedure execution with the help of the remote operations interface of the Repair API (see section 0).

Some hosts in the network are accessible from *Central* indirectly, via intermediate hosts called *hops* (About.com 2015). Such topology may result from security issues (the hosts

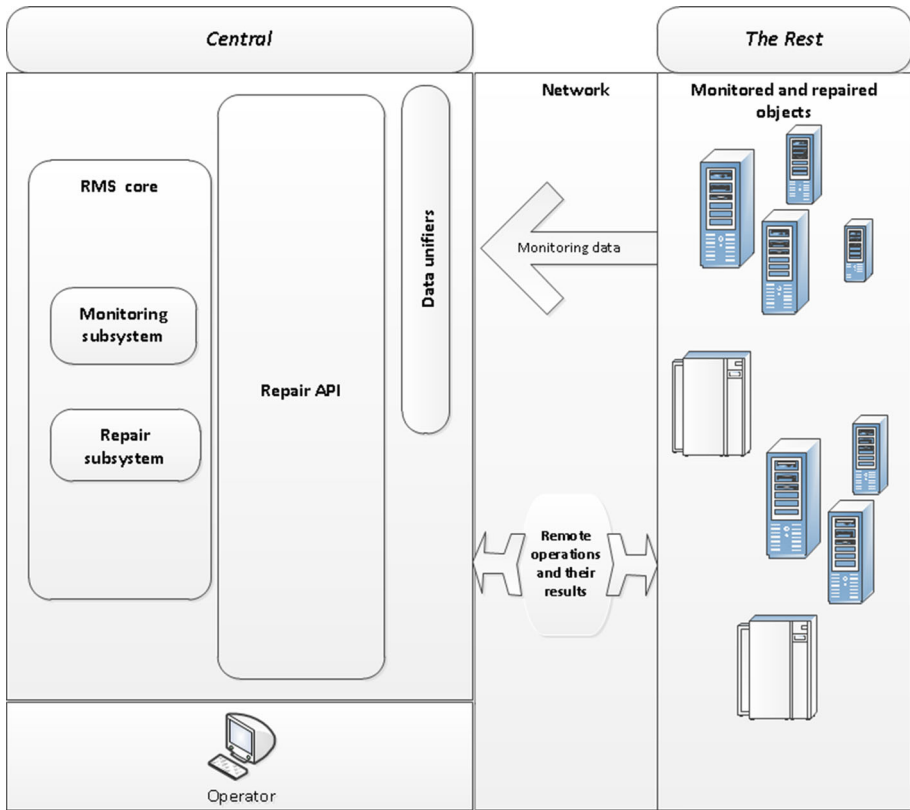


Fig. 7 Architecture of RMS

owned by a customer are located in her/his internal network). In such case, one of the hosts of the internal network is made directly accessible and serves as a gate to other hosts. This situation is shown in Fig. 8.

The structure shown in Fig. 8 is called *connections tree*. It covers all hosts and provides for identification of the shortest access paths. Traversing these paths is transparent to programmers writing the repair procedures. For example, a procedure attempting to connect to the *test machine* (of customer X) resolves the following path:

start from *the RMS network interface*, then connect to *gate* of customer X, then connect to *production server*, then connect to *test machine*,

which results in establishing a connection needed to conduct the repair.

To increase efficiency while distributing the repair steps to their destination machines over the network, the steps are grouped in *actions* and kept in files being the units of distribution. This optimization is shown in Fig. 9.

7 Repair API

The architecture of the repair API is multi-layered (see Fig. 10).

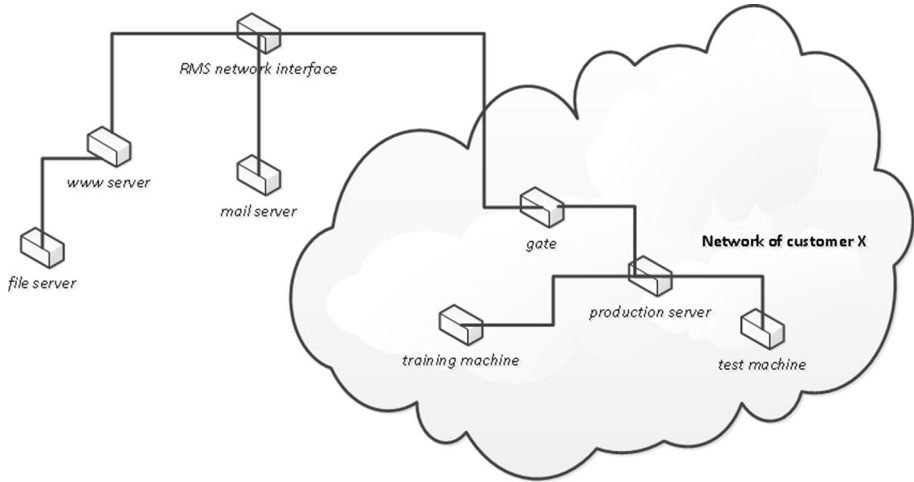


Fig. 8 Accessing a protected subnetwork

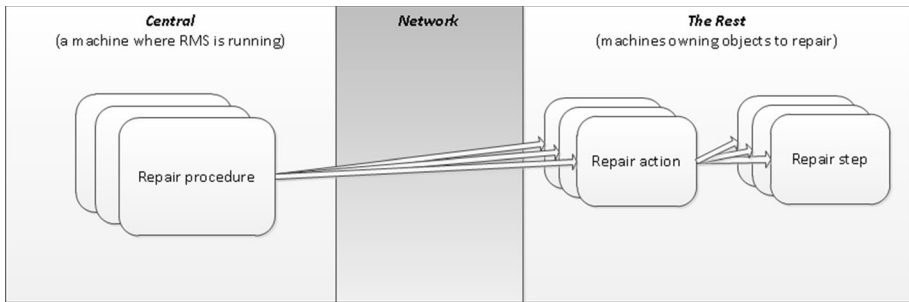


Fig. 9 Splitting repair procedures into actions and steps

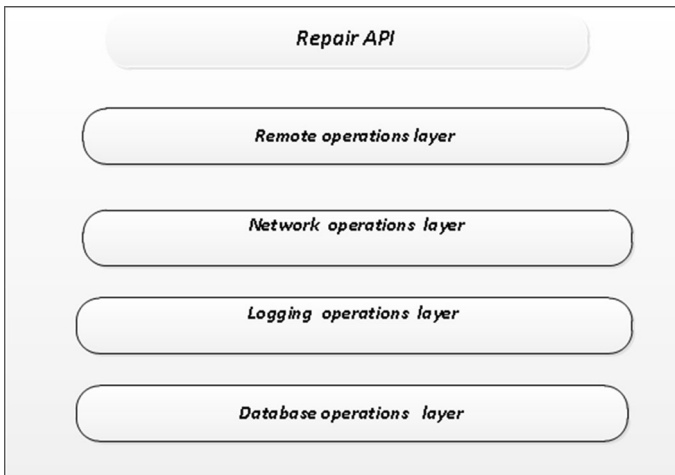


Fig. 10 Architecture of repair API

Only the highest layer (remote operations layer) is accessible to the programmers writing the repair procedures. It provides an interface for programming remote operations. This interface facilitates distribution and running of the repair steps on remote machines.

Other layers are implementation dependent and not accessible to the repair procedure programmers. Examples of the lower-level operations executed by the API implicitly include:

- Registering and unregistering repairs in the RMS database,
- Removing temporary files used for passing bigger arguments,
- Changing states of repairs,
- Terminating failed repairs and changing their status,
- Finishing a multi-threaded repair, after its last thread has terminated.

Operations of the API are logged at six different levels of severity: *FATAL ERROR*, *ERROR*, *WARNING*, *INFO*, *DEBUG* and *LOG* to support debugging of the repair procedures.

These lower layers of repair API will not be further discussed in this paper.

7.1 Remote operations interface

The remote operations layer groups routines accessible to the repair procedure programmers. This layer includes:

Remote operations:

- *putFile*: sending a file to a host,
- *putData*: sending a data item to a host,
- *getFile*: getting a file from a host,
- *getData*: getting a data item from a host,
- *executeStep*: executing a step on a host with optional *local* arguments,
- *rollbackStep*: rolling action performed by a particular step back.

Grouping repair steps in actions:

- *setAction*: establishing an action consisting of several repair steps,

Remote arguments:

- *setArgument*: creating a *global* argument (an argument valid and visible to any step),
- *unsetArgument*: destroying a global argument.

Outside world communication:

- *writeOutput*: sending output by the *running* repair,
- *readInput*: retrieving input by the *suspended* repair.

Monitoring interface:

- *getSubnodesNames*: getting names of all problem-related hosts, containers and subcontainers,
- *getNodeMeasurementTime*: getting time of a measurement of a specified leaf container of the scope,

- *getNodeMeasurementResult*: getting a measurement and its optional details (for instance, the log file related to a failed optimization task of a particular database).

Configuration management:

- *setUser*: setting the user account that will be used to execute the nearest remote operation,
- *setHost*: setting the host that will be used to execute the nearest remote operation,
- *finishRepair*: informing that (and—optionally—how) a repair has been finished.

At any time a running repair can connect with only one `user@host` specified earlier in the calls of *setUser* and *setHost*. To be invoked by calling *executeStep*, a repair step must first be enabled by calling *setAction* with proper parameter.

Repair API maintains namespaces of global and local arguments of repair steps. It also transports bigger arguments over the network and removes their temporary files, after they are no longer needed.

7.2 Repair procedure template

Abstract specifications of the repair procedures (see Sect. 2.2) provide for their reusability in different contexts. They can be implemented in a chosen programming language and deployed in a specific target environment.

The template of a typical repair procedure is given in Fig. 11. The technical details refer to PERL implementation of repair API (it may look slightly different if other programming languages were chosen).

8 Validation

In this section we explain the process of selecting the repair procedures and present the available results related to their effectiveness and efficiency.

8.1 Selection of repair procedures

Based on the analysis of historic data (976 incident reports covering the period of 23 months) from an industrial helpdesk we have identified different categories of helpdesk activities. The summary is presented in Table 1 [the column labelled (%) presents the occurrence rate of a corresponding activity]. The coverage of these activities by the repair procedures presented in this paper (Sect. 2.2) is shown in the columns labelled 1, 2, 3 and 4 (ticks indicate the activities covered by a particular procedure).

The details related to the potential impact and complexity of the repair procedures described in Sect. 2 are given in Table 2.

The complexity of the procedures is characterized by the following metrics: L denotes the size in PERL lines, S denotes the number of the repair steps of the procedure, and U denotes the number of user interactions during procedure execution.

The last column of Table 2 gives estimates of the potential impact on the monitored system(s) of the related repair problem. These estimates were obtained by interviewing the helpdesk operators.

	<i>Expected behaviour of RMS</i>	<i>Repair procedure code</i>
<p><i>Mandatory beginning</i></p> <ul style="list-style-type: none"> -reading a configuration -connecting with RMS database -registering a thread of a repair -reading a scope of a problem into variable \$context <ul style="list-style-type: none"> -reading values of nodes of the monitoring tree into variables <ul style="list-style-type: none"> -indicating action to be executed -choosing a host -choosing a user -setting global arguments for a step <ul style="list-style-type: none"> -executing the step name with local arguments <ul style="list-style-type: none"> -rolling the step name back -writing information about progress of the repair -reading an information from the outer world -sending a data to a defined host -retrieving data from a defined host 		<pre> use modules::api; \$host = getNode(\$context); \$inst = getNode(\$context->{\$host}); ... useAction("fix-small-quota-problem"); setHost(\$host); setUser("root"); setArgument("name1", "value1"); ... setArgument("nameM", "valueM"); executeStep("name", ("nameX" => "valueA", "nameY" => "valueB")); rollbackStep("name", { ... }); writeOutput("line", "or", "lines"); @input = readInput(); putData(@data, "file", "directory"); @data = getData("file", "directory"); unregisterTask("exit status"); OR finishRepair("exit status") </pre>
<p><i>Mandatory ending</i></p> <ul style="list-style-type: none"> -finishing a repair (with an optional exit status) 		

Fig. 11 Template of a typical repair procedure

8.2 Formal specifications and ARM implementation

We have developed formal specifications in Z for all essential functionalities of RMS including:

- The mechanism to invoke and run repair procedures,
- The mechanism of distributing repair steps to their destination hosts,
- The mechanism to execute remote operations (repair steps).

The formal specifications comprise 406 lines in Z and cover 21 data types and 33 Z schemas. In addition, the problem detecting predicates related to the repair procedures of Sect. 2.2 were specified in accordance with the notation given in Sect. 3.1.

The formal specifications were subjected to various analyses with the help of the Z/EVES (Z-EVES 2015; Saaltink 1997, 1999), an interactive tool supporting editing and analysing of Z specifications (the scope of the analyses supported by Z/EVES covers syntax, type and domain checking, schema expansion, precondition calculation, refinement proofs and proving theorems). The advantage of having formal specifications is that they give an abstract definition of ARM which can then be reused while developing implementations of the method for different target environments.

Table 1 Typical generic activities of a helpdesk

Generic helpdesk activities	(%)	1	2	3	4
Restoring the DBMS	19.1		✓		
User accounts management	16.1				
Optimizing database structures	14.2			✓	
Handling data replication	11.1				✓
Dealing with unsuccessful backup	10	✓	✓		
Disc space clean-up	7.6		✓	✓	
Improving performance	5			✓	✓
Massive data modifications	4.7				✓
DBMS failures	2.7		✓		✓
Failures of applications	2.6				
Restoring connections	2		✓		✓
Removing data from database tables	1.7			✓	
Providing technical advice	0.9				
Regular maintenance	0.8	✓	✓	✓	
Handling blocked sessions	0.7				✓
Requests for script activation	0.5	✓	✓	✓	✓
Other	0.3				

Table 2 Details related to the repair procedures described in Sect. 2

Repair procedure	Complexity			Impact
	L	S	U	
1. Repairing corrupted scheduler table	49	3	0	Very high
2. Bringing DBMS to working status	378	8	4	High
3. Optimizing database structures	755	4	1	Medium
4. Emergency replication of time-critical data	356	6	1	High

The verified specifications have been used to develop an actual implementation of ARM and to validate ARM in a real industrial context. ARM was implemented and deployed in the environment where a productive, NAGIOS-based, monitoring system Opsview is running. The monitored systems are related to flight planning. The implementation language for the repair procedures was PERL. In the present implementation the results of monitoring are still interfaced to RMS with the help of system operators. The repair mechanisms of RMS have been fully implemented and are presently in use. The corresponding interfaces for automatic delivery of the monitoring results have been already created and tested and are ready to be used (interfacing monitoring and repair without human intervention is presently a business-level decision and is not constrained by technical issues).

So far, some 16 different generic repair problems (including those described in Sect. 2.1) have been identified and the corresponding repair procedures have been specified and implemented. In addition to the four repair problems already listed in Sect. 2 they include: installing monitoring packages and configuring monitored hosts, user accounts maintenance, preparing lists of created databases and installed versions of flight planning



software, executing SQL queries and/or complex OS scripts/commands, and archiving and downloading remote files.

The sizes of the implemented repair procedures vary from 21 to 755 lines in PERL. The average size is 127 lines, so implementing an already identified and specified repair procedure is not highly resource consuming. Identifying and specifying the repair procedures is not an easy task, however. It needs a significant insight into the monitored systems and the related problems, and essentially it is based on expert judgement. The already implemented repair procedures cover the reoccurring problems which have been identified during the helpdesk exploitation over several years.

8.3 ARM evaluation

The question of particular interest concerns the gain in efficiency resulting from the repair procedures automation. To provide for answering this question we have run experiments and also analysed the historical data, if they were available. The experiments involved:

- Executing each of the four repair procedures described in Sect. 2 in a testing environment (the testing environment is arranged to simulate the production environment and is used to train helpdesk operators),
- Executing the same procedure manually with the same inputs and similar environmental conditions as in the previous case, by a trained helpdesk employee.

In both cases we measured the interval of time needed to complete the procedure. The result was two metrics (measured in seconds): PRT_A —problem resolution time with automatic execution and PRT_M —problem resolution time with manual execution.

In addition to this we examined the historical data (ITIL reports) related to the manual executions of the procedures, if they were available. This resulted in the third metric, PRT_H —problem resolution time derived from historic data.

From these data we have calculated two efficiency gain metrics, EG_M and EG_H , which characterize the improvement of efficiency resulting from repairs automation:

$$EG_M = \frac{PRT_M - PRT_A}{PRT_M} \times 100\%, \quad EG_H = \frac{PRT_H - PRT_A}{PRT_H} \times 100\%$$

From these metrics we have calculated EG_A —the efficiency gain resulting from automation, as the mean value of EG_M and EG_H if both metrics were available (otherwise we assumed that $EG_A = EG_M$). The results are presented in Table 3.

Table 3 Efficiency gain resulting from repairs automation

Repair procedure	PRT_A (s)	PRT_M (s)	PRT_H (s)	EG_M (%)	EG_H (%)	EG_A (%)
1. Repairing corrupted scheduler table	28	240	NA	88	NA	88
2. Bringing DBMS to working status	967	1151	2961	16	67	42
3. Optimizing database structures	116	589	247	80	53	67
4. Emergency replication of time-critical data	108	900	NA	88	NA	88

We have collected the validation data over the period of 1.5 year (since ARM introduction in the first half of 2012). During this period RMS conducted some 5300 repairs (about ten repairs daily). The most commonly used repair procedures include the procedure of changing the scheduler table (executed massively, and in parallel, against multiple productive environments several times per year). Another example is the Opsview's packages installing and updating executed with the frequency of more than 100 times per month.

9 Conclusion

In the paper we presented the ideas and models related to monitoring and repair of IT systems and explained how they have been integrated in the ARM method. The advantage of ARM is that it provides for smooth integration of the existing and proven monitoring mechanisms with the repair processes in a way that reduces the need for human intervention.

The monitoring model of ARM unifies the monitoring data in a way that is independent of the details of specific monitoring systems. This unification provides for application of generic problem detectors which automatically initiate execution of the corresponding repair procedure, if a problem needing a repair intervention occurs.

Configuring the monitoring system is out of the scope of this paper in which we focus on the problem of how to use the measurements to conduct the repairs. Nevertheless, the timing nature of the repair problems should be taken into account while analysing the problems and writing the corresponding repair procedures. The programmer of the procedure can use the *getNodeMeasurementTime* function offered by the repair API (see Sect. 7.1) to check the exact measurement times and can repeat the measurements at the beginning of the procedure body to make sure that the problem really occurs in case the measurements do not meet some specific proximity criteria.

Repair procedures are executed automatically, invoking the necessary repair steps on the remote machines requiring repair actions. Prior to a repair procedure execution, the corresponding repair steps are automatically sent to their destinations. To provide for execution of the repair steps in a distributed environment, the repair API has been defined which provides for integrating the repair steps with the repair procedure body written in a chosen programming language.

The repair procedures define solutions to the repair problems that have already been identified. This identification is not part of ARM. We assume that it results from the experience collected while providing maintenance services to the IT systems of interest.

In general, repair problems are not orthogonal, so it is always possible that while repairing one problem we can affect another problem. Measurements and consequently the problem detecting predicates are just 'sniffing' the problems instead of diagnosing them exactly, and it may happen that two or more predicates become true on the same set of measurements. In the present system this problem is solved by involving a human operator in the decision loop. A possible strategy that could provide further improvement could be to assign priorities to the problems and then repeating the measurements after repairing a given problem to make sure that the remaining problems still appear.

A new procedure is initially specified at a high level of abstraction and is then implemented in a chosen programming language and inserted into RMS. The procedure may communicate with the external world and conduct a repair in an interactive way. This

supports more complex repair scenarios compared to a trivial execution of so-called event handler (in case of an exceptional situation). ARM provides for conducting multi-stage repairs, initiated by a combination of multiple undesired conditions (represented by the corresponding problem detecting predicate). The repair API specification provides for implementing the repair procedures in any modern programming language (in our implementation it was PERL).

In ARM both repair steps and repair procedures need to be programmed. From the repair procedure perspective the repair steps are executed by calling *executeStep* and the effects of a given step can be rolled back by calling *rollbackStep*. Each *executeStep* call returns a diagnostic message informing if the call was successful, so rollback can be initiated from the repair procedure level, if necessary. This rolling back will be effective only if the repair step programmer had foreseen such a possibility by, for instance, making proper backups before the step body is being executed. However, in general the repair steps are not atomic and therefore we cannot be 100 % sure that the rollback will always be successful. For instance, if a given repair step involves a Unix command *mv* which moves files from filesystem A to filesystem B, it may happen that after moving 75 % of files (from A to B) the step fails and simultaneously the filesystem A overflows (for some other reasons). Then the rollback becomes infeasible (there is no room for moving the files from B back to A). At the repair procedure level the programmer still has a chance to repeat measurements and re-evaluate the problem detecting predicate to make sure that the problem has been successfully dealt with.

The architecture of RMS explains how the monitoring and repair subsystems are combined to form a runnable system. A distinguishing feature of RMS is that it can be interfaced to a wide range of practically used monitoring systems and complement them with the repair functionality with a high level of automation.

The main components of ARM, including RMS, repair procedures and repair steps, were specified formally using a proven formal specification method Z. The advantages of having such a formal specification include high level of precision and unambiguity and a possibility of choosing an abstraction level which is free of unnecessary details making the specification implementation independent. This in turn provides for reusability of the specification in case ARM is implemented in a different target environment.

The formal specifications of the ARM components were subjected to verification with the help of the Z/EVES tool. Then the specifications were used to implement ARM in the context of a real helpdesk service to validate the method. The helpdesk provides monitoring and repair services for systems of varying criticality, including calculation of flight parameters for international airlines. Initially, the validation was carried out in a testing environment (parallel to the production environment of the helpdesk). Then, gradually, the thoroughly tested repair procedures were moved to the production environment of the helpdesk.

The experimentally validated benefits resulting from ARM include:

- Efficiency gains in repair procedures execution (in case of the procedures described in this paper the gain was from 42 to 88 %).
- New repair procedures can be included following the well-defined steps of ARM.
- Repairs can be conducted in parallel which saves time.
- In many cases a repair can start automatically and immediately after detecting the related problem which reduces the time to repair comparing the manual integration of the monitoring and repair processes.



- Repair procedures are more reliable. Formal specifications help to avoid and detect errors preventing them from occurring in a repair procedure code.
- Formalization of repairs provides for more effective and efficient communication and training of the personnel.
- Easy integration with the existing systems. ARM fits well into the existing industrial reality and extends it with new functionality and potential of automation. ARM does not require changes in the existing monitoring infrastructure.
- RMS can be also used as a stand-alone (i.e. separated from the monitoring functionality) repair system which performs complex remote tasks (such as administrative tasks) and helps in their automation. In such case the triggering signals are sent manually by the operators.
- The monitoring components can be easily exchanged (provided they conform to the monitoring model of ARM). This facilitates deployment of ARM in different monitoring contexts.

We see limitations of ARM in the following:

- Designing a complex repair procedure requires extensive domain expertise regarding the related problem as well as the target environment of the procedure.
- Implementing a repair procedure requires sufficient programming skills (in the chosen programming language).
- A faulty procedure can cause severe harm. A possibility of automation should not result in quality assurance relaxation, and human operators should be involved in the most critical decisions (even if the procedure could have been executed fully automatically).
- Too much automation can bring the additional risk of reducing attention and the erosion of competencies of human operators. To some extent it can be remedied by regular training.
- Agentless architecture and public-key infrastructure chosen in our implementation (not discussed in this paper) bring a security risk that breaking into a machine running RMS facilitates breaking into the dependant systems, supervised by RMS. Installing RMS in a secure network (VPN), behind firewalls and limiting connectivity helps to reduce this risk.
- Connectivity overhead is yet another issue as execution of remote operations needs additional resources. Although we strived to reduce these overheads in our implementation (for instance, by reducing the communication needed to invoke external routines), we still see a potential for improvement in this respect.

Acknowledgments This work was partially supported by the statutory Grant B01/030765/K05-204/1/K01 by Polish Ministry of Science and Higher Education.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

About.com. (2015). Hop definition. http://compnetworking.about.com/od/networkdesign//bldef_hop.htm. Accessed August 2015.



- Albrand, S., Düllman, D., & Millar, P. (2005). *Metadata monitoring requirements*, GridPP (pp. 3–5). <http://www.gridpp.ac.uk/datamanagement/metadata/Documents/MonitoringReq.pdf>. Accessed August 2015.
- Ansible. (2015). (Open Source IT automation software). <http://www.ansible.com/home>. Accessed August 2015.
- Aslett, M. (2006). Open management, TechNews. <http://www.netdotwork.co.za/article.aspx?pkarticleid=3962>. Accessed August 2015.
- Barth, W. (2008). *Nagios. System and network monitoring* (2nd ed., pp. 619–623). San Francisco: No Starch Press.
- Bauer, K., & Campi, N. (2009). *Automating Linux and Unix system administration*. New York: Apress.
- Cacheda, F., Carneiro, V., Formoso, V., & Valiño, J. (2007). Rembassy: Open source tool for network monitoring. In Proceedings of the IADIS international conference on telecommunications, networks and systems (pp. 67–74), July 03–05, Lisbon: IADIS Press.
- CA technologies. (2006). Ingres MDB housekeeping recommendations, New York. http://supportconnectw.ca.com/public/impcd/r11/MDBMain/MDB_Housekeeping_Ingres.htm. Accessed August 2015.
- Ceccanti A., & Panzieri F. (2004). Content-based monitoring in grid environments. In Proceedings of the 13th IEEE international workshops on enabling technologies: Infrastructure for collaborative enterprises (pp. 255–259), June 14–16. Modena: University of Modena and Reggio Emilia, IEEE.
- Cottrell, L. (2000). *Network monitoring tools*. California: Stanford University. <http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html>. Accessed February 2015.
- Davies, J., & Woodcock, J. (1996). *Using Z—specification, refinement, and proof* (pp. 1–5). New Jersey: Prentice Hall.
- Gerlan, D., Shang-Wen, C., & Schmerl, B. (2003). *Increasing system dependability through architecture-based self-repair, architecting dependable systems* (pp. 61–89). Berlin: Springer.
- Goettsche, L. D. (2005). *Maintenance of instruments and systems*. North Carolina: ISA.
- Ingres Corporation. (2006). *Ingres 2006 System Administrator Guide* (p. 35), California. <http://downloads.actian.com/download/sysadm.pdf>.
- Jones, B. (2004). SysAdmin to SysAdmin: Service monitoring with Nagios, Linux.com. <http://www.linux.com/archive/articles/113803>. Accessed August 2015.
- Josephsen, D. (2007). *Building a monitoring infrastructure with Nagios* (pp. 23–26). New Jersey: Prentice Hall.
- Opsview. (2015). *Puppet module*. <http://www.opsview.com/resources/downloads/puppet-module>. Accessed August 2015.
- Pervilä, M. (2007). Using Nagios to monitor faults in a self-healing environment. In *Spring seminar on self-healing systems* (pp. 1–6). Helsinki: University of Helsinki. <http://www.cs.helsinki.fi/u/niklande/opetus/SemK07/paper/pervila.pdf>. Accessed August 2015.
- Puppet. (2015). (Open Source IT automation software). <http://puppetlabs.com/puppet/what-is-puppet>. Accessed August 2015.
- Radič, B., & Škiljan, Z. (2004). *Monitoring systems: Concepts and tools*. In Proceedings of the 6th CARNET users conference, Zagreb: CARNET. http://cuc.carnet.hr/cuc2004/program/radovi/f3_radic/f3_full.pdf. Accessed February 2015.
- Retkowski G. (2006). *Building a self-healing network*. California: O'Reilly—ONLamp.com Portal. <http://onlamp.com/lpt/a/6589>. Accessed August 2015.
- Saaltink, M. (1997). *The Z/EVES system. Lecture notes in computer science* (Vol. 1212, pp. 72–85). Berlin: Springer.
- Saaltink M. (1999). *The Z/EVES 2.0 User's Guide*. Ottawa: ORA Canada. <http://www.cs.york.ac.uk/ftplib/pub/leo/mefes/zeves/manuals/userguide.pdf>. Accessed August 2015.
- Sakellariou, R., & Zaniolas, S. (2005). A taxonomy of grid monitoring systems. *Journal of Future Generation Computer Systems*, 21(1), 163–188.
- Savage, S. (1999). Sting: A TCP-based Network Measurement Tool. In *Proceedings of the 2nd USENIX symposium on internet technologies and systems* (pp. 71–80), October 11–14. California: USENIX Association.
- Sellens, J. (2000). System and network monitoring. *Login: The Magazine of USENIX and SAGE* 25(5). California: USENIX Association. <https://www.usenix.org/system/files/login/articles/monitoring2000-8.pdf>. Accessed August 2015.
- The PostgreSQL Global Development Group. (2013). *PostgreSQL 8.3.23 documentation*. <http://www.postgresql.org/files/documentation/pdf/8.3/postgresql-8.3-US.pdf>. Accessed August 2015.
- Turnbull, J. (2006). *Pro Nagios 2.0*, Apress, New York, USA 2006.
- Whalen, E. (2006). *Oracle database 10g Linux administration*. California: McGraw-Hill Professional. 2006.
- Zabbix. (2008). *Zabbix Manual v1.6* (pp. 36–38, 159–161). <http://www.zabbix.com/downloads/ZABBIX%20Manual%20v1.6.pdf>. Accessed August 2015.
- Z-EVES. (2015). <http://www.z-eves.software.informer.com/>. Accessed August 2015.



Janusz Górski is a professor at the Department of Software Engineering, Gdansk University of Technology, in Poland where he is leading the Information Assurance Group (IAG) <http://iag.pg.gda.pl/iag/>. His present interests are in trust, safety and security, software engineering, software quality and information assurance. For several years he was serving as a member ENISA (European Network and Information Security Agency) Permanent Stakeholders Group providing advice to the Director of the Agency. He is also providing expertise to EC Joint Research Centre in Ispra on critical infrastructures protection and offers consultancy to companies on information security, software quality assurance and risk management. He is a cofounder of Argevide company bringing the results of the IAG research to the market.



Marek Kaminski is a senior database administrator in an international IT company providing aeronautical services worldwide. He has over 10 years of experience in supporting productive IT systems. His work is focused on maintenance of aeronautical operations. In this domain he obtained his Ph.D. from Gdansk University of Technology in Poland where he also was involved in teaching activities. He has published a number of research papers covering different aspects of monitoring and repair of IT systems. In his private life he is a husband and a father of two children, and in a free time he enjoys walking, biking and building and programming in NXC his own contraptions with Lego Mindstorms bricks. He is also interested in a Christian apologetics.