A Framework Based on Learning Techniques for Decision-making in Self-adaptive Software

Frank José Affonso, Gustavo Leite
Dept. of Statistics, Applied Mathematics and Computation
Univ Estadual Paulista - UNESP
Rio Claro, SP, Brazil
frank@rc.unesp.br, gustavoleite.ti@gmail.com

Rafael A. P. Oliveira, Elisa Yumi Nakagawa
Dept. of Computer Systems
University of São Paulo - USP
São Carlos, SP, Brazil
{rpaes, elisa}@icmc.usp.br

***Abstract*—The development of Self-adaptive Software (SaS) presents specific innovative features compared to traditional ones since this type of software constantly deals with structural and/or behavioral changes at runtime. Capabilities of human administration are showing a decrease in relative effectiveness, since some tasks have been difficult to manage introducing potential problems, such as change management and simple human error. Self-healing systems, a system class of SaS, have emerged as a feasible solution in contrast to management complexity, since such system often combines machine learning techniques with control loops to reduce the number of situations requiring human intervention. This paper presents a framework based on learning techniques and the control loop (MAPE-K) to support the decision-making activity for SaS. In addition, it is noteworthy that this framework is part of a wider project developed by the authors of this paper in previous work (i.e., reference architecture for SaS [1]). Aiming to present the viability of our framework, we have conducted a case study using a flight plan module for Unmanned Aerial Vehicles. The results have shown an environment accuracy of about 80%, enabling us to project good perspectives of contribution to the SaS area and other domains of software systems, and enabling knowledge sharing and technology transfer from academia to industry.**

***Keywords*-Self-adaptive software; Reference Architecture; Framework; Learning Techniques; Decision-making.**

I. Introduction

Over recent years, one has observed a significant increase in the complexity of software systems and their computational environments. In general, such systems share functional, nonfunctional, physical, and virtual requirements. The human ability to manage systems has shown as inadequate when their complexity increases. Moreover, involuntary injection of faults has often configured as one of the major causes of system failures (especially in the context of Self-adaptive Software – SaS). In SaS, the design decisions are moved towards runtime to control dynamic behavior and individual reasons of such systems about their states and environments.

Reference Architectures (RAs) refer to a special class of software architecture that have become an important element to systematically reuse architectural knowledge [2], [3]. Thus, in previous work [1], [4] we have proposed Reference Architecture for SaS (RA4SaS) – an architecture that provides a guideline set for SaS development and an automated approach for self-adaptation of the software entities[1] at runtime without human intervention.

Based on the presented context aimed at improving the quality of development processes for SaS, this paper presents a framework based on learning techniques (classifiers and association rules) [5] and the MAPE-K (Monitor, Analyze, Plan, Execute over Knowledge base) [6], [7] control loop for decision-making in SaS. The main purpose of this framework is to classify and analyze sensory data to autonomously detect and mitigate faults at runtime. Thus, we believe that the needs for systems to interface with human administrators may be reduced, alleviating operational-human costs and, ideally, improving upon existing mitigation techniques. Moreover, based on the preliminary results, we believe that our framework may be used in the knowledge management of other types of software systems. For instance, we have applied this framework in the monitoring and eventual corrections of flight plan for Unmanned Aerial Vehicles (UAVs).

In this context, the primary propose of this paper is to supply the industry with supporting strategies to systematize and automate the functionalities of SaS, contributions from Software Engineering (SE) and Knowledge Engineering (KE) are necessary. Other contributions are: (1) the evaluation of a solution for the problem of classifying and recommending solutions at runtime; (2) a flexible strategy for SaS modeling; and (3) regarding the adaptive module, a feasible strategy to rebuild classifiers and rules from specific points where they were interrupted.

Following the introduction, this paper is organized as follows: Section II presents the background and some related work associated to our study; Section III provides a description of RA4SaS and the framework for decision-making for SaS; a case study designed to validate our approach is presented in Section IV; and finally, Section V summarizes our findings, conclusions, and perspectives for further research.

II. Background and Related Work

This section presents the background (i.e., standard concepts and definitions on SaS and RA) and related work on our study. SaS has specific features in comparison to traditional systems since this type of software system constantly deals with adaptations at runtime, fixing new needs of both users and/or execution environment. Moreover, the SaS development has

[1] From this point onwards, SaS may be also referred to as software entities or simply entities.

(DOI reference number: 10.18293/SEKE2015-125)

boosted self-⋆ properties in general-purpose software systems, such as self-managing, self-configuring, self-organizing, self-protecting, self-healing, and so on. These properties allow systems to automatically react against users' needs or to respond as soon as these systems meet execution environment changes [3], [8], [9], [10].

RA is a special type of architecture that provides major guidelines for the specification of concrete architectures of a class of systems [11]. Some studies [12], [13], [14], [15] have established different investigations to systematize the design of such architectures, guidelines, and processes. Moreover, the effective knowledge reuse of RA depends not only on raising the domain knowledge, but also documenting and communicating this knowledge efficiently through an adequate architectural description. Commonly, architectural views have been used, together with UML (Unified Modeling Language) techniques, to describe RAs. Considering their relevance as the basis of the software development, a diversity of RAs has been proposed and used, including for (self-⋆) software.

As related work, Schneider et al. [16] presented a survey on self-healing systems frameworks. According to these authors, these systems can combine machine learning techniques and control loops to reduce human intervention, since such systems are costly to develop and they can autonomously detect and recover themselves from faulty states. The study presented a classification of self-healing frameworks per three categories (techniques): (i) learning methodology (supervised, semi-supervised, and unsupervised); (ii) management style (bottom-up and top-down); and (iii) computing environment (n-tier traditional, cloud, virtualized, and grid/p2p). In Psaier & Dustdar [7], a survey on self-healing systems was conducted. This survey showed that the number of approaches for the research on self-healing has been very active. Moreover, a selection of current and past self-healing approaches was addressed, as well as explanations for the origins, principles, and theories of self-healing for such approaches. These two studies provided the theoretical basis for the design of our framework.

Qun et al. [17] stated that architecture-based self-healing approaches were used in the architectural model as basis for system adaptation. Such approaches were based on architectural reflection, and their software architectures are observable and controllable. Cheng et al. [18] purposed a software architecture-based adaptation for grid computing. Technically, the study designed a framework based on a software architectural model. This model allows the analysis of the necessity of adaption in an application, enabling repairs to be written in the context of the architectural model and propagated (applied/designated) to the running system. Zadeh & Seyyedi [19] suggested an architecture based on failure-prediction in architectures based on web services. The main goal of this study is to repair the execution process after detection of a failure. In a similar context, Psaier et. al [20] developed a self-healing approach that enables recovering mechanisms to avoid degraded or stalled systems. Thus, the study designed VieCure – a framework to support self-healing principles in mixed service-oriented systems. In this context, one can highlight that the literature has revealed important initiatives for the context of this paper.

III. Reference Architecture and Framework for Decision-Making

This section presents a brief description of our RA to support the development of SaS [1]. Moreover, our approach addresses a framework based on learning techniques and control loop for decision-making in such systems. This framework is part of the aforementioned architecture, whose main purpose is to support the identification of anomalies (symptoms), and propose solutions (treatments) for SaS at runtime.

A. Reference Architecture: RA4SaS

Figure 1 shows the general representation of our RA4SaS [1]. This architecture is composed of four external modules and a core for potential adaptation (dotted line), which represents an "adaptation bus" of the software entities at runtime in an automated approach. In short, our RA works with a controlled adaptation approach, i.e., the software engineer must insert annotations in each software entity so that the automatic mechanisms in the environment execution can identify the adaptability level of each entity. These levels contain parameters that determine where the new changes may be applied. Thus, when an entity is developed, an automatic mechanism performs a scan process, to inspect if such annotations were correctly inserted. After a validation process, such entities can be stored in the entities repositories (execution environment) so that they may be invoked in future adaptations. Next, a brief description of this architecture is addressed.

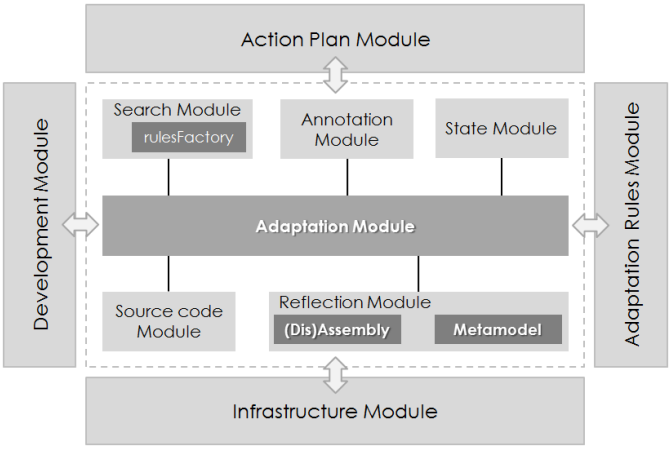

Fig. 1. Reference architecture for self-adaptive software [1]

The **Development Module** provides a guideline set for the development of software entities (SaS). Such guidelines act on requirement analysis, design, implementation, and evolution (i.e., adaptation of the software entities at runtime). The **Action Plan Module** aims at assisting in the adaptation activity of software entities. This module must be able to control:(i) dynamic behavior, (ii) individual reasons, and (iii) execution state in relation to the environment. To do so,

a framework based on learning techniques (classifiers and association rules) [5] and the MAPE-K control loop [6], [21] to support the decision-making of SaS is also part of this module. Section III presents details on the design and implementation of this framework. The **Adaptation Rules Module** provides a rule set (metrics) for adaptation of the software entities. Such rules are stored in the repositories (rule base) and reused when a search for adaptation is performed. The **Infrastructure Module** provides support for software entities adaptation at runtime, i.e., a mechanism set for the dynamic compiling and dynamic loading of software entities. Finally, the core of adaptation represents a logic sequence of well-defined steps so that the adaptation of the software entities is conducted with no human intervention, i.e., all activities of this process are conducted by an automated process as an "assembly line".

B. Framework for Decision-making

This section presents details of a framework for decision-making in SaS. In short, this framework acts as a non-intrusive supervision modality, i.e., a supervisor system (meta-level) can be coupled to a software entity (base-level) to monitor its internal state of operation or the execution environment in which it is inserted. Besides such supervision modality, this framework incorporates an extension of the MAPE-K control loop and three modules were designed: (i) classification of problems; (ii) recommendation of solutions; and (iii) test of solutions. In addition, sensors and effectors are also components of this framework, since they represent a means of interaction between supervisor system and supervised entities. Sensors are responsible for capturing parameters from the execution environment for the supervised system. Next, the classification module classifies these parameters to identify the changes occurred in each software entity from within the execution environment. Based on this classification and the collected data, an adaptation plan is prepared by the recommendation module to establish a solution for the identified problem. Before it becomes an effective solution, such recommendation must be tested in order to ensure that no "collateral effects" will be propagated to the software system (i.e., other software entities). Effectors deal with the "selected solution" after its testing activities are performed, applying it to the system. Figure 2 shows the control loop utilized in our framework.

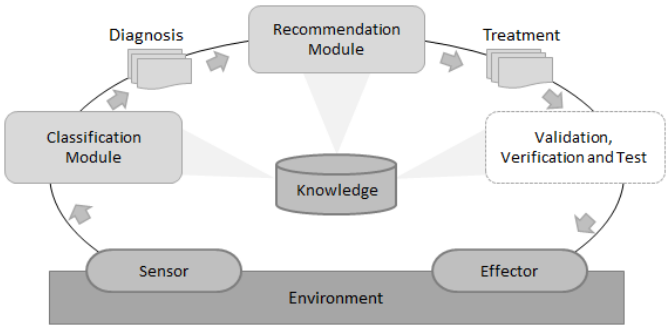

Fig. 2. MAPE-K control loop (Adapted from [6])

Section III-B1 and Section III-B2 present operational details of the classification and recommendation modules. Due to space limitations, details on the framework testing module are not widely detailed in this paper. However, in short, it is possible to mention that the framework testing module involves a test case selection based on information provided by logs during the system adaption. Section III-B3 provides details on the framework design.

1) Classification module: Figure 3 shows the classification module of our framework, whose main purpose is to present a classification for a set of data collected from sensors at runtime. Preliminarily, software engineers must specify the application domain, mapping the "main points" of a software entity (i.e., software system or software architecture) that will be monitored. This specification details the number of attributes of an instance and values that can be assigned to them. Such specification must be stored in a "`.arff`" file in the "Specification" component and mapped to a database aiming to store all interactions occurred during the execution cycle of our framework. Based on this specification and a set of labeled initial data (Step 1), an incremental classifier is generated (Step 2) in the "Incremental Classifier" component. Next, new data can be collected from the execution environment and sent to this classifier for identification of symptoms (Step 3). Finally, these data are stored in the database as collected and classified (Step 4) after the validation by the recommendation module.

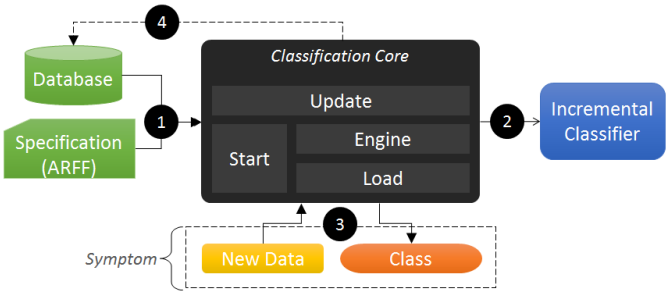

Fig. 3. General representation of the classification module

In the following, we present a brief description of the classification core: (i) **Start:** aims to initialize this module by means of the "Engine" component. As result, an incremental classifier is generated ("Incremental Classifier" component) based on the specification and initial knowledge provided by the specialist; (ii) **Load:** attempts to load data stored in the database, which is organized in two types: (i) specification, i.e., initial knowledge provided by the specialist; and (ii) acquired knowledge, i.e., data obtained during the execution cycle; (iii) **Engine:** represents an abstraction of the incremental classifier algorithm that performs the data classification collected from the execution environment, or from the initial knowledge provided by the specialist; and (iv) **Update:** updates the database after new data has classified. Such update is performed when a message from the recommendation module is received indicating that both data and classification can be stored.

2) Recommendation module: The recommendation module supports the selection of an effective treatment for the problem reported in the previous step (classification module). This module has similar operations in comparison to the previous one. In Step 1, an assumption is required: the specification of the domain/problem must be provided by the specialist. To do so, both database of symptoms and domain specification ("Specification" component) must be reused from classification module. Moreover, a treatment database must be created, since it will store the solutions to the problems identified by the previous module. From these databases (symptoms and treatment) and specifications are generated a rule set (Step 2), which intends to map the problems (symptoms) and solutions (treatment). The main purpose for the use of association rule [22] in this module is to detect more significant statistically correlations, via support and confidence, among the symptoms and treatments in order to operate the recommendation of treatments for a symptom set [23]. Why? It is worth noting that there is no interest in a specific attribute (i.e., in a specific treatment), since a symptom set may present some alternatives of treatments. After creation of the rule set, new data (classification module) can be inserted (Step 3) in this module so that one or more treatments may be identified (Step 4). At the end, this module may recommend one or more treatments for the symptom identified. Whether there is more than one treatment, the approach presents a list that must be ordered by the support and confidence criteria of the rules. Thus, one can select these rules one-by-one (i.e., from highest to lowest criterion). After that, the cycle tests the solution as a feasible solving for the identified problem (symptom).

Further, regarding the recommendation module, the internal components of this module (Start, Load, and Update) have the same functionalities as the classification module. Therefore, such components are not presented in the same level of detail. The "Engine" component represents an abstraction for the rule algorithm. Similar to the classification module, other rule algorithms can be coupled to this module as a strategy to compare/evaluate results (i.e., statistical measures).

3) Framework Design: Figure 4 shows the main structure of classes of our framework, which is organized in three layers: (i) infrastructure for control loop, containing the core and module packages; (ii) classification and rule algorithms, represented by the algorithms package; and (iii) external resources, represented by the dotted line because they contain a package set developed by third parties. Next, a brief description of these packages in each layer is addressed.

The `core` package contains two interfaces (`Observer` and `Subject`) and a class named `AbstractObserver`. Such interfaces represent the observer and observed roles for the classes of the `module` package (`AbstractModule`). Finally, it is noteworthy that the `AbstractObserver` class implements both interfaces of this package, i.e., implements the methods of the `Subject` interface and delegates the implementation of the `Observer` interface for classes inherited.

The `module` package is composed of a class set that represents the MAPE-K implementation (Figure 2) for the

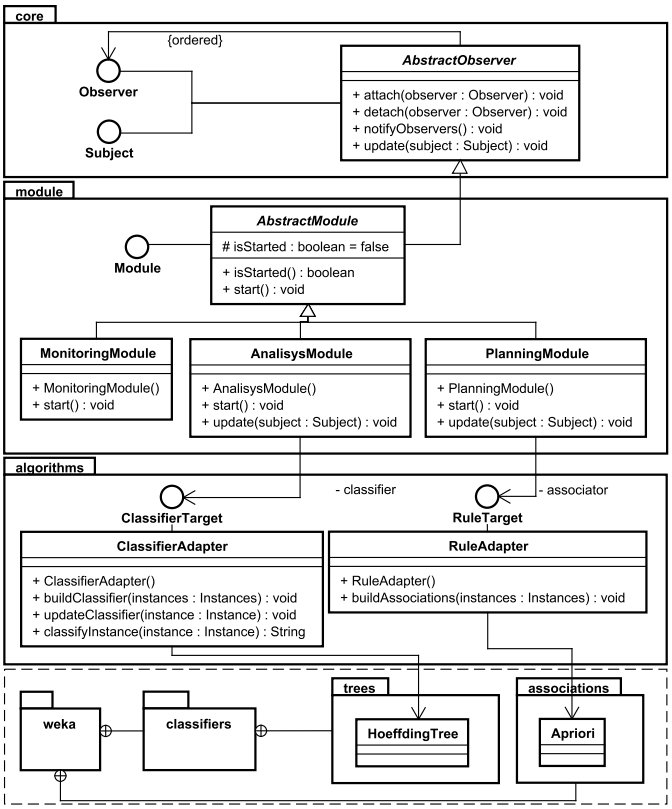

Fig. 4. Framework UML model

classification and recommendation modules. Such implementation is based on the *Observer* design pattern [24], i.e., all classes in this package are, at the same time, a subject and an observer. Thus, three configurations can be created: (i) the `MonitoringModule` class is an observer for the sensors of a software entity and a subject for the `AnalisysModule` class; (ii) the `AnalisysModule` class is an observer for the `MonitoringModule` class and a subject for the `PlanningModule` class; and finally, (iii) the `PlanningModule` class is an observer for the `AnalisysModule` class and a subject for the test module. This strategy enables the classes of this package to be decoupled, acting through event notification by the previous class via a single interface (`Module`).

The `algorithms` package contains a class set that represent the classification and recommendation algorithms. Such classes implement the *Adapter* design pattern [24] so that a common interface is available for both algorithms. The classification module is composed of an interface (`ClassifierTarget`) and a class (`ClassifierAdapter`). This class is an abstraction for the classification algorithm (`HoeffdingTree` class) implemented in the `trees` package. Similarly, the recommendation module was implemented using the aforementioned pattern.

Finally, the packages inside the dotted line represent the concrete classes of our framework, which were developed by third parties. According to *Adapter* design pattern [24],

the `ClassifierTarget` class is a `Target` class in the pattern, `ClassifierAdapter` is an `Adpter`, and `HoeffdingTree` is an `Adaptee`. Thus, other algorithms can be coupled to our framework without additional implementation in our system; only the `ClassifierAdapter` and `RuleAdapter` classes will be subtly modified. Moreover, the new packages and classes should be represented in the same format as the current ones (dotted lines).

IV. Case Study

To evaluate the applicability, strengths, and weaknesses of our framework this section presents a case study we have conducted. As subject application for our empirical analysis, we have selected an application addressed to the management of an UAV in a simulated environment, as shown in Figure 5. In short, the UAV architecture is organized in three layers: UAV, Communication, and Client. The UAV layer is composed of a UAV set that contains the following components: 3D glasses with radio frequency transmitter; autopilot; navigation camera; day and night vision camera; parachute; solar board; thermal sensor camera and so on. The communication layer contains the servers for communication between UAVs and clients, and time synchronization (NTP – Network Time Protocol). The client layer represents the UAV controllers in different operating systems.

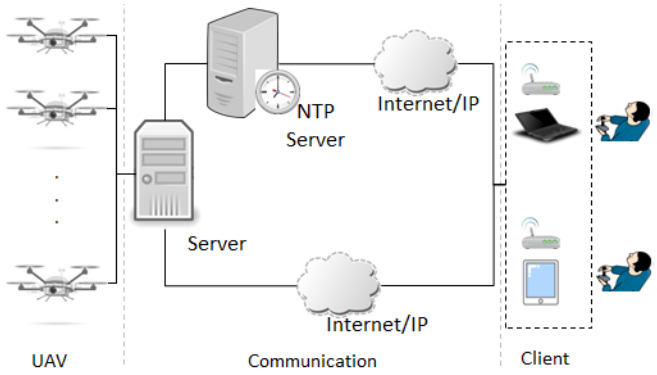

Fig. 5. General architecture for UAV

Operationally, we have instantiated our framework into server (Figure 5), enabling us to collect data from the environment via sensors, and transferring it for classification. In this context, when a problem is detected, a set of useful solutions is presented for correcting the flight plan. In extreme cases, the system may exhibit a recommendation to abort the operation. This last case is recommended when the UAV integrity may be compromised. Then, the UAV location is provided for our system, enabling the vehicle to be rescued. Modifications are made in the flight plan when the collected data tell us that something unplanned is changing in the environment. Thus, even if no decision is taken, the mission of the UAV may be compromised.

The UAVs used in the scope of this empirical study are equipped with seven sensors: (i) altitude and direction, (ii) barometer, (iii) battery level, (iv) humidity, (v) latitude and longitude, (vi) speed, and (vii) temperature. Some of these sensors provide numerical information that must be discretized, since both algorithms (classification and recommendation) of our framework require data in the form of categorical attributes [5]. Due to space reasons, only one of the sensors was used to show the discretization process. Thus, we have chosen the battery level as our target sensor since it is the power source for all components of an UAV. In addition, the information provided by this sensor represents an estimation of the flight range of the UAVs. Flight range is the time that an UAV can remain flying and, consequently, through this trip autonomy, one can get an estimate on feasible distance of flight. Table I presents the categories for the battery level sensor. The first column shows the range to classify the battery level (second column) on a scale of six to seven percentage points (i.e., A with six points and B and C with seven points). The third column presents a classification in a scale of 20 percentage points. However, it is noteworthy that a classification has three levels, i.e., the A level is the best state of a classification, the B level can be considered as a stability region, and the C level represents a transition stage. Since the discretization process requires a nominal category, we combine the first letter of each classification with respective battery charge levels, as shown in column 4. Finally, it is important to highlight that we have applied the same strategy for the remaining sensors,

TABLE I
Classification for the Battery Level Sensor

Interval	Level	Classification	Class
95 - 100	A	Excellent	E.A
88 - 94	B		E.B
81 - 87	C		E.C
75 - 80	A	Good	G.A
68 - 74	B		G.B
61 - 67	C		G.C
55 - 60	A	Regular	R.A
48 - 54	B		R.B
41 - 47	C		R.C
35 - 40	A	Bad	B.A
28 - 34	B		B.B
21 - 27	C		B.C
14 - 20	A	Critical	C.A
7 - 13	B		C.B
0 - 6	C		C.C

After discretization of the variables, the modeling activity is started. Thus, each sensor will be transformed into an attribute and its respective class in values for this attribute. Next, an initial knowledge must be provided to the databases of symptoms and treatment (i.e., classification and recommendation modules), setting a limitation for our approach. When new data were collected from the environment to be classified by our framework, an environment accuracy rate of 80% was obtained. However, it is noteworthy that although the number may be expressive, this rate can be optimized depending on the initial knowledge provided, since in previous studies this rate ranged from 87 to 94%.

Although no validation process has been used to obtain the results presented in this section, such percentages pro-

vide evidence that the imbalanced data may negatively affect the behavior of both modules (i.e., classification and recommendation). According to our expertise, the following activities must be conducted to overcome such adversity: (i) the domain specialist should conduct the modeling of the problem, i.e., the selection of attributes and values as shown in the discretization process for the battery level attribute; (ii) next, an initial knowledge should be provided so that both modules can be started. It is noteworthy that this knowledge is closely related to the problem and must not be generalized; and (iii) finally, a calibration process of such data must be performed by the specialist, since each problem has specific features and behaviors that should be considered in the execution of both modules. According to [5], [7], [20] this process can optimize the performance of the algorithms of both modules. Finally, we consider the particularity of our subject application as a threat to the validity of our results. Practitioners have been exploring different adaptation rules and creating SaS with different features, limiting the wider generalization of empirical analysis.

V. Conclusions and Future Work

This paper presented a framework for decision-making in SaS. The main contributions of this paper are: (i) SaS area for providing a feasible solution for classification of problems and recommendation of solution at runtime. Our study uses learning techniques as a means to implement the MAPE-K control loop [6], [21]. Moreover, the extension of this loop must be highlighted, since all solutions must be tested before being inserted into the execution environment to avoid that collateral effects regarding to adaptation activity are propagated; (ii) Development facilities with this framework, since an application can be modeled and instantiated without a high level of knowledge by the developers; (iii) Algorithm coupling flexibility, since some applications may require other information or measure for treatment of a problem; and (iv) From operational view point, the reconstruction of classifier and rules for the same point they were interrupted, since all data are labeled as Initial Knowledge (IK) and Acquired Knowledge (AK). Moreover, the databases (symptoms and treatment) are updated when a solution is confirmed as feasible, otherwise the data must be evaluated by the specialist.

As future work, three goals are intended: (i) conduction of more case studies intending to completely evaluate our framework; (ii) evaluation of this framework with other algorithms for both modules classification and recommendation; and (iii) use of this framework in the industry, since it is intended to evaluate its behavior when it is applied in larger real environment of development and execution. Therefore, it is expected that a positive scenario of research, intending to have this framework become an effective contribution to the software development community.

Acknowledgment

This research is supported by PROPe/UNESP and Brazilian funding agencies (FAPESP, CNPq and CAPES).

References

[1] F. J. Affonso and E. Y. Nakagawa, "A reference architecture based on reflection for self-adaptive software," in *SBCARS' 2013*, 2013, pp. 129–138.

[2] E. Y. Nakagawa, F. Oquendo, and M. Becker, "RAModel: A reference model of reference architectures," in *ECSA/WICSA' 2012*, Helsinki, Finland, 2012, pp. 297–301.

[3] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE' 2007*, may 2007, pp. 259 –268.

[4] F. J. Affonso, M. C. V. S. Carneiro, E. L. L. Rodrigues, and E. Y. Nakagawa, "Adaptive software development supported by an automated process: a reference model," *Salesian Journal on Information Systems*, vol. 12, pp. 8 – 20, 2013.

[5] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[6] IBM, "An architectural blueprint for autonomic computing," [*Online*], *World Wide Web*, 2005, in http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf (Access in 03/13/2015).

[7] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jan. 2011.

[8] D. Weyns, S. Malek, and J. Andersson, "Forms: a formal reference model for self-adaptation," in *ICAC' 2010*. New York, NY, USA: ACM, 2010, pp. 205–214.

[9] L. Liu, S. Thanheiser, and H. Schmeck, "A reference architecture for self-organizing service-oriented computing," in *ARCS' 2008*, U. Brinkschulte, T. Ungerer, C. Hochberger, and R. Spallek, Eds. Springer Berlin / Heidelberg, 2008, vol. 4934, pp. 205–219.

[10] D. Weyns, S. Malek, and J. Andersson, "On decentralized self-adaptation: lessons from the trenches and challenges for the future," in *SEAMS' 2010*. New York, NY, USA: ACM, 2010, pp. 84–93.

[11] S. Angelov, P. Grefen, and D. Greefhorst, "A classification of software reference architectures: Analyzing their success and effectiveness," in *WICSA/ECSA' 2009*, 2009, pp. 141–150.

[12] J. Bayer, T. Forster, D. Ganesan, J.-F. Girard, I. John, J. Knodel, R. Kolb, and D. Muthig, "Definition of reference architectures based on existing systems," Fraunhofer IESE, Tech. Rep., 2004, technical Report 034.04/E.

[13] E. Y. Nakagawa, R. M. Martins, K. R. Felizardo, and J. C. Maldonado, "Towards a process to design aspect-oriented reference architectures," in *CLEI' 2009*, 2009, pp. 1–10.

[14] M. Galster and P. Avgeriou, "Empirically-grounded reference architectures: a proposal," in *QoSA-ISARCS' 2011*, New York, NY, USA, 2011, pp. 153–158.

[15] S. Angelov, P. Grefen, and D. Greefhorst, "A framework for analysis and design of software reference architectures," *Information and Software Technology*, vol. 54, no. 4, pp. 417 – 431, 2012.

[16] C. Schneider, A. Barker, and S. Dobson, "A survey of self-healing systems frameworks," *Software: Practice and Experience*, pp. n/a–n/a, 2014.

[17] Y. Qun, Y. Xian-chun, and X. Man-wu, "A framework for dynamic software architecture-based self-healing," in *ICSMC' 2005*, vol. 3, Oct 2005, pp. 2968–2972 Vol. 3.

[18] S.-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu, "Software architecture-based adaptation for grid computing," in *HPDC-11' 2002*, 2002, pp. 389–398.

[19] M. H. Zadeh and M. A. Seyyedi, "A self-healing architecture for web services based on failure prediction and a multi agent system," in *ICADIWT' 2011*, Aug 2011, pp. 48–52.

[20] H. Psaier, F. Skopik, D. Schall, and S. Dustdar, "Behavior monitoring in self-healing service-oriented systems," in *COMPSAC' 2010*, July 2010, pp. 357–366.

[21] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, no. 2, pp. 223–259, Dec. 2006.

[22] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93, New York, NY, USA, 1993, pp. 207–216.

[23] C. Tew, C. Giraud-Carrier, K. Tanner, and S. Burton, "Behavior-based clustering and analysis of interestingness measures for association rule mining," *Data Mining and Knowledge Discovery*, vol. 28, no. 4, pp. 1004–1045, 2014.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.