ISSN 2278-5612

Using Product Lines Techniques to Specify Self-Adaptative Systems

Chiraz BOUZID[1], Naoufel KRAIEM[2], Camille SALINESI[3]
RIADI Lab, ENSI, Campus of Manouba, Tunisia
Chiraz_bouzid@yahoo.fr
RIADI Lab, ENSI, Campus of Manouba, Tunisia
naoufel.kraiem@gmail.com
Université Paris 1 Panthéon – Sorbonne, Centre de Recherche en Informatique
camille.salinesi@univ-paris1.fr

ABSTRACT

Dynamic software adaptability is one of the central features leveraged by autonomic computing. However, developing software that changes its behavior at run time in response to dynamically varying user needs and resource constraints is a challenging task. With the emergence of mobile and service oriented computing, such variation is becoming increasingly common, and the need for adaptivity is increasing accordingly. Software product line engineering has proved itself as an efficient way to deal with varying user needs and resource constraints. In this paper we present an approach to specifying adaptive systems based on product line oriented technique such as variability modeling: we propose to combine goal modeling techniques to represent architectural and environmental variability, with constraint programming to provide the analyst with a means to identify the system variants best suited to the various environmental contexts that a system might encounter at runtime.

KEYWORDS

Dynamic Software Adaptability, Software Product Line Engineering, variability modeling, Dynamic Software Product Line.

Council for Innovative Research

Peer Review Research Publishing System

Journal: International Journal of Management & Information Technology

Vol. 5, No. 2
editor@cirworld.com
www.cirworld.com, member.cirworld.com

1. INTRODUCTION

Today's society increasingly depends on software systems deployed in large companies, banks, airports, telecommunication operators, and so on. These systems must be available 24/7 for very long period [15]. To sustain the availability, the system should be able to adapt to different execution contexts, with no interruption, and with no human intervention. To be able to run for a long period, systems should be open to evolution. Indeed it is impossible to predict what the user requirements will be in 10 years. A promising approach is to implement such critical systems as Dynamically Adaptive Systems (DASs), which are able to adapt according to their execution context, and even evolve according to changing user requirements. Software systems must then become more and more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments and system requirements.

The topic of self-adaptive systems has been studied within the different research areas of computer science from software engineering, including, requirements engineering [21], software architectures, middleware [25], component-based development [14], and programming languages [22] to Artificial Intelligence and robotics [23]. However most of these initiatives have been isolated and until recently without a formal forum for discussing its diverse facets [1].

With the emergence of mobile, pervasive, and service oriented computing, such dynamic variation in user needs and available resources is becoming more and more widespread. This raises a series of questions such:

How to specify requirements of self-adaptive system? What aspects of the environment should the self-adaptive system monitor? Clearly, the system cannot monitor everything. And exactly what should the system do if it detects a less than optimal pattern in the environment? Presumably, the system still needs to maintain a set of high-level goals that should be maintained regardless of the environmental conditions. But non critical goals could well be relaxed, thus allowing the system a degree of flexibility during or after adaptation. And how do designers specify how system feature impacts Quality of Service (QoS) properties?

One of the main challenges that self-adaptation poses is that when designing a self-adaptive system, we cannot assume that all adaptations are known in advance--that is, we cannot anticipate requirements for the entire set of possible environmental conditions and their respective adaptation specifications.

Product line engineering has proved itself as an efficient way to deal with varying user needs and resource constraints [24]. In fact, software product lines (SPLs) and adaptive systems aim at variability to cope with changing requirements. However, A self-adaptive system that uses architectural adaptation may be conceptualized as a *dynamic* SPL [2] in which each configuration is one of the possible variants of the SPL. In a self-adaptive system, these variants may be bound dynamically and emergent.

Software Product Lines (SPL) are families of software products sharing common behavior and differentiating in base of functionalities called features. SPL engineering has the goal of reducing time and effort in the development of applications in the same family. Traditional techniques use Feature-oriented Programming (FOP) to reason about features combinations and representing features in the language [8]. Dynamic Software Product Lines (DSPL) [9] have been recently explored to support adaptive systems by switching among the available features at run time [6, 13]. However, the approaches proposed so far are at the architectural level.

In this paper, we present our approach to specify adaptative systems; this approach is based necessarily on ideas from software product line engineering. To handle the questions in particular the problem of specifying DSPL, we propose to combine goal modeling techniques with constraint programming. On the one hand, goal modeling supports reasoning on quality of service (QoS) and helps represent architectural and environmental variability. On the other hand, feature modeling provides the analyst with a means to specify which system variants best suit the various environmental contexts that the system shall encounter at runtime.

The remainder of this paper is structured as follows. Section 2 places our work in the context of related work. Section 3 discusses the idea that Adaptative system can be conceptualized as Dynamic SPL. Section 4 presents the first step of our approach which consists in Goal Modeling. Section 5 presents the second step which consists on Constraint Modeling. Section 6 concludes the paper with a summary and highlights future work.

2. EXISTING WORK

As demonstrate by Cheng and Atlee [1] Requirements engineering for self-adaptive systems appears to be a wide open research area with only a limited number of approaches yet considered. They explain how preliminary work on personalized and customized software can be applied to adaptive systems. In addition, some research approaches have successfully used goal models as a foundation for specifying the autonomic behavior [5] and requirements of self adaptive systems [4]. One of the main challenges of self-adaptation is that when designing a self-adaptive system, we cannot assume that all adaptations are known in advance: it is difficult to anticipate the needs and all possible environmental conditions and their respective specifications of adaptation. The consequences are manifold. On the one hand, the requirements for self-adaptive systems may involve degrees of uncertainty or may necessarily be specified as "incomplete". The specification of a complete collection of requirements should cope with: (1) incomplete information on the environment (2) the diversity of behaviors that the system must adopt and (3) the changing demands while software is running.

Self-adaptation is currently an important issue in several research areas. Mobile computing, grid computing, SOA (service oriented architecture), and autonomic computing are such areas that require system adaptability at run-time. Several approaches to self-adaptation have been proposed, but not all support essential properties such as software extensibility and reusability in a satisfying way [30]. Programming language features, such as conditional expressions,

parameterization and exceptions, are widely used. In these approaches, adaptation and application behaviors are intertwined. The lack of separation of concerns makes the software complex and evolution difficult. Differently, external approaches where adaptation mechanisms are realized by reusable application independent middleware components relieve the applications from the adaptation concerns [31], [32], [33]. To enable the middleware to adapt an application, a representation of the application architecture model that describes variability must be made available for the middleware.
MADAM [12, 13] uses the adaptation capabilities offered by a middleware platform, and treats DASs as software product lines [13]. MADAM takes into account the benefits of coarse-grained variability mechanisms. In the MADAM approach, variants are treated as configurations, not simply components, in the same way that component frameworks support variants in Genie and OpenCOM. MADAM also uses the configurator pattern for event-based reconfiguration. Our approach is more general since the focus of MADAM is restricted to mobile computing applications, which Genie can also support [14].
Dynamic Software Product Lines (DSPL) [9] have been recently explored to support adaptive systems by switching among the available features at run time [13, 6]. This approach was evaluated using the GridStix case study [10, 11]. GridStix is a wireless sensor network for flood prediction that has been deployed on the River Ribble in North West England.
Several works have proposed techniques to simplify the configuration of software products [19-20] but they focus on configuration at design or launch time and do not address reasoning and reconfiguration at runtime.

3. DYNAMIC SOFTWARE PRODUCT LINE

3.1. The Principle of DSPL

In the software reuse community, Dynamic Software Product Line (DSPL) [2] has been proposed as an effective paradigm to develop self-adaptive systems with the principle of software product line (SPL) engineering. DSPL identifies the reusable and dynamically reconfigurable core assets at development time which are explicitly modeled as dynamic variability. At runtime, DSPL application proposes to configure and reconfigure runtime instances by the variability customization, which means to adapt the binding decisions of the variations within the current system during execution. The business logic of a DSPL application covers the adaptable behaviors which can be represented as a domain model.

3.2. DSPL Example of a Course Selection System (CSS) [29]

The Course Selection System is a DSPL example whose feature model (in Figure 1) as well as its adaptation rules (in Table 1) is identified before the system is running. The system is endowed with the capability of self-adaptation so that it can provide a stable online service facing the course-selecting demand from thousands of students in a campus. The adaptation capabilities are formalized as the ECA (*On* Event *If* Condition *Do* Action) [28] rules which indicate the operations upon the dynamic variation points in the feature model.

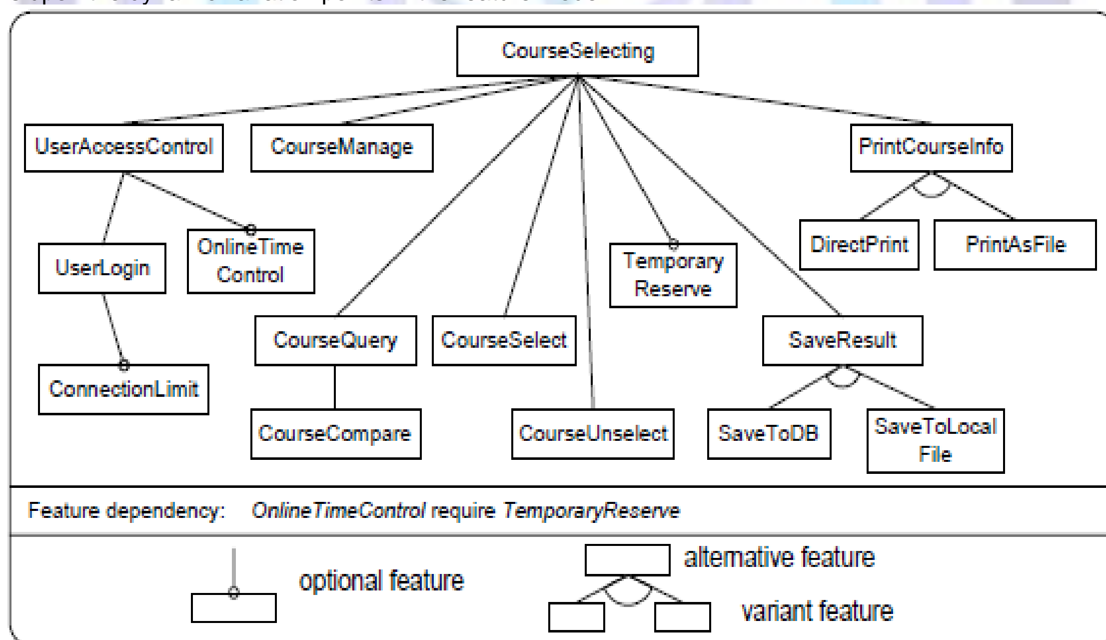

Fig 1: Feature model of the course-selecting system

The self-management capability helps to generate the variations in the business model which represents the possible configurations that the system may behave at runtime. For example in the figure, the feature *OnlineTimeControl* is an optional feature that can be bound or unbound according to the changing concurrent accessing number specified in the first rule. *TemporaryReserve* which can keep the unsaved user operations for a period of time if the user is disconnected is required by *OnlineTimeControl*. It means the former cannot be bound if the latter is not bound. *ConnectionLimit* is another optional feature whose binding status depends on the available server memory. *SaveResult* is an alternative

feature whose variants are *SaveToDB* and *SaveToLocalFile* separately. Thus, the saving mode can be changed at runtime conforming to the availability status of the database which is evaluated through the response time shown in the third rule. *PrintCourseInfo* is similar with the previous alternative feature that its binding strategy of its variants depends on the availability of the printer (the forth rule).

Table 1: The ECA rules for CCS

ON	IF	DO
The concurrent accessing number (CAN) changes	CAN > 500	Bind *OnLineTimeControl*
	CAN < 450	Unbind *OnLineTimeControl*
The memory utilization (MU) changes	MU > 90%	Bind *ConnectionLimit*
	MU < 80%	Unbind *ConnectionLimit*
Database response time (DRT) changes	DRT > 3s	Bind *SaveToLocalFile*
	DRT < 1s	Bind *SaveToDB*
Printer state (PS) changes	PS = out-of-srevice	Bind *PrintAsFile*
	PS = in-service	Bind *DirectPrint*

3.3. Adaptative Systems as Dynamic Software Product Line

A dynamically adaptive system (DAS) developed using architectural adaptation [3] (affecting the structure of the application) can be conceptualized as a dynamic software product line (DSPL) in which variability is bound at run-time, and each component based configuration can be considered as a product or variant of the DAS. This means that variants of the DAS product line can be produced at runtime. The DAS product line defines a core reference architecture that constrains each product variant's component configuration. This is an attractive notion because work on SPLs has resulted in understanding of a number of ways to represent and reason about **architectural variability**.

Conventional SPL modeling notations allow the analyst to model architectural variability but they cannot easily model two properties essential for the class of DSPLs that interest us; *environmental variability and Quality of Service (QoS).*

The efficiency of SPLE approaches was well evaluated for static systems, but it was little explored for self-adaptatif systems. It would be therefore desirable to explore this way. Our goal is to allow systems to respond more flexibly to changing environmental contexts. This object and the problem of specifying a DSPL can be reduced to constraint satisfaction problem. By combining goal modeling techniques with constraint programming, to provide the analyst with a means to identify the system variants best suited to the various environmental contexts that a system might encounter at runtime.

Goal modeling supports the modeling of environmental variability and QoS (Quality of Sevice) as well as a subset of the concepts needed for the effective representation of architectural variability [7]. A goal model can be mapped to a constraint program, which has the added capability to represent properties such as excludes relationships that are hard to represent in goal models.

4. MODELING VARIABILITY IN DSPL

Central to the modeling of variability is the notion of *feature*, originally defined by Kang et al. as *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"* [26]. Customers and engineers refer to product characteristics in terms of what features a product has or delivers, so it is natural to express any commonality and variability between products also in terms of features [27].

In our approach, for each feature of the system, we should precise how it impacts QoS properties. We should also specify which properties to optimize in which contexts. At runtime, the system should find the best selection of features, which best optimize the properties that are important in the current context.

Environmental variability may be considered to be an outcome of DSPL domain engineering [7]. In fact environmental variability needs to be represented explicitly and context variables identified that can be monitored at runtime in order to detect when an adaptation needs to be triggered. It is also critically important to model QoS, and how the required QoS trade-offs vary with context.

Goal modeling (where goals derive requirements) has been used for modeling variability [16]; Requirement is used to identify what specific adaptation mechanisms are needed to realize the adaptations among requirement models. Goal-Oriented Requirements Engineering (GORE) languages such as i* and KAOS are based on the Beliefs, Desires and Intentions (BDI) model developed for agent based systems, and BDI maps well to self-adaptive systems. With BDI, desires represent the system goals, and intentions represent how the goals can be realized. Desires and intentions are concepts with utility for understanding the requirements for any system, self-adaptive of otherwise. However, the belief concept is particularly useful for self-adaptive systems because beliefs represent a model of the environment used to inform a system's goals. In GORE languages, goals correspond to desires, goal operationalizations represent intentions and a variety of means can be used to represent beliefs. A key advantage of GORE languages is the support they provide

for reasoning about how different goal operationalizations satisfy QoS requirements (which can be modeled directly as *soft*goals).
Figure 1 shows a simplified goal model represented as a variant of KAOS [17].

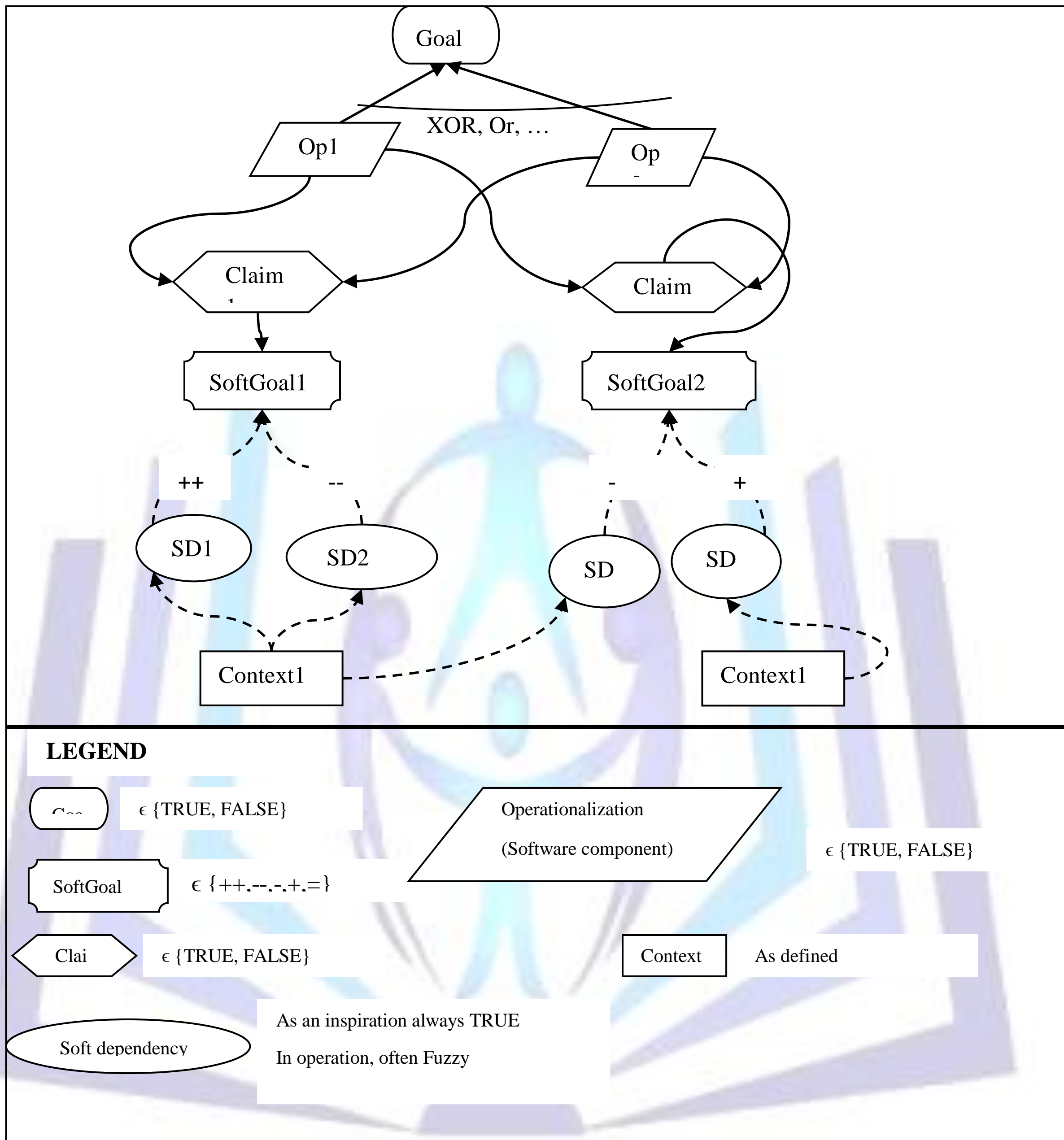

Fig 2 : Goal Model

The main elements in the models are:
Goals indicate the purpose of the system. Goals are either satisfied (true) or denied (false).
Softgoals are goals that are not that sharpely defined [base on clear-cut, black-and-white notion of goal achievement], but are goals nonetheless. They are not clear-cut because their meaning is not objectively known. The extent to which a soft goal is satisfied is modeled on a five-point ordinal scale: complete denial (--), partial denial (–), neutral or undefined (=), partial satisfaction (+), complete satisfaction (++).
*Goal **operationalizations**,* specify how software components, subsystems or humans can be assigned responsibility for satisfaction of a goal. Where there are more than one operationalizations linked to a goal, it indicates that there is more than one way to achieve the goal. In this case the goal acts as a *variation point* and the set of possible system configurations, as defined by the set of possible operationalizations that may be bound at any instant in time, represent the system's *architectural variability*.
Claims (which we model using KAOS assumptions) indicate assumptions about how operationalizations contribute to the satisfaction of softgoals.

Context, shown as rectangles, is an abstraction over a part of the system's environment, and is monitored at runtime by sensing. Context represents *environmental variability*.
Soft constraints express a required level of softgoal satisfying in a particular context. They are soft in the sense that it may prove impossible to satisfy them in all contexts.
Note that claims and soft dependencies are orthogonal. Claims specify the QoS expected from particular operationalizations, while soft dependencies specify the QoS required under particular contexts.

5. CONSTRAINT MODELING

Constraint programming, and in particular Boolean constraint programming, has been used so far to support analysis of variability models such as Feature-Oriented Domain Analysis (FODA) and the like [18].

In our approach we propose to transform the goal model into an executable Constraint Program (CP) that can be executed by a tool. The transformation follows a set of mapping rules where goals, softgoals, operationalizations and contexts are modeled as variables, with claims and soft constraints modeled as constraints within the constraint program.

Our approach consists in first place on constructing a goal model of the DSPL and verifying the model to avoid defects, at design time. We then seek a variant for each context that best satisfies the soft dependencies, even if none exists that can satisfy them all. To achieve this, we can transform the goal model into a constraint program (CP). Following a set of mapping rules, this transformation models goals, soft goals, operationalizations, and context variables as variables, and claims and soft constraints as constraints, within the CP. The CP can also represent SPL properties such as operationalization incompatibilities (excludes relationships) that goal models cannot easily represent [7].

Once deployed, the DSPL operates an execute-monitore valuate-adapt control loop. Our focus here is on the decision-making element that takes the result of monitoring as input and triggers adaptations as output.

6. CONCLUSION AND FUTURE WORK

We have argued that as mobility and pervasiveness invades computing, the typical execution environment for software systems is characterized by dynamic change, and adaptivity is rapidly becoming a necessary property of most software systems. To alleviate the development of adaptive systems we have proposed an approach leveraging ideas inherited from product line engineering. Furthermore, we have argued that a DAS can be regarded as a product family line in which variabilities are bound at runtime instead of at pre-delivery time, and a self-adaptive system that uses architectural adaptation may be conceptualized as a *dynamic* SPL in which each configuration is one of the possible variants of the SPL.

Our contribution to respond to the research questions and to the evolving understanding of the problems posed by DSPLs, which is essentially that the conventional SPL modeling notations cannot easily model two properties essential for the class of DSPLs that interest us; environmental variability and QoS, our contribution is to combine goal modeling techniques with constraint programming. Goal modeling supports the modeling of environmental variability and QoS. A goal model can be mapped to a constraint program, which has the added capability to represent properties such as excludes relationships that are hard to represent in goal models. Moreover, a constraint program, when solved using a tool is able to solve the constraints.
In our future work, we aim to apply our approach to different classes of DSPL not only to those based on dynamic architectural adaptation. We will propose a methodological approach that allows to derive an implementation of a self-adaptive product line produced from user requirements and initial models. We will make things more concrete by an empirical evaluation of the solution through a case study of controlled experiments, reviews by experts. Indeed, we should define a product line for evaluation systems, including modeling and explicit expressions for reuse and variability.
We will show what has been accomplished, where more work is needed, and where additional evaluation is required. We will end by discussing the usefulness of the new approach for self-adaptative system.s

REFERENCES

[1] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. 5525:1–26, 2009.

[2] S.Hallsteinsen, M.Hinchey, S.Park and K.Schmid. Dynamic Software Product Line. In:Computer, 41(4): pp. 93-95 (2008).

[3] Mckinley, P., Sadjadi, S., Kasten, E., Cheng, B. (2004) "Composing adaptive software" IEEE Computer 37(7)

[4] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H.C.Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS), 2008.*

[5] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *CASCON'06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research,* page 7, New York, NY, USA, 2006. ACM.

[6] B. Nelly, S. Pete, B. Gordon, G. Paul. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. *Computing department, InfoLab21, Lancaster University, LA1 4WA, United Kingdom, 2008.*

[7] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, D. Hughes. Constraint Programming as a Means to Manage Configurations in Self-Adaptative Systems. In: Computer. IEEE Computer Society 2012, pages 56-63.

[8] BATORY, D. 2004. Feature-oriented Programming and the AHEAD tool suite. In Proceedings of the 26th International Conference on Software Engineering. ICSE '04. IEEE Computer Society, Washington, DC, USA, 702–703.

[9] CETINA, C., GINER, P., FONS, J., AND PELECHANO, V. 2010. Designing and prototyping dynamic software product lines: techniques and guidelines. In Proceedings of the 14th international conference on Software product lines: going beyond. SPLC'10. Springer-Verlag, Berlin, Heidelberg, 331–345.

[10] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: A middleware approach to network heterogeneity. In *Proc. 3rd ACM International EuroSys Conference*, Glasgow, Scotland, 2008.

[11] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. An intelligent and adaptable flood monitoring and warning system. In *5th UK E-Science All Hands Meeting (AHM06) (Best Paper Award)*, 2006.

[12] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.

[13] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 141–150,Washington, DC, USA, 2006. IEEE Computer Society.

[14] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective component-based technologies to support dynamic variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08)*, 2008.

[15] Nelly Bencomo. On the Use of Software Models during Software Execution. In MISE'09: Proceedings of the Workshop on Modeling in Software Engineering, at ICSE'09, 2009.

[16] Semmak, F., Gnaho, C., Laleau, R. (2010) "Extended KAOS Method to Model Variability in Requirements" in (Maciaszek, L., González-Pérez, C., Jablonski, S. Eds) Evaluation of Novel Approaches to Software Engineering, Springer Berlin Heidelberg.

[17] Van Lamsweerde, A. (2009) Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley & Sons, Chichester, UK.

[18] C. Salinesi, R. Mazo, O. Djebbi, D. Diaz , A. Lora-Michiels, Constraints: the Core of Product Line Engineering,RCIS 2011.

[19] J. Amilhastre and H. Fargier, "Handling interactivity in a constraint-based approach of configuration", in *Proc. of the 14th European Conference on Artificial Intelligence (ECAI 2000), Berlin, Germany*, 2000.

[20] F. Bergenti, "Product and Service Configuration for the Masses," presented at Proc. of the 14th European Conference on Artificial Intelligence (ECAI 2000), Berlin, Germany, 2000.

[21] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H.C.Cheng. Goal-based modeling of dynamically adaptive system requirements. In 15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS), 2008.

[22] G. SALVANESCHI, C. GHEZZI, M. PRADELLA, Politecnico di Milano. Context-Oriented Programming: A Programming Paradigm for Autonomic Systems. the European Community's IDEAS-ERC Programme, Project 227977 (SMSCom)

[23] I. Bouassida, J. Lacouture, and K. Drira. Semantic driven self-adaptation of communications applied to ERCMS. In The 24th IEEE International Conference on Advanced Information Networking and Applications (AINA 2010), Perth (Australia), 201

[24] K. Pohl, G. Böckle, and F. J. v. d. Linden, Software Product Line Engineering Foundations, Principles and Techniques Springer, 2005.

[25] N. Bencomo. Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability, PhD Thesis. PhD thesis, 2008.

[26] K. C. Kang, S. G. Cohen, J. A. Hess,W. E. Novak, and A. S. Peterson, "Featureoriented domain analysis (foda) feasibility study," Tech. Rep., Carnegie-Mellon University Software Engineering Institute, 1990.

[27] I. Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman, Redwood City, Calif, USA, 2004.

[28] K.R.Dittrich, S.Gatziu and A.Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In: LNCS 985, Springer, pp. 3-20, (1995).

[29] Liwei Shen, Xin Peng, Jindu Liu and Wenyun Zhao. Towards Feature-oriented Variability Reconfiguration in Dynamic Software Product Lines. In ICSR'11 Proceedings of the 12th international conference on Top productivity through software reuse pages 52-68

[30] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "Architecture-based approach to selfadaptive software", IEEE Intelligent Systems and Their Applications, vol. 14, pp. 54-62, 1999.

[31] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure", IEEE Computer, vol. 37, pp.46-54, 2004.

ISSN 2278-5612

[32] P. Grace, G. S. Blair, and S. Samuel, "Middleware awareness in mobile computing", in Proc. of the 23rd International Conference on Distributed Computing Systems Workshops, Providence, RI, USA. IEEE, 2003, pp. xxxi+971.
[33] M. Roman, F. Kon, and R. H. Campbell, "Reflective middleware: from your desk to your hand", IEEE Distributed Systems Online, vol. 2, 2001.