Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 321 (2016) 67–88

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Model to Guide Dynamic Adaptation Planning in Self-Adaptive Systems

Andrés Paz [1,2,3]

Université du Québec, École de Technologie Supérieure, Montréal, Canada

Hugo Arboleda [1,2,4]

Universidad Icesi, I2T Research Group, Cali, Colombia

Abstract

Self-adaptive enterprise applications have the ability to continuously reconfigure themselves according to changes in their execution contexts or user requirements. The infrastructure managing such systems is based on IBM's MAPE-K reference model: a *Monitor* and an *Analyzer* to sense and interpret context data, a *Planner* and an *Executor* to create and apply structural adaptation plans, and a *Knowledge manager* to share relevant information. In this paper we present a formal model, built on the principles of constraint satisfaction, to address dynamic adaptation planning for self-adaptive enterprise applications. We formalize, modify and extend the approach presented in [1] for working with self-adaptation infrastructures in order to provide automated reasoning on the dynamic creation of structural adaptation plans. We use a running example to demonstrate the applicability of such model, even in situations where complex interactions arise between context elements and the target self-adaptive enterprise application.

Keywords: Self-Adaptive Enterprise Applications, Dynamic Adaptation Planning, Automated Reasoning.

1 Introduction

Currently many Enterprise Applications (EAs) live in dynamic execution contexts, interacting with other systems, and under the influence of stimuli from sources inside or outside the system scope. This may affect their behavior or the levels at which they satisfy agreed quality; however, regardless of these impacts, they still have to fulfill their service quality agreements. On the one hand, the fulfillment of quality agreements is completely and utterly dependent on system architectures,

[1] This work has been partially supported by grant 0369-2013 from the Colombian Administrative Department of Science, Technology and Innovation (Colciencias) under project SHIFT 2117-569-33721.

[2] We thank Miguel Jiménez and Gabriel Tamura for their contributions on the project and the architecture of the SHIFT framework.

[3] Email: afpaz@icesi.edu.co

[4] Email: hfarboleda@icesi.edu.co

http://dx.doi.org/10.1016/j.entcs.2016.02.005

1571-0661/© 2016 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

which comprises software architecture, hardware and network infrastructure. On the other hand, in response to ever increasing needs for strengthened responsiveness and resiliency, quality agreements may evolve to reflect this business reality.

Autonomic computing deals with the management of independent components capable of handling both external resources and their internal behavior, which are constantly interacting in accordance with high-level policies. Its required infrastructure usually integrates an autonomic manager, an implementation of the generic control feedback loop from control theory, and managed components. Most autonomic managers are based on the MAPE-K reference model [2], allowing software systems to be adapted to context changes in order to ensure the satisfaction of agreed Service Level Agreements (SLAs). Five elements make up the reference model: *Monitor*, *Analyzer*, *Planner*, *Executor* and *Knowledge Manager*. The *Monitor* continuously senses context conditions and the *Analyzer* interprets and compares the sensed data with SLAs, the *Planner* synthesizes and creates adaptation plans when required, and the *Executor* alters the system's behavior by modifying its structure in accordance with a given adaptation plan. All of them share information through the *Knowledge Manager* element.

In this paper we present a formal model, built on the principles of constraint satisfaction, to address the task of the *Planner* element, *i.e.* dynamic adaptation planning for self-adaptive enterprise applications. Our work in this paper is focused around changing quality agreements while EAs are already operational. This task, however, has a direct impact on system architecture. We consider in this work only the relationships of such quality agreements with software architecture in order to plan the necessary structural adaptations to meet the new quality specifications. We use a running example to demonstrate the applicability of such model, even in situations where complex interactions arise between context elements and the target self-adaptive enterprise application. In the context of product line engineering, decision and resolution models have been used for planning the composition of core assets according to variable configurations that include user requirements, *e.g.*, [3,4]. All of such approaches, however, deal with problems related to product configuration without taking into account the problem of planning dynamic adaptation of systems.

Some authors have explored different trends for generating reconfiguration plans. For instance [5,6] use artificial intelligence based on hierarchical task networks and situation calculus, respectively, to plan new web service compositions in an attempt to overcome faults. [7] calculates fuzzy values of quality of service (QoS) levels for available service variants and selects the variants with the nearest QoS levels that fit the context and user requeriements. There are other approaches that implement dynamic adaptation of service compositions, *e.g.*, [8,9,10]; however, they neither provide implementation details nor formal specifications of any formal model for planning activities.

In previous work [1], we presented an approach based on constraint satisfaction for product derivation planning in model-driven software product lines. There, we modeled the problem of planning the transformation workflow to derive products as a constraint satisfaction problem. In this paper, we base on such model and we fur-

ther formalize, modify and extend it for working with self-adaptation infrastructures in order to provide automated reasoning on the creation of structural adaptation plans.

The remainder of this paper is organized as follows. Section 2 introduces the background of this work. Section 3 presents our motivating case along with an illustrative example which we use as a running example throughout the following sections. Section 4 details our formal model, including the necessary definitions and specifications. Section 5 describes the automated reasoning that we currently provide. Section 6 discusses related work. Finally, Section 7 sets out conclusions and outlines future work.

2 Background

2.1 Autonomic Computing

In [11], IBM researchers Kephart and Chess introduced an architectural approach to realize autonomic computing based on independent elements capable of managing both external resources and their internal behavior. In light of this, autonomic systems are compositions of these autonomic *elements*, constantly interacting in accordance with high-level policies. Each autonomic element is composed of an autonomic manager, an implementation of the generic control feedback loop from control theory, and a managed element, a hardware or software resource, such as a server, a service or a set of interconnected software components.

The autonomic manager, based on the MAPE-K reference model [2], is the infrastructure that allows the software systems to be adapted to unforeseen context changes in order to ensure the satisfaction of agreed Service Level Agreements (SLAs). Comprising this infrastructure is (i) a *Monitor* element that continuously senses relevant context and system control data; (ii) an *Analyzer* element that interprets monitoring events reported by the *Monitor* to determine whether the SLAs are being fulfilled; (iii) a ***Planner*** element that creates a configuration from the variability model according to the context conditions delivered by the *Analyzer* to generate an adaptation plan, which defines the modification required by the deployed system structure and the required parameters to reach a desired system state; (iv) an *Executor* element that realizes adaptation plans, which alters the system's behavior; and (v) a *Knowledge Manager* element sharing relevant information among the other elements.

2.2 Dynamic Software Product Line Engineering

Software Product Line Engineering (SPLE) is an expanding approach that aims at developing a set of software systems that share common features and satisfy the requirements of a specific domain [12]. While having much in common, product line members still differ in functional and quality requirements. Variability management is the key process in SPLE that is in charge of dealing with the analysis, modeling, design and realization of variants while considering adequate decision making

support for building products by using reusable assets.

Variability Models. Variability in SPLE is captured in variability models, such as the Orthogonal Variability Model (OVM) [13,12]. An OVM is a variability model designed to only document variability; we use OVMs in this paper to document variability in our running example described in Section 3.2. In OVMs like the one presented in Figure 1, a variation point (p) represents a variable item in a system and is depicted as a triangle. A variant (v) represents a particular option to instance the variation point and is depicted as a rectangle linked to the variation point by one of three types of relationships. Relationships between variants and variation points may be mandatory, optional or set. A mandatory relationship, depicted in Figure 1 as a solid line, states that if a variation point p is present its child variant v must be present too. An optional relationship, depicted as a dotted line, states that if a variation point p is present its child variant v^p may or may not be present. A set of children variants $\{v_i \mid i = 1, \ldots, z\}$ has a set relationship with their parent variation point p when an interval $[x, y]$ of its children v_i can be included $\{v_i \mid x \leq i \leq y\}$ if their parent is present. This type of relationship is illustrated as variants grouped by an angular solid line with a label describing the interval. Relationships can also exist between variants of different variation points. Such relationships are, namely, requires and excludes; they are drawn as single arrow line and double arrow line respectively. A requires relationship is a cross variant constraint that states that if variant requires variant v_b then if v_a is present, v_b must be present too. An excludes relationship is a cross variant constraint that states that if variant v_a excludes variant v_b then the variants cannot be present at the same time. We give a formal definition of each relationship when we present our proposed model in Section 4.

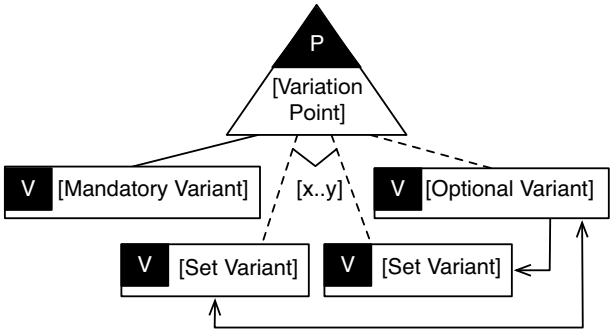

Fig. 1. Orthogonal Variability Model

Dynamic SPLE. Dynamic SPLE [14] extends current product line engineering approaches by moving their capabilities to runtime, helping to ensure that system adaptations lead to desirable properties. It is concerned about the management of reusable and dynamically reconfigurable core assets, facing the challenge of binding variants to such assets, at runtime, when software is required to be adapted according to context changes.

Decision and Resolution Models. When variants are selected by architects at design time (in the context of SPLE), or defined by context conditions at runtime

(in the context of dynamic SPLE), concrete core assets must be selected as part of the (re)composition plan. In practice, there is a significant gap between variability at a conceptual level (variation points and variants) and variability at the implementation level (concrete core assets to be deployed). With the objective of closing that gap, *decision* and *resolution models* are used [4,15]. A decision model relates open decisions and possible resolutions to define the necessary actions to derive product line members in accordance with configurations, which are sets of selected variants. A resolution model is the instance of a decision model, and it is used to create a product line member. In a resolution model all the decisions captured in a decision model are resolved, thus, it defines a product line member including a subset of chosen variants, the core assets required to derive the desired product, and the adaptation that must be performed on the core assets to obtain such product line member.

Variant Interactions. Decision models rapidly become very complex artifacts in the face of many variants and, specially, when *variants interactions* appear. When several variants are combined interactions between them may occur; this means, the presence of one variant affects the behaviour of another. Let suppose a variant v_i is related to a software component c_i, and a variant v_j is related to a software component c_j, an interaction exists when the presence of v_i and v_j in one configuration raises a problem when composing c_i and c_j. Some variant interactions may be benign, planned or desirable, but others, in turn, may have unwanted effects that may disrupt the user from obtaining the expected behavior. Since the variant interactions problem can be arbitrarily complex and computationally difficult to treat, a formal approach is an appropriate and flexible option.

2.3 Constraint Satisfaction

A great variety of combinatorial problems can be expressed as searching for one or several elements in a vast space of possibilities. In general, the search space is defined as all combinations of possible values for a predefined set of variables. Elements to be searched for are particular values of these variables. In most cases the desired values of the elements are implicitly specified by properties they should satisfy. These properties are known as constraints, which are usually expressed as predicates over some set of variables. Roughly speaking, a problem formulated in this frame is known as a Constraint Satisfaction Problem (CSP) [16].

Solving a CSP consists of two steps: modeling the problem (logical specification) and finding its solutions through a form of search (in this paper we perform a basic backtracking). Modeling involves basically the specification of the variables, their domains and the constraints among them. Solving the CSP through backtracking is an attempt at trying to incrementally build resolution candidates by assigning possible values to the variables. Partial candidates that cannot become a valid solution are discarded. If all variables are bound, a resolution candidate has been found. If, after exploring all possibilities no resolution candidate has been found, then the problem does not have a solution.

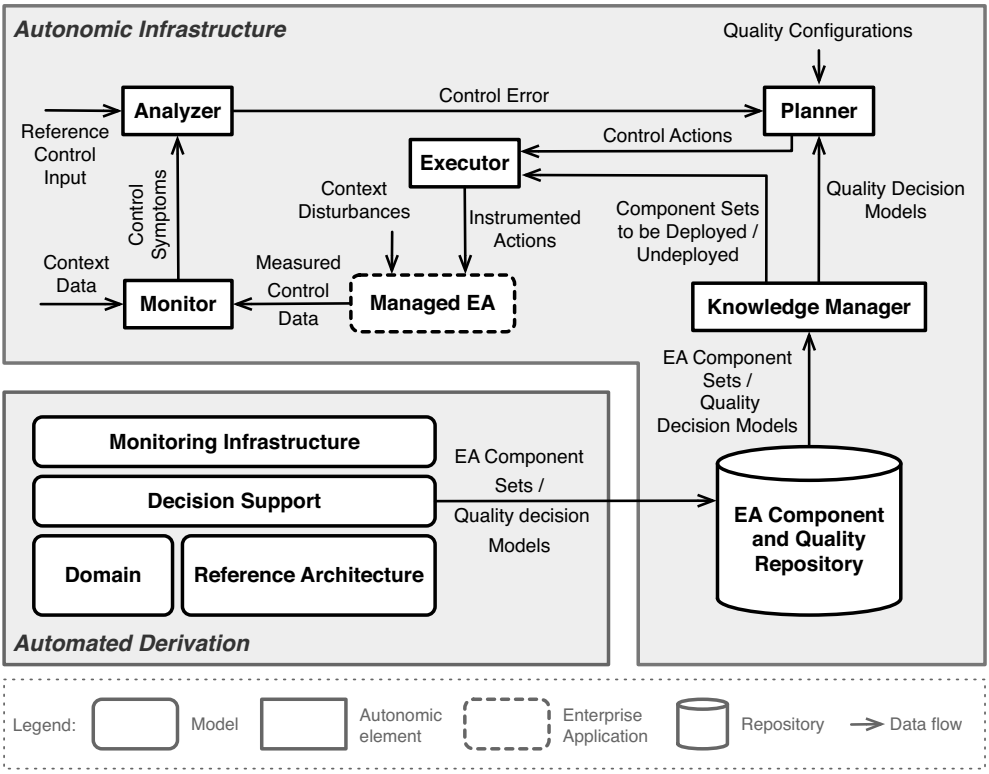

Fig. 2. High-level architectural view of the SHIFT elements.

3 Motivating Case

3.1 The SHIFT Framework

Our research group has proposed independent approaches and implementations in the contexts of autonomic computing with the DYNAMICO reference model [17], quality of service (QoS) contract preservation under changing execution conditions with QoS-CARE [18], model-based product line engineering with the FieSta approach [4,15], automated reasoning for derivation of product lines [1], and the recent (unpublished) contributions regarding quality variations in the automated derivation process of product lines [19]. The required integration of all these efforts in a move to approach automation and quality awareness along the life cycle of enterprise applications has motivated the creation of what we call the *SHIFT Framework*. Figure 2 presents a high-level architectural view of SHIFT's constituting elements.

The `Automated Derivation` region is concerned with providing support for functional and quality configuration and derivation of deployable enterprise applications components and monitoring infrastructure. Generated components are stored in the `Component Repository`, which is managed by a `Knowledge Manager` element; they are an input for the adaptation planning process. The monitoring infrastructure is deployed as part of the `Autonomic Infrastructure` region, which implements the adaptation feedback loop of the DYNAMICO reference model [17].

As part of the `Planner` element, our focus in this paper, SHIFT considers the need for dynamically planning adaptations to application structure based upon quality configurations. Realizing the adaptation plans in the deployed and operating managed Enterprise Application (EA) considers transporting components from their source repository to the corresponding computational resource, undeploying previous versions of them, deploying them into the middleware or application server, binding their dependencies and services, and executing them. In addition, if necessary, to recompile system source code to make measurement interfaces available to the monitoring infrastructure.

In order to obtain the best possible selection of composable components, or *optimum resolution*, when planning an adaptation, we propose in this paper addressing dynamic adaptation planning through a model built on the principles of constraint satisfaction, which will help reasoning upon the set of constraints defined by reachable quality configurations and their relationships with the components in the component repository. Following Section 4 will refer to the relationships between components in the component repository and the reachable quality configurations as *decision models*, and all the possible adaptation plans that can be derived from a *decision model* given a specific quality configuration as *resolution models*.

3.2 Running Example

To illustrate the problem of adapting an EA, while at runtime, when the set of quality agreements (captured as quality scenarios as explained by Bass *et al.* in [20]) changes, we use the case of a large-scale e-commerce application. We use this case as a running example throughout the following sections. The following sections give the details regarding how the `Planner` element of the SHIFT Framework captures adaptation constraints and reasons upon them to determine possible adaptation plans to satisfy changing context conditions.

With our example e-commerce application there is the need to handle component compositions and adaptations driven by different system quality levels in accordance with varying shopping activities (*e.g.*, special offers on certain products, shopping frenzies). This implies working with varying quality scenarios. Thus, we use the OVM in Figure 3 to capture the different quality scenarios that can be configured for the e-commerce EA. The quality attribute, environment and stimuli fields of a quality scenario represent a variation point. The response field represents a variant. Figure 3 illustrates 3 variation points with all of their variants linked with optional relationships.

Suppose the e-commerce application has been initially deployed fulfilling the requirement of purchase by credit card and the quality configuration corresponds to the selection of quality scenarios `V2` and `V4` detailed in Table 1. The *time-behavior* scenario determines an average latency of 6 seconds for purchases with credit card under a load of 1,000 purchases per minute, stochastically. The *confidentiality* scenario specifies all available sensitive information is encrypted to prevent unauthorized access.

A component diagram for the implementation of the purchase with credit card

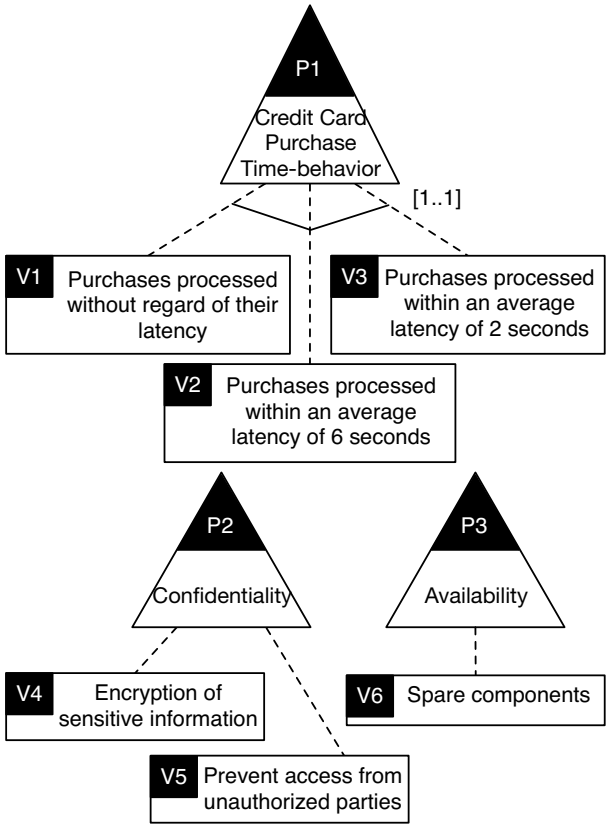

Fig. 3. Variability Model

requirement is illustrated in Figure 4. This implementation comprises (i) a `Purchase` component that manages the workflow performed for any purchase, (ii) a `Credit Card Authorization` component in charge of performing the workflow to get approval for the transaction with the issuing bank (or credit card association), (iii) a `Risk Tool` component responsible for validating credit card information provided by the customer and the responses sent from the issuing bank, (iv) a `Credit Card Settlement` component that requests the transfer of funds from the issuing bank into the merchant's account, (v) a `Cryptography Manager` component that processes the encryption and decryption of information to and from the issuing bank, and (vi) a `Payment Processor` component managing all communications to the multiple issuing banks. The payment processing behavior exhibited by the previous implementation is specified step by step in Figure 5.

For a first adaptation setting suppose now that, while in operation, the application's initial quality configuration has been changed due to an expected peak in system load caused by an upcoming *Cyber Monday* shopping season. Particularly quality scenario `V2` has been replaced by quality scenario `V3`, presented in detail in Table 2. Quality scenario `V4` remains selected. In turn, the application's constituent components must be changed to new ones developed with the modified quality configuration in mind.

Table 1
Quality scenarios for the e-commerce application

Quality Attribute	*Performance – Time behavior*
Environment	The application provides a set of services available to concurrent users over the Internet under normal operating conditions.
Stimuli	Users initiate 1,000 purchases with credit card as payment method per minute, stochastically.
Response	Every purchase is processed with an average latency of 6 seconds.

Quality Attribute	*Security – Confidentiality*
Environment	The application provides a set of services that makes sensitive information available to other applications over the Internet.
Stimuli	Another application intercepts data by attacking the network infrastructure in order to obtain sensitive information.
Response	The architecture does not control the other application's access, but information is encrypted in order to prevent access to sensitive information.

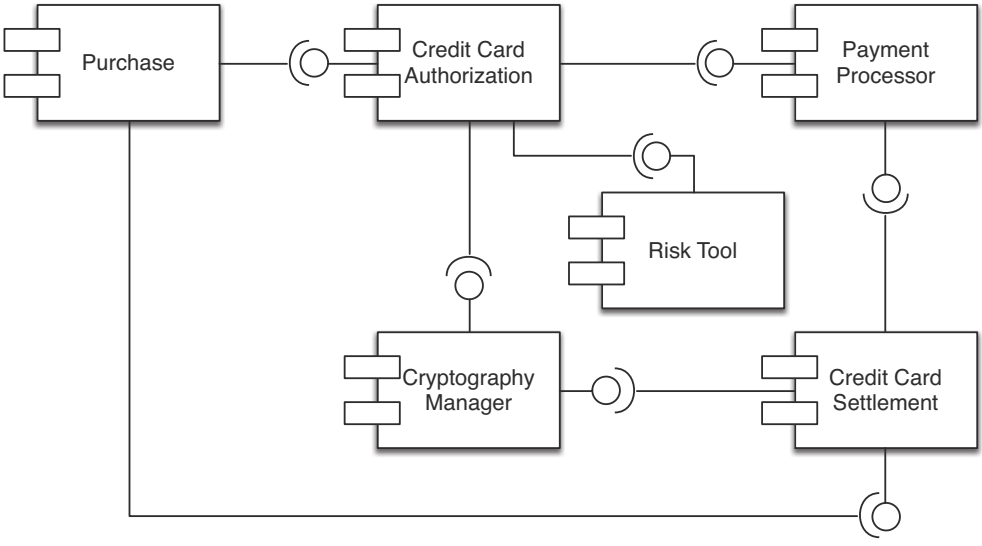

Fig. 4. Partial set of components for initial e-commerce application

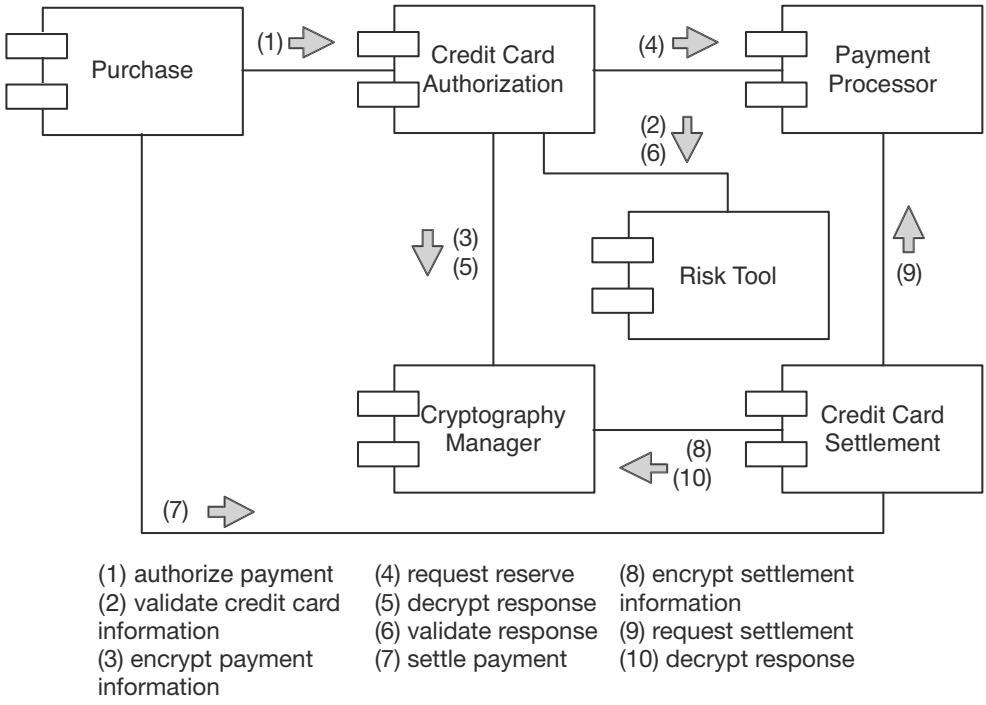

Fig. 5. Component collaboration for initial e-commerce application

Table 2
Modified time-behavior scenario for the e-commerce application

Quality Attribute	*Performance – Time behavior*
Environment	The application provides a set of services available to concurrent users over the Internet under normal operating conditions.
Stimuli	Users initiate 20,000 purchases with credit card as payment method per minute, stochastically.
Response	Every purchase with credit card as payment method is processed with an average latency of 2 seconds.

The adapted implementation for the purchase with credit card requirement is illustrated in Figure 6. This implementation comprises modified versions of the `Purchase`, `Credit Card Authorization` and `Credit Card Settlement` components. These modified components are marked with an asterisk symbol (`*`). A new component appears, the `Order Manager` component, which provides a consolidated and automated processing of orders. The `Payment Processor` component remains unchanged. The behavior of this implementation is changed due to the structural adaptation performed that streamlined the workflow in comparison with

the initial deployment. Figure 7 shows the collaboration steps between the new set of components.

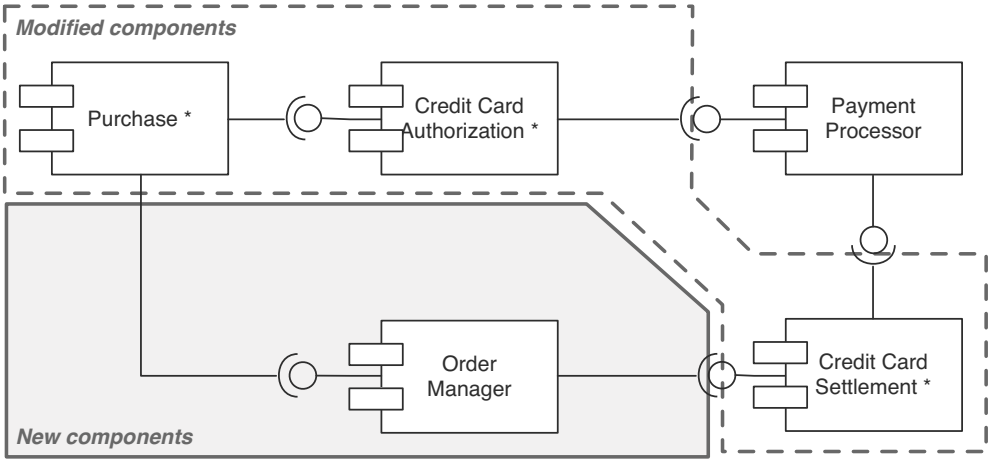

Fig. 6. Partial set of components for adapted e-commerce application

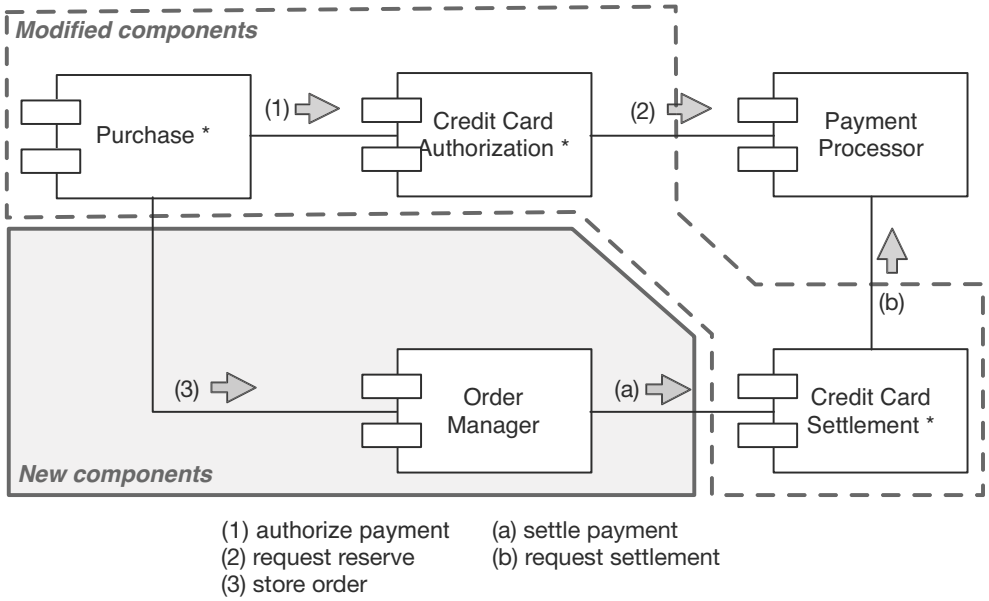

Fig. 7. Component collaboration for adapted e-commerce application

Suppose now, for a second adaptation setting, that to further strengthen the application to cope with the coming sales burst a new quality configuration has been specified selecting quality scenarios `V3`, `V4` and `V6`. The new *availability* scenario `V6` in Table 3 states that the system initializes and puts into operation spare components when part of the application becomes unavailable. The spare components are initialized from a persistent state before entering into operation. Thus, this

response requires the use of persistent storage (*e.g.*, database) to maintain application state and be able to replace failed components. Let's assume that the `Payment Processor` helps meet the *time-behavior* scenario in Table 2 due to its use of a cache to avoid requests to the database. A variant interaction arises when trying to fulfill both quality scenarios, as the *availability* scenario makes accessing the database mandatory and no caches are permitted. Hence, the *availability* scenario cannot be promoted with the planned adaptation shown in Figure 6. A new solution needs to be designed or the quality scenario needs to be either redefined or dropped.

Table 3
New quality scenario for the e-commerce application

Quality Attribute	*Reliability – Availability*
Environment	A subsystem of the application becomes unavailable.
Stimuli	Users initiate transactions to the affected subsystem.
Response	Spare components are initialized and placed into operation.

4 Self-Adaptation Planning

The previous e-commerce application provides an interesting example of the decisions that need to be taken when planning an adaptation to satisfy changing quality scenarios. Manually evaluating all component compositions, their relationships to quality scenarios and quality scenario interactions are costly, time consuming and error-prone; even more when the software system is already operational. In this section we propose an approach addressing dynamic adaptation planning built on the principles of constraint satisfaction.

Benavides *et al.* in [21] propose mapping variability models, particularly feature models, to an equivalent CSP representation in order to deal with the automated analysis of such models. To be able to analyze varying quality scenarios for the creation of adaptation plans we translate input OVMs holding the quality scenarios into a specific CSP representation. Definition 4.1 formally describes this CSP as a quality model. It is modified from the one presented for the translation of feature models into CSP in [21].

Definition 4.1 A quality model μ is a three-tuple of the form (Q, W, R); where Q is a finite set of l variables made up of h variation points p and i variants v; W is a finite set of domains made of the variants' configuration states, with a state of 1, if the quality scenario is unselected, or 2, if the quality scenario is selected; and R is a finite set of constraints defined on Q.

$$Q = \{\{\langle p_k \rangle \mid k = 1, \ldots, h\}, \{\langle v_i \rangle \mid i = 1, \ldots, n\}\}$$

$$W = \left\{ \left\{ W_{p_k} = [1..2] \mid \begin{pmatrix} 1 \text{ if } p_k \text{ is unselected} \\ 2 \text{ if } p_k \text{ is selected} \end{pmatrix} \right\}, \left\{ W_{v_i} = [1..2] \mid \begin{pmatrix} 1 \text{ if } v_i \text{ is unselected} \\ 2 \text{ if } v_i \text{ is selected} \end{pmatrix} \right\} \right\} \tag{1}$$

$$R = \{r_{mandatory}, r_{optional}, r_{set}, r_{requires}, r_{excludes}\} \tag{2}$$

The set R of Equation 2 contains the following relationship constraints:

Mandatory. A mandatory relationship states that if a variation point p is present its child variant v must be present too.

$$r_{mandatory} = \langle v \geq 2 \Leftrightarrow p \geq 2 \rangle$$

Optional. An optional relationship states that if a variation point p is present its child variant v may or may not be present.

$$r_{optional} = \langle p < 2 \Rightarrow v < 2 \rangle$$

Set. A set of children variants $\{v_i \mid i = 1, \ldots, z\}$ has a set relationship with their parent variation point p when a number of them can be included if their parent is present.

$$r_{set} = \langle x \in [0..f], y \in [1..g]((g \leq z \wedge p \geq 2) \Rightarrow ((x \times 2) \leq (\sum_{i=1}^{z} v_i) \leq (y \times 2))) \rangle$$

Requires. A requires relationship is a cross variant constraint that states that if variant v_a requires variant v_b then if v_a is present, v_b must be present too.

$$r_{requires} = \langle a, b \in [1..n]((a \neq b) \wedge (v_a \Rightarrow v_b)) \rangle$$

Excludes. An excludes relationship is a cross variant constraint that states that if variant v_a excludes variant v_b then the variants cannot be present at the same time.

$$r_{excludes} = \langle a, b \in [1..n]((a \neq b) \wedge \neg(v_a \wedge v_b)) \rangle$$

In accordance to Definition 4.1, the OVM in Figure 3 can be translated to the quality model described in Equation 3.

$$\mu_{e-commerce} = (Q_{e-commerce}, W, R) \tag{3}$$

Where

$$Q_{e-commerce} = \{p_1, p_2, p_3, v_1, v_2, v_3, v_4, v_5, v_6\}$$

W and R are as specified in Equations 1 and 2, respectively.

In order to plan an adaptation, values must be assigned to the variables in the Q set conforming to the selection and unselection of quality scenarios. We call this a *quality configuration*. The quality configurations matching the initial scenario and the two adaptation settings for Cyber Monday in Section 3 are as specified in Equations 4, 5 and 6, respectively.

$$\begin{aligned} Q^{initial}_{e-commerce} = \{&p_1 = 2, p_2 = 2, p_3 = 1, \\ &v_1 = 1, v_2 = 2, v_3 = 1, v_4 = 2, v_5 = 1, v_6 = 1\} \end{aligned} \tag{4}$$

$$\begin{aligned} Q^{timebehavior}_{e-commerce} = \{&p_1 = 2, p_2 = 2, p_3 = 1, \\ &v_1 = 1, v_2 = 1, v_3 = 2, v_4 = 2, v_5 = 1, v_6 = 1\} \end{aligned} \tag{5}$$

$$\begin{aligned} Q^{timebehavior+availability}_{e-commerce} = \{&p_1 = 2, p_2 = 2, p_3 = 2, \\ &v_1 = 1, v_2 = 1, v_3 = 2, v_4 = 2, v_5 = 1, v_6 = 2\} \end{aligned} \tag{6}$$

Promoting a quality scenario may often require several composed components, thus, in this paper we refer as a *componentset* (see Definition 4.2) to the composition of components promoting a quality scenario. We denote the composition operator as $\oplus$.

Definition 4.2 A *componentset* c is a composition of g components e.

$$c = \bigoplus_{u=1}^{g} e_u$$

For the example e-commerce application we have identified five *componentset*s: c_1 (see Equation 7), c_2 (see Equation 8), c_3 (see Equation 9), c_4 (see Equation 10) and c_5 (see Equation 11).

$$\begin{aligned} c_1 = \text{Purchase} \oplus \text{Credit Card Authorization} \oplus \\ \text{Credit Card Settlement} \oplus \text{Risk Tool} \end{aligned} \tag{7}$$

$$c_2 = \text{Cryptography Manager} \tag{8}$$

$$c_3 = \text{Payment Processor} \tag{9}$$

$$c_4 = \text{Purchase}^* \oplus \text{Credit Card Authorization}^* \oplus \text{Credit Card Settlement}^* \tag{10}$$

$$c_5 = \text{Order Manager} \tag{11}$$

Table 4 shows the relationships established between the quality scenarios (in the remainder of this paper we refer to every variant, *i.e.* response alternative, as one quality scenario) and the identified *componentset*s as presented in Section 3. A ✓ indicates the *componentset* requires the quality scenario to be selected in the configuration; on the contrary, an ✗ indicates that the *componentset* requires the quality scenario to be unselected. A "-" indicates the *componentset* is not constrained by the presence of the quality scenario.

Table 4
Relationships between quality scenarios and components

Quality Scenarios	*componentset*s				
	c_1	c_2	c_3	c_4	c_5
v_1	✓	-	✓	✗	✗
v_2	✓	-	✓	✗	✗
v_3	✗	✗	✓	✓	✓
v_4	-	✓	-	✓	-
v_5	✗	✗	✗	✗	✗
v_6	✗	✗	✗	✗	✗

One of the main elements of the proposed approach is the decision model. Decision models in our approach relate *componentset*s stored in a component repository (see Figure 2) and quality scenarios to define the necessary actions to adapt an enterprise application in accordance to a configuration of such quality scenarios.

Definition 4.3 A decision model D is a finite set of $m \times n$ decisions. Each decision d relates one *componentset* c_j with one quality scenario v_i.

$$D = \{\langle d_j^i \rangle \mid j = 1, \ldots, m \wedge i = 1, \ldots, n\}$$

Where

$$d_j^i = \begin{cases} 0 \text{ if } v_i \text{ does not constrain the deployment of } c_j \\ 1 \text{ if } c_j \text{ requires } v_i = 1 \\ 2 \text{ if } c_j \text{ requires } v_i = 2 \end{cases}$$

Table 4 maps to the decision model in Equation 12.

$$\begin{aligned} D_{e-commerce} = \{&d_{c_1}^{v_1} = 2, d_{c_1}^{v_2} = 2, d_{c_1}^{v_3} = 1, d_{c_1}^{v_4} = 0, d_{c_1}^{v_5} = 1, d_{c_1}^{v_6} = 1, \\ &d_{c_2}^{v_1} = 0, d_{c_2}^{v_2} = 0, d_{c_2}^{v_3} = 1, d_{c_2}^{v_4} = 2, d_{c_2}^{v_5} = 1, d_{c_2}^{v_6} = 1, \\ &d_{c_3}^{v_1} = 2, d_{c_3}^{v_2} = 2, d_{c_3}^{v_3} = 2, d_{c_3}^{v_4} = 0, d_{c_3}^{v_5} = 1, d_{c_3}^{v_6} = 1, \\ &d_{c_4}^{v_1} = 1, d_{c_4}^{v_2} = 1, d_{c_4}^{v_3} = 2, d_{c_4}^{v_4} = 2, d_{c_4}^{v_5} = 1, d_{c_4}^{v_6} = 1, \\ &d_{c_5}^{v_1} = 1, d_{c_5}^{v_2} = 1, d_{c_5}^{v_3} = 2, d_{c_5}^{v_4} = 0, d_{c_5}^{v_5} = 1, d_{c_5}^{v_6} = 1\} \end{aligned} \tag{12}$$

A resolution model is a decision model instance, which defines an adaptation plan.

Definition 4.4 A resolution model S is a finite set of s *componentset* deployments. The deployment s_j is 0 if the *componentset* j should not be deployed, and 1 if the *componentset* j should be deployed.

$$S = \{\langle s_j \rangle \mid j = 1, \ldots, m\}$$

Where

$$s_j = \begin{cases} 0 \text{ if } c_j \text{ should not be deployed} \\ 1 \text{ if } c_j \text{ should be deployed} \end{cases}$$

The resolution for the first adaptation setting in our e-commerce example is presented in Equation 13. The adaptation plan represented in this resolution model indicates that the *componentset*s c_3, c_4 and c_5 should be the ones deployed in order to promote the configured quality scenarios (see Equation 5). According to the decision model in Equation 12, with the available *componentset*s there is no adaptation that can meet the configured quality scenarios in Equation 6 corresponding to the second adaptation setting.

$$S_{e-commerce} = \{s_1 = 0, s_2 = 0, s_3 = 1, s_4 = 1, s_5 = 1\} \tag{13}$$

However, not every possible resolution model is a valid resolution model. A valid resolution model must satisfy the following constraints:

Definition 4.5 Deployment constraint. A *componentset* must be deployed satisfying the respective deployment condition in the decision model.

$$\forall j \in [1..m] s_j = 1 \Rightarrow \forall i \in [1..n](c_j^i = 0 \vee (c_j^i \neq 0 \wedge c_j^i = v_i))$$

Definition 4.6 Non-exclusion constraint. Two deployable *componentset*s must not exclude each other.

$$\forall j_1, j_2 \in [1..m](s_{j_1} = s_{j_2} = 1 \wedge j_1 \neq j_2) \Rightarrow \forall i \in [1..n](c_{j_1}^i = 0 \vee c_{j_2}^i = 0 \vee c_{j_1}^i = c_{j_2}^i)$$

Definition 4.7 Completeness constraint. All deployable *componentsets* must take into account all the quality scenarios' states in the quality configuration.

$$\forall i \in [1..n] \exists j \in [1..m] (s_j = 1 \wedge c_j^i \neq 0))$$

Definition 4.8 A self-adaptation plan is a three-tuple of the form (L, T, P); where L is a finite set of variables made up of the quality configuration Q (see Definition 4.1), the decision model D (see Definition 4.3) and the set of possible resolution models M (see Definition 4.4); T is a finite set of domains made up of the domains for the quality configuration (see Definition 4.1), decision model (see Definition 4.3) and resolution models (see Definition 4.4); and P is a finite set of constraints defined on L (see Definitions 4.1, 4.5, 4.6 and 4.7).

$$\omega_Q^D = (L, T, P)$$

Definition 4.9 Let ω_Q^D be a self-adaptation plan of the form (L, T, P), its solution space denoted as $sol(\omega_Q^D)$ is made up of all its possible solutions (possible resolution models M). An adaptation is satisfiable if the solution space of ω_Q^D is not empty.

$$sol(\omega_Q^D) = \{\langle S \rangle \mid \forall s_j (s_j \in S \Rightarrow P(s_j) = true)\}$$

5 Automated Reasoning

This section presents how automated reasoning is provided in the *Planner* element. Due to interactions between quality scenarios, and since different component compositions may be available; conflicts between *componentsets* may arise. Automated reasoning seeks to cope with this issue by providing additional information to get the best possible selection of *componentsets* when determining an adaptation plan. The proposed approach is able to answer the following questions.

Application. Given a decision model, a quality configuration and a self-adaptation plan, there should be a way of verifying the resolution model's applicability to adapt the specified enterprise application.

Definition 5.1 Let D be a decision model and Q a quality configuration, a resolution model S is applicable if it is an element of the solutions of the equivalent CSP ω_Q^D.

$$applicable(S) \Leftrightarrow (S \in sol(\omega_Q^D)) \tag{14}$$

Possible resolutions. Once a quality configuration is defined, there should be a way to obtain the potential sets of *componentsets* that promote it.

Definition 5.2 Let D be a decision model and Q a quality configuration, the potential resolution models that promote Q from D are equal to the solutions of the equivalent CSP ω_Q^D.

$$resolutions(Q, D) = \{\langle S\rangle \mid S \in sol(\omega_Q^D)\} \tag{15}$$

Number of resolutions. A key question to be answered is how many potential resolution models a decision model contains to adapt an enterprise application. The higher the number of resolutions, the more flexible and complex becomes the decision model.

Definition 5.3 Let D be a decision model and Q a quality configuration, the number of potential resolution models that promote Q from D, or cardinal, is equal to the solution number of its equivalent CSP ω_Q^D.

$$cardinal(Q, D) =\mid sol(\omega_Q^D) \mid \tag{16}$$

Validation. A valid decision model is a model where at least one resolution model can be selected to adapt an enterprise application. That is, a model where ω_Q^D has at least one solution.

Definition 5.4 A decision model D is valid to adapt an enterprise application promoting quality configuration Q if its equivalent CSP is satisfiable.

$$valid(Q, D) \Leftrightarrow resolutions(Q, D) \neq \emptyset \tag{17}$$

Flexible *componentsets*. A flexible *componentset* is a *componentset* that can be applied in self-adaptation plans for the same quality scenario with different combinations of other *componentsets*. Given a set of possible resolution models, there should be a way to find the *componentset*s appearing more than once in such set.

Definition 5.5 Let M be the set of possible resolution models, the set of flexible *componentset*s in M is equal to the *componentset*s selected to be applicable found in the intersection of M.

$$flexible(M) = \{\langle s\rangle \mid s = 1 \wedge s \in \bigcap M\} \tag{18}$$

Inflexible *componentsets*. An inflexible *componentset* is a *componentset* that only makes part of one resolution model. Given a set of possible resolution models, there should be a way to find the inflexible *componentset*s in such set.

Definition 5.6 Let M be the set of possible resolution models, the set of inflexible *componentset*s in M is equal to the *componentset*s selected to be applicable not found in the intersection of M.

$$inflexible(M) = \{\langle s\rangle \mid s = 1 \wedge s \notin \bigcap M\} \tag{19}$$

Optimum resolution. Finding out the best resolution model according to a criterion is an essential task for self-adaptation in the proposed approach. Given a set of possible resolution models, there should be a way to find the solution that

matches the criteria of an objective function. Two objective functions were taken into account. On the one hand, the function that outputs the resolution model with the greater number of applicable *componentset*s to self-adapt an enterprise application; namely *max*. On the other hand, the function that outputs the resolution model with the least number of applicable *componentset*s to self-adapt an enterprise application; namely *min*.

Definition 5.7 Let M be the set of possible resolution models and O an objective function, the optimum solution (max or min) is equal to the optimum space of ω_Q^D.

$$\begin{aligned} max(M, O) &= max(\omega_Q^D, O) \\ min(M, O) &= min(\omega_Q^D, O) \end{aligned} \tag{20}$$

Definition 5.8 Let ω_Q^D be a CSP, its optimum space, denoted as $max/min(\omega_Q^D, O)$, is made up of all the solutions that maximize or minimize O, respectively.

$$\begin{aligned} max(\omega_Q^D, O) &= \{\langle S\rangle \mid \forall S'((S' \in sol(\omega_Q^D) \wedge S' \neq S) \Rightarrow (O(S) \geq O(S')))\} \\ min(\omega_Q^D, O) &= \{\langle S\rangle \mid \forall S'((S' \in sol(\omega_Q^D) \wedge S' \neq S) \Rightarrow (O(S) \leq O(S')))\} \end{aligned} \tag{21}$$

6 Related Work

There are some approaches that have used CSPs for the manipulation of variability models in SPL Engineering. One of the most representative works on the subject was presented in [21], where the authors presented an algorithm to transform feature models into a CSP. The authors proposed to use CSPs to reason on feature models in such a way that they can answer questions such as number of products, filters based on user selections, valid configurations, among others. Several other contributions have been made since then, presenting CSPs as a good complement to SPLs (*e.g.*, [22,23]). All of them, however, deal with problems related to product configuration without taking into account the problem of planning composition of products.

Some authors have explored different trends for generating reconfiguration plans. For instance, Moore *et al.* [5] use artificial intelligence (AI) based on hierarchical task networks; McIlraith *et al.* [6] propose an AI planner built by adapting and extending Golog [24], which is a logic programming language based on the situation calculus, built on top of Prolog. Other planners, like SHOP2 [25] are hierarchical task network planners, based on the situation calculus. When composing Web services, high level generic planning templates (subplans) and complex goals can be represented by Golog. These approaches, however, do not provide any support for self-adaptive infrastructures. On the other hand, Beggas *et al.* propose in [7] the use of fuzzy logic in adaptation planning. Adaptation controllers calculate fuzzy values for the QoS levels of available service variants, the current context state and user requirements. The variants with the nearest QoS levels that fit the current context state and user requirements will be selected for application.

There are approaches that implement dynamic adaptation of service compositions at the language level *e.g.*, [26,27]; these can be complex and time-consuming, and with low-level implementation mechanisms for every element of the adaptation infrastructure. Our work is more closely related to approaches using models at runtime, *e.g.*, [8,9,10], which implement, tacit or explicitly, the MAPE-K reference model. The recent work of Alférez *et al.* [10] summarizes good practices implementing the MAPE-K reference model. They center their attention on service re-composition at runtime using dynamic product line engineering practices for assembling and re-deploying complete applications according to context- and system-sensed data. Application changes are reflected into the service composition by adding or removing fragments of Business Process Execution Language (WS-BPEL) code, which can be deployed at runtime. In order to reach adaptations, the authors argue that they use Constraint Programming for verifying at design time the variability model and its possible configurations; however, they neither provide implementation details nor formal specifications of any CSP model for planning activities.

7 Conclusions and Future Work

In this paper we presented a formal model based on the principles of constraint satisfaction for supporting the creation of *Planner* elements in self-adaptation infrastructures. We use CSPs to reason on the set of constraints defined by reachable configurations and their relationships with the components stored in the component repository. We provided formal definitions of the concepts of *quality model*, *decision model*, *resolution model*, *deployment constraint*, *non-exclusion constraint*, *completeness constraint*, *self-adaptation plan* and *solution space*. Our formal model of the *Planner* allows us to answer the following questions: *application*, *possible resolutions*, *number of resolutions*, *validation*, *flexible componentsets*, *inflexible componentsets*, and *optimum resolution*. We used a running example, in the context of enterprise applications and a self-adaptive framework, to demonstrate the applicability of the model, even in situations where complex interactions arise between context elements and the target self-adaptive enterprise application.

As future work, we will extend the model for reasoning on the process of binding components while they are redeployed on system infrastructures. We will also implement a support tool for the model and integrate it into a self-adaptation infrastructure. Other challenges to face in the near future are to perform a validations of our implementation with a case study.

References

[1] H. Arboleda, J. F. Díaz, V. Vargas, and J.-C. Royer, "Automated reasoning for derivation of model-driven spls," in *SPLC'10 MAPLE'10*, 2010, pp. 181–188.

[2] IBM, "An architectural blueprint for autonomic computing," *IBM White Paper*, 2006.

[3] P. Tessier, S. Gérard, F. Terrier, and J. M. Geib, "Using Variation Propagation for Model-Driven Management of a System Family," ser. LNCS 3714, 2005, pp. 222–233.

[4] H. Arboleda, R. Casallas, and J.-C. Royer, "Dealing with Fine-Grained Configurations in Model-Driven SPLs," in *Proc. of the SPLC'09*. San Francisco, CA, USA: Carnegie Mellon University, Aug. 2009, pp. 1–10.

[5] C. Moore, M. Xue Wang, and C. Pahl, "An architecture for autonomic web service process planning," in *Emerging Web Services Technology Volume III*, ser. Whitestein Series in Software Agent Technologies and Autonomic Computing, W. Binder and S. Dustdar, Eds. Birkhaeuser Basel, 2010, pp. 117–130.

[6] S. A. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, Toulouse, France, 2002, pp. 482–496.

[7] M. Beggas, L. Médini, F. Laforest, and M. T. Laskri, "Towards an ideal service qos in fuzzy logic-based adaptation planning middleware," *Journal of Systems and Software*, vol. 92, pp. 71 – 81, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121213001738

[8] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," *IEEE Trans. on Software Engineering*, vol. 37, no. 3, pp. 387–409, 2011.

[9] D. Menasce, H. Gomaa, S. Malek, and J. P. Sousa, "Sassy: A framework for self-architecting service-oriented systems," *IEEE Software*, vol. 28, no. 6, pp. 78–85, 2011.

[10] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," *Systems and Software*, vol. 91, no. 1, pp. 24–47, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.06.034

[11] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[12] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[13] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig, "A Meta-model for Representing Variability in Product Family Development," in *Software Product-Family Engineering*, 2004, pp. 66–80.

[14] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, April 2008.

[15] H. Arboleda and J.-C. Royer, *Model-Driven and Software Product Line Engineering*, 1st ed. ISTE-Wiley, 2012.

[16] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[17] N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas, "DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems," *LNCS*, vol. 7475, pp. 265–293, 2013.

[18] G. Tamura, R. Casallas, A. Cleve, and L. Duchien, "QoS contract preservation through dynamic reconfiguration: A formal semantics approach," *Science of Computer Programming*, vol. 94, pp. 307–332, 2014.

[19] D. Durán and H. Arboleda, "Quality-driven software product lines," Master's thesis, Universidad Icesi, 2014. [Online]. Available: http://bibliotecadigital.icesi.edu.co/biblioteca{_}digital/handle/10906/77492

[20] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2012.

[21] D. Benavides, A. Ruiz-Cortés, and P. Trinidad, "Automated reasoning on feature models," *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, vol. 3520, pp. 491–503, 2005.

[22] J. White, D. Benavides, B. Dougherty, and D. Schmidt, "Automated reasoning for multi-step software product line configuration problems," in *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*, San Francisco, US, August 2009, pp. 1–10.

[23] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro, "Automated error analysis for the agilization of feature modeling," *J. Syst. Softw.*, vol. 81, no. 6, pp. 883–896, 2008.

[24] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59 – 83, 1997, reasoning about Action and Change.

[25] U. Kuter, E. Sirin, B. Parsia, D. Nau, and J. Hendler, "Information gathering during planning for Web Service composition," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, pp. 183 – 205, 2005.

[26] L. Baresi and S. Guinea, "Self-supervising bpel processes," *IEEE Trans. on Software Engineering*, vol. 37, no. 2, pp. 247–263, 2011.

[27] N. C. Narendra, K. Ponnalagu, J. Krishnamurthy, and R. Ramkumar, *Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming.* Springer, 2007.