Development and Instrumentation of a Framework for the Generation and Management of Self-Adaptive Enterprise Applications[1]

Desarrollo e instrumentación de un marco de trabajo para generar y gestionar aplicaciones empresariales autoadaptativas[2]

Hugo Arboleda[3]
Andrés Paz[4]
Miguel Jiménez[5]
Gabriel Tamura[6]

doi:10.11144/Javeriana.iyu20-2.difg

How to cite this article:
H. Arboleda, A. Paz, M. Jiménez, and G. Tamura, "Development and instrumentation of a framework for the generation and management of self-adaptive enterprise applications," *Ing. Univ*., vol. 20, no. 2, pp. 303-333, 2016. http://dx.doi.org/10.11144/Javeriana.iyu20-2.difg

---

[1] Submitted on: December 18th, 2015. Accepted on: March 8th, 2016. This article is derived from the research project SHIFT: *Framework for the assisted generation of self-adaptive enterprise software*, with register code 2117-569-33721, developed by the research group I2T of the Universidad Icesi, Cali, Colombia.

[2] Fecha de recepción: 18 de diciembre de 2015. Fecha de aceptación: 8 de marzo de 2016. Este artículo se deriva de un proyecto de investigación denominado *SHIFT: Framework para la generación asistida de software empresarial en tiempo de ejecución*, con código de registro 2117-569-33721, desarrollado por el grupo de investigación I2T de la Universidad Icesi, Cali, Colombia.

[3] Ingeniero de sistemas, Universidad del Valle, Cali, Colombia. Magíster en Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia. Doctor en Informática, École des Mines de Nantes, Nantes, Francia. Doctor en Ingeniería, Universidad de los Andes. Director de la Maestría en Gestión de Informática y Telecomunicaciones y profesor asistente e investigador Grupo I2T, Universidad Icesi, Cali, Colombia. E-mail: hfarboleda@icesi.edu.co

[4] Ingeniero de sistemas, Universidad Icesi, Cali, Colombia. Magíster en Informática y Telecomunicaciones, Universidad Icesi. Asistente de investigación, Université du Québec, École de Technologie Supérieure, Montreal, Canadá.
E-mail: andres.paz@me.com

[5] Ingeniero de Sistemas, Universidad Icesi, Cali, Colombia. Estudiante Maestría en Informática y Telecomunicaciones, Universidad Icesi. Investigador Grupo I2T, Universidad Icesi. E-mail: majimenez@icesi.edu.co

[6] Ingeniero de Sistemas y Computación, Pontificia Universidad Javeriana, Cali, Colombia. Magíster en Sistemas y Computación, Universidad de los Andes, Bogotá, Colombia. Doctor en Informática, Université Lille 1, Lille, Francia. Doctor en Ingeniería, Universidad de los Andes. Profesor asociado e investigador Grupo I2T, Universidad Icesi, Cali, Colombia. E-mail: gtamura@icesi.edu.co

Abstract

Operations of companies have become over-dependent on their supporting enterprise software applications. This situation has placed a heavy burden onto software maintenance teams who are expected to keep these applications up and running optimally in varying execution conditions. However, this high human intervention drives up the overall costs of software ownership. In addition, the current dynamic nature of enterprise applications constitutes challenges with respect to their architectural design and development, and the guarantee of the agreed quality requirements at runtime. Efficiently and effectively achieving the adaptation of enterprise applications requires an autonomic solution. In this paper, we present SHIFT, a framework that provides (i) facilities and mechanisms for managing self-adaptive enterprise applications using an autonomic infrastructure, and (ii) automated derivation of self-adaptive enterprise applications and their respective monitoring infrastructure. Along with the framework, our work led us to propose a reference specification and architectural design for implementing self-adaptation autonomic infrastructures. We developed a reference implementation of SHIFT; our contribution includes the development of monitoring infrastructures, and dynamic adaptation planning and automated derivation strategies. SHIFT, along with its autonomic infrastructure and derived enterprise application, can provide a cost-effective mean to fulfill the agreed quality in these types of applications.

Keywords

self-adaptive enterprise applications; software product lines; component configurations

Resumen

Las operaciones de las empresas se han vuelto excesivamente dependientes en sus aplicaciones empresariales. Esta situación ha puesto una carga sobre los equipos de mantenimiento de *software*, de quienes se espera que mantengan estas aplicaciones disponibles y funcionando óptimamente en diferentes condiciones de ejecución. Sin embargo, esta alta intervención humana hace subir los costos totales de propiedad del *software*; además, la actual naturaleza dinámica de las aplicaciones empresariales constituye retos respecto a su diseño arquitectónico y su desarrollo, y el cumplimiento en tiempo de ejecución de los escenarios de calidad acordados. Para lograr adaptar las aplicaciones empresariales con eficiencia y eficacia se requiere una solución autonómica. Este artículo presenta SHIFT, un marco de trabajo que provee: 1) servicios y mecanismos para la gestión de aplicaciones empresariales autoadaptativas mediante una infraestructura autonómica, y 2) derivación automatizada de aplicaciones empresariales autoadaptativas y su respectiva infraestructura de monitoreo. Junto con el marco de trabajo, el trabajo lleva a proponer una especificación de referencia y un diseño arquitectónico para implementar infraestructuras autonómicas para autoadaptación. Se desarrolló una implementación de referencia de SHIFT. Se incluye el desarrollo de infraestructuras de monitoreo y estrategias de planeación dinámica de adaptaciones y derivación automatizada. SHIFT, junto con su infraestructura autonómica y aplicaciones empresariales derivadas, puede proporcionar un mecanismo costoefectivo para cumplir con la calidad acordada en este tipo de aplicaciones.

Palabras clave

aplicaciones empresariales autoadaptativas; líneas de producto de software; configuraciones de componentes

Introduction

Enterprise Applications (EAs) have become key assets for all modern organizations, holding huge amounts of data and providing concurrent user access to such information as well as processing services for it. They live in dynamic execution contexts and are no longer isolated but instead interact with other systems. With operations of companies increasing the dependency on their EAs, any disruption to them translates into severe direct and indirect financial losses (*e.g.*, lost transaction revenues, increased labor costs, legal penalties, lost business opportunities, brand damage) [1]. Modern EAs are, thus, expected to maintain functional and quality agreements despite the fact that their dynamic nature implies they are constantly under the influence of external, unforeseeable stimuli (*i.e.* disturbances) from various sources inside or outside the system scope that may affect their behavior or the levels at which they satisfy agreed quality. Regardless of the intrinsic uncertainty of disturbances and their possible sources, EAs still have to fulfill the customers' quality agreements. This has generated a growing interest concerning support of infrastructures for autonomic adaptation, as well as flexible architectural designs conceived for allowing recomposition at runtime. However, achieving self-adaptation in EAs requires a proper framework and tooling to cope with two challenges (i) the fulfillment of the agreed quality at runtime, and (ii) the design, development, and evolution of such self-adaptive enterprise applications.

In this paper, we consider the problem of implementing self-adaptation support in EAs. Targeting this we disclose SHIFT [2], a framework that provides (a) facilities and mechanisms as part of an autonomic infrastructure for managing self-adaptive enterprise applications based on the adaptation feedback loop of the DYNAMICO reference model [3] and (b) support for automated derivation of self-adaptive enterprise applications considering possible quality and monitoring variations. A number of proposals have achieved to develop autonomic infrastructures for self-adapting applications (*e.g.*, [4]–[7]); however, they do not fully address all the concerns related to the first challenge, and, furthermore, they do not approach the second challenge. Our goal is to offer a comprehensive proposal.

Our first technical contribution is to provide a low-level, complete and consistent specification and architectural design for building autonomic infrastructures with a minimum set of functional and non-functional requirements. Our requirements come mostly from IBM's architectural blueprint for autonomic computing [8], but we modify some of them, include new ones and specify them with the necessary technical details for their straightforward implementation. Our second technical contribution is to provide facilities that dynamically and non-intrusively measure relevant data from the EAs and their execution contexts. Our third technical contribution is to provide automated reasoning at runtime regarding context- and system-sensed data to determine and apply necessary adaptations to the EA, considering deployment and undeployment tasks. Our last technical contribution is to assist software architects and engineers in the specification and development of monitoring infrastructures, and the design and development of EAs. We contemplate the modeling of functional and quality variations, and support the automated derivation of EA components and their respective monitoring infrastructures.

We organize the remainder of this paper as follows. Section 1 introduces the context of our study and shows related work. Section 2 provides a high-level design of the SHIFT framework as well as an overview of its reference implementation. Section 3 describes the specification and architectural design of SHIFT's autonomic infrastructure, and presents our monitoring infrastructure and adaptation planning implementations. Section 4 exposes the mechanisms for SHIFT's assisted derivation of monitoring infrastructures and enterprise applications. Last section sets out a summary of our contributions and outlines future work.

1. Background and Related Work

Our general context comprises self-adaptive software systems, autonomic infrastructures providing self-adaptation support, and automated derivation of software systems. Subsection 1.1 presents a short overview of the general responsibilities and comprising elements of autonomic infrastructures. Subsection 1.2 briefly describes the evolution of generative software development. Subsection 1.3 discusses related works and contrasts them with our proposed framework to qualitatively determine its soundness.

1.1. Autonomic Infrastructures

An autonomic infrastructure [8] is the infrastructure that allows a managed system to be adapted to unforeseen context changes in order to ensure the satisfaction of agreed quality. Composing this infrastructure are five basic elements: a *monitor* element that continuously senses relevant context data; an *analyzer* that interprets monitoring events reported by the *monitor* to determine whether the SLAs are being fulfilled or predict future shortcomings by correlating current measurements with historical data; and the *planner* and *executor* elements that synthesize and realize (respectively) action plans to alter the system's behavior, either by modifying the system structure or by varying parameters to reach a desired system state. These four components share relevant information through the *knowledge manager* element. The autonomic infrastructure interfaces with the *managed system* through a set of touchpoint elements, namely *sensors* and *effectors*. *Sensors* collect measurements of variables of interest from the *managed system*; *effectors* provide the necessary interfaces to modify the resources or artifacts of the *managed system*. The DYNAMICO reference model [3] comprises three autonomic infrastructures characterizing three identified levels for self-adaptation: (i) control objectives, (ii) managed system, and (iii) monitoring infrastructure. Due to extension limitations, in this paper we do not delve into the details of DYNAMICO; for more information refer to [3].

We have identified a lack of a detailed, standard reference specification and architectural design for building autonomic infrastructures. In light of this, our previous work in [9] gives a first step towards this with the design of a component-based architecture for the five basic elements previously introduced. However, such a work does not provide a complete reference specification that sets out detailed functional scope, restrictions, and quality concerns. It is our interest to build on this work and present a reference specification and architectural design for the implementation of an autonomic infrastructure, including its functional scope, restrictions, and quality concerns, and a reference implementation for it.

1.2. Automated Derivation

Automated derivation of software seeks to automatically generate software assets from given written specifications [10]. The combination of Software Product Line Engineering (SPLE) and Model-Driven Engineering (MDE) has attracted attention as an important automated derivation approach. On the one hand, SPLE [11] aims to derive high-quality software through a (semi-)automatic development process that models families of closely related software systems in terms of their shared common features and their variations. Then, it builds their implementations by assembling reusable assets promoting the desired features. The conceptual problem space captures a family's functional and non-functional requirements in terms of variability models (*e.g.*, feature models, orthogonal variability models), which also govern product configuration knowledge. In the solution space, SPLE approaches use a variant derivation mechanism to transform a product configuration into a concrete product.

On the other hand, MDE's principle is to use domain specific models representing software system specifications as first-class artifacts during the whole development process [12]. The development of domain-specific models may be guided by: metamodels or domain-specific languages (DSLs) [13]. The former involves an abstract representation of domain concepts and their relationships. The latter involves a context-free grammar that determines the syntax (abstract and concrete) and semantics for a textual language. Generators transform such models (incrementally or in one step) into source code. The generators make use of model-to-model transformations, which take a model and transform it into another model with a different representation, and model-to-text transformations, which take a model and transform it into source code representation.

It is our interest to follow an MD-SPLE approach for the automated derivation of component-based enterprise applications, additional deployable enterprise application components when an adaptation requires them, and the artifacts making up sensor and monitor elements of the autonomic infrastructure.

1.3. Related Work and Conceptual Validation

Current approaches implement dynamic adaptation of service compositions at the language level [14]–[16], or using models at runtime [17]–[20]. The first ones have specific facilities, tied to the languages themselves, to handle the definition of constraints and conditions that regulate the replanning of compositions at runtime. Despite these flexibilities, they can be complex and time-consuming, and with low-level implementation mechanisms. Model-based approaches for dynamic adaptation of service compositions, on the other hand, implement, tacit or explicitly, the five basic elements of autonomic infrastructures [8]: (i) a Monitor, (ii) an Analyzer, (iii) a Planner, (iv) an Executor, and (v) a Knowledge manager. Our work is related to approaches that use models at runtime.

The recent work of Alférez *et al.* [21] summarizes good practices implementing autonomic infrastructures and gives implementation details about reconfiguration mechanisms. They center their attention on service recomposition at runtime using (dynamic) product line engineering practices for assembling and redeploying complete applications according to context- and system-sensed data. However, model-based approaches for dynamic adaptation of service compositions (*e.g.*, [17], [21], [22]) do not consider changing requirements over Service-Component Architecture (SCA) composites, Enterprise Java Beans (EJB), or OSGi models. This triggers new challenges given the complexity of deployment at the stage of adapting composites, EJB, and bundle bindings. The work of van Hoorn *et al.* [23] goes in this direction by proposing an adaptation framework operating over component-based software systems. Nonetheless, their proposal remains at a high level without working with specific component models and their framework is centered around component migration and load balancing, while our interest is component recomposition over SCA composites, EJB components, and OSGi bundles.

The work of Cedillo *et al.* in [17] is also closely related to ours. They propose a middleware for monitoring cloud services defined around a monitoring process that uses models at runtime capturing low- and high-level non-functional requirements from Service Level Agreements (SLAs). Their middleware only provides a partial implementation of an autonomic infrastructure, specifically of the monitor and analyzer elements. Their proposal derives the monitoring code from the input model at runtime. The monitoring code is used by the middleware during the monitoring process. Heinrich *et al.* [22] also work around monitoring cloud applications; however, they are only concerned with triggering change events when the observation data model is populated at runtime.

Other approaches do not take into account the design, development, and evolution of self-adaptive applications. Our goal is to grant support for the assisted derivation of EA components and their associated monitoring infrastructures. This is important in order to efficiently provide standard mechanisms to control the monitors' behavior. Assisted derivation of both EA components and monitoring infrastructure also guarantees relevance of the complete self-adaptive architecture in changing context conditions of system execution [3].

In previous works, we proposed independent approaches and implementations in the contexts of the engineering of highly dynamic adaptive software systems with the DYNAMICO reference model [3] model-based product line engineering with the FIESTA approach [24]–[26]; automated reasoning for derivation of product lines [27]; and the recent (unpublished) contributions regarding quality variations in the automated derivation process of product lines [26]. The SHIFT framework is motivated by the required integration of all these efforts as part of the SHIFT research project in a move to approach automation and quality awareness along the life cycle of enterprise applications.

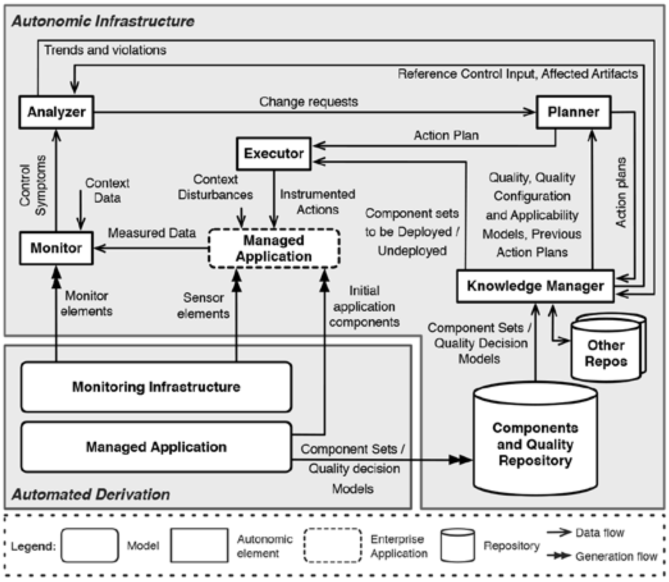
2. The SHIFT Framework

The SHIFT Framework has two layers: *Autonomic Infrastructure* and *Automated Derivation*. Figure 1 depicts the high-level architectural view of the SHIFT Framework. Subsection 2.1 describes the scope of the *Autonomic Infrastructure* layer. Subsection 2.2 describes the scope of the *Automated Derivation* layer. Following sections delve into key elements of these two layers.

2.1. Autonomic Infrastructure

The *Autonomic Infrastructure* layer (see Figure 1 top) provides an implementation of the adaptation feedback loop of the DYNAMICO reference model [3]. As part of this layer, SHIFT deploys a monitoring infrastructure bound to the *Managed Application*. The monitoring infrastructure comprises a set of *Sensor* and *Monitor* elements for allowing the measurement of actual service executions in the *Managed Application*. Monitoring rules define when a control symptom should be reported to the *Analyzer* element for further analysis. The *Analyzer* element is in charge of deciding when an adaptation is needed to ensure the fulfillment of performance SLAs. When the *Analyzer* element identifies an adaptation symptom, the framework considers the need for dynamically deploying and undeploying components in response. Thus, the *Planner* element provides automated reasoning on the dynamic creation of structural solutions. In order to obtain the best possible selection of components when configuring an adaptation to a deployed product, we rely on constraint satisfaction to reason on the set of constraints defined by reachable quality scenarios configurations and their relationships with the component sets implementing them. Interactions between quality scenarios may occur, and since different component sets may be available, conflicts between component sets may arise. Through automated reasoning, the *Planner* element may cope with this issue by taking into account additional information to get the best possible selection of component sets when determining an action plan to preserve the fulfillment of performance SLAs, when possible.

Figure 1. High-level architectural view of the SHIFT components



Source: authors' own elaboration

Realizing an action plan is the responsibility of the *Executor* element. This task includes, sequentially or concurrently, 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. All of these while redirecting new requests for the application's components to the new instances being deployed, and allowing existing requests and sessions to properly terminate. The *Executor* element performs these actions over SCA composites, EJB components, and OSGi bundles by means of the introspection capabilities in the FraSCAti middleware [28] and the *dynamic redeployment in operational environment* features in the GlassFish and Equinox middleware. In addition, The *Executor* element is able to recompile the system's source code, if necessary, to make measurement interfaces available to the monitoring infrastructure. Accordingly, these deployment tasks are applied to the *Monitor* element to effectively ensure dynamic quality awareness.

Some conceptual constraints, nonetheless, will limit the reach of the framework in the *Autonomic Infrastructure* layer. For instance, the measurement of quality attributes is a challenging field and many of them are particularly difficult to measure (*e.g.*, the security quality attribute). Additionally, the *Autonomic Infrastructure* elements are inherently tied to the Managed Application at different extents, particularly the *Analyzer*, the *Planner*, and the *Executor* elements are closely related to it. We are currently focused on the performance quality attribute and, thus, automated measurement support is bound to the provided performance sensors, any other measurement will require the manual development of the corresponding sensor. For the *Planner* element, with the use of the principles of constraint satisfaction we have detached the concerns related to the managed application into a model representation called PISCIS [29] derived, and stored in a repository managed by the *Knowledge Manager* element.

2.2. Automated Derivation

As for the *Automated Derivation* layer (see Figure 1 bottom left), SHIFT is concerned about provisioning, through automated derivation, (i) component-based self-adaptive enterprise applications and their respective monitoring infrastructures, and (ii) artifacts (*i.e.* models and deployable enterprise application components) that are input of the adaptation processes initiated by the autonomic infrastructure.

In this layer SHIFT uses two interrelated models. On the one hand, the *Monitoring Infrastructure* model captures the monitoring infrastructure scope, *i.e. Sensor* elements that will be attached to the *Managed Application* through a non-intrusive strategy based on aspect-oriented programming, and event-based *Monitor* elements that collect context data. On the other hand, the *Managed Application* model captures the functional, quality, and architectural scopes of the EAs. Generated component sets and quality decision models, relating component sets with quality scenarios, are stored in the *Components and Quality Repository*, which is managed by the *Knowledge Manager* element. Although the process of generating EA components is automated, binding the functional, quality, and architectural scopes of an EA requires the intervention of a software architect since complex interactions may arise.

3. Autonomic Infrastructure

Pertaining the *Autonomic Infrastructure* layer in Figure 1 (top), SHIFT contains a reference implementation for the autonomic infrastructure necessary to realize the adaptation feedback loop of the DYNAMICO reference model. Subsection 3.1 presents the proposed reference specification for building autonomic infrastructures for self-adaptation and from which SHIFT was built. Subsection 3.2

describes in detail the monitoring infrastructure. Subsection 3.3 exposes our dynamic adaptation planning strategy for the *Planner* element.

3.1. Reference Specification

We build from the design proposed in [9] and present a reference specification and architectural design for the implementation of an autonomic infrastructure, including its functional scope and quality considerations, and structural and behavioral architectural designs.

3.1.1. Functional Scope

An autonomic infrastructure is required to implement the following set of functional requirements, based on IBM's architectural blueprint for autonomic computing in [8].

Sensor requirements:

S-1 A Sensor element must collect measurements of variables of interest (from now on referred to as *sensed data*) (*e.g.*, quality attributes specified in the series of standards ISO 25000 [30] like performance of a service, availability of resources, topology information, configuration properties) in the context in which it is located, *i.e.* its execution context or the context of the domain to which it belongs.

S-2 A Sensor element must temporarily store sensed data.
Rationale. The Monitor elements' responsiveness relies on the timely availability of the sensed data. This availability can be achieved by supporting temporary storage, which would allow Monitor elements to gather data at any moment. Nonetheless, Sensor elements can use up memory space assigned to the Managed Application, thus, other storage options should be taken into consideration.

S-3 A Sensor element must expose a subset of the sensed data to the set of Monitor elements, whether both Monitor and Sensor elements have been deployed jointly or independently.

S-4 A Sensor element must remove a subset of the sensed data being stored temporarily when instructed by a Monitor element.

S-5 A Sensor element must perform primitive operations (*e.g.*, count repetitions of a measurement in a given time interval) on a subset of the sensed data.
Rationale. The ongoing transmission of sensed data from Sensor elements to Monitor elements can overuse network resources, thereby hindering the Managed Application's regular operation. Placing primitive operations in Sensor elements can considerably reduce the amount of data transmitted through the network when Monitor elements do not require the entire collection of sensed data but, instead, calculations over it.

Monitor requirements:

M-1 A Monitor element must obtain the sensed data from one or more Sensor elements where it has been captured through the required access modes, *i.e.* by request (pull) or per occurrence (push).

M-2 A Monitor element must calculate metrics (based on sensed data) related to the variables of interest to characterize the current state of the Managed Application. Said calculation can be made periodically or whenever a new measurement happens, which would produce average or instant calculations, respectively. This calculation can also involve the composition or correlation of metrics calculated by other Monitor elements.

M-3 A Monitor element must make the calculated metrics available, through the Knowledge Manager element, to other Monitor elements so they can compose their own calculations.

M-4 A Monitor element must filter the calculated metrics before being reported to the Analyzer element. The filter must be done through the application of a set of domain-dependent monitoring rules over the calculated metrics.

M-5 A Monitor element must report to the Analyzer element control symptoms, *i.e.* the metrics (simple or compound) that meet the conditions set by the monitoring rules.

M-6 A Monitor element must allow changing the periodicity in which it calculates its metrics.

M-7 A Monitor element must allow to update the set of monitoring rules it applies to perform the filter of metrics. Such update may be triggered by, for example, a structural change of the Managed Application, or a change in the quality scenarios.
Rationale. Business and system's operation can make a variable of interest gain or lose relevance, thereby requiring flexibility against such behavior at runtime. Furthermore, providing elements with operations to control their internal behavior help support such flexibility.

Analyzer requirements:

A-1 An Analyzer element must evaluate reported control symptoms against reference values previously established (corrective behavior). Reference values must be recovered using the Knowledge Manager element. The evaluation should identify violations that occur with respect to these reference values. A violation indicates an adaptation symptom.

A-2 An Analyzer element must store a record of trends and violations through the Knowledge Manager.

A-3 An Analyzer element must reason about the reported control symptoms taking into account the historical records of trends and violations (recovered using the Knowledge Manager element) to identify observable degradation trends with respect to the reference values (also recovered using the Knowledge Manager element) to avoid future violations (predictive behavior). The evaluation can employ time-series forecasting and queuing models. An observable degradation trend indicates an adaptation symptom.

A-4 An Analyzer must create and send one or more change requests to the Planner element if adaptation symptoms are detected. Such request must include which variable of interest is at risk of being (predictive behavior) or has already been (corrective behavior) violated, the variable's corresponding value, the motive for the request (*e.g.*, violation, risk of violation), and the set of artifacts under the scope of the variable of interest (*i.e.* affected artifacts).

Planner requirements:

P-1 A Planner element must reason about the variable of interest, the degree of the violation, and the set of affected artifacts to identify a reachable, optimum resolution. For this reasoning, the Planner element must take into account the quality, quality configuration, and artifact applicability models. This information must be recovered using the Knowledge Manager.

P-2 A Planner element must perform a gap analysis to determine the necessary, high-level actions (*e.g.*, deploy new artifacts, redeploy existing artifacts, replace existing artifacts with alternate ones, remove existing artifacts, update configuration setting) to reach the identified resolution.

P-3 A Planner element must create and send an action plan to the Executor element. Such action plan must include the set of high-level control actions determined with the gap analysis that will modify the Managed Application.

P-4 A Planner element must store a record of optimum resolutions and their corresponding action plans through the Knowledge Manager.

P-5 A Planner element must recover a previous action plan through the Knowledge Manager if the reachable optimum resolution identified matches to one of the action plans stored.

Executor requirements:

E-1 An Executor element must perform the realization of the action plan given by the Planner element through the scripting of executable commands (*e.g.*, compile, deploy, redeploy) by the corresponding Effector elements.

E-2 An Executor element must use the corresponding Effector element to run commands over the Managed Application.

Effector requirements:

Ef-1 An Effector element must allow managing a resource or set of resources (*e.g.*, manage a middleware to deploy, redeploy, and undeploy components).

Knowledge manager requirements:

K-1 A Knowledge Manager element must perform create, retrieve, update, and delete operations over the repositories where the information of interest to the other elements of the autonomic infrastructure is stored.

K-2 A Knowledge Manager element must provide support operations for the analysis of the information managed by it.

3.1.2. Quality considerations

An autonomic infrastructure must satisfy a set of quality concerns impacting its functioning. As Sensor elements are placed inside of the Managed Application, the application's regular operation is subject to be impacted as well. In addition to the previous functional requirements and their supporting quality rationale, we identified the quality scenarios in Tables 1 and 2... describing concerns regarding the development of autonomic infrastructures.

Table 1. Quality scenario for Accountability and Analyzability of adaptation symptoms

Quality Scenario	1.	Traceability of adaptation symptoms
Quality attribute	Security – Accountability; Maintainability – Analyzability	
Justification	The lack of information in an adaptation symptom can obscure its root cause.	
Stimulus	A new change request is sent to the Planner element	
Source of stimulus	Analyzer element	
Environment	The Managed Application is not in a desired state; an adaptation is under way.	
Artifact	Planner element	
Response	The change request contains all the necessary information such that all adaptation symptoms can be traced back to the originating sources, in order for the Planner element to know what artifacts from the Managed Application to consider in the adaptation.	

Source: authors' own elaboration

Table 2. Quality scenario for Co-existence, Interoperability, and Modularity of Sensor elements

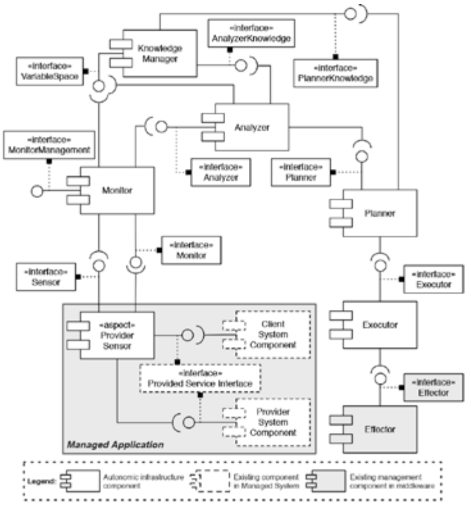
Quality Scenario	2.	Introduction of Sensor elements
Quality Attribute	Compatibility – Co-existence and Interoperability; Maintainability – Modularity	
Justification	Sensor elements can be entangled with the Managed Application, thereby increasing the complexity of the Managed Application's maintainability.	
Stimulus	A new Sensor element is introduced into the Managed Application	
Source of Stimulus	A change in an existing quality scenario or the occurrence of a new scenario.	
Environment	Managed Application and Autonomic Infrastructure are under operation.	
Artifact	The intended component to be sensed in the Managed Application.	
Response	The Sensor element is introduced into the Managed Application in a non-intrusive way.	

Source: authors' own elaboration

3.1.3. Architectural Design

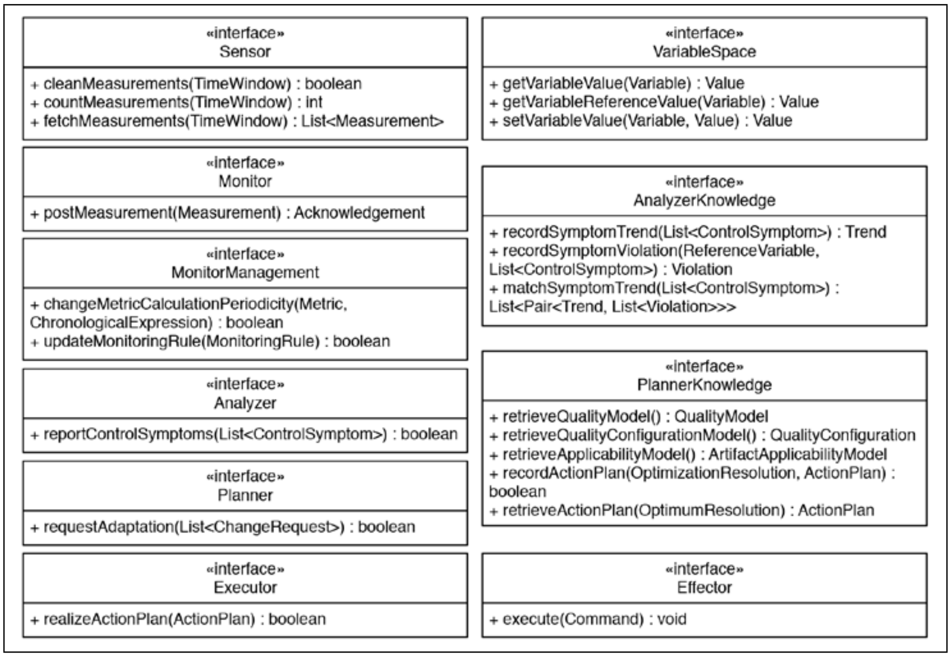
The architectural design of the autonomic infrastructure satisfies the previous requirements specification and follows a component-based model. The component diagram in Figure 2 presents the structural view of the autonomic infrastructure's components. Due to space restrictions, communication interfaces are only named in Figure 2; Figure 3 presents the definition of such interfaces in a class diagram representation.

Figure 2. Component diagram for the reference architecture of an autonomic infrastructure


«interface» AnalyzerKnowledge
Knowledge Manager
«interface» VariableSpace
«interface» PlannerKnowledge
Analyzer
«interface» MonitorManagement
«interface» Analyzer
«interface» Planner
Monitor
Planner
«interface» Monitor
«interface» Sensor
«interface» Executor
«aspect» Provider Sensor
Client System Component
Executor
«interface» Provided Service Interface
«interface» Effector
Provider System Component
Effector
Managed Application
Legend:
Autonomic infrastructure component
Existing component in Managed System
Existing management component in middleware

Source: authors' own elaboration

Figure 3. Class diagram for the communication interfaces in Figure 2


«interface»
Sensor
+ cleanMeasurements(TimeWindow) : boolean
+ countMeasurements(TimeWindow) : int
+ fetchMeasurements(TimeWindow) : List<Measurement>
«interface»
VariableSpace
+ getVariableValue(Variable) : Value
+ getVariableReferenceValue(Variable) : Value
+ setVariableValue(Variable, Value) : Value
«interface»
Monitor
+ postMeasurement(Measurement) : Acknowledgement
«interface»
AnalyzerKnowledge
+ recordSymptomTrend(List<ControlSymptom>) : Trend
+ recordSymptomViolation(ReferenceVariable, List<ControlSymptom>) : Violation
+ matchSymptomTrend(List<ControlSymptom>) : List<Pair<Trend, List<Violation>>>
«interface»
MonitorManagement
+ changeMetricCalculationPeriodicity(Metric, ChronologicalExpression) : boolean
+ updateMonitoringRule(MonitoringRule) : boolean
«interface»
Analyzer
+ reportControlSymptoms(List<ControlSymptom>) : boolean
«interface»
PlannerKnowledge
+ retrieveQualityModel() : QualityModel
+ retrieveQualityConfigurationModel() : QualityConfiguration
+ retrieveApplicabilityModel() : ArtifactApplicabilityModel
+ recordActionPlan(OptimizationResolution, ActionPlan) : boolean
+ retrieveActionPlan(OptimumResolution) : ActionPlan
«interface»
Planner
+ requestAdaptation(List<ChangeRequest>) : boolean
«interface»
Executor
+ realizeActionPlan(ActionPlan) : boolean
«interface»
Effector
+ execute(Command) : void

Source: authors' own elaboration

The autonomic infrastructure components are arranged in a 'pipes and filters' architectural style, where each component performs subsequently the specific functions of the corresponding autonomic element in the autonomic infrastructure. Each component, thus, exposes a very simple interface to receive an inbound message, process it and forward the result to the next component. The *Analyzer*, *Planner*, and *Executor* elements are implemented by one component. Generally, one would expect to have multiple *Sensor* and *Monitor* components depending on the variables of interest. The Effector component is usually provided by the middleware supporting the *Managed Application*, although one would have to be built if the middleware does not provide one. If the *Managed Application* is distributed among different middleware, an *Effector* component is required for each middleware. The adaptation processing behavior exhibited by the autonomic infrastructure is specified step by step in the collaboration diagram in Figure 4. The internal designs of the components' structure and behavior are not detailed in this paper (except for the Planner, which is shown in Subsection 3.3); these will be part of future publications.

3.2. Monitoring Infrastructure

The monitoring infrastructure is supported at runtime by the PASCANI library, a collection of classes that allows implementing the monitoring elements of the reference architecture, namely *Sensors* and *Monitors*. *Sensors* are introduced into the system, their services bound dynamically and then started, therefore allowing to measure actual service executions. *Monitors* contain the necessary logic to abstract single context events (*i.e.* events arising from *Sensors*) into complex and relevant monitoring data to be analyzed by the *Analyzer* and other components (*e.g.*, log components and monitoring dashboards).

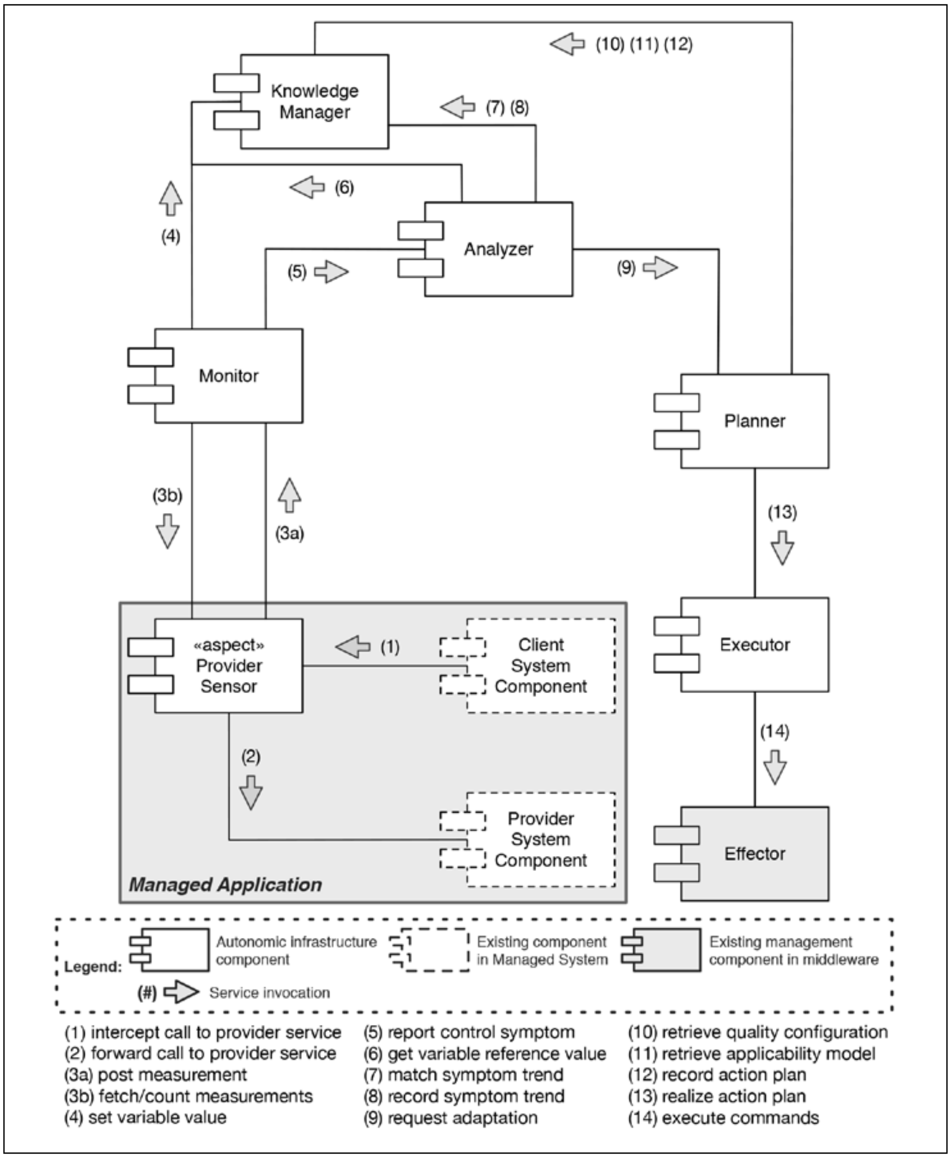
Both *Sensors* and *Monitors* are supplied with standard traceability and controllability mechanisms to (i) prevent the monitoring infrastructure from introducing considerable overhead in the system's regular operations, and (ii) feed knowledge sources with relevant monitoring data. Controlling the produced executable monitoring components is important when the *Managed Application* reaches critical quality levels, given that it can end up breaching quality agreements or overusing system resources. This is also important in order to keep the monitoring data relevant through time, as new variables of interest can emerge as product of system and business evolution. Adding support for monitoring new variables at runtime requires introducing new *Sensors* into the *Managed Application* and monitoring rules into the monitoring infrastructure. The interaction between *Sensors* and *Monitors* is event-based, and is specified in a single source file.

Besides *Sensors* and *Monitors*, PASCANI includes a shared variable model containing relevant monitoring variables holding both reference values (*e.g.*, Service Level Indicators contracted in SLAs) and values describing the current state of the system (*e.g.*, current system throughput). *Monitors* and other components can read and update these values; additionally, they can observe changes in them, by defining events in the monitoring specifications (see Subsection 4.1).

3.3. Dynamic Adaptation Planning

The *Planner* element of the *Autonomic Infrastructure* layer in Figure 1 (top) is a key factor in the SHIFT Framework. SHIFT's *Planner* element takes advantage of PISCIS, a formal model based on constraint satisfaction for adaptation planning we presented in [29]. The *Planner* element includes automated reasoning facilities that help design solutions that alter the system's behavior by modifying its structure or by varying parameters to reach a desired system state. In order to obtain the best possible selection of components to alter the system's behavior we use the PISCIS model, which relies on the principles of constraint satisfaction to (i) capture the set of constraints that define reachable architectural adaptations, and (ii) provide information to reason over the best possible solution. We have adapted and extended the definitions presented in {27} for the PISCIS model. Figure 5 provides a look at the internal behavior of the *Planner* element in the SHIFT Framework.

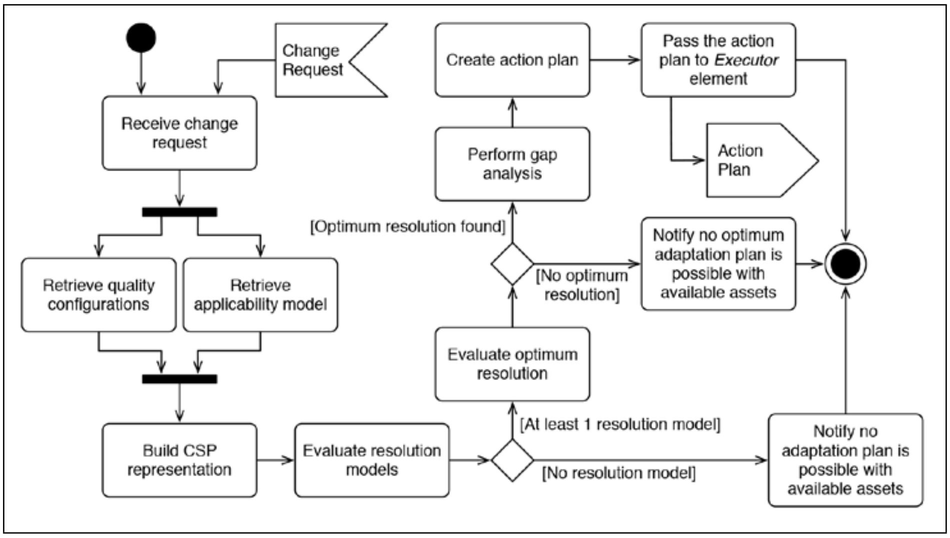
Figure 4. Component collaboration for the autonomic infrastructure in Figure 2



Source: authors' own elaboration

The process of planning an adaptation initiates with a change request received from the *Analyzer* element. The *Planner* then retrieves the EA's quality configuration and applicability model, inputs to the PISCIS model. A quality configuration consists of a finite set of quality scenarios classified as *unselected* (*i.e.* with a state of 1) or *selected* (*i.e.* with a state of 2). We relate on *applicability models* the information of applicable *component sets* promoting such quality scenarios in order to define the necessary actions to derive adaptation plans in accordance with a quality configuration. Implementing a quality scenario in an application may often require several composed components, thus, we refer as a *component set* to the set of composed components implementing a quality scenario. An *applicability model* is a finite set of weighted application relationships between one *component set* and one quality scenario. The application relationship may be 0 if the quality scenario does not constraint the application or deployment of the *component set*, 1 if the *component set* requires the quality scenario to be *unselected*, and 2 if the *component set* requires the quality scenario to be *selected*.

Figure 5. Activity diagram for the Planner element


Change Request
Create action plan
Pass the action plan to *Executor* element
Receive change request
Perform gap analysis
Action Plan
[Optimum resolution found]
Notify no optimum adaptation plan is possible with available assets
[No optimum resolution]
Retrieve quality configurations
Retrieve applicability model
Evaluate optimum resolution
[At least 1 resolution model]
Build CSP representation
Evaluate resolution models
[No resolution model]
Notify no adaptation plan is possible with available assets

Source: authors' own elaboration

With these models, the Planner builds a PISCIS model, *i.e.* a constraint satisfaction problem (CSP) representation, to evaluate the set of constraints defined by reachable quality configurations and their relationships with *component sets*. A *resolution model* is an applicability model instance (*i.e.* a solution to the CSP), which binds variability and defines the system's future structure, *i.e.* the resulting adaptation plan. A *resolution model*, or adaptation plan, is a finite set of *component set* applications. The application is not planned if the *component set* should not be deployed, and planned if the *component set* should be deployed. However, not every possible *resolution model* is a valid solution. A valid solution must satisfy the following constraints: (i) a *component set* must be deployed satisfying the respective application relationship of the applicability model; (ii) two deployable *component sets* must not exclude each other; and (iii) all applicable *component sets* must take into account all the quality scenarios' states in the configuration.

Since many valid solutions may be found, we have formulated in [29] some operations on the previous CSP representation to provide the *Planner* element with additional information in order to determine the best possible solution. The *application* operation takes an applicability model, a quality configuration, and a resolution model to verify the resolution model's applicability as a solution. The *possible resolutions* operation calculates all the potential solutions from the given quality configuration and applicability model. The *number of resolutions* operation calculates the number of potential resolution models from the given quality configuration and applicability model. This operation gives an indication of flexibility and complexity of the applicability model. The *validation* operation indicates if a given applicability model can provide at least one resolution model.

If no valid resolution model is found the *Planner* should notify no adaptation is possible with the available assets; thus, new assets should be derived. If at least one resolution model is found, the *Planner* should evaluate for an optimum resolution. Three operations are used in this task. The *flexible component sets* operation determines the component sets shared by a given set of possible resolution models. The *inflexible component sets* operation gives the opposite result of the flexible component sets operation, *i.e.* the component sets unique to each resolution model in a set of possible resolution models. The *optimum resolution* operation finds the best resolution model within a set of possible resolution models through the use of a maximizing or minimizing function depending on whether the greater or the least number of component sets, respectively, is more fit to adapt the *Managed EA*. If no optimum resolution is found, the *Planner* will notify of the event. If an optimum resolution is found, a gap analysis is performed to compare the actual deployed artifacts with the identified solution and determine the necessary, high-level actions that will achieve the target deployment. The action plan is created and finally passed to the *Executor* element for realization.

4. Automated Derivation

The *Automated Derivation* layer in Figure 1 (bottom) contains two interrelated models: *Monitoring Infrastructure* and *Managed Application*. Subsection 4.1 outlines the *Monitoring Infrastructure* model and illustrates how *Sensor* and *Monitor* components are derived from such model. Subsection 4.2 explains the *Managed Application* model and describes how EA components are derived from such model.

4.1. Specification and Derivation of Sensor and Monitor Elements

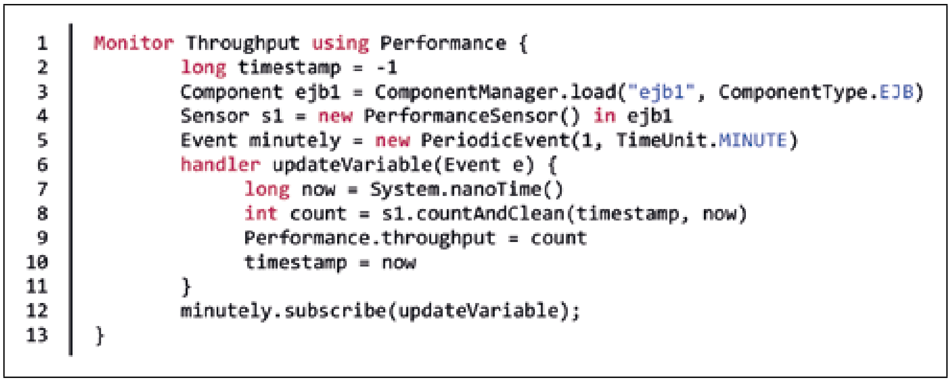
In SHIFT, the specification and generation of monitoring infrastructures, deployable at runtime, is performed through PASCANI. PASCANI is a Domain-Specific Language (DSL) still under development that allows defining and generating sensors and monitors.

Monitoring specifications can be parameterized and derived in an automated way for those quality attributes with clear definition of metrics and measurement methods [30]. In SHIFT's current state, we have already designed a mechanism for automatically generating PASCANI components for the performance quality attribute. This mechanism takes place in the automated derivation phase, and produces the monitoring component and its corresponding deployment descriptors. A monitoring specification comprises a Variables Store and a Monitor skeleton for each performance factor covered in the quality submodel of the Decision Support model. EA developers should use these Monitor skeletons as templates to declare the actual *Managed Application* components that will be subject of measurement.

One of most useful features of PASCANI is the standard abstraction between measurement mechanisms and event-based monitoring logic, which is based on the Event-based and Implicit invocation architectural style [31]. This separation of concerns allows PASCANI to monitor different quality attributes, as far as sensors implementing the necessary measurement methods exist. In our current implementation, sensors to measure performance factors and exceptional behaviors are automatically generated and inserted into the *Managed Application*'s components.

Figure 6 shows a simple specification for monitoring the latency of an EJB. Line 1 declares that monitor *Throughput* will read and update variables within the *VariableSpace Performance*. As illustrated in Figure 7, a variable space reserves space for variable names and values. This facilitates the aggregation of monitoring variables from different monitors. Line 4 of Figure 6 creates and associates a sensor element to the EJB *ejb1*. Additionally, it provides the monitor with direct access to the sensor. Line 5 defines a periodic event to be raised every minute; each time it is raised, the *throughput* variable is updated according to the number of service executions during the last minute (see lines 8 and 9).

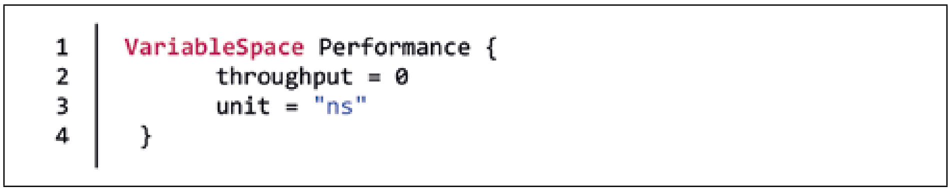
Figure 6. Monitor specification to monitor Service Time-behavior



```
 1 | Monitor Throughput using Performance {
 2 |         long timestamp = -1
 3 |         Component ejb1 = ComponentManager.load("ejb1", ComponentType.EJB)
 4 |         Sensor s1 = new PerformanceSensor() in ejb1
 5 |         Event minutely = new PeriodicEvent(1, TimeUnit.MINUTE)
 6 |         handler updateVariable(Event e) {
 7 |               long now = System.nanoTime()
 8 |               int count = s1.countAndClean(timestamp, now)
 9 |               Performance.throughput = count
10 |               timestamp = now
11 |         }
12 |         minutely.subscribe(updateVariable);
13 | }
```

Source: authors' own elaboration

Figure 7. Variable Space specification for holding and sharing throughput values



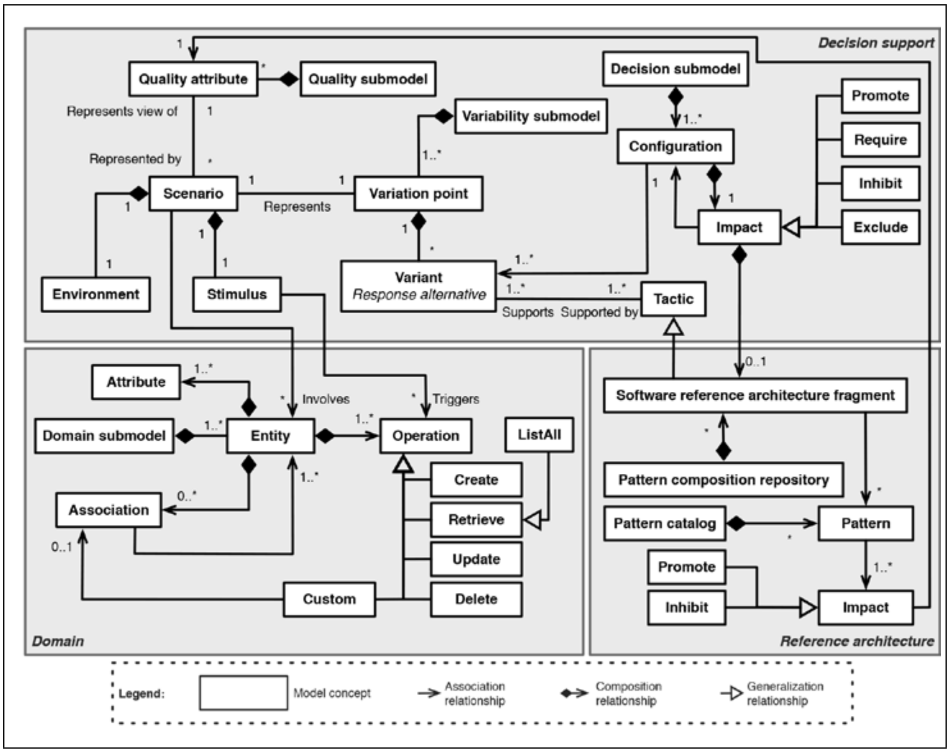
```
VariableSpace Performance {
        throughput = 0
        unit = "ns"
 }
```

Source: authors' own elaboration

4.2. Specification and Derivation of Managed Applications

Figure 8 shows the *Managed Application* model through a UML-like notation diagram. Quality concerns of an EA are captured in the *Decision Support* scope under the *quality submodel* concept (see Figure 8 top center) as quality *scenarios*. A quality *scenario* may involve various functional artifacts in the EA being stimulated. The *Domain* scope (see Figure 8 bottom left) comprises concepts for capturing the functional scope of EAs in terms of business *entities*, and their *associations* and *operations* [26]. The Domain scope may be extended to include more complex business logic representations. A quality *scenario*, thus, relates to a stimulated *entity* and the corresponding *stimulus* triggers an *operation* defined in such entity. Since quality concerns for a self-adaptive EA may (and do) vary over time, a *variability submodel* (see Figure 8 top center) in the *Decision Support* scope is focused on supporting this information. Quality *scenarios* in the *variability submodel* are represented as *variation points* and the possible *alternative responses* are the *variants*.

Figure 8. Interrelated models for the derivation of component sets



Source: authors' own elaboration

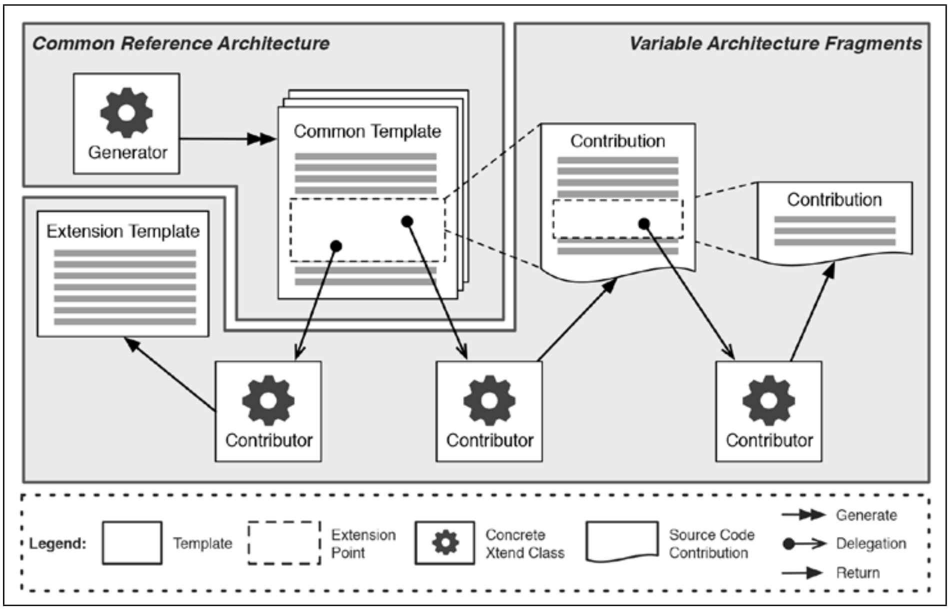
The *Reference Architecture* scope (see Figure 8 bottom left) is focused on supporting the modeling of software architectural implementations for quality variations. In order to associate architectural implementations for quality variants, we select design *patterns* in their pure form or we compose them. Resulting structures are documented as variable *software reference architecture fragments* that are later composed and made concrete during the derivation process of components and complete applications. In that way, we compose patterns respecting a base (common) reference architecture, over which variable reference architecture fragments are integrated before deriving concrete implementations. We also document the *impact* of patterns over quality attributes, *i.e.* a pattern *promotes* or *inhibits* a *quality attribute*. This is useful information when choosing design implementations for quality variants. The *Reference Architecture* scope may be extended to include support for modeling other non-software related architectural concepts for hardware or network architecture by extending the *tactic* concept.

The *Decision Support* scope provides support for assisted reasoning regarding achievable quality configurations and their interactions. Our *decision submodel* (see Figure 8 top right) is a collection of (partial) reachable product quality *configurations*, expressed as sets of quality *variants*, and the modeling of their *impact* on other configurations. The impact of one configuration over another is expressed in terms of *promote*, *require*, *inhibit*, and *exclude* relationships. For every impact of one configuration over another, a *reference architecture fragment* should be associated, if a reasonable solution that accommodates both configurations can be achieved. Such architecture fragments model resulting structures and behaviors that produce the composition of patterns associated to variants involved in the related configurations. Concrete architectures of reusable components and complete applications are created as a composition of a common reference architecture and reference architecture fragments. Composition rules are managed in model-based artifacts that will be introduced in the following subsection.

Components result from transforming into source code a set of functionalities with a configuration of quality levels and structured by a composition of common and variable software reference architecture fragments. The transformation process satisfies the constraints and conditions dictated by a common reference architecture and the variable reference architecture fragments that contribute to the overall architecture (see Figure 8 bottom left). EA developers should use the derived components as partial implementations conforming to a reference architecture promoting the desired quality attributes and must complete such implementations in order to fulfill their project's specific functional requirements.

Our derivation strategy is based on a composable templates approach that delegates responsibilities on the templates themselves to reduce the need for a separate, bulky control logic to weave common and variable reference architecture fragments. Each template will contain a set of model-to-text transformations. Figure 9 depicts such interaction. The common reference architecture is associated to a set of *Common Templates* in charge of orchestrating the concrete architecture composition. Such templates know the specific points where contributions can be made, *i.e.* they mark out extension points. Concrete contributions determined by the corresponding configured quality variants are inserted into these extension points. Thus, common templates delegate the code declaration in an extension point to one or more *Contributors*, which are concrete Xtend classes able to return source code fragments, call other standalone extension templates, or, in turn, mark out their own extension points and delegate onto other contributors the responsibility of returning required source code.

Figure 9. Delegation strategy



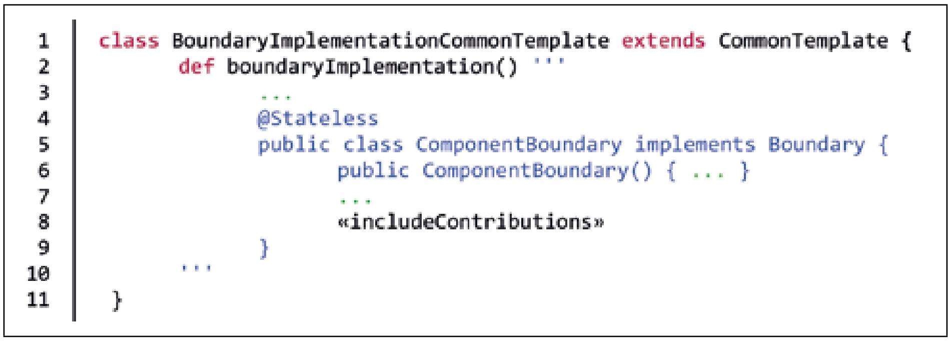
Source: authors' own elaboration

We have developed a library as tool support for describing and weaving required contribution compositions. The library includes facilities for defining extension points in templates and contributions, and registering contributors. It also includes an engine for weaving code fragments returned by contributors. We have implemented the library as a set of Eclipse plug-ins. Currently, we generate JEE7 components under the EJB 3.2 specification. The generation of SCA composites and OSGi bundles is part of our roadmap. The specification, design, and derivation of quality-concerned enterprise application is part of our recent (unpublished) work available in [26].

Figure 10 shows a simple *Common Template* for generating the Boundary element of a component with an Entity-Control-Boundary microarchitecture as proposed by Bien in [32]. Line 1 declares the template as a *CommonTemplate*. Line 8 shows an extension point marking the space for including contributions from the set of registered

Contributors. As illustrated in Figure 11, *Contributors* registered to a specific extension point will be asked to perform their contribution (see Line 3). Figure 12 defines a *Contributor* that provides a reference to a Fast Lane Reader pattern implementation (see Lines 3 through 8) for retrieving all the occurrences of an entity.

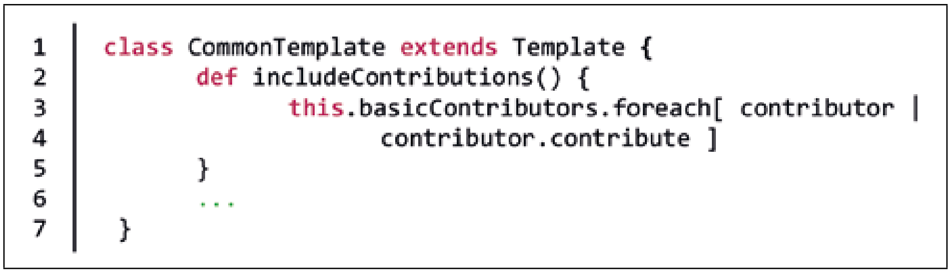
Figure 10. Common template example for deriving an EJB component skeleton



```
class BoundaryImplementationCommonTemplate extends CommonTemplate {
      def boundaryImplementation() '''
            ...
            @Stateless
            public class ComponentBoundary implements Boundary {
                  public ComponentBoundary() { ... }
                  ...
                  «includeContributions»
            }
      '''
}
```

Source: authors' own elaboration

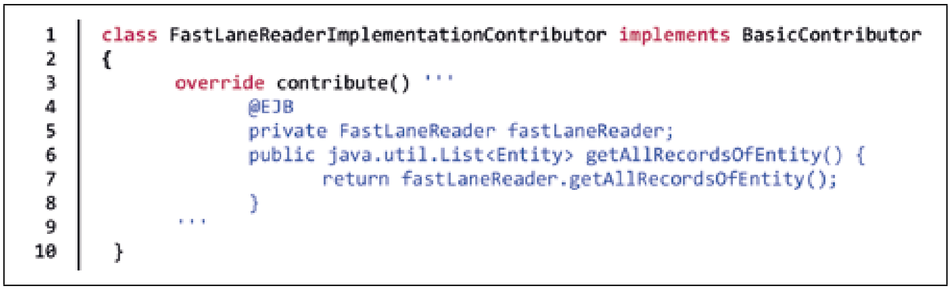
Figure 11.. Including contributions from contributors



```
1  class CommonTemplate extends Template {
2        def includeContributions() {
3              this.basicContributors.foreach[ contributor |
4                    contributor.contribute ]
5        }
6        ...
7  }
```

Source: authors' own elaboration

Figure 12. Example of a contribution to an EJB component



```
1  class FastLaneReaderImplementationContributor implements BasicContributor
2  {
3        override contribute() '''
4              @EJB
5              private FastLaneReader fastLaneReader;
6              public java.util.List<Entity> getAllRecordsOfEntity() {
7                    return fastLaneReader.getAllRecordsOfEntity();
8              }
9        '''
10 }
```

Source: authors' own elaboration

Conclusions and Future Work

The over-dependence of companies on their software applications forces an uninterrupted satisfaction of agreed software quality. The high human intervention needed along with the dynamic nature of enterprise applications requires, however, a more cost-effective approach. The use of software self-adaptation for activities previously done by software maintenance teams is seen as a promising alternative. Nonetheless, a proper framework and tooling are necessary.

In this paper we presented SHIFT, a framework for the generation and management of self-adaptive enterprise applications. SHIFT has a high-level architecture based on the DYNAMICO [3] reference model. A key point that lets our framework go much further than other approaches is the introduction of the two layers, *Autonomic Infrastructure* and *Automated Derivation*, that cover automation and quality awareness across the life cycle of enterprise applications. An important contribution of our work is the proposed reference specification and architectural design for implementing self-adaptation support as an autonomic infrastructure, and it is from which we base the framework's *Autonomic Infrastructure* layer. Our framework, through PASCANI, provides a rich DSL for defining *Sensors* and *Monitors* that abstract metrics and measurement implementations. The introduced *Planner* element and its associated PISCIS model, built on the principles of constraint satisfaction, provides a means to automated reasoning able to find the best configuration of components necessary to preserve SLAs. The *Automated Derivation* layer, as we have illustrated, is SHIFT's basis for supporting changing functional and quality requirements. It offers support for the assisted derivation of enterprise application components and their associated monitoring infrastructures. A more detailed qualitative contrast of the SHIFT framework against other approaches can be found in Section 1.3.

Some conceptual constraints are still present that need to be addressed. For instance, measuring quality attributes is an open research field since many of them are particularly difficult to measure (*e.g.*, the security quality attribute). Additionally, the elements composing the *Autonomic Infrastructure* layer are inherently tied to the managed system at different extents, particularly the *Analyzer*, the *Planner*, and the *Executor* elements are closely related to it. We are currently focused on the performance quality attribute and, thus, automated measurement support is bound to the provided performance monitor probes; any other measurement will require the manual development of the necessary monitor probes. For the *Planner* element with the use of the principles of constraint satisfaction we have detached the concerns related to the managed system into a CSP representation derived and stored in a repository managed by the *Knowledge Manager* element. The composition of software design patterns in reference architectures still requires the intervention of a software architect since complex interactions may arise.

As future work, we will be working on refining the reference specification and architectural design as well as on the design of the framework and completing the concrete implementations for all the elements presented, including the complete autonomic infrastructure and its interoperability with Java middleware. In addition, we will propose and perform the validation of our SHIFT framework and the implemented tool support with an industrial case study.

Acknowledgments

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.

References

[1] Aberdeen Group, "Downtime and data loss: How much can you afford?," 2013. [Online]. Available: http://www.aberdeen.com/research/8623/ai-downtime-disaster-recovery/content.aspx.

[2] H. Arboleda, A. Paz, M. Jiménez, and G. Tamura, "A framework for the generation and management of self-adaptive enterprise applications," in *10th Computing Colombian Conference (10CCC)*, 2015.

[3] 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," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7475 LNCS, pp. 265–293, 2013.

[4] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, "Kinesthetics eXtreme: An external infrastructure for monitoring distributed legacy systems," in *Proc. Autonomic Computing Workshop*, 2003, pp. 22–30.

[5] D. Ameller and X. Franch, "Service Level Agreement Monitor (SALMon)," in *Proc. the Seventh Int. Conf. Composition-Based Software Systems (ICCBSS'08)*, 2008, pp. 224–227.

[6] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola, "MOSES: A framework for QoS Driven runtime adaptation of service-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1138–1159, 2012.

[7] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the Effectiveness of the Rainbow Self-adaptive System," in *Proc. of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09)*, 2009, pp. 132–141.

[8] IBM, "An architectural blueprint for autonomic computing," 2006. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20 V7.pdf.

[9] L. Castañeda and G. Tamura, "A reference architecture for component-based self-adaptive software systems," Master's Thesis, Universidad Icesi, Colombia, 2012.

[10] K. Czarnecki, "Overview of generative software development," in *Unconventional Programming Paradigms*, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Berlin: Springer, 2005, pp. 326–341.

[11] C. Seidl, S. Schuster, and I. Schaefer, "Generative software product line development using variability-aware design patterns," in *Proc. of the 2015 ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences*, 2015, pp. 151–160.

[12] A. Paz and H. Arboleda, "Towards a framework for deriving platform-independent model-driven software product lines," *Ing. e Investig.*, vol. 33, no. 2, pp. 70–75, 2013.

[13] B. Rumpe, M. Schindler, S. Völkel, and I. Weisemöller, "Generative software development," in *Proc. the 32nd Int. Conf. on Soft. Eng. (ICSE 2010)*, 2010, pp. 473–474.

[14] M. Colombo, E. Di Nitto, and M. Mauri, "Scene: A service composition execution environment supporting dynamic changes disciplined through rules," in *Proc. of the ICSOC'06*, Springer, 2006, pp. 191–202.

[15] L. Baresi and S. Guinea, "Self-supervising bpel processes," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 247–263, 2011.

[16] 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*. Berlin: Springer, 2007.

[17] P. Cedillo, J. Gonzalez-Huerta, S. Abrahao, and E. Insfran, "Towards monitoring cloud services using models@run.time," in *Proc. of the 9th Workshop on Models@run.time*, 2014, pp. 31–40.

[18] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS management and optimization in service-based systems," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 387–409, 2011.

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

[20] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair, "An aspect-oriented and model-driven approach for managing dynamic variability," in *Model Driven Engineering Languages and Systems*. Berlin: Springer, 2008, pp. 782–796.

[21] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," *J. Syst. Softw.*, vol. 91, no. 1, pp. 24–47, 2014.

[22] R. Heinrich, E. Schmieders, R. Jung, K. Rostami, A. Metzger, W. Hasselbring, R. Reussner, and K. Pohl, "Integrating Run-Time Observations and Design Component Models for Cloud System Analysis," in *Proc. of the 9th Workshop on Models@run.time*, 2014, pp. 41–46.

[23] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring, "An adaptation framework enabling resource-efficient operation of software systems," in *Proc. of the Warm Up Workshop for ACM/IEEE ICSE 2010*, 2009, pp. 41–44.

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

[25] H. Arboleda, R. Casallas, J.-C. Royer, and J.-C. Arboleda, Hugo and Casallas, Rubby and Royer, "Dealing with fine-grained configurations in model-driven SPLs," in *Proc. of the 13th Int. Soft. Product Line Conf. (SPLC'09)*, 2009, pp. 1–10.

[26] D. Durán and H. Arboleda, "Quality-driven software product lines," Master's Thesis, Icesi University, Colombia, 2014.

[27] H. Arboleda, J. F. Diaz, V. Vargas, and J.-C. Royer, "Automated reasoning for derivation of model-driven SPLs," in *Proc. of the 14th Int. Soft. Product Line Conf. (SPLC 2010), Volume 2, 2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010)*, 2010, pp. 181–188.

[28] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A component-based middleware platform for reconfigurable service-oriented architectures," *Softw. Pract. Exp.*, vol. 42, no. 5, pp. 559–583, 2012.

[29] A. Paz and H. Arboleda, "A model to guide dynamic adaptation planning in self-adaptive systems," *Electron. Notes Theor. Comput. Sci.*, vol. 321, pp. 67-88, 2016.

[30] ISO/IEC, "ISO/IEC 25000:2014 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE," Information technology standard, ISO/IEC, 2014.

[31] D. Garlan and M. Shaw, "An introduction to software architecture," *Knowl. Creat. Diffus. Util.*, vol. 1, no. January, pp. 1–40, 1994.

[32] A. Bien, *Real World Java EE Patterns-Rethinking Best Practices*, 2 Ed. lulu.com, 2012.