Infrastructure Based on Template Engines for Automatic Generation of Source Code for Self-adaptive Software Domain

Gabriele Salgado Benato[1] and Frank José Affonso[2]
Dept. of Statistics, Applied Mathematics and Computation
São Paulo State University – UNESP
PO Box 178, 13506-900, Rio Claro, SP, Brazil
[1]gabibenato@gmail.com, [2]frank@rc.unesp.br

Elisa Yumi Nakagawa
Dept. of Computer Systems
University of São Paulo – USP
PO Box 668, 13560-970, São Carlos, SP, Brazil
elisa@icmc.usp.br

***Abstract*—Self-adaptive Software (SaS), a special class of software system, constantly deals with some type of changes (i.e., structural and/or behavioral) to meet the user's new needs or autonomously react to modifications in its execution environment. Software adaptation, when manually performed, becomes an onerous and error-prone activity. Automated approaches have been adopted as a feasible alternative to overcome such adversities because they reduce the human involvement to a minimum. Based on this scenario, a Reference Architecture for SaS (RA4SaS) [1], [2] was designed in previous work. This architecture enables the adaptation of SaS at runtime by means of a controlled adaptation approach. In this sense, an automated process and a complete infrastructure to allow the phases of such process were also developed. This paper presents the design and implementation of a new module for automatic generation of source code for our architecture based on template engines [3]. In short, the main benefits of this module can be summarized in three items: (i) higher design flexibility, maintenance and evolution of SaS; (ii) introduction of new capabilities for automatic generation of source code; and (iii) performance improvement. To present the operation of this new module, a case study was conducted. As result, we have observed that our architecture has good perspective to efficiently contribute to the SaS area.**

***Keywords*-Automated Process, Reference Architecture, Self-adaptive Software, Template Engine.**

I. Introduction

Over recent years, a significant increase in the complexity of software systems and their computational environments has been observed. In general, such systems share functional, nonfunctional, physical, and virtual requirements. The human ability to manage systems becomes insufficient as their complexity increases. Moreover, involuntary injection of faults or uncertainties has often configured as one of the major causes of system failures, especially in the context of Self-adaptive Software (SaS). Software systems that can autonomously react to changes in their environment or modify its structure and/or behavior to meet the user's new needs are called SaS [4], [5].

Software adaptation, when manually performed, becomes an onerous and error-prone activity. Adversities, such as time, effort, money and the involuntary injection of uncertainties by developers, may be pointed as the main negative factors that have hampered the adaptation conducted by humans [6], [7]. To overcome such 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], [8]. According to Salehie and Tahvildari [4], SaS development may be based on an external approach, which segments this type of software in two layers: (i) adaptable software, which represents software entities that will be adapted; and (ii) adaptation engine, which contains the adaptation logic. Regarding the second layer, control loops [9] have been used to ensure that the adaptation engine dynamically adjusts the adaptable software.

Based on this context, a Reference Architecture for SaS (RA4SaS) [1], [2] was designed in previous work. In short, this architecture is based on computational reflection [10] and an external adaptation approach [4], which enables the adaptation of software entities at runtime by means of a controlled adaptation modality. The software engineers define the adaptation levels of each software entity by means of annotations. Next, the adaptation of each entity is conducted by an automated process, since capabilities of human administration show decreasing relative effectiveness. For instance, some tasks have been difficult to manage, introducing potential problems, such as change management and simple human error.

Although our architecture (RA4SaS) has already been instantiated for a concrete architecture, the generation of source code, a crucial activity for the execution of this automated process, has shown some limitations. For instance, developer's interventions were required to meet different application domains, i.e., the software engineers were responsible for making adjustments to meet the requirements of an application domain, architectural models, among other requirements. In this sense, this paper presents the design and implementation of a new module for the RA4SaS based on Template Engine (TE). According to Bergen [3], TEs can be used in software development that needs automatically generation of source code according to specific purposes. Regarding the design and implementation of this new module, the main benefits are: (i) higher flexibility of design, maintenance and evolution of the software entities; (ii) introduction of new capabilities for automatic generation of source code; and (iii) performance improvement.

The paper is organized as follows: Section II presents the

DOI reference number: 10.18293/SEKE2017-147

background and related work; Section III provides a description of our architecture (RA4SaS); Section IV shows the design and implementation of an infrastructure for automatic generation of source code for the RA4SaS; Section V presents a case study to show the applicability of this new module; 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-⋆ systems, reference architecture and TE) and related work on our paper.

Self-⋆ Systems. SaS has specific features in comparison to traditional ones because this type of software constantly deals with structural and/or behavioral changes at runtime. Some of them deal with management of complexity, reliability 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 to the users' needs or to respond as soon as these systems meet execution environment changes [11], [12]. According to Silva and De Lemos [13], there is a set of goals to be achieved so that structural and behavioral modifications are performed in the SaS without affecting its execution states. For these authors, an adaptation plan is a feasible solution to define which procedures shall be adopted so that such changes are implemented.

Reference Architecture. According to Nakagawa et al. [14], Reference Architectures (RA) refer to a special type of software architecture that have become an important element to systematically reuse architectural knowledge. The main purpose of such architectures is to facilitate and guide [14]: (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 derived from a different 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 [15] and architectures for software engineering environments [16] are some of RAs found in the literature. Other areas/domains have also proposed RAs, including self-⋆ software (e.g., RA4SaS [1]).

Template Engines. According to Bergen [3], "TEs are used in software development scenarios where it is necessary to automatically generate text and format it according to specific processing rules". This definition fits perfectly the purposes of our RA and its automated process of adaptation, which aims to manage the changes (i.e., user's new needs or modifications in the execution environment) without human intervention. The main reason for this type of TE is that our RA is based on reflection and, subsequently, it was instantiated in Java programming language. Thus, after a detailed analysis, the FreeMarker TE was chosen to be used in this work for the following reasons [3], [17]: (i) it is an open source software; (ii) it is a general-purpose template; (iii) it is faster than others; (iv) it provides facilities of use (e.g., JSON support, shared variables, template loaders, among others); (v) it has automated support in Java IDE (Integrated Development Environment); and (vi) its templates are not compiled to classes, i.e., a template can be loaded or reloaded during runtime without redeploying the application.

As related work, Cheng et al. [5] presented a study that aims to summarize the state-of-the-art on software engineering for SaS, identifying the main gaps and challenges. According to these authors, the Model-Driven Development (MDD) was pointed as a challenge for software adaptation, since a key issue of this approach is to keep the runtime models synchronized with system modifications. In this sense, the authors emphasized the importance of the code generators to maintain the integrity between source code and models. Silva and De Lemos [13] developed a framework for automatic generation of processes for SaS based on workflow, model-based approaches and artificial intelligence planning techniques. A specific contribution of this work is the usage of model-based technology for the generation of adaptation plans, since different planning techniques, according to the needs of the application domain, may be used. Daniele et al. [18] developed a Service Oriented Architecture (SOA)-based, platform-specific framework for context-aware mobile applications. This framework is composed of a methodology based on Model-Driven Architecture (MDA) principles that relies on SOA for context-aware mobile applications. Hallsteinsen et al. [19] developed a framework for the development of self-adapting applications in ubiquitous computing environments. This framework is based on MDD and it aims to facilitate the design and implementation of context-aware adaptive applications by the reuse of modeling artifacts and adaptive components. According to these initiatives, model-driven approaches and source code generators have been used in the development of self-adaptive applications in different domains. As our RA is based on metamodel for adapting software entities, only a source code generator based on TE was designed and it shall be presented in this paper.

III. Reference Architecture for SaS

Figure 1 shows the general representation of our RA, which is composed of four external modules and a core of adaptation [1]. This RA works with a controlled adaptation modality, i.e., the software engineer must insert annotations in each software entity so that the automatic mechanisms can identify their adaptability levels. In short, such levels contain parameters that determine where the new changes can 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 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 presented.

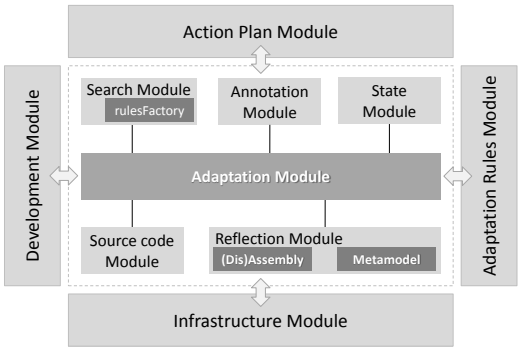

Figure 1. Reference architecture for SaS [1]

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 to assist in the adaptation activity of such entities, providing controls for: (i) dynamic behavior, (ii) individual reasons, and (iii) execution state in relation to the environment. Thus, a framework [2] based on learning techniques [20] and the MAPE-K control loop [9], [21] and a tool called *DmS*-Modeler [22] were developed to assist the development of Decision-making Systems (DmS) for SaS, providing facilities for modeling, calibration of such system, and automatic generation of execution infrastructure. The "Adaptation Rules Module" provides a set of rules (i.e., metrics) for adapting software entities. The "Infrastructure Module" provides support for software entities adaptation at runtime, i.e., a set of mechanisms for the dynamic compiling and dynamic loading of software entities. 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 [1]. To do so, this core was organized in six modules: (i) search, (ii) annotation, (ii) state, (iv) source code, (v) reflection, and (vi) adaptation. In order to detail the adaptation modality of our RA, the reflection and source code module will be detailed in this paper.

The reflection module is organized into two submodules. The first assists in the "disassembly" and "assembly" of the software entities by using the annotation module for obtaining adaptation levels supported by each entity. Next, structural and behavioral information is recovered via reflection and inserted into a metamodel, which is inside the second submodule. New information can be either added or removed from this metamodel, according to adaptation interests. This new metamodel is then transferred to the source code module so that the new software entities are generated. Figure 2 shows the UML model of this metamodel, which enables the adaptation of object-oriented software systems or make use of the structure of classes as units of software systems, as it can notice the presence of classes (`Class`), attributes (`FieldClass`), methods (`MethodClass`), among other. In this metamodel, classes which have "`Annotation`" prefix are used to manage the annotations responsible for the adaptation levels of each entity. Similarly, classes with the "`Persistence`" prefix are used to manager the annotations responsible for the persistence of software entities. To delimit the scope of this metamodel in relation to the persistence of entities, an Object/Relational Mapping (ORM) framework was used. This framework enables us to develop persistent classes following natural object-oriented idioms as our metamodel. Thus, developers do not have to worry about the generation of storage structures, since such they are automatically generated by the framework. This strategy can be considered relevant for developers, since they can develop self-adaptive software entities in a notation to their expertise area.

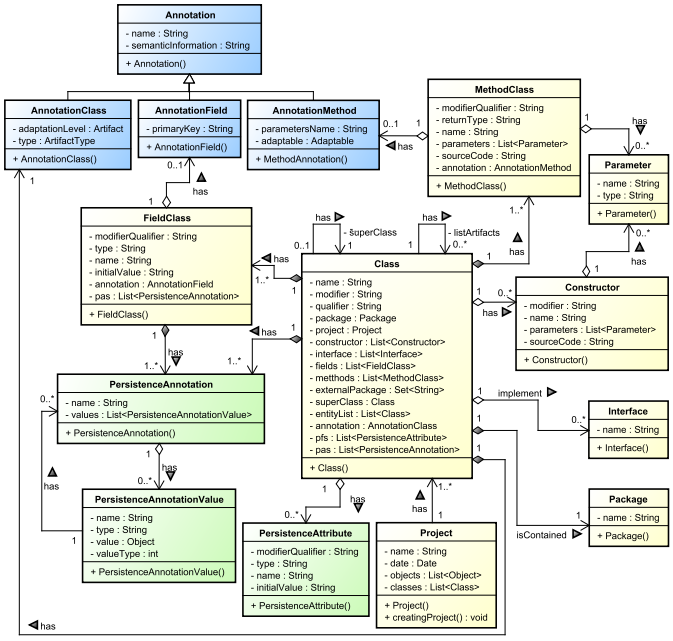

Figure 2. RA4SaS Metamodel for software entities

IV. Infrastructure for Automatic Generation of Source Code

The new generator module of source code is responsible to create the new software entities so that they can be compiled and inserted in the execution environment. Thus, the main purpose of this section is to present details on design and operation of such module in Section IV-A and a comparative analysis emphasizing the advantages of this module compared to the previous one in Section IV-B.

A. Design of the Generator Module of Source Code

Figure 3 provides an overview of the information flow contained in the generator module of source code, which is organized in three levels: (i) Metamodel, (ii) Template Engine, and (iii) Source Code. The first level represents the new metamodel with the adaptation changes (Figure 2). The second level contains all logic of this module, which must be able to generate the software entities according to the target domain and architectural models used in their development. Thus, software engineers and domain specialists must specify

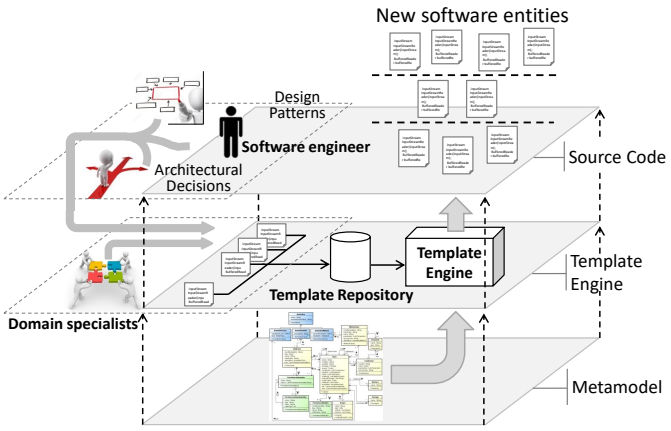

Figure 3. Source code module

design patterns, establish architectural decisions for the SaS development, and provide domain information to create a set of templates that will insert in the template repository so that the source code of each software entity can be generated by the "Template Engine" component. Design patters describe general solutions for design problems that occurs repeatedly in many projects including SaS domain. Architectural decisions represent implications for the environment in which the architecture will be deployed. Domain information encompasses the previous two, since the domain specialists must have a special knowledge to design each template according to requirements and restrictions and limitations imposed by the architectural and design models. Finally, the third level represents the source code for the new software entities.

The "Template Engine" component, illustrated on the second level, is a subsystem whose purpose is to generate source code for the software entities. Figure 4 illustrates the UML model of this subsystem, which is composed of three logical classes (`FileManager`, `FreeMarkerUtils`, and `TemplateManager`), an enum of constants (`FileType`), two external packages (`external`, and `freemarker.template`) and four stereotypes, i.e., two boundary classes (`Metamodel Boundary` and `Template Boundary`) and two control classes (`Metamodel Control` and `Template Control`). These packages contain a symbolic class (`Jacobe`) that represents the code formatter (`external` package) and three logical classes of the FreeMarker TE (`freemarker.template` package). The aforementioned stereotypes are used to represent: (i) the metamodels for the self-adaptive software entities, which are provided by the reflection module after insertion of adaptation changes; and (ii) the templates for generating software entities, since they may have been developed based on design patterns and architectural models; however, they may require a set of templates to generate the source code. Among of the logical classes, `TemplateManager` can be considered the main class of this subsystem because it is responsible for "orchestrating" activities so that the source code of software entities is generated. Initially, *Step 1*, the `start` method (`TemplateManager` class) receives as input a metamodel (from `Metamodel Control` and `Metamodel Boundary`) containing the software entities that will be generated. Next, *Step 2*, the `createEntity` method is invoked by the `start` method so that the software entities contained in this metamodel are fragmented and inserted in the entity list (`entities` attribute). In this step, the `FileType` enum is used, since provides a set of useful constants to maintain the nomenclature of such entities and to assist in the configuration of the code type that will be generated. Thus, for each retrieved entity, the `generateCode` method is invoked (*Step 3*). The purpose of this method is to match three important items: (i) nomenclature information (`FileType` enum), (ii) metamodel of each entity (`entities` attribute), and (iii) templates (from `Template Boundary` and `Template Control`). Regarding the operation, this method allows generating source code for three categories: logic classes, persistence classes, and business classes. These categories are identified by annotations inserted in the development phase (`AnnotationClass` – Figure 2). As *Step 4*, the `generateCode` method performs a call to `parser2Template` method of the `FreeMarkerUtils` class, transferring an entity map and template name that must be used to generate each entity. This method uses the templates provided by the software engineers, which are represented by stereotypes `Template Boundary` and `Template Control`. Moreover, the `FreeMarkerUtils` class can be considered as an abstraction for our subsystem because it encapsulates the classes of the FreeMarker TE (`freemarker.template` package). The source code files generated by the `FreeMarkerUtils` class are returned to `TemplateManager` class (`generateCode` method) so that they are formatted and inserted in their respective packages (*Step 5*). Operations for managing directories and files, both necessary in the source code generation process, are performed by the `FileManager` class. The `formatFile` method uses the `Jacobe` symbolic class to format the generated source code according to the standard established in the development phase. Finally, it is noteworthy that this process (Step 2 to 5) must be performed until the entity list (`entities` attribute – `TemplateManager` class) has been generated. Thus, the source code can be transferred to the compile module to be compiled and inserted into the execution environment.

B. Comparative Analysis

This section presents a brief comparative analysis between both modules (new and old). Next, a brief description of each feature is provided, as well as our analysis on both modules in relation to each of them, comparing their advantages and limitations.

Both modules have been developed based on the Java programming language. This language was selected because our RA is based on reflection [1]. Although the new module have been designed based on FreeMarker TE and uses the FreeMarker Template Language (FTL) for developing templates, the benefits highlighted in the next features are worthwhile. In addition, we can add the advantages already presented in Section II and the automated support by the main IDEs as complementary benefits.

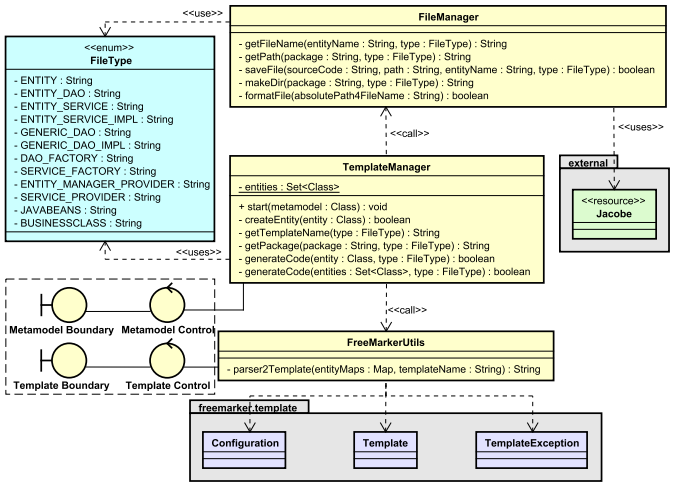

Figure 4. UML model of the source code module

Regarding the design, developers were obliged, during the development of the old module, to elaborate the "embedded templates" and processing logic (i.e., engine) of such templates in parallel. In this new version, the processing logic was abstracted to a simple class (`TemplateManager` class) and developers may devote their time in the preparation of code templates for software entities. Thus, a significant gain can be obtained when our RA is instantiated in a different domain that needs of changes for generating new entities.

As development approaches, two types were identified. The old module was developed based on internal approach and the new one based on external approach. As reported in previous feature, "embedded templates" and engine were developed by the developers in first approach (internal). This task was considered too costly (e.g., time and cost) to the developers when our RA had to be instantiated for a new application domain. On the other hand, a well-defined segmentation between code templates and engines was developed in the second approach (external). Thus, developers can coordinate their efforts in the development of code templates, since the engines of such templates were already implemented.

In the context of this paper, flexibility and maintainability can be considered related features because the first may assist or impair the second or vice-versa. According to previous paragraph, this new module has been more susceptible to change than the old one. In other words, when our RA is instantiated to meet a new domain, only the required entity templates should be developed. In addition, a significant minimization of code lines for processing templates in this new version compared to previous one must be highlighted. Therefore, this aspect can be considered highly favorable to maintain this new module.

V. Case Study

In this section, we present a case study to evaluate the applicability of the generator module of source code. The main purpose of this study is to demonstrate the real value of this module for our RA and, hence, for the SaS development. In previous work, we have instantiated our RA in Java and some validation types were conducted. Therefore, we will approach the same studies in this paper; however, emphasizing the use of the module presented in this paper. Next, a brief description of our subject application and the empirical strategies adopted for conducting this case study is addressed.

Subject Application. RA4SaS enables structural and behavioral adaptation at runtime [1]. Thus, for space reasons, only the structural adaptation was chosen for our empirical analysis, since this type of adaptation is sufficient to show the applicability of this module for the generation of source code. Thereby, a software entity will be used in the scope of this empirical study to meet two levels of granularity: (i) *association of entity*, which represents the addition of new information (i.e., entities) through the following relationships: aggregation, composition or association. In this paper, the composition relationship will be presented; and (ii) *extension of entity*, which represents the addition of new information (i.e., entities) by means of an inheritance relationship.

Empirical research strategy. Figure 5 illustrates the `Customer` software entity and the changes (i.e., association of entity and extension of entity) that will be applied to it. Before describing the case study, it is noteworthy that this entity belongs to the information system context for the bookstore management. Thus, to illustrate the first type of adaptation (i.e., *association of entity*), the original entity (`Customer`), initially developed to act in a local system, will be adapted to act in a web system with authentication. Based on this context, a `Login` entity with two attributes (`username` and `password`) must be created and added to the `Person` entity by means of a composition relationship. This entity (`Person`) receives the adaptation changes because it has an annotation (from "Annotation" module) that determines where such changes can be inserted. Then, a brief description of the adaptation process will be presented, but, for space reasons, the implementation commands are not shown in this paper.

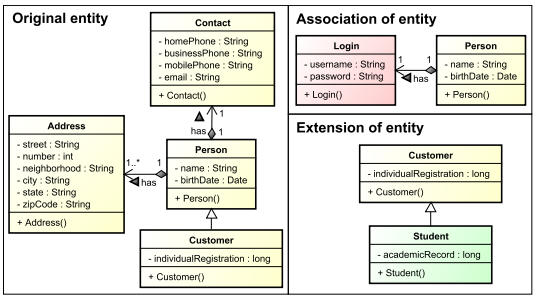

Figure 5. Adaptation types for the "Customer" software entity

To conduct the adaptation of the `Person` entity, the adaptation process of our RA will be used. In short, a metamodel (Figure 2) for this entity must be instantiated by the "Reflection" module so that the changes to be incorporated into it. Thus, a `Login` entity (i.e., an instance of `Class` – Figure 2) composed of two attributes (i.e., `username` and `password` – instance of `FieldClass`) is created in other metamodel (i.e., in a different plan). To associate these entities, the `Person`

metamodel must receive an attribute from `Login` metamodel, i.e., the entity list attribute (`entityList`) of the `Class` class. Metadata as type of relationship (e.g., composition), multiplicity (e.g., `ONE-to-ONE`), and navigability (e.g., bidirectional), also are provide in this step.

To show the *extension of entity*, it will be considered the modification of new `Customer` software entity. Now, this entity acts in a web system with authentication for bookstore management and it will be adapted to act in school management system. Similarly to the previous adaptation, the `Person` metamodel was instantiated so that a new entity was added via inheritance relationship. To do so, a metamodel for `Student` entity with `academicRecord` attribute must be created and associated with `Person` metamodel. Finally, specifically for this type of relationship, metadata determines the type of association (e.g., `EXTENDS`).

VI. Conclusions and Future Work

This paper presented the design and implementation of a module for automatic generation of source code for SaS. This module is part of a wider project, i.e., a RA4SaS that aims to support the development and adaptation of software entities at runtime [1]. Using this RA, software entities are transparently monitored and adapted at runtime without user's perception and developer's involvement. This RA uses a set of modules that coordinately work in an "assembly line", i.e., a software entity is automatically disassembled, adapted, and reassembled by the modules in an automated process. As discussed in Section IV, this new module aims to optimize the generation of source code and, at the same time, to boost the SaS development. Thus, the main contributions of this paper are: (i) SaS area, since we have a means that enables the development of self-adaptive software entities and it allows to adapt them at runtime; (ii) software architecture, because we have proposed the first RA for SaS based on reflection and, most importantly, we have worked in its evolution; (iii) software engineering community, since we believe that our RA may be adequately used together with software development processes that have been used by companies, since they seem to be complementary.

As future work, some activities are being planned: (i) the first is related to case studies that will be conducted for evaluation of the module presented in this paper, since the case study reported in Section V does not cover all cases and does not solve all problems related to software adaptation at runtime; however, other types of adaptation must be investigated; (ii) the second is related to the use of our RA in different domains, since we can get some indicators as flexibility and maintainability of this module; (iii) the third is related to execution performance of this module compared to previous one; and (iv) the fourth is related to the use of our RA in the industry, since we intend to evaluate the behavior of this module when it is applied in larger environments of development and execution. Finally, we have a positive scenario of research, intending to make this module (and our RA) become an effective contribution to the SaS 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' 13*, 2013, pp. 129–138.

[2] 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.

[3] J. V. Bergen, "Velocity or freemarker?" [*On-line*], *World Wide Web*, 2007, available in: http://www.javaworld.com/article/2077797/open-source-tools/velocity-or-freemarker.html, Accessed on March 08, 2017.

[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] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–26.

[6] F. J. Affonso and E. L. L. Rodrigues, "A proposal of reference architecture for the reconfigurable software development," in *SEKE' 2012*, San Francisco, USA, 2012, pp. 668–671.

[7] G. Coulson, G. Blair, and P. Grace, "On the performance of reflective systems software," in *PCCC' 2004*, 2004, pp. 763–769.

[8] F. J. Affonso, M. C. V. S. Carneiro, E. L. L. Rodrigues, and E. Y. Nakagawa, "A reference model as automated process for software adaptation at runtime," *IEEE Latin America Transactions*, vol. 13, no. 1, pp. 214–221, 2015.

[9] IBM, "An architectural blueprint for autonomic computing," [*On-line*], *World Wide Web*, 2005, available in: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf, Accessed on March 08, 2017).

[10] P. Maes, "Concepts and experiments in computational reflection," in *Object-oriented Programming Systems, Languages and Applications (OOPSLA' 87)*, New York, NY, USA, 1987, pp. 147–155.

[11] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *FOSE' 07*, 2007, pp. 259–268.

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

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

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

[15] 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, Accessed on March 08, 2017.

[16] 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.

[17] A. FreeMarker, "What is apache freemarker?" [*On-line*], *World Wide Web*, 2016, available in: http://freemarker.incubator.apache.org/, Accessed on March 08, 2017.

[18] L. M. Daniele, E. Silva, L. F. Pires, and M. van Sinderen, *A SOA-Based Platform-Specific Framework for Context-Aware Mobile Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 25–37.

[19] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, 2012.

[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, 2006.

[22] F. J. Affonso, G. Leite, and E. Y. Nakagawa, "Dms-modeler: A tool for modeling decision-making systems for self-adaptive software domain," in *SEKE' 16*, 2016, pp. 617–622.