Ensuring Continuous Data Accuracy in AISEMA Systems

Irina Diana Coman, Alberto Sillitti, Giancarlo Succi
Center for Applied Software Engineering
Free University of Bolzano
Bolzano, Italy
{IrinaDiana.Coman, Alberto.Sillitti, Giancarlo.Succi}@unibz.it

Abstract — Automated In-process Software Engineering Measurement and Analysis (AISEMA) systems are powerful tools to monitor and improve the software development process. However, to be useful, it is required that such tools never stop working. Therefore, they need the support of advanced monitoring systems able to detect and locate malfunctions and inform automatically human operators providing all the information required to solve the problem. This paper describes the approach and the tools developed to support a specific AISEMA system developed to support both managers and developers in implementing continuous process improvement initiatives.

AISEMA; development process; monitoring.

I. INTRODUCTION

The success of software measurement programs is strongly dependant on the automation of the related data collection (Pfleeger, 1993; Daskalantonakis, 1992; Offen and Jeffery, 1997; Hall and Fenton, 1997; Iversen and Mathiassen, 2000). Manual data collection suffers from several limitations including: it is time consuming, tedious, error prone, and often biased or delayed (Johnson and Disney, 1999). Semi-automated data collection is better (tools such as LEAP (Moore, 1999)) but there are still context switching problems (Johnson et al., 2003) with a negative impact on the performance of the developers since it requires to switch continuously between working activities and data collection.

A new generation of tools (such as PROM (Sillitti et al., 2003) and Hackystat (Johnson et al., 2003)) has been developed to overcome such limitations providing a fully automated, non-invasive data collection. Such tools allow data collected from on-going projects to be used for improvement of the same project, therefore they are also called Automated In-process Software Engineering Measurement and Analysis (AISEMA) systems.

AISEMA systems aim at automatically collecting the data, but also at providing tailored analyses for decision support. They reduce the cost of data collection, as they run in the background and let people focus on their work without any additional workload or distractions. They can collect a large variety of data. Based on these data, they propose: support for process management (Remencius et al., 2009; Danovaro et al., 2008), assessment of low-level processes (Coman and Sillitti, 2009), etc.

Ensuring continuous data accuracy is one of the main challenges during the usage of an AISEMA system (Coman et al., 2009). The changes in the environment (such as software updates, software crashes, hardware failures, changes in security policies, etc.) affected sometimes the accuracy of data, mainly by disabling some of the data collection components, hindering data transfer or causing data loss. Not all such events are avoidable. Consequently, small amounts of data might be lost from time to time. However, it is very important to limit as much as possible these missing data and to have detailed information on the cause why they are missing and on their type. Such information helps to assess whether the missing data invalidate or not a specific analysis, thus ensuring reliable results of data analyses.

In most of the cases, existing systems have already error prevention mechanisms located at each of theirs components. This is the case of the PROM system (Sillitti et al., 2003). Such mechanisms ensure a good functioning of the components. However, they cannot prevent, for instance, a silent disable of the components as result of repeated crashes of the host system. Moreover, in some cases, the disabling of the components is perfectly acceptable (for instance, when a developer chooses not to collect some specific data). Thus, to assess whether the disabled status of a component represents a failure of the system or not, additional context information is needed.

In PROM, as the components interact with the server over the network medium, the correct functioning of the system as a whole does not depend only on the correct functioning of each individual component. Because the client components are not aware of their broader context (and are not meant to be aware), there is a need for a separate component that monitors the functioning of the system as a whole. Such component should identify potential problems and use local information from specific components to localize the actual cause of the problem. Additionally, such component should log the occurrence of the problem and, if the solution is known, to proceed with solving the problem. If the solution is unknown, the component should notify the maintainers of the system.

The initial solution used during most of the case-study was to have a human performing such monitoring. However, this is
extremely time consuming and costly. Moreover, ideally, the monitoring should be continuous and thus requires an automated solution. The solution developed was to enhance the existing AISEMA system with characteristics of autonomic computing (Horn, 2001; Kephart and Chess, 2003) such as self-monitoring and self-healing. This paper presents in detail the concrete approach and its implementation. It is organized as follows: Section 2 presents the related work; Section 3 introduces our approach; Section 4 describes the proposed implementation; Section 5 summarizes the results obtained; finally, Section 6 draws the conclusions and presents future work.

II. RELATED WORK

The initial manifesto of autonomic computing (Horn, 2001) proposed it as the single approach able to mitigate the effects of an increasingly acute software complexity crisis. The manifesto pointed out that the complexity of the IT infrastructure grows constantly and threatens to get beyond human ability to manage it. Continuously increasing number of connections, dependencies and interactions between software components make the maintenance and management of the software systems increasingly complex, up to the point of surpassing human ability of managing these tasks. Thus, the only solution resides in building software systems that are able to configure and manage themselves, adapting to changing environments and adjusting their behavior for most efficient use of their resources to solve the required tasks.

Autonomic computing identifies four key aspects that self-managing software should possess: self-configuration, self-optimization, self-healing, and self-protection. Of these four, the self-healing is the main aspect that AISEMA systems should have to ensure continuous data accuracy.

Self-healing is defined as the capability of a software system to automatically detect, diagnose, and repair localized problems resulting from bugs or failures in software and hardware (Kephart and Chess, 2003). Koopman (2003) proposes a taxonomy that identifies four main aspects of a self-healing strategy: fault model, system response, system completeness, and design context.

The fault model (also called a fault hypothesis) identifies the faults that the system should be able to tolerate. The system response represents the strategy of the system to detect and treat faults. The system completeness and design context include the factors that influence the scope of self-healing capabilities.

Each of these aspects can be further described by a set of properties. Table I gives an overview of the characteristics of our approach (PROM) in terms of the aspects and properties defined in the taxonomy of Koopman.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Property Property</th>
<th>PROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault model</td>
<td>Fault duration</td>
<td>Permanent</td>
</tr>
<tr>
<td></td>
<td>Fault manifestation</td>
<td>Corrupted or missing data</td>
</tr>
</tbody>
</table>

TABLE I. CHARACTERISTICS OF THE APPROACH IN TERMS OF A TAXONOMY FOR SELF-HEALING PROBLEM SPACES


There are already many projects exploring various self-healing strategies (Hoover et al., 2001; Raz et al., 2002a; Shelton et al., 2003; Shehory, 2006; Mikic-Rakic et al., 2002). The proposed approaches focus on reconfigurable architecture (Shehory, 2006; Mikic-Rakic et al., 2002), graceful degradation (Shelton et al., 2003), optimized usage of resources (Hoover et al., 2001), or inferring specifications from underspecified components (Raz et al., 2002a). Most of the proposed approaches focus on distributed systems and address software failures in one or more of the components. By contrast, the approach proposed here focuses on failures due to environment factors such as disable of components or loss of connectivity. The proposed approach makes also use of the specificities of the domain, namely the characteristics of the automatically collected data.

Among the existing self-healing approaches, one that is very similar to the approach presented here is that of Raz et al. (2002a). They focus also on detecting failures by searching for semantic anomalies in the data. In their case, the data are dynamic data feeds that change outside user control and are usually under-specified. Their approach has two phases: setup...
During setup, the approach aims at inferring invariants from data presented as input. These invariants are used during the usage phase to detect anomalies in the data. During the setup phase, human intervention is required to find a training set size, establish parameter values, select attributes to take into account, and select from the proposed invariants. During the usage phase, human intervention is also desired to eliminate from the results the false positives (normal data falsely declared as anomalous).

Similarly to the approach of Raz et al., the approach proposed here also detects failures by searching for anomalies in the data. However, the domain is very different and it allows to take one step further by taking action to correct faults based on the detected anomalies.

By contrast to the approach of Raz et al., the specifications of the data are very well known as the data come from the components of the system itself. Thus, data invariants can be defined by the administrators of the system, together with the possible causes and steps to take for restoring proper functioning. Moreover, the definition of data invariants can also take advantage of overlapping data coming from different sources. Once the invariants have been defined, the system can perform fully automatically the cycle of fault detection, cause identification, and fault resolution. Human intervention might be required again only in cases when none of the proposed solutions solve the problem.

Another contribution of the approach proposed here is a model for defining data invariants together with potential causes of violation and actions to take for resolution of the violation. The model also allows the combination of basic invariants to easily specify more complex ones while hiding the complexity of the definition. While the current implementation expects predefined data invariants, the approach can be easily extended with automated inference of data invariants.

III. THE APPROACH

The main goal is to ensure continuous high accuracy of automatically collected data. The approach proposed here is to perform regularly a set of checks on the collected data to detect anomalies and then to take action to prevent further anomalous data. Anomalies in the data indicate a problem in the functioning of the system. When anomalous data are detected, the system investigates several possible causes to localize the actual problem. To do so, it performs a set of tests regarding the status of the network and of the components installed on the client machine from which data originated. After identifying the cause(s), the system proceeds to heal itself. Depending on the actual cause, the actions taken can consist in warm or cold reboot of components on the client machine, restoring corrupted files or notifying the user that user action is needed. Moreover, the system logs all checks and tests performed and their results to provide a complete view on the status of the data at any given time.

An anomaly is defined as a violation of one or more data invariants. A data invariant expresses a condition that all accurate data should satisfy. The definition of data invariants is based on domain knowledge. As the collected data are well defined and the system specifications are known, the easiest approach is to use such knowledge for data invariant definition.

Thus, simple data invariants can define the domain of values for each type of data collected (for instance, the time spent in some activity should be positive, and below 24 hours during a single day), the required fields (according to the known data structure), or the frequency with which they are collected (according to the specifications of the system).

The above type of invariants describes mainly the structural properties of data, rather than the semantic ones. The violation of such an invariant can help identify severe problems (such as no network connection or hardware failure) but do not reveal more subtle issues (such as data from one component with legal but incorrect values). Additional data invariants that define relations in the data can help detect also semantic anomalies. Such invariants are usually called adaptive invariants to distinguish them from invariants that impose certain specific values rather than relations.

The automatically collected data come from various components. As such, the data usually overlap to some degree. Data invariants that make use of these overlaps are used to identify anomalies in data coming from one component. Moreover, as different data represent different views on the same artifact, there are some relations in the data that should be respected. Such relations that should hold at any time are usually called adaptive invariants. The violation of an adaptive invariant means that the data are not semantically consistent.

A. The Data Invariant Model

The definition of an invariant contains the data property or the data relation that has to be ensured, a description of the problem to be reported if the invariant is violated, and a list of the potential causes of the identified problem. The actual computation of a data invariant can be a direct check of a data property, or it can build on the results obtained on checking several other data invariants. This gives particular strength to the model as it allows the definition of extremely complex invariants while making their definition easy by hiding their computational complexity from the user.

The self-healing ability requires not only that anomalous data are detected, but also that the cause of anomaly is removed. There is not a one-to-one relation between anomalies and causes. The violation of a data invariant can be due to one or more of the causes listed for that invariant, or even to a new cause, not yet identified. Thus, each cause is defined together with a set of tests that should fail only when the cause is present. Such tests usually concern local conditions on the machine which generated anomalous data. However, the only limits to what such tests check depend only on the possibility of actually performing the test.

The various tests associated with a potential cause is useful not only to identify the presence of a cause, but also to distinguish between malign and benign manifestations of the same cause. The benign manifestation of a cause means a false-positive in the sense that the violation of the adaptive invariant is not due to a problem in the functioning of the system, but rather to special circumstances (for instance no upload of data...
due to user being on a sick leave). The malign manifestation of a cause represents a real failure of the system, one of its components or the network medium.

Finally, each test has to define also a list of actions to take if the test fails. Such actions should remove the cause of anomalous data and should restore the proper functioning of the system.

Thus, an invariant’s definition should contain such information as shown in Table II. Table III shows an example of a simple data invariant and Table IV provides an example of the definition of one of the possible causes of violation of such an invariant.

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
<th>Intended user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Short, descriptive name for the data invariant. Helps in reports for</td>
<td>Human</td>
</tr>
<tr>
<td></td>
<td>administrators.</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>A more detailed description of the invariant and, if needed, the reasons</td>
<td>Human</td>
</tr>
<tr>
<td></td>
<td>justifying it.</td>
<td></td>
</tr>
<tr>
<td>Computation</td>
<td>The actual function or formula that returns true is the invariant is</td>
<td>System</td>
</tr>
<tr>
<td></td>
<td>satisfied and false if it is violated.</td>
<td></td>
</tr>
<tr>
<td>Problem reported</td>
<td>A description of the problem identified by a violation of the invariant.</td>
<td>Human</td>
</tr>
<tr>
<td>if the invariant is</td>
<td></td>
<td></td>
</tr>
<tr>
<td>violated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>List of possible</td>
<td>A list with the possible causes of the invariant’s violation</td>
<td>System</td>
</tr>
<tr>
<td>cause(s)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE III. EXAMPLE OF A DATA INVARIANT’S DEFINITION**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data upload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>For each user, there should be at least one upload of data each day.</td>
</tr>
<tr>
<td>Computation</td>
<td>AnyUploadCheck (method)</td>
</tr>
<tr>
<td>Problem</td>
<td>NO uploads during the day.</td>
</tr>
<tr>
<td>Possible causes</td>
<td>1. AISEMA system inactive on user’s machine.</td>
</tr>
<tr>
<td></td>
<td>2. Transfer scheduler NOT running.</td>
</tr>
<tr>
<td></td>
<td>3. Corrupted configuration of connection on user’s machine.</td>
</tr>
<tr>
<td></td>
<td>4. Server component down.</td>
</tr>
<tr>
<td></td>
<td>5. Server unreachable from client machine.</td>
</tr>
</tbody>
</table>

**TABLE IV. EXAMPLE OF A POSSIBLE CAUSE’S DEFINITION**

<table>
<thead>
<tr>
<th>Cause</th>
<th>Test</th>
<th>Type</th>
<th>Action to take if test fails</th>
</tr>
</thead>
<tbody>
<tr>
<td>AISEMA system inactive on user’s</td>
<td>Check flag of system activity to be set to</td>
<td>Benign</td>
<td>Just annotate that the user has deactivated the system on his/her</td>
</tr>
<tr>
<td>machine</td>
<td>active.</td>
<td></td>
<td>machine.</td>
</tr>
<tr>
<td></td>
<td>Check that data are collected on user’s</td>
<td>Malign</td>
<td>Warm reboot of the client components.</td>
</tr>
<tr>
<td></td>
<td>machine (there are files stored).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IV. IMPLEMENTATION**

The approach is implemented in two components: PROM Data Inspector (server-side component) and PROM Console (client-side component). Both components are written in Java. PROM Data Inspector implements the actual problem detection and takes steps to solve the identified issues. PROM Console provides additional information on the status of the clients when needed and it propagates to the client the actions recommended by PROM Data Inspector to solve the identified problems. Figure 1 shows the relations between the existing architecture of PROM and these two components.

The definitions of data invariants are stored in the PROM database. At present, the implementation of the formula of the invariant is in a separate module of the Data Inspector. However, PROM Data Inspector can be easily extended to use formulas implemented in external modules. This makes the actual structure of the data storage transparent to the Data Inspector, ensuring that the self-healing mechanism is independent from the way in which data are stored.

Data invariants can be of two types: basic and summary invariants. The basic invariants have a simple, single data property to check. They are usually meant for testing fixed data invariants rather than adaptive invariants that require more complex checks of data relations. For such checks, the summary invariants are more appropriate. A summary invariant has a computation formula that usually involves other basic invariants. This allows to check for complex data relations. Moreover, if desired, several distinct invariants can be defined to allow a more precise identification of the actual cause of violation.

PROM Data Inspector runs at regular intervals, loads the invariants from the database and performs a check of all defined data invariants. For any invariant that is violated, the component goes through the list of possible causes and runs the associated tests. For every failed test, the corresponding action is taken and its result is stored. If the execution of the action fails, an error notification is sent to the administrators of the system. After successful execution of an action, the system performs again the test that had previously failed. However, the system has to wait until the next round of overall check of data invariants to be sure that it solved the problem that was actually causing anomalous data. Thus, the overall algorithm has the following steps:

1. Check all data invariants and store results.

Figure 1. PROM Architecture and the relationship with PROM Data Inspector and PROM Console
2. For each data invariant that was violated:
   a. For each possible cause:
      i. Perform all associated tests
         1. If a test fails, run the corresponding action;
         2. If the action completed successfully, perform again the test;
         3. If the action could not be run or the test failed again after action has been taken report to the administrators.
   b. If all tests of all possible causes passed from the first run (i.e., no cause has been identified as responsible for the observed violation), notify the administrators.

The PROM Data Inspector stores all the results of the check of invariants in the database for future reference, together with a timestamp. Optionally, it can also generate a report and send it by email to the administrators. The detailed results of the checks performed can be used for having a clear view on the status of the data at any given time. The Data Inspector ensures the compatibility with the graphical interface PEM (PROM Experience Manager), so that the status and results of all tests can be displayed in a chart allowing for fast identification of days and users with problematic data.

The PROM Console resides on the client machine and communicates with the Data Inspector on the server. It gathers information about the local context, such as the PROM components installed, the state of their configuration files and the data that are collected. It sends such data to the Data Inspector which uses them to perform the tests for cause identification. Upon request from the Data Inspector, PROM Console notifies the user that some specific action needs to be taken, or it implements directly non-invasive actions such as restoring of corrupted configuration files.

V. Results

PROM Data Inspector and PROM Console were in usage during the last 2 months of data collection in the an experiment in a company (Coman et al., 2009). In a first step, Data Inspector represented mainly an automation of the data checks that were previously performed manually. However, it soon became obvious that it can also perform more complex checks that are very tedious to be performed manually. Thus, the summary invariants were introduced, allowing the combination of many basic invariants.

The initial reports of the PROM Data Inspector were just listing all the tests performed together with the results obtained. However, such reports were hard to read without knowing the inner functioning of Data Inspector. The current version addressed this problem by providing at the beginning summaries of the identified problems, together with the possible causes. This new form of reports proved to be understandable also by people that were not directly involved in the definition of data invariants or in the development of the Data Inspector. Figure 2 shows an example of the contents of a report generated for a single user showing an identified problem and listing the possible causes. In the report, each test represents in fact a data invariant. This change of name makes the report more intuitive to people and does not require users to know what a data invariant is.

```
1. Problem: NO uploads during the day for users:
   John (68)
   Possible causes:
   PROM inactive on user's machine;
   Transfer Scheduler NOT running;
   PROM server down or unreachable;
   wrong configuration of the Transfer on the user's machine

FAILED TESTS:
John (68): CheckClearedEffort: 8<=0
John (68): CheckPlugIns: trace-NVD/Plugins=NO;devmen.exe=1D
John (68): CheckUploades: last upload:NONE before last upload:NONE
John (68): CheckStoryPoints: source1=() and source2=(12452=8)

SUCCESSED TESTS:
John (68): CheckClearedEffort: 0/108800
John (68): CheckClearedCompileTime: compile_time=0 and VS_effort<=$

TESTS NOT RUN:
NONE
```

Figure 2. Example of PROM Data Inspector report showing a detected data anomaly

VI. Conclusions and Future Work

This paper reported on the first step towards transforming AISEMA systems into autonomous systems, by ensuring self-healing capabilities. The proposed approach is to detect data anomalies and to trace them back to the software or hardware failure that caused them. The data anomalies are modeled as violations of data invariants. The data invariants are based on the knowledge of the system and of the data collected. They can be very simple (e.g., time recorded should always be positive) or more complex, combining several basic invariants for testing of relations into the data.

To solve the issues that cause data anomalies, each data invariant contains also a list of possible causes. Each cause has a list of tests that help identify the exact problem and whether it is benign (not a failure but special circumstances) or malign (a failure that has to be solved). For each test there is also a list of actions to be taken when the test fails. This model ensures flexibility (new invariants can be added at any time), simplicity (the complexity of invariants can be hidden from the user by building on underlying levels of invariants), and traceability of issues that affected the collected data (all actions are clearly defined and the results are always stored).

The current implementation uses only user-defined invariants. As future work, it would be valuable to explore the possibilities of automatically extracting invariants from previously collected data. Techniques such as the one used in Daikon (Ernst et al., 2001) or Mean (Raz et al., 2002b) have already been applied to automated detection of data invariants in online data feeds. However, applying such techniques is not
straight forward, as the characteristics of the AISEMA data are quite different from those of online data feeds (i.e., most of the data are not normally distributed, the volume of data is bigger, complex relations between various types of data have to be taken into account).

To preserve the character of self-healing of the proposed approach, an automated detection of data invariants should also be complemented by an automated detection of possible causes and of appropriate measures to be taken for each identified cause. However, this is still an open issue.

REFERENCES


