Software Architecture Evolution in the Open World through Genetic Algorithms

Myriam Torres and Germán H. Alférez
Facultad de Ingeniería y Tecnología
Universidad de Montemorelos
Apartado 16-5 Montemorelos N.L. Mexico

Abstract - *In the open world, it is impossible to know at design time all the possible context events that can arise (e.g. sudden security attacks, decreased performance, or service unavailability). Moreover, it is unthinkable to fix these situations manually because of the rapid responses required in modern systems (e.g. in banking or trading). In this paper, we propose an approach to support the dynamic evolution of software architectures in the open world by means of genetic algorithms and models at runtime. Dynamic evolution actions try to preserve the expected quality attributes of the software architecture when facing unknown context events. Models at runtime are used to reason about the quality attributes to be preserved and the tactical functionality to preserve these requirements. Genetic algorithms inject the tactical functionality into the running software architecture. Our approach is supported by a running prototype.*

Keywords: Autonomic Computing, Genetic Algorithms, Dynamic Evolution, Models at Runtime, Open World.

1 Introduction

In recent years, there has been a trend to self-adapt software architectures in order to face arising context events (e.g. events in the computer infrastructure, such as security attacks). Instead of carrying out manual adjustments, which can be slow and error-prone, self-adaptation facilitates how the system can respond to changing contexts.

Most of the solutions to achieve self-adaptive software have focused on the closed-world assumption. When developing software for the closed-world, the set of all possible adaptations is known at design time. However, in the open world, it is impossible to know beforehand (i.e., at design time) all the possible context events that can arise. This is specially truth in ubiquitous and pervasive computing settings, for example in software based on cloud computing or in wearables. Therefore we need approaches to develop systems that self-adjust in the open world.

Our contribution is an approach to support the evolution of software architectures in the open world by means of Genetic Algorithms (GAs) and models at runtime. On one hand, a GA is a search heuristic that mimics the process of natural selection. GAs are especially useful for optimization, machine learning, and business applications. On the other hand, models at runtime are used during execution to reason about the quality attributes that need to be preserved at runtime and how these attributes can be preserved.

When an unexpected event occurs in the open world, GAs are used to find out the most appropriate software architecture. First, a set of possible software architectures is auto-generated. Then, these architectures are mated and a set of abstract tactics is used to mutate them. We propose the concept of *tactics* in order to extend the software architecture with new functionalities that can be used to keep the Quality of Service (QoS) level. Then, the utility function, or fitness level, of each software architecture is evaluated. The most suitable solutions are used to create new populations.

We use GAs in our approach because they are useful to search for the most appropriate solution in the large and uncertain open world. Specifically, in the open world there is a large space of possible context events (foreseen and unforeseen) and a large number of software architecture configurations that can be used to face these events. Also, the proven efficiency of GAs offers an attractive option to get a resulting software architecture in a short amount of time. Last but not least, since GAs are supported by an evolutionary approach, they offer a logical way to guide the dynamic evolution of software architectures. In this work, we see the concept of *evolution* as the gradual and continuous growth of the software architecture. *Dynamic evolution* does not imply just punctual adaptations to punctual context events but a gradual structural or architectural growth into a better state.

The remainder of this paper is organized as follows. Section II presents the background. Section III presents a motivating scenario. Section IV describes our approach for the dynamic evolution of software architectures. Section V presents the tool support. Section VI describes related work and Section VII presents conclusions and future work.

2 Underpinnings of Our Approach

Nowadays, an important trend in software engineering is to support the automation of decision-making at runtime with Autonomic Computing [1]. Instead of doing manual reconfigurations of software architectures to adapt to changes in the context, it is desirable that the software architecture self-adapts during execution. In order to achieve this goal, we propose an approach based on the following concepts:

1. ***Context:*** Applications should be aware of their contexts and automatically adapt to their changing contexts in order to provide adequate services to users. Specifically, "context is any information that can be used to characterize the situation of an entity" [2]. A system is context-aware if it can extract, interpret and use context information and adapt its functionality to the current context of use.

2. ***Dynamic adaptation vs. Dynamic evolution:*** The self-adaptive-software community is concerned with the increasing complexity of the context [3]. This complexity is caused by systems that are moving from the closed world to the open world (e.g. ubiquitous and pervasive computing).

Under the closed-world assumption, the possible context events are fully known at design time. These events will eventually trigger the dynamic adaptation of the software architecture. Nevertheless, it is difficult to foresee all the possible situations arising in uncertain contexts where the system runs. Therefore, the software architecture should react to continuous and unanticipated changes in complex and uncertain contexts for a better functioning.

We define *dynamic evolution* as the process of moving the software to a new version, which cannot be supported by predefined dynamic adaptations, in order to manage unknown context events at runtime [4]. We refer to *unknown context events* as those arising situations in the context that have not been foreseen at design time. *Uncertainty* is caused by how the software architecture should deal with these unknown context events.

3. ***Autonomic Computing:*** It is a branch of software engineering concerned with creating software systems capable of self-management [1]. The term *autonomic* is derived from human biology. For example, the autonomic nervous system monitors heartbeat without any conscious effort. In a similar way, self-managing autonomic capabilities try to resolve problems with minimal human intervention.

4. ***Models at Runtime:*** Models at runtime can be defined as "causally connected self-representations of the associated system that emphasize the structure, behavior, or goals of the system from a problem-space perspective" [5]. In response to changes in the context, the system itself can query these models to determine the necessary modifications in the underlying architecture.

5. ***Evolutionary Algorithms:*** Evolutionary Algorithms (EA) are a branch of artificial intelligence for solving search and optimization problems. EA includes search methods that allows to treat optimization problems where the objective is to find a set of parameters for a function adaptation. From this set, one of the most popular models are GAs [6].

GAs work with a set (population) of candidate solutions (individuals) with the best capability (adaptation) [7]. The evolution usually starts from a population of randomly generated individuals and happens in generations. The population changes as an iterative process (generations) where individuals that have better capabilities are likely to survive and move to the next generation and participate of the genetic operators. In each generation, the fitness of every individual in the population is evaluated. Multiple individuals are selected from the current population (based on their fitness), and modified to form a new population. New individuals are created by means of using genetic operators, namely selection, crossover, and mutation.

3 Motivating Scenario

In order to illustrate the need for dealing with unexpected context events in the open world, we introduce a critical system that supports on-line product shopping at EUROTECH, a multinational retailer of technology products. In Figure 1, the UML is used to design the software architecture in terms of software components.

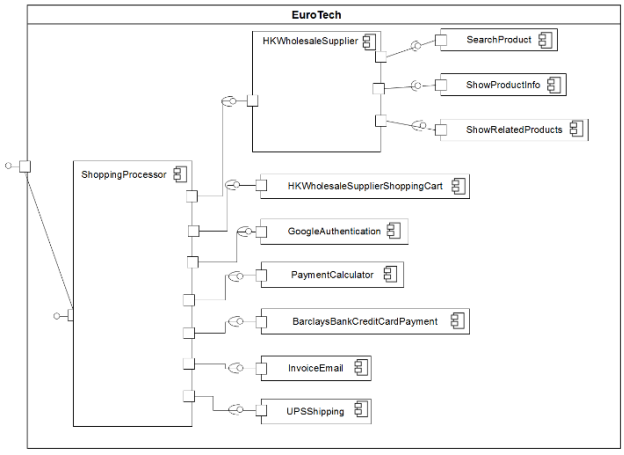

Figure 1. EUROTECH software architecture

The operation for product searching is provided by the SEARCHPRODUCT component, which is part of the HKWHOLESALESUPPLIER component. The product information is sent to the customer by the SHOWPRODUCTINFO component and the information for other related products is listed by the SHOWRELATEDPRODUCTS component. Customers can add products to the shopping cart through the HKWHOLESALESUPPLIERSHOPPINGCART component. When the customer is ready to checkout, he or she is authenticated by the GOOGLEAUTHENTICATION component. The PAYMENT CALCULATOR component calculates the total amount to be paid. The payment is done through the BARCLAYSBANKCREDITCARDPAYMENT component. Finally, the in-house EMAILINVOICE component sends an e-mail to the customer with the invoice and the UPSSHIPPING component is invoked to deliver the product.

In order to support the dynamic adaptation of the system to keep this process available 24/7, systems engineers have programmed a set of predefined adaptation actions for specific context events. For instance, if the BARCLAYSBANKCREDITCARDPAYMENT component is unavailable, then other components can be invoked instead.

Nevertheless, implementing predefined adaptation actions has the following drawback: If there are not predefined adaptation actions for a particular context situation, then no adaptation is carried out. This situation helps us to identify the following *challenges* for context-aware systems in the open world: 1) context-aware systems should be able to count on corrective actions that trigger the dynamic evolution of the system to preserve the expected requirements when facing unknown context events; and 2) the dynamic evolution of the software architecture should be carried out by auto-generated evolution actions in order to avoid human intervention. The following section describes our approach to face these challenges.

4 Dynamic Evolution of Software Architectures

Manual dynamic adaptation of software is unfeasible in complex computational scenarios that require prompt response and high availability. Moreover, critical software, such as software that supports power grids, cannot be stopped in order to inject new functionality or make changes. Therefore, our approach is focused on supporting autonomic adjustments of the software architecture. To this end, our solution is based on IBM's reference model for autonomic control loops [1] (which is sometimes called the MAPE-K loop). By following the principles of the MAPE-K loop, it is possible to inject autonomic behavior into the system.

Figure 2 describes the pieces of our approach in terms of the MAPE-K loop. The CONTEXT MONITOR retrieves information from the context by means of sensors (e.g. one sensor measures performance and another one observes security levels). In turn, the EVOLUTION PLANNER analyzes the collected context information and looks for quality attributes that can be negatively affected by unknown context events (i.e., unforeseen at design time). Also, the EVOLUTION PLANNER plans dynamic evolutions by looking for surviving tactics to preserve the software architecture despite unknown context events. Afterwards, the GA EVOLVER executes the dynamic evolution of the software architecture by means of GAs.

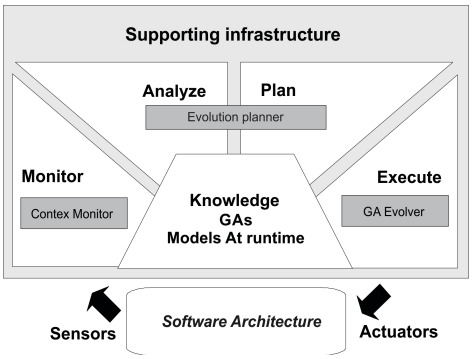

Figure 2. Our approach for dynamic evolution

The steps that are carried out by our approach to face unknown context events in the open world are as follows:

1. **Observe the Context:** The objective of this step is to get information from the context. This information will be used later to trigger dynamic adjustments on the software architecture. To this end, we propose a CONTEXT MONITOR that processes context information that is collected by sensors and updates an ontology accordingly. The CONTEXT MONITOR captures the basic metrics of significant quality attributes from the context. The CONTEXT MONITOR and the underlying ontology are described in our previous work [8, 9].

2. **Look for Quality Attributes that Can Be Affected.** The objective of this step is to look for the *quality attributes* that can be negatively affected by unknown context events. Quality attributes are the basis of software architectures [10]. Therefore, quality attributes, such as accuracy, security, reliability, availability, and performance must be preserved at runtime despite arising unknown context events.

In order to find the quality attributes that can be negatively affected by unknown context events, we propose the EVOLUTION PLANNER, which uses forward chaining. Forward chaining evaluates arising context facts

against rule premises, which are defined at design time and kept in a knowledge base. In our previous work, we define how forward chaining can be used to realize if a particular fact (an unknown context event) can trigger an evolution [4]. For example, it is possible to find out that the HKWHOLESALESUPPLIER component can affect the HIGH SECURITY quality attribute when this component has rapidly increased its execution time (an unknown context event).

3. **Look for a Tactic to Face the Unknown Context Event:** In our approach, *tactics* are considered as the last resort to be used when the system does not have predefined adaptation actions to deal with arising context events. These tactics are expressed in a *requirements model.* This model is leveraged at runtime to count on the representation of the requirements, including quality attributes, which the context-aware system must preserve at runtime. Requirements in this model have to be fulfilled despite arising unknown context events. Since we are particularly interested in keeping quality attributes or non-functional requirements (NFRs) at runtime (e.g. performance), the EVOLUTION PLANNER uses GRL [11].

Figure 3 depicts the requirements model for the on-line product shopping system at EUROTECH. This model has softgoals that describe the quality attributes to be kept by the context-aware system in order to reach the top-level goal (e.g. the HIGH SECURITY softgoal). It also contains tasks that specify specific surviving tactics to reach the softgoals (e.g. the DECEPTION task). Since tasks represent core assets to keep the QoS of the system, they make a positive contribution to softgoals. The tactics with the highest contributions (i.e., the ones with "++") are chosen first by the EVOLUTION PLANNER.

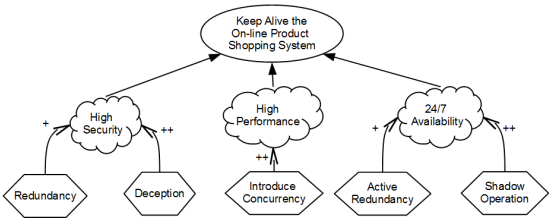

Figure 3. Requirements model

4. **Generate an Initial Population:** In this step, an initial population of chromosomes is auto-generated by the GA EVOLVER at runtime. Each chromosome is given a random collection of genes. Each gene represents a particular software component (see Figure 4).

A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. Each chromosome in the initial population represents a fully-functional software architecture.

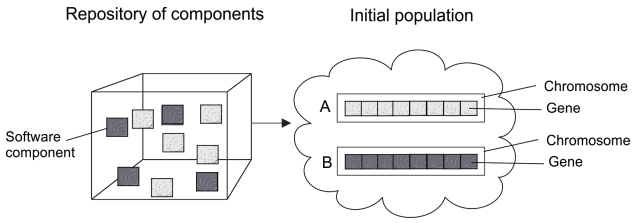

Figure 4. Generation of the initial population of components

Genes are taken from a repository of components. A *repository* is a storage site for objects of some sort. In other words, a repository stores information about an organization's assets. It can store information, store-in-depth documentation, support for versioning and change control, generate name conventions, etc.

For example, in our repository we have common and variant software components. *Common components* have to be present in all the generated chromosomes. However, *variant components* are present in particular chromosomes. For example, on one hand, the GOOGLEAUTHSERVICE is a common component that is present in all the chromosomes. On the other hand, the functionality to look for products can be implemented by means of two alternative variant components, the HKWHOLESALESUPPLIER and the AMERICAN SUPPLIES Co. components. Any of these variants can fulfill the expected functionality.

In order to decide which genes (components) have to be present in each chromosome, we make use of a feature model [12]. This model describes the dynamic software architecture configurations, the variants of the software architecture, and constraints among functionalities [8]. Feature modeling was chosen because it offers coarse-grained variability management and has good tool support.

5. **Crossover and Mutation:** Crossover combines the properties of two chromosomes of the previous population to create new chromosomes (software architectures in our case). Mating is achieved by selecting two parents and taking a "splice" from each of their gene sequences. Mutation is used to introduce new genetic material into the population. In our approach, mutations are the tactical functionalities that are used to evolve a chromosome in order to preserve the quality attributes that can be negatively impacted by unknown context events.

For instance, Figure 5 shows how software architectures A and B are mated. The result is software architecture C. In the second step of this section, we described an example in which our solution discovers an unknown context event: the HKWHOLESALESUPPLIER

component can affect the HIGH SECURITY quality attribute. In this case, GAs are used to insert the functionality of the DECEPTION tactic into the resulting software architecture (see Figure 3). We consider this insertion as a mutation. This tactical mutation will try to preserve the security quality attribute in a particular generated architecture.

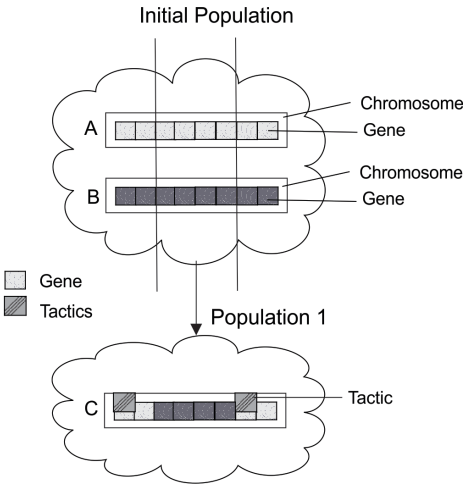

Figure 5. Crossover and mutation of tactical functionalities

6. **Selection:** In this step, the population is subjected to a selection process that favors individuals better adapted. Each chromosome in the population must be evaluated. This is done by evaluating its "fitness" or the quality of its solution. The fitness is determined through the fitness (or utility) function, making a summation of the functionality (together with the mutated tactics) of the genes that make up the chromosome. Based on the fitness level of each chromosome, it is possible to select the chromosomes that will mate, or those that have the "privilege" to mate in further generations. For example, security levels can be evaluated with different metrics in each resulting chromosome. The chromosomes with the best results will have the best fitness.

Steps 5 and 6 are repeated until the best solution has not changed for a preset number of generations. Finally, our GA approach returns a software architecture that can be used to face the arising unknown context events.

5 Tool Support

A prototype system validates the feasibility of our approach. The CONTEXT MONITOR and the EVOLUTION PLANNER are described in our previous work [4, 8, 9]. The CONTEXT MONITOR uses the SPARQL Protocol and RDF Query Language (SPARQL)[1] to query the ontology with context information. The EVOLUTION PLANNER uses the EMF Model Query (EMFMQ)[2] to manage the feature model and the requirements model at runtime [4]. The GA EVOLVER is implemented in Java. Changes in the evolved software architecture can be applied on the underlying system by means of different technologies, such as OSGi bundles[3]. These bundles can be activated or deactivated at runtime according to the evolutionary process.

Figure 6 shows the screenshot of our running prototype, a dynamic evolution manager that implements the components of IBM's MAPE-K loop for the open world. This prototype presents the logs of the activities that are carried out by the CONTEXT MONITOR, the EVOLUTION PLANNER, and the GA EVOLVER when an unknown context event is detected.

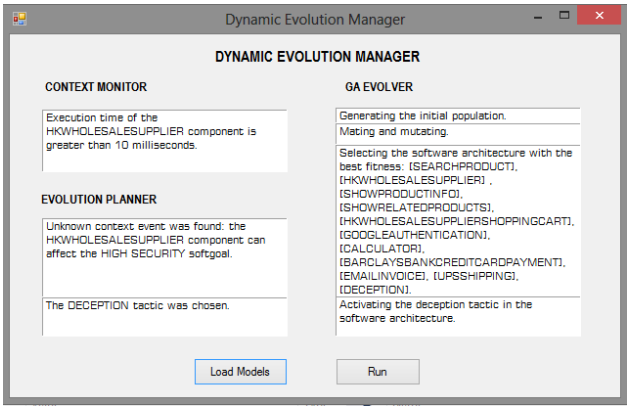

Figure 6. Running prototype

In the scenario of Figure 6, the CONTEXT MONITOR detects that the execution time of the HKWHOLESALESUPPLIER component is greater than 10 milliseconds. This is an event that was not predefined at design time. Afterwards, the EVOLUTION PLANNER realizes that this unknown context event can negatively affect the HIGH SECURITY quality attribute. As a result, it looks for a tactic to preserve this quality attribute and it finds the DECEPTION tactic. Then, the GA EVOLVER uses this information to choose the software architecture with the best fitness. At the end of the process, our GA-oriented approach mutates the original software architecture with the DECEPTION tactic. This tactic will try to deceive the hackers that are trying to violate the security of the HKWHOLESALESUPPLIER component at a particular time. As a result, the expected HIGH SECURITY quality attribute is preserved.

[1] http://www.w3.org/TR/rdf-sparql-query/: SPARQL
[2] http://www.eclipse.org/modeling/emf/: EMF Model Query
[3] http://www.osgi.org/Main/HomePage/: OSGi

Preliminary evaluation results show that using GAs to evolve the software architecture is a promising option for the unpredictable open world. This is because of the fact that GAs can be used to optimize a large set of non-linear systems (software architectures) with a large number of variables (software components).

6 Related Work

Over the past decade, a large number of research works have been concerned with self-adaptation. We give an overview of relevant approaches that support self-adaptation.

The MUSIC project [13] focuses on developing self-adaptive mobile applications. The authors use an explicit representation of the environment, in particular an ontology-based model. MUSIC adopts utility functions as adaptation mechanism. MUSIC uses a system model based on a system component meta-model representing the system structure. The variability is implicit in the system model. The main variability mechanism consists in loading different implementations for each component type of the architecture. The system, environment and adaptation representations are fixed at design time. However, MUSIC does not provide mechanisms to manage unexpected changes in the architecture at runtime.

Morin et al. [14] propose a combination of model-driven engineering and aspect-oriented modeling to support self-adaptation. This approach keeps an explicit representation of the system. The system is modeled using a base model that contains the common functionalities and a set of variant models that can be composed with this base model. The variant models capture the variability of the adaptive system. An adaptation model specifies which variant have to be selected depending on the environment of the running application. As adaptation mechanisms they use adaptation rules that specify how the system should adapt to its context. All the models are fixed at design time and cannot be extended at runtime to incorporate unanticipated elements.

RAINBOW [15] is an architecture-based framework to support self-adaptation of software systems. At runtime, the authors use an explicit representation of the system, specifically an abstract architectural model. The adaptation mechanism is explicit and it is based on ECA rules. Since the variability is implicit, it is encapsulated into the adaptation rules. RAINBOW also uses an implicit representation of the context. RAINBOW does not provide mechanisms to allow the modification of the context, system, and rules representations at runtime.

Zhang et al. [16] introduce an approach for creating formal models to support self-adaptive system behavior. Specifically, they use state machine based models (such as Petri nets) to model the system's adaptive behavior. Contextual changes guide the transitions among system states. Then, the adaptation mechanism is explicit through rules. In this work, the underlying mechanisms for dynamic adaptations are fixed at design time and cannot be extended at runtime.

PLASTIC [17] focuses on service-oriented systems. PLASTIC maintains an explicit model of the system. Application alternatives are stored in a repository. However, no new alternatives can be added at runtime. PLASTIC does not provide mechanisms to extend the system, the context and system variants at runtime.

CAPucine [18] builds adaptive systems based on services. CAPucine considers the environment implicitly inside the system. The variability is managed explicitly by means of a feature model. The system is represented explicitly through models. The adaptation mechanism is explicit. A series of rules are stored in a repository. This approach defines all the underlying elements at design time.

CASA [19] provides a framework for enabling dynamic adaptation of applications executing in dynamic contexts. Adaptation mechanisms are defined explicitly by a set of adaptation policies. CASA does not provide mechanisms to modify or extend the specification of the software architecture at runtime.

In the aforementioned approaches, adaptation is fully foreseen at design time. Systems have a fixed set of adaptation actions and new behaviors cannot be introduced during runtime in the software architecture. The trend has focused on self-adaptive mechanisms that are not open to evolution in the open world. According to our best knowledge, our work presents the first generic approach based on GAs to handle unknown context events through the dynamic evolution of the software architecture.

7 Conclusions and Future Work

In this paper, we have presented an approach for the dynamic evolution of software architectures in the open world through GAs. Specifically, our approach deals with unexpected context events and has several benefits: 1) it can guide the creation of context-aware systems that self-evolve when facing unknown context events in order to preserve quality attributes; 2) it can be applied to different domains; and 3) human workload is reduced thanks to the autonomic evolution of the system.

As future work, we will answer the following questions related to the verification of the evolved software architecture: Does a merged tactic accomplish its objective at runtime? Or does the quality attribute continue decreasing? How to verify that the software architecture does not grow excessively with a large amount of merged tactics, which can make the software architecture complex or slow? In this way, some tactics could be automatically removed when the software architecture has reached a stable state.

8 References

[1] J. O. Kephart, D. M. Chess, 2003. The vision of autonomic computing, Computer, vol. 36, no. 1, pp. 41–50.

[2] A.K. Dey, 2001. Understanding and Using Context. Personal and Ubiquitous Computing Journal, Vol. 5 (1), pp. 4–7.

[3] M. Prehofer, C. Schäfer, W. Schlichting, R. Smith, D. Sousa, J. Tahvildari, L. Wong, K. Wuttke, J., 2013. Software engineering for self-adaptive systems: A second research roadmap. In: Lemos, R., Giese, H., Müller, H., Shaw, M. (Eds.), Software Engineering for Self-Adaptive Systems II. Vol. 7475 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–32.

[4] G. H. Alférez, V. Pelechano, 2012. Dynamic evolution of context aware systems with models at runtime. In: R. France, J. Kazmeier, R. Breu, C. Atkinson, (Eds.), Model Driven Engineering Languages and Systems. Vol. 7590 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 70–86.

[5] G. Blair, N. Bencomo, and R. B. France, October 2009. Models@run.time, Computer, vol. 42, pp. 22-27.

[6] D. Ashlock, 2006. Evolutionary Computation for Modeling and Optimization. Springer.

[7] M. Mitchell, 1996. An Introduction to Genetic Algorithms. MIT Press.

[8] G. H. Alférez, V. Pelechano, 2011. Context-aware autonomous Web services in software product lines. In: Proceedings of the 2011 15th International Software Product Line Conference. SPLC'11. IEEE Computer Society, Washington, DC, USA, pp. 100–109.

[9] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, D. Diaz, May 2014. Dynamic adaptation of service compositions with variability models, Journal of Systems and Software, Volume 91, Pages 24-47.

[10] L. Bass, P. Clements, R. Kazman. 2013. Software Architecture in Practice, Third Edition. Pearson Education, Inc.

[11] L. Liu, E. Yu, 2004. Designing information systems in social context: a goal and scenario modelling approach. Inf. Syst. 29, 187–203.

[12] D. Batory, 2005. Feature Models, Grammars, Propositional Formulas. In Software Product Lines Conference, ser. Lecture Notes in Computer Sciences, vol. 3714, Springer-Verlag. Springer-Verlag, 2005, p. 7-20.

[13] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, U. Scholz, 2014. MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments, in Software Engineering for Self-Adaptive Systems, ser. Lecture Notes in Computer Science, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, Eds. Springer Berlin / Heidelberg, vol. 5525, pp. 164–182.

[14] B. Morin, O. Barais, G. Nain, J. M. Jezequel, 2009. Taming dynamically adaptive systems using models and aspects. In IEEE 31st International Conference on Software Engineering. ICSE'09. pp. 122–132.

[15] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, oct. 2004. RAINBOW: architecture-based self-adaptation with reusable infrastructure, Computer, vol. 37, no. 10, pp. 46–54.

[16] J. Zhang, B. H. C. Cheng, 2006. Model-based development of dynamically adaptive software, in Proceedings of the 28th international conference on Software engineering, ser. ICSE'06. New York, NY, USA: ACM, pp. 371–380.

[17] M. Autili, P. Di Benedetto, P. Inverardi, 2009. Context-aware adaptive services: The PLASTIC approach, in Fundamental Approaches to Software Engineering, ser. Lecture Notes in Computer Science, M. Chechik, M. Wirsing, Eds. Springer Berlin / Heidelberg, vol. 5503, pp. 124–139.

[18] C. Parra, X. Blanc, L. Duchien, 2009. Context awareness for dynamic service-oriented product lines, in Proceedings of the 13th International Software Product Line Conference, ser. SPLC'09. Pittsburgh, PA, USA: Carnegie Mellon University, pp. 131–140.

[19] A. Mukhija, M. Glinz, 2005. Runtime adaptation of applications through dynamic recomposition of components, in Systems Aspects in Organic and Pervasive Computing. ARCS'05, ser. Lecture Notes in Computer Science, M. Beigl, P. Lukowicz, Eds. Springer Berlin / Heidelberg, vol. 3432, pp. 124–138.