DmS-Modeler: A Tool for Modeling Decision-making Systems for Self-adaptive Software Domain

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

Elisa Yumi Nakagawa
Dept. of Computer Systems
University of São Paulo - USP
São Carlos, SP, Brazil
elisa@icmc.usp.br

***Abstract*—The ability to modify its own structure and/or behavior at runtime is a native feature in the development of Self-adaptive Software (SaS). In previous work, a Reference Architecture for SaS (RA4SaS), an automated process for adaptation, and a framework for decision-making were developed to assist the development of SaS. Although such initiatives have collaborated with evolution of SaS, the design of the Decision-making Systems (DmS), element of first class for SaS, is manually conducted. Therefore, this paper presents a tool called *DmS-Modeler*, which aims to assist the development of DmS for SaS, providing facilities for modeling, calibration of such system, and automatic generation of infrastructure (i.e., source code and databases). Aiming to present the applicability of our tool, a case study was conducted and the results enable us to have good perspectives of contribution to the SaS area and other domains of software systems.**

***Keywords*-Self-adaptive software; Reference Architecture; Tool; Decision-making System.**

I. Introduction

The complexity of software systems and their computational environments has increased in the last years. In general, our society is becoming increasingly dependent of such systems, which must be able to work in 24/7 mode (i.e., 24 hours per day, seven days per week). Moreover, most of them must be prepared to operate in adverse conditions, maintaining their integrity of execution. Therefore, features as robustness, reliability, scalability, and flexibility have been increasingly required by these systems in the presence of changes (e.g., new needs of their users and/or modifications in the execution environment). These features and examples have been associated with the area of Self-adaptive Software (SaS), which are systems capable of self-configuration, self-adaptation and self-healing, self-monitoring and self-tuning, and so on. In short, they enable the incorporation of new features or functionalities at runtime without interruption in their execution [1], [2].

Software adaptation, when manually performed, becomes an onerous (e.g., regarding time, effort, and money) and error-prone activity (e.g., involuntary injection of uncertainties by the developers) [3], [4]. To overcome these adversities, automated approaches have been adopted as a feasible alternative to maximize the speed of SaS implementation and, at the same time, minimize the developers' involvement [4], [5].

In parallel, Reference Architectures (RA) refer to a special class of software architecture that have become an important element to systematically reuse architectural knowledge [6], [7]. Thus, in previous work [3], [8] we have proposed a Reference Architecture for SaS (RA4SaS)[8] – an architecture that provides a set of guidelines for the SaS development and an automated process for self-adaptation of the software entities at runtime without human intervention. From this point onwards, SaS may be also referred to as software entities or simply entities. Moreover, as part of this RA, a framework for decision-making was developed [9], whose purpose is to identify anomalies and propose one or more solutions to solve them. The RA and this framework have improved the SaS development; however, the design of Decision-making Systems (DmS), i.e., instances of such framework, was still manually conducted. According to Whitehead [5], Nakagawa et al. [6] and Gray [10], software engineering tools, among several purposes, can significantly reduce the time and development cost in a software project. In this sense, the main contribution of this paper is to present a tool called *DmS-Modeler*, which aims to support the DmS development for SaS. As result, to the best of our knowledge, there is no other complete solution as RA4SaS that enables the SaS development in both design and infrastructure.

The paper is organized as follows: Section II presents the background and related work; Section III provides a description of RA4SaS and the automated process for SaS adaptation; Section IV shows the *DmS-Modeler* tool and its approach of development; Section V presents a case study to show the applicability of our tool; and Section VI summarizes the contributions and presents perspectives for further research.

II. Background and Related Work

This section presents the background (i.e., concepts and definitions on self-$\star$ systems and RA) and related work on our paper.

DOI reference number: 10.18293/SEKE2016-184

Self-⋆ Systems. SaS has specific features in comparison to traditional one because this type of software system constantly deals with structural and/or behavioral changes at runtime. Some of them deal with management of complexity, robustness in handling unexpected conditions (e.g., failure), changing priorities and policies governing the goals, and context conditions (e.g., execution environment). The SaS development has 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 [1], [11]. According to Silva and De Lemos [12], there is a set of goals to be achieved so that structural and behavioral modifications are performed in the SaS without impacting its execution states. For these authors, an adaptation plan is a feasible solution to define which procedures will be adopted so that such changes are implemented.

Reference Architecture. According to Nakagawa et al. [7], RAs refer to a special type of software architecture that have become an important element to systematically reuse architectural knowledge. The main purpose of these architectures is to facilitate and guide [7]: (i) the design of concrete architectures for new systems; (ii) the extensions of systems of neighbor domains of a RA; (iii) the evolution of systems that were derived from the RA; and (iv) the improvement in the standardization and interoperability of different systems. Considering their relevance for the software development, different domains have proposed RAs. For instance, service-oriented architectures such as IBM's foundation architecture [13] and architectures for software engineering environments [6] are some of RAs found in the literature. Other areas/domains have also proposed RAs, including the self-⋆ software (e.g., RA4SaS [8]).

As related work, Schneider et al. [14] presented a survey on frameworks for self-healing systems. According to this study, these systems can combine machine learning techniques and control loops to reduce human intervention, since such systems can autonomously detect and recover themselves from faulty states. The authors proposed a classification of self-healing frameworks in three categories: (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 and Dustdar [15], a survey on self-healing systems was conducted. This study 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 were 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 [9].

Qun et al. [16] reported that architecture-based self-healing approaches were used in the architectural models as basis for system adaptation. Such approaches were based on architectural reflection so that their software architectures are observable and controllable. Cheng et al. [17] proposed a software architecture-based adaptation for grid computing. Technically, the study designed a framework based on a software architectural model, which allows the analysis of the adaptation needs in an application, enabling repairs to be written in the context of the architectural model and propagated to the running system. Zadeh and Seyyedi [18] 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 [19] developed a self-healing approach that enables recovering mechanisms to avoid degraded or stalled systems. As result a framework called VieCure was designed 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 for SaS

According to Affonso and Nakagawa [8], the RA4SaS (Figure 1) is composed of four external modules and a core of adaptation. This 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 adaptation 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, these entities are inserted in the execution environment (i.e., entity repositories) so that they may be invoked in future adaptations. Next, a brief description of this architecture is presented.

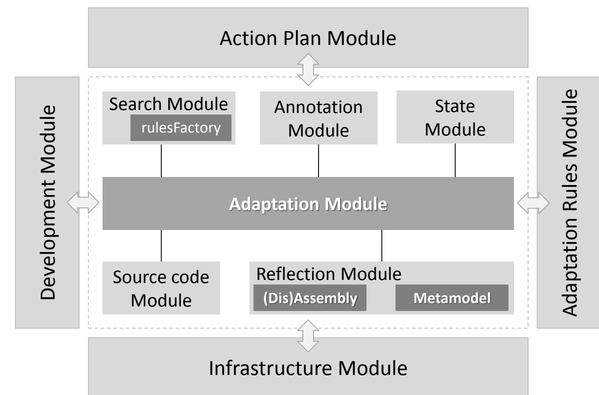

Fig. 1. General View of the RA4SaS [8]

The **Development Module** provides a set of guidelines for requirement analysis, design, implementation, and adaptation of the software entities at runtime. The **Action Plan Module** aims at assisting in the adaptation of such entities, providing controls for: (i) dynamic behavior, (ii) individual reasons, and (iii) execution state in relation to the environment. Thereby, a

framework based on learning techniques [20] and the MAPE-K loop [21], [22] was developed in previous work as part of this module to support the decision-making [9]. Section IV presents details on this framework as well as the design and implementation of the *DmS-Modeler* tool. The **Adaptation Rules Module** provides a set of rules (i.e., metrics) for adaptation of the software entities. The **Infrastructure Module** provides support for the adaptation of software entities at runtime, i.e., a set of mechanisms for the dynamic compiling and dynamic loading of such entities. Finally, the **Core of Adaptation** represents an automated process composed of a logic sequence of nine well-defined steps so that the adaptation of the software entities is conducted with no human intervention [8].

IV. *DmS-Modeler* TOOL

As presented in Section III, the RA4SaS has an automated process to accommodate the structural or behavioral changes in a software entity. According to authors of this RA [8], the modification of a software entity is a complex activity, since an action plan must established so that the software entities are adapted. In short, this plan must be elaborated based on new requirements (e.g., structural and/or behavioral) and adaptation level of each software entity. Based on this scenario, a framework for decision-making was developed by Affonso et al. [9] to support the generation of such plan identifying anomalies and proposing one or more solutions to solve them. Concerning the design, MAPE-K loop [22], learning techniques [23], [24], and an external approach [4] of adaptation were the resources used in this framework.

The MAPE-K loop is a reference model (i.e., mature solution) that has been adopted to provide an autonomous behavior in SaS. In short, the *Monitor* process receives data collected from sensors and convert such data in “symptoms”. The *Analyze* process aims to correlate the collected data and to model complex situations so that the autonomous systems can learn from the environment and predict future situations. The *Plan* process is responsible to create a plan for adaptation, i.e., what will be adapted and how to apply the changes in a software entity. The *Execute* process must provide the mechanisms that can execute the action plan (i.e., proposed solutions). *Sensors* and *Effectors* are components to generate a collection of data reflecting the system state to rely on in vivo mechanisms or autonomous subsystems to apply changes [4], [21], [22].

Concerning the learning technique, an incremental classifier and association rules were used in the design of the classification and recommendation modules. The incremental classifier analyzes the training data and produces an inferred function, which can be used for mapping new instances [23], [24]. The association rule [23] aims to detect more significant statistically correlations, via support and confidence, among the occurred changes in order to operate the recommendation of solutions for such changes [24]. It is worth noting that there is no interest in a specific attribute (i.e., specific solution), since a set of changes may present a set of solutions.

Finally, the external adaptation approach enables the organization of a SaS in two layers: (i) adaptation engine, which contains the logical for adaptation; and (ii) adaptable software, which represents entities that can be adapted. Therefore, our framework [9] acts as a non-intrusive supervision modality, i.e., a supervisor system (engine) can be coupled to a software entity (adaptable software) to monitor its internal state of operation or the execution environment in which it is inserted.

Although our framework has been designed to assist the process of decision-making and propitiate its reuse in different systems (e.g., SaS and other software domains), the development of a DmS is manually conducted and, hence, it can be considered onerous and error-prone activity. Therefore, the design of a tool that meets the following requirements is a feasible alternative: (i) modeling of the problem and definition of the main points of monitoring for a software system (SaS); (ii) automatic generation of the classification and recommendation modules; (iii) insertion of initial knowledge into such modules; and (iv) data calibration of the DmS. According to these requirements, a tool called *DmS-Modeler* was developed and a set of guidelines elaborated to support the development of DmS for SaS, as illustrated in Figure 2. In short, the development of this system type is composed of three phases: design, execution, and instruction mapping. The first is organized in two modeling steps: (a) *classification module*, which main purpose is to present a classification for a data set collected via sensors from execution environment; and (b) *recommendation module*, which aims to present a solution set ranked by statistical measures for a problem reported by the classification module. The second represents the complete instantiation of the DmS for the calibration process into its execution environment. The third consists of mapping knowledge (i.e., recommendation module) to instructions in source code. Next, a description of each step is reported.

During the design of the classification module, the stakeholders (i.e., software engineer and domain specialist) must define the “main points” of monitoring for a software entity. These points represent the attribute number of an instance and values that can be assigned to them. In this sense, the *DmS-Modeler* tool has a wizard (i.e., a visual front-end) that enables the creation of attributes and their values, which may be numerical or nominal. In order to make compatible the values collected from the execution environment in relation to ones required by the algorithms used in the implementation of both modules, a discretization process of values was implemented in our tool. After the definition of attributes and values, the stakeholders must save the specification of the DmS in a “`.arff`” file, which is represented by the “Specification Repository” component. This file will be used by our tool to generate the database (“Problems” component) of each system. Based on this specification and a set of labeled initial data, an incremental classifier is generated in the “Classifier” component. Next, new data can be collected from execution environment and sent to this classifier for identification of anomalies. The data is stored in the database as collected and classified after the validation by the test module [9].

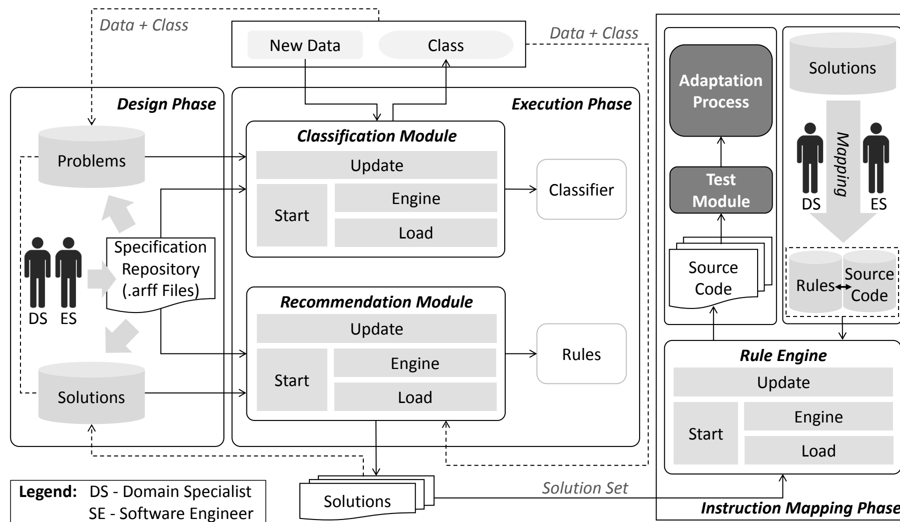

Fig. 2. Development phases of the *DmS-Modeler* tool

The design of the recommendation module must be conducted in a similar way to previous one [9]. The stakeholders must conduct the specification of the solutions and stored them in the "Specification Repository" component as a "`.arff`" file, which will be used by the *DmS-Modeler* tool to generate the database ("Solutions" component). The insertion of labeled initial data is different from the previous module, since the stakeholders must provide one or more solution for each instance of problem. From these databases (i.e., Problems and Solutions) and specifications are generated a rule set ("Rules" component), which intends to map the problems and solutions. The design of the rule engine is conducted in parallel to this module, since the main purpose of this engine is to provide a solution in source code for each instance of problem. Thus, each one must be labeled by a class to receive one or more solution in source code ("Source Code" repository), generating a rule in the "Rules" repository. Before it becomes an effective solution, a recommendation must be tested ("Test Module") in order to ensure that no "collateral effects" will be propagated to the software system. For space and scope reasons, the design of this module will not be detailed in paper.

Regarding the internal components of each module, a brief description is addressed [9]: (i) **Start:** aims to initialize each module by means of the "Engine" component. As result, an incremental classifier ("Classifier" component), rule set ("Rules" component), and a rule engine are generated based on the specifications and initial knowledge provided by the specialist; (ii) **Load:** attempts to load data stored in each database, which is organized in two types: (a) specification, i.e., initial knowledge provided by the specialist; and (b) acquired knowledge, i.e., data obtained during the execution cycle; (iii) **Engine:** represents an abstraction for the algorithms of classification and recommendation, which were developed by third parties; and (iv) **Update:** updates the database after new data has classified and validated. Such update is performed when a notification from the test module is received. Finally, it is noteworthy that our tool has a flexible engine, since it enables that other algorithms can be coupled to it without additional implementation in its modules.

V. Case Study

In this section we present a case study to evaluate the applicability of the *DmS-Modeler* tool. The main purpose of this study is to demonstrate the real value of our tool for the stakeholders during the development of a DmS. In previous work, we have applied our framework for the monitoring and eventual corrections of flight plan for Unmanned Aerial Vehicles (UAVs) [9], since this system type requires unforeseen context changes. Therefore, we will approach the same system in this paper; however emphasizing the use of the *DmS-Modeler* tool. Next, a brief description of our subject application and the empirical strategies adopted for conducting this case study is presented.

Subject Application. For our empirical analysis we have selected an application addressed to the management of an UAV in a simulated environment. In short, this application is organized in three layers: (i) *UAV*, which is composed of a set of UAVs; (ii) *Communication*, which contains the servers for communication between UAVs and clients; and (iii) *Client*, which represents the controllers of the UAVs in different operating systems. The UAVs used in the scope of this empirical study are equipped with five sensors: (i) altitude, (ii) battery level, (iii) distance, (iv) speed, and (v) temperature. These sensors provide numerical information that must be discretized, since both algorithms of our tool require data in the nominal form [20]. Operationally, this application enables us to collect data from the environment via sensors, and transferring it for classification. Thus, 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.

Empirical research strategy. Figure 3 illustrates the systematization for modeling of a DmS, which is organized in two steps: (i) modeling of the problem; and (ii) calibration of the modules. Initially, the stakeholders must identify the monitoring points for such system (Step 1). Next, for each point, a verification is conducted to obtain its output (i.e., *Is it a numerical or nominal value?*). The main difference between the modeling of numerical and nominal values consists of definition of a label (i.e., nominal value) for an interval of numerical values, as illustrated in the table fragment "battery level sensor" (Step 3.1). For space reasons, only the discretization of this sensor will be presented. 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 points in relation to first column. We also defined 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 region of stability, and the C level represents a transition stage. Next, we combine the first letter of each classification with respective battery charge levels to create a nominal category, as shown in column 4. After discretization, each sensor must be transformed in an attribute and its label in value for it. Step 3.2 represents the insertion of monitoring points (i.e., attributes) by the stakeholders into our tool. Step 4 represents the generation of a complete infrastructure for the classification module (i.e., database containing all the attributes and source code for this module). Step 5 represents the mapping of solutions for each type of problem. In this wizard, the stakeholders can select the project developed in the previous step so that the recommendation module is generated (Step 6). Next, an initial knowledge must be provided to the databases of the classification and recommendation modules (Step 7). Our tool provides a table for insertion of knowledge for the classification module and a list for mapping of solutions. Each line of this table represents a system state in the execution environment. To overcome possible adversities during its execution cycle, one or more solutions can be created and associated to each state to maintain a system in execution. Our tool also enables the validation of the inserted knowledge by means of a wizard (Step 8), which aims to ensure that a problem has one or more solution. After the insertion of knowledge, the calibration process can be conducted to evaluate the precision of a classification of problems and the recommendation of solutions [9], [15], [19], [20].

VI. Conclusions and Future Work

This paper presented *DmS-Modeler* tool that intends to support the SaS development. This tool incorporates a framework for decision-making [9] and, hence, the benefits of our tool can also be extended for other communities of software development. As reported in this paper, an approach was proposed to systematize the development of a DmS using our tool. Based on this scenario, the main contributions of this paper are:

- For the SaS area by providing a means that facilitates the development of DmS for SaS;
- For the use of our framework [9], providing a wizard for modeling of the monitoring point. For instance automatic generation of the classification and recommendation modules, insertion of initial knowledge, and calibration of the DmS are other functionalities provided by our tool that can executed by means of a wizard. Moreover, the automation of the activities for the software development tends to minimize human' involvement and involuntary generation of uncertainties, which are considered negative factors for the software development, especially for the SaS [5], [6], [8], [10], [9]; and
- For the RA area, since we have proposed the first RA based on reflection [8] and the implementation of this tool aims to optimize our initiative for the development of such systems.

As future work, three goals are intended: (i) conduction of more case studies intending to completely evaluate our tool, including different software domains; (ii) implementation of a new wizard for mapping of solutions to instructions (i.e., source code), which would complete the development cycle presented in Figure 2; and (iii) use of this tool 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 tool 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] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE' 07*, 2007, pp. 259–268.
[2] P. Maes, "Concepts and experiments in computational reflection," *SIGPLAN Notice*, vol. 22, no. 12, pp. 147–155, December 1987.
[3] F. J. Affonso and E. L. L. Rodrigues, "A proposal of reference architecture for the reconfigurable software development," in *SEKE' 12*, 2012, pp. 668–671.
[4] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, pp. 1–42, 2009.
[5] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering*, 2007, pp. 214–225.
[6] E. Y. Nakagawa, F. C. Ferrari, M. M. F. Sasaki, and J. C. Maldonado, "An aspect-oriented reference architecture for software engineering environments," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1670–1684, 2011.
[7] E. Y. Nakagawa, F. Oquendo, and M. Becker, "RAModel: A reference model of reference architectures," in *WICSA/ECSA' 12*, 2012, pp. 297–301.
[8] F. J. Affonso and E. Y. Nakagawa, "A reference architecture based on reflection for self-adaptive software," in *SBCARS' 13*, 2013, pp. 129–138.
[9] F. J. Affonso, G. Leite, R. A. P. Oliveira, and E. Y. Nakagawa, "A framework based on learning techniques for decision-making in self-adaptive software," in *SEKE' 15*, 2015, pp. 1–6.

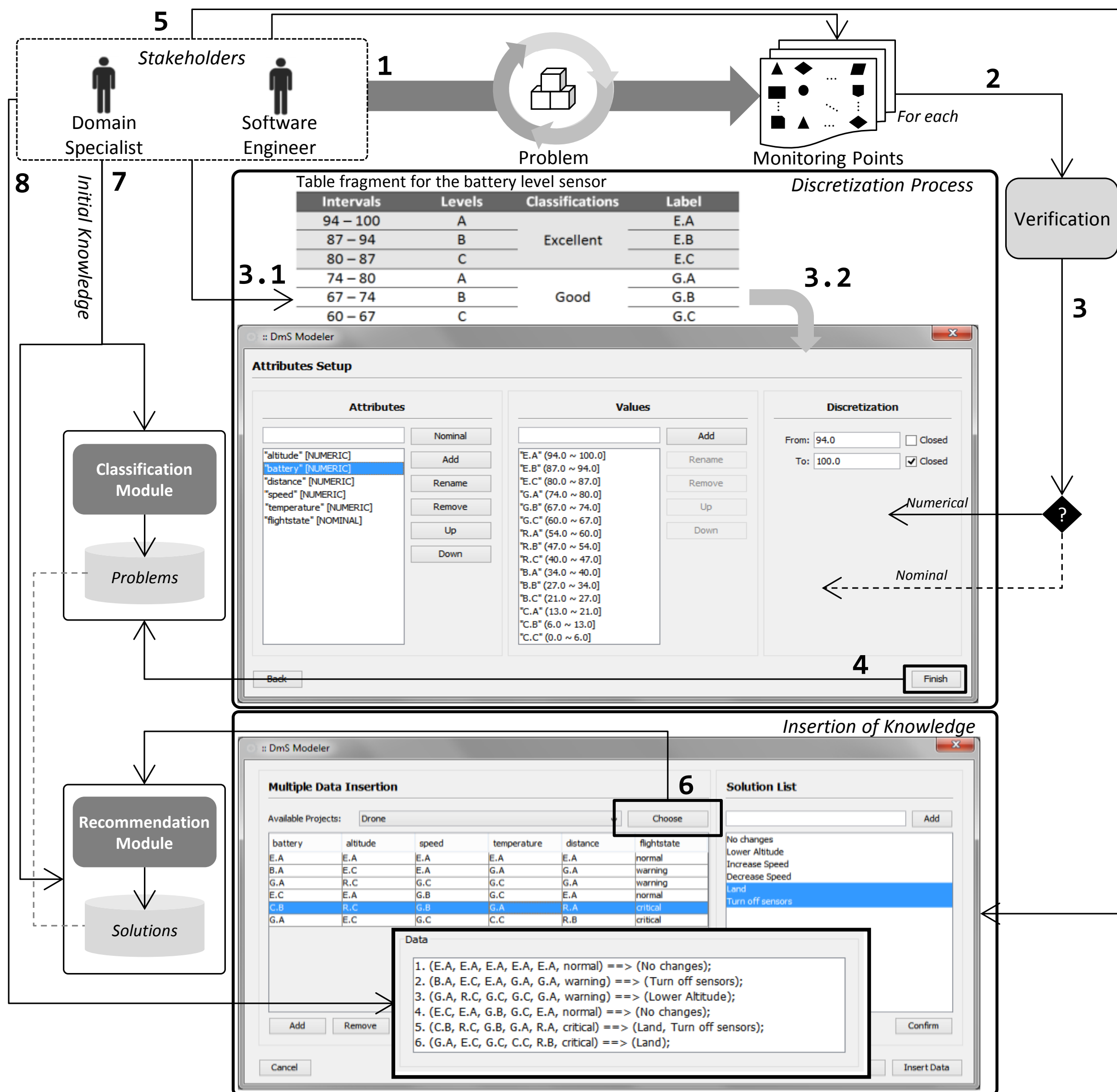

Fig. 3. Case study

[10] J. Gray, "Software engineering tools," in *HICSS' 00*, January 2000, pp. 3300–3301.

[11] D. Weyns, S. Malek, and J. Andersson, "Forms: a formal reference model for self-adaptation," in *ICAC' 10*, 2010, pp. 205–214.

[12] C. E. Silva and R. Lemos, "A framework for automatic generation of processes for self-adaptive software systems," *Informatica Journal*, vol. 35, no. 1, pp. 3–13, 2011.

[13] R. High, S. Kinder, and S. Graham, "Ibm's soa foundation - an architectural introduction and overview," 2005, available in: http://signallake.com/innovation/soaNov05.pdf (Access on March 1, 2016).

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

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

[16] Y. Qun, Y. Xian-chun, and X. Man-wu, "A framework for dynamic software architecture-based self-healing," in *SMC' 05*, vol. 3, October 2005, pp. 2968–2972 Vol. 3.

[17] S.-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu, "Software architecture-based adaptation for grid computing," in *HPDC' 02*, 2002, pp. 389–398.

[18] 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' 11*, August 2011, pp. 48–52.

[19] H. Psaier, F. Skopik, D. Schall, and S. Dustdar, "Behavior monitoring in self-healing service-oriented systems," in *COMPSAC' 10*, July 2010, pp. 357–366.

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

[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 Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, December 2006.

[22] IBM, "An architectural blueprint for autonomic computing," [*Online*], *World Wide Web*, 2005, available in: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf (Access on March 1, 2016).

[23] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *ACM SIGMOD' 93*, 1993, pp. 207–216.

[24] 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.