Article

DOI: 10.1111/j.1468-0394.2012.00620.x

Knowledge representation of the software architecture design process based on situation calculus

María Luciana Roldán, Silvio Gonnet and Horacio Leone
Instituto de Desarrollo y Diseño, Universidad Tecnológica Nacional - CONICET, Avellaneda 3657, 3000 Santa Fe, Argentina
Email: lroldan@santafe-conicet.gov.ar, sgonnet@santafe-conicet.gov.ar, hleone@santafe-conicet.gov.ar

Abstract: *Software architecture design is an interactive, complex, decision-making process. Such a design process involves the exploration, evaluation, and composition of design alternatives. Increasingly, new computer-aided tools are available to help designers in these complex activities. However, these tools do not know how design is actually done, in other words, by means of which design activities the final artefact was obtained. In fact, the architectural design knowledge exclusively rests in the mind of designers, and there is an urgent need to move it, as much as possible, to a computer-supported environment that enables the capture of this type of knowledge. This contribution addresses this need by introducing a model for capturing how products under development are generated and transformed along the software architecture design process. The proposed model follows an operational perspective, where architectural design decisions are modelled by means of sequences of operations that are applied on the design products. Situation calculus is used to formally express the existence of an object in a given state of a design process. In addition, this formalism allows us expressing without ambiguities when an operation can be performed in a specific state of the design process.*

Keywords: design process representation, software architectures, situation calculus

1. Introduction

Software architecture design process (SADP) includes many tasks such as exploration, evaluation, and composition of design alternatives, which make it a difficult, complex process (Jansen & Bosch, 2005). In the last few years, numerous design methods, modelling languages, and computer-aided tools (Garlan *et al.*, 2000; Bachmann *et al.*, 2003; Hofmeister *et al.*, 2007; Medvidovic *et al.*, 2007; Clements *et al.*, 2010) have been proposed to face these complex activities. These strategies, languages, and tools are basically focused on assisting a designer in describing an architecture model, maintaining intellectual control over its design, and on generating a software architecture model to satisfy a set of requirements. However, such tools keep only part of the knowledge applied and generated during the architecture design process. Most of the knowledge still rests in the minds of experienced designers, it is not a product of the process and consequently, it is lost with time. Therefore, there is a need for tools able to capture and retrieve the software architecture design process. These tools should be capable of recording each designer's decision and its impact on the architectural model, thus making possible both the tracking and tracing of the SADP and the analysis of its underlying rationale. In this way, the expert's knowledge could be captured, thus providing the foundations for learning and training activities and for future reuse.

To set the basis for the development of tools with such characteristics, it is needed a model that provides the elements for capturing the evolution of the products that arise from SADP and the design decisions that generated them.

This contribution adopts the situation calculus (McCarthy, 1963; Reiter, 2001) for modelling that evolution. Such a formalism is useful to make explicit the states of SADP and how they were obtained. Based on this proposal, the authors have developed a tool for capturing and tracing engineering design processes (Roldán *et al.*, 2010). Although such a tool was intended to supporting generic engineering design processes, its flexibility enables being applied to software architectures domain.

The paper is organized as follows. In Section 2, the problem and its scope are defined. Then, in Section 3, the core model of this proposal is presented. After that, in Section 4, an extension is proposed were the particular operations of SADP are included. The proposed model is validated in Section 5 by designing a monitoring system for an industrial process. In Section 6, related work is compared with our approach and some limitations of the approach are discussed. Finally, Section 7 summarizes the main issues presented.

2. Problem definition

Software architecture design is one of the most complex and critic activities in the software life cycle, involving the consideration and resolution of many creative problem-specific tasks, which require a lot of decision making. A software architecture is intended for representing and communicating the system structure and behaviour of a system to all the stakeholders. In addition, software architecture captures early design decisions of a system. The architecture description, along with the architectural design decisions, constitutes architectural knowledge that should be represented (Kruchten *et al.*, 2006), thus captured and documented in some way. Different technical tools have emerged to support such architectural knowledge, which make emphasis in some of the activities of SADP. Particularly, some notations have been proposed to describe architectural models, and then, based on these notations numerous computational environments emerged to support architectural visual modelling (Medvidovic *et al.*, 2007). Other recent tools

(Tang *et al.*, 2010; Weinreich & Buchgeher, 2011) have been proposed to support architectural knowledge management in some activities of SADP life cycle, mainly architectural analysis, synthesis, and evaluation. By using different approaches, these tools capture design decisions and their relationships with requirements and architecture design. However, they share a common problem: users find out that capturing or documenting design decisions requires an additional effort, and in consequence, they ignore them. If design decisions are not documented, they remain tacit knowledge, which is easily lost (Jansen & Bosch, 2005), thus causing knowledge vaporization that is the key problem of design erosion. Eventually, vaporization of architectural knowledge, results in reduced stakeholder communication, increased system maintenance costs, and limited reusability of architecturally significant entities.

The approach proposed in this work addresses the depicted problem. It is focused in capturing not only the final architecture model, but also all the design decisions that generated the intermediate products during the design process. In spite of the value of the design decision rationale, we know that it is practically infeasible to obtain from architects the textual documentation of their design decisions. Therefore, our proposal considers that a design decision is materialized, also documented, by the execution of sequences of design operations on a predecessor version of the architectural model. The design operations have been defined as part of the design domain model and enclose a specific piece of knowledge of it. Such design operations can be as basic as adding or deleting a component, or as complex as applying an architectural pattern, but always have a concrete impact on the software architecture model, causing the arising of a new model version.

We do not separate the outcome of the software architecture design process from the process itself. Basically, this work considers the outcome of the design process as a 'rational reconstruction' of that process (Dingsøyr & van Vliet, 2009). Therefore, we propose a model to represent a SADP by explicitly capturing the architectural knowledge given by: (1) the design products generated and their evolution; and (2) the design decisions made during the process and materialized by the application of sequences of domain operations. The SADP is executed using product types and operations that were defined in the design domain model.

3. Representing how the SADP is performed

The proposed scheme considers the SADP as a sequence of activities that operate on the products of the design process and materialize the architect decisions. Therefore, these products, which are called *design objects*, are the result of decisions made by the designer and constitute an abstraction of what is eventually desired to be realized in the real world (Taylor & van der Hoek, 2007). Typical design objects are architectures of the software being designed, specifications and scenarios to be met, and components and connectors that form an architectural view. Naturally, these objects evolve as the SADP takes place, giving rise to several *versions*. Consequently, this contribution focuses on representing the various states (versions) of the design objects along their life cycle, and how these states were derived.

3.1. Representing design objects

As it was previously introduced, a *design object* represents any entity that can evolve during a SADP. It is represented at two levels, the *repository* and the *versions level*. The *repository* level keeps a unique entity for each *design object* that has been created and/or modified during a SADP. This object is regarded as a *versionable object* (o). On the other hand, the *versions level* keeps the different versions of each *design object*. These are called *object versions* (v).

The concepts of the proposed scheme are formally defined by introducing appropriated first-order logic predicates and writing axioms that capture their semantics. To represent *design objects*, unary predicates *versionableObject* and *objectVersion* are proposed. The relationship between a *versionable object* and one of its *object versions* is represented by the *version*(v, o) predicate, which expresses that the v object version is a version of the o versionable object. Below, axioms (1)–(3) define the semantic of such a relationship.

$$(\forall o)\, versionableObject(o) \Rightarrow (\exists v : ObjectVersion)\, version(v, o) \tag{1}$$

$$(\forall v)\, objectVersion(v) \Rightarrow (\exists\, o : VersionableObject)\, version(v, o) \tag{2}$$

$$(\forall v : ObjectVersion)(\exists\, o_1, o_2 : VersionableObject)\; version(v, o_1) \wedge version(v, o_2) \Rightarrow (o_1 = o_2) \tag{3}$$

It should be also noted, that an object version is always linked to a versionable object, from which is version, as it is represented in (2). In addition, this versionable object is unique, which is expressed in (3).

At the *repository* level, also the associations among the different versionable objects are maintained. These associations (a_k) correspond to the rules that allow associating objects to develop syntactically valid models. A ternary predicate *association*(a_k, o_i, o_j) is included to express such repository associations between o_i and o_j versionable objects.

3.2. The operational perspective for model evolution

This section focuses on describing how the design process evolution is represented. The situation calculus (McCarthy, 1963; Reiter, 2001) is adopted for modelling such a process. Situation calculus is a second-order language, sometimes described as a first-order language with some second-order features, for representing dynamically changing worlds (Scherl & Levesque, 2003). All changes to the world are the result of named *actions*. Possible world histories, which are sequences of actions, are represented by first-order terms called *situations*. By convention, S_0 is used to denote the initial situation, which is comprised of the empty sequence of actions. There is a binary symbol do; *do*(*action*, s), which denotes the successor situation to s resulting from performing an *action*. Relations whose truth values vary from situation to situation, called *fluents*, are denoted by expressions with a predicate symbol that take a situation term as the last argument. We find the ontology provided by situation calculus for modelling dynamic domains very useful for modelling the software architecture design process.

We abstract the SADP as a history made up of discrete situations. In this way, at a given stage during the execution of a software architecture design project, the set of *versions* assumed by the relevant *design objects* supplies a snapshot description of the state of the design process at that point. Each state of the design process is called *model version*. Then, a new model version m_n is generated when a design decision *dd* is made. A design decision *dd* is materialized by a sequence of operations ϕ. The new model version m_n is the result of applying such a sequence ϕ to the elements of a previous model version m_p. In consequence, using the situation calculus ontology, we assimilate *situations* to the different *model versions* generated during a SADP, *actions* to *operations*, and *complex actions* to *sequence of operations*. Consequently, a new model version m_n is achieved by performing the following evaluation: $apply(\phi, m_p) = m_n$.

The *apply* function is defined in (4), where *SequenceOfOperations* is the set of all possible operation sequences ϕ and *ModelVersion* is the set of all possible *model versions m*. In (5), a sequence of operations is defined in a recursive way, where λ is an empty sequence of operations, *op* is an operation, and • is the function that create a sequence of operations from an operation and another sequence of operations.

$$apply : SequenceOfOperations \times ModelVersion \rightarrow ModelVersion. \tag{4}$$

$$\phi = \begin{cases} \lambda \\ op \bullet \phi \end{cases} \tag{5}$$

Then, in (6) the *apply* function is defined by induction. *do*: $A \times S \rightarrow S$ is a binary function defined by situation calculus (Reiter, 2001; Scherl & Levesque, 2003), where A is the set of all possible actions and S is the set of all possible situations. This can be expressed as follows:

$$\begin{aligned}&ModelVersion \subseteq S\\ &Operation \subseteq A\\ &do(\lambda, m) = m\\ &do(op, m) = m_i, m \neq m_i\\ &apply(op\,\phi, m) = apply(\phi, do(op, m)).\end{aligned} \tag{6}$$

In order to represent the transformation of *model versions* during SADP, a set of primitive operations is proposed. The operations are *add*, *delete*, and *modify*, which can be combined to form a sequence of operations ϕ to be applied on a given model version. By using the *add(v)* operation, an *object version* that did not exist in a previous *model version* can be incorporated into a successor one. Conversely, the *delete(v)* operation eliminates an *object version* that existed in the previous *model version*. Also, if a *design object* has a version v_p, the *modify*(v_p, v_s) operation creates a new *version* v_s of the existing *design object*, where v_s is a successor *version* of v_p. Thus, an *object version v* belongs to a *model version* that arises after applying the sequence of operations ϕ to *model version m*, if and only if one of the following conditions is met:

(1) *v* is added when the new *model version* is created by means of an *add* or *modify* operation (7); or

(2) *v* already belonged to the previous *model version m* (*belong*(*v, m*)) and it is not deleted when ϕ is applied to it (8).

$$(\forall \phi, v, m)(add(v) \in \phi \vee (\exists v_p) modify(v_p, v) \in \phi) \Rightarrow added(v, apply(\phi, m)). \tag{7}$$

$$(\forall \phi, v, m)(delete(v) \in \phi \vee (\exists v_s) modify(v, v_s) \in \phi) \Rightarrow deleted(v, apply(\phi, m)). \tag{8}$$

From these definitions, and using the format of successor state axioms proposed by Reiter (2001), a formal specification of the cases in which an *object version* belongs to a *model version* is specified in (9). It states that an *object version v* belongs to the version model that results from applying the sequence of operations ϕ to model version *m* if an only if *v* already was present in *m* and it is not deleted when ϕ is applied to *m*, or *v* is added when the new model version is created from applying sequence of operations ϕ on *m*. In (9), the predicate *belong(v, m)* is true when *object version v* belongs to model version *m*.

$$\begin{aligned}&(\forall \phi : SequenceOfOperations, v{:}ObjectVersion,\\ &m : ModelVersion) belong(v, apply(\phi, m))\\ &\quad \Leftrightarrow (belong(v, m) \vee added(v, apply(\phi, m))) \wedge\\ &\quad (\neg deleted(v, apply(\phi, m))).\end{aligned} \tag{9}$$

From this expression, the *object versions* that belong to a certain *model version* can be determined. Then, it is possible to reconstruct a *model version* m_{i+1} by applying all the sequences of operations from the initial *model version* m_0.

Each sequence of *operations* applied to a *model version* incorporates the necessary information to trace a model evolution. This information is represented by *modelHistory* predicate between the model versions and the applied sequence of operations (10).

$$\begin{aligned}&(\forall \phi : SequenceOfOperations, m_p,\\ &m_s : ModelVersion)\\ &apply(\phi, m_p) = m_s \Leftrightarrow modelHistory(\phi, m_p, m_s)\end{aligned} \tag{10}$$

It is worth mentioning that the proposed model makes possible representing the design objects evolution since a *versionable object* can be expressed by one or more *object versions*, but they belong to different *model versions* (11).

$$\begin{aligned}&(\forall o : VersionableObject, m{:}ModelVersion)\\ &(\exists v_i : ObjectVersion)\\ &version(v_i, o) \wedge belong(v_i, m)\\ &\quad \Rightarrow [(\forall v_k : ObjectVersion)\ version(v_k, o) \wedge\\ &\quad belong(v_k, m) \Rightarrow (v_i = v_k)]\end{aligned} \tag{11}$$

The relationships existing among *object versions* in a model version can be also found out. For that, it should be noted that in this proposal, *object versions* belonging to a *model version* are not explicitly associated to other versions belonging to the same model version. On the contrary, if an association between two design objects exists, such a link is represented at the repository level, without a situation term. Consequently, the relationship existing between two object versions must be inferred from the relationship established between the versionable objects that have been versioned by them. This fact is represented in (12), in which an association a_k is inferred between two object versions v_1 and v_2 belonging to the same model version m (*inferredAssociation*(a_k, v_1, v_2, m)), if and only if there exists an association a_k between the two versionable objects o_1 and o_2 (*association*(a_k, o_1, o_2)), of which v_1 and v_2 are versions, respectively (*version*(v_1, o_1) and *version*(v_2, o_2)).

$$\begin{aligned}&(\forall v_1, v_2, m, a_k) inferredAssociation(a_k, v_1, v_2, m)\\ &\quad \Leftrightarrow (\exists o_1, o_2) belong(v_1, m) \wedge belong(v_2, m) \wedge\\ &\quad version(v_1, o_1) \wedge version(v_2, o_2) \wedge\\ &\quad association(a_k, o_1, o_2). \end{aligned} \tag{12}$$

Situation calculus provides precondition axioms to specify the conditions under which an action can be performed (Reiter, 2001). Thus, it is necessary to extend them to specify the preconditions to apply a sequence of operations ϕ to a given model version m, fact that is expressed by the $poss_{so}(\phi, m)$ predicate in (13).

$$\begin{aligned}&(\forall \phi, m) poss_{so}(\phi, m)\\ &\quad \Leftrightarrow (\forall op_i, op_i \in \phi) poss_o(op_i, \phi, m). \end{aligned} \tag{13}$$

A sequence of operations ϕ may be applied to a model version m if each operation op_i belonging to ϕ can be applied to m, as well as op_i can be applied in all the situations generated by applying the $i-1$ previous operations in the sequence, where op_i is the ith operation belonging to ϕ. This fact is defined by the $poss_o(op_i, \phi, m)$ predicate (14), which was previously introduced in (13). Therefore, a sequence ϕ can be applied to m if each operation of ϕ is applicable to m and does not violate the preconditions of the other operations belonging to ϕ.

$$\begin{aligned}&(\forall \phi, m)(\forall op_i, op_i \in \phi, \exists \phi_1, \phi_2, \phi = \phi_1 \bullet op_i \bullet \phi_2)\\ &\quad poss_o(op_i, \phi, m)\\ &\qquad \Leftrightarrow (\forall m_i, m \leq m_i < apply(\phi_1 \bullet op_i, m))\\ &\qquad poss(op_i, m_i). \end{aligned} \tag{14}$$

The *poss*(op, m) predicate expresses that an operation op is applicable to a given model version m. This fact is represented by the following axioms: (1) operation *add*(v) can be applied to model version m if the object version v does not belong to m (15); (2) operation *delete*(v) can be applied to the model version m if the object version v belongs to m (16); (3) operation *modify*(v_i, v_j) can be applied to m if v_i belongs to m and v_j does not belong to it (17).

$$(\forall v, m) poss(add(v), m) \Leftrightarrow \neg belong(v, m). \tag{15}$$

$$(\forall v, m) poss(delete(v), m) \Leftrightarrow belong(v, m). \tag{16}$$

$$\begin{aligned}&(\forall v_i, v_j, m) poss(modify(v_i, v_j), m)\\ &\quad \Leftrightarrow belong(v_i, m) \wedge \neg belong(v_j, m). \end{aligned} \tag{17}$$

3.3. Design process products representation

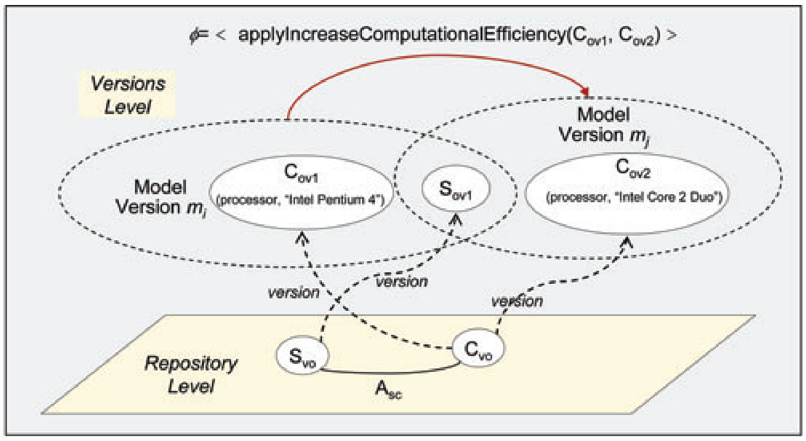
Let us regard a fragment of a hypothetical design process of a software architecture, which is illustrated in Figure 1. In this example, the designer has obtained m_i model version, where an object version (C_{ov1}), that represents a physical component, is part of the deployment view (Clements *et al.*, 2010) of the architecture for a system (S_{ov1}). At this point of the design process, the designer decides to improve the computational capacity of the component in order to satisfy a performance requirement imposed on the system. This decision is translated in the applying of a sequence of operations ϕ on m_i, which encloses only one *applyIncreaseComputationalEfficiency* operation. Such an operation is an extension of *modify* primitive operation (Section 4 address how to define complex operations like *applyIncreaseComputationalEfficiency*) and adds a new object version (C_{ov2}) from an existing object version (C_{ov1}). This fact is represented by the effect axiom of Figure 1(d). As a consequence of this sequence of operations, model version m_j is generated.

This simple model evolution generates new knowledge that is represented with the facts stated in Figure 1. Facts included in Figure 1(b) keep what was true before the execution of ϕ. Then, facts in Figure 1(c) describe the resulting knowledge after obtaining model version m_j. It can be observed that at the repository level, the component and system design objects are represented by versionable objects C_{vo} and S_{vo} respectively, whereas at versions level they assume the versions C_{ov1} and S_{ov1}, respectively. In addition, at the repository level, an association A_{sc} indicates that the component is part of the system under design since an *association* fact that links the two versionable objects C_{vo} and S_{vo} exists. After applying the sequence of operations ϕ to obtain m_j, new facts represent the new generated version (C_{ov2} in this case).

It should be noted that the facts in Figures 1(b) and 1(c) do not have a situation term. Therefore, nothing is said about to which model versions the object versions belong. However, that information can be obtained by means of the successor state axiom *belong* (9). In this example, *belong*(S_{ov1}, m_i) and *belong*(S_{ov1}, m_j) are true since S_{ov1} belonged to m_i and was not deleted after applying ϕ. This can be viewed in Figure 1(a), where S_{ov1} version participates in both m_i and m_j model versions. In addition, it is possible to know that C_{ov1} is part of model version m_i but not of m_j (*belong(*C_{ov1}, m_i*)* is true and *belong(*C_{ov1}, m_j*)* is false) because C_{ov1} is deleted as a consequence of performing *applyIncreaseComputationalEfficiency* operation (see effect axiom Figure 1(d)). Similarly, it can be determined that C_{ov2} does not participate in model version m_i but it is present in m_j due to the effects of that operation.

4. Defining a SADP domain

A model for capturing and tracing SADP must be flexible enough to enable the definition and execution of the several



(a) Graphical representation of versions and repository level

Facts at Versions' Level
objectVersion(S_{ov1}).
version(S_{ov1}, S_{vo}).
objectVersion(C_{ov1}).
version(C_{ov1}, C_{vo}).
propertyValue (C_{ov1}, *processor*, "Intel Pentium 4").

Facts at Repository Level
versionableObject(S_{vo}).
versionableObject(C_{vo}).
association(A_{sc}, S_{vo}, C_{vo}).

(b) Facts expressing partial knowledge before applying sequence of operations ϕ to model version m_i

Facts at Versions' Level
objectVersion(S_{ov1}).
version(S_{ov1}, S_{vo}).
objectVersion(C_{ov1}).
version(C_{ov1}, C_{vo}).
propertyValue (C_{ov1}, *Processor*, "Intel Pentium 4").
objectVersion(C_{ov2}).
version(C_{ov2}, C_{vo}).
propertyValue (C_{ov2}, *Processor*, "Intel Core 2 Duo").

Facts at Repository' Level
versionableObject(S_{vo}).
versionableObject(C_{vo}).
association(A_{sc}, S_{vo}, C_{vo}).

(c) Facts expressing partial knowledge after applying sequence of operations ϕ to model version m_i

$$(\forall\ \phi, v_p, v_s, m)\ applyIncreaseComputationalEfficiency(v_p, v_s) \in \phi \Rightarrow$$
$$added(v_s, apply(\phi, m)) \wedge deleted(v_p, apply(\phi, m))\ .$$

(d) Effect axiom of *applyIncreaseComputationalEfficiency* operation

Figure 1: $m_j = apply(\{applyIncreaseComputationalEfficiency(C_{ov1}, C_{ov2})\}, m_i)$.

operations involved in the design decisions made during a design process. In addition, this contribution not only provides the capabilities for representing each performed operation but also its resulting products. To that end, the model must include the elements that make possible defining the types of design objects to be captured and the possible relationship types than can exist among them. The model must also support the definition of the several design operations applicable to the types of products that can be managed in a domain, which materialize the design decisions.

As it was introduced in the previous section, in this paper, situation calculus is used for specifying the preconditions and the effects of the design operations and for capturing and reconstructing how a design process took place. Therefore, first-order logic is also used for representing additional knowledge about SADP domains.

4.1. *Defining design object types for SADP*

Before developing a SADP project, it is necessary to identify which types of design objects should be captured. Possible design objects must be classified according to the different types managed in the domain (e.g., component, port, system, quality requirement, etc.).

In order to identify them for defining a SADP domain, it is important to consider the numerous architecture description languages (ADLs) that have been proposed in the literature, specially those adopted by industry practitioners and those familiar to the project's architects. ADLs embody a set of concepts to be modelled (from which design object types can be derived) and a particular approach to the specification and evolution of an architecture (Medvidovic *et al.*, 2002; Medvidovic *et al.*, 2007). Besides, we consider relevant to include also some concepts relative to the architectural design method preferred by the designer.

To this end, the proposed model is extended defining a possible SADP domain that includes different design object types derived from concepts taken from ADD method (Bass *et al.*, 2003) and the architectural description language ACME (Garlan *et al.*, 2000).

The first set of design object types for the SADP domain is identified from concepts present in the ADD method proposed by SEI (Bass *et al.*, 2003). ADD method is based on a decomposition process, where architectural patterns are chosen at each iteration to fulfil a set of requirements (*functional* and *quality requirements*). From the instantiation of an architectural pattern, several elements (like *components* and *connectors*) are included in the architectural model. The inputs to ADD are *quality requirements*, which are expressed as a set of system specific *quality scenarios*, and *functional requirements*, which are translated into a set of *responsibilities* that are assigned to *components*. Quality scenarios and responsibilities can be delegated to other components when the original component is refined. When a method iteration is finished, the designer verifies how approached is the architecture to each proposed scenario and set an *assessment*.

At the end of an ADD iteration, a partial architecture model is obtained. The IEEE recommended practice for architectural description of software-intensive systems (IEEE, 2000) recommends describing an architectural model by using different *viewpoints*. In the scope of this contribution, Component&Connector views are considered (Clements *et al.*, 2010). However, this proposal supports the definition of several design object types relative to any view. Then, to represent Component&Connector views, the ACME architectural description language is chosen (Garlan *et al.*, 2000). ACME defines a *component* as a computational element and data store of a system. A *component* may have multiple interfaces, each of which is termed *port*. The *connectors* represent interactions among *components* and have interfaces that are defined by a pair of *roles*. A *system* in ACME comprises *components* and *connectors*, by setting *attachments* between *roles* and *ports*. Moreover, ACME proposes elements to document extra-structural properties of a system's architecture, which are named *characteristics.*

The concepts (design object types) introduced above can be formally defined in a SADP domain by introducing appropriate first-order logic constructs and writing axioms to capture their properties and possible relationships with other concepts. We introduce the unary predicate *designObjectType.* This predicate has one term, which is the name of the concept that constitutes the design object type. Using such a predicate, the concepts defined above are included in the SADP domain by means of the facts in (18).

$$\begin{aligned}
&designObjectType(Requirement).\\
&designObjectType(QualityRequirement).\\
&designObjectType(FunctionalRequirement).\\
&designObjectType(QualityScenario).\\
&designObjectType(Assessment).\\
&designObjectType(System).\\
&designObjectType(Component).\\
&designObjectType(Connector).\\
&designObjectType(Responsibility).\\
&designObjectType(Port).\\
&designObjectType(Role).\\
&designObjectType(Characteristic). \qquad (18)
\end{aligned}$$

The SADP domain also states which are the types of the relationships that rule the valid ways in which design objects can be associated. This is expressed by means of *domainRelationship* predicate. *domainRelationship* predicate has three terms, where the first one represents the name of a relationship type and the two others are the types of design objects that are involved in the relationship type. In (19), some relevant domain relationships are included in the SADP domain.

$$\begin{aligned}
&domainRelationship(RSystQReq, System,\\
&\quad QualityRequirement).\\
&domainRelationship(RSystFReq, System,\\
&\quad FunctionalRequirement).\\
&domainRelationship(RSystComp, System,\\
&\quad Component).\\
&domainRelationship(RSystConn, System,\\
&\quad Connector).\\
&domainRelationship(RCompResp, Component,\\
&\quad Responsibility).\\
&domainRelationship(RCompPort, Component,\\
&\quad Port).\\
&domainRelationship(RConnRole, Connector,\\
&\quad Role). \qquad (19)
\end{aligned}$$

Design object types have a set of versionable characteristics denominated *properties*. This means that object versions will have particular values for the properties defined by its design object type. Each property (*p*) of a design object type (*dot*) is explicitly stated in the SADP domain by means of a binary predicate *property*(*dot*, *p*). In (20), the facts that describe the properties of *Component* and *Connector* design object types are listed. Particularly, in this domain, besides the property *Name* for *Component*, two additional properties were defined: *MaxCantOfTables* and *IndexingType*, which are characteristics of databases components. On the other hand, a *Connector* type defines properties for communicating two components in an architectural model, like a name (given by *Name* property in (20)), the type of connection indicating whether it is physical or logical (given by *Type* property in (20)), and the adopted communication protocol (given by *Protocol* property).

$$\begin{aligned}
&property(Component, Name).\\
&property(Component, MaxCantOfTables).\\
&property(Component, IndexingType).\\
&property(Connector, Name).\\
&property(Connector, Type).\\
&property(Connector, Protocol). \qquad (20)
\end{aligned}$$

When design object types share common properties, it is worthwhile to generalize these concepts by defining an abstract design object type, or, with a different perspective, to specialize a concept in a set of new concepts that inherit its properties. A binary predicate *isSubtypeOf*(*dot*1, *dot*2) allows expressing that design object type *dot*1 is a subtype of design object type *dot*2. Examples are given in (21) by specializing the possible requirements of a system in *FunctionalRequirement* and *QualityRequirement*.

$$\begin{aligned}
&isSubtypeOf(Requirement,\\
&\quad QualityRequirement).
\end{aligned}$$

$$isSubtypeOf\,(Requirement,\\ FunctionalRequirement). \qquad (21)$$

Sometimes, an expert concerned in the definition of a SADP domain, might decide to include a new design object type to represent the associations rules that exist between two types. This means the transformation of a domain relationship in a design object type, which is called *reification*. By doing that, a domain relationship becomes a design object type. An example is given by *RCompResp* domain relationship, which can be transformed in a design object type called *RHasResp* that links *Component* and *Responsibility* design object types by means of *RHRComponent* and *RHRResponsibility* domain relationships. These facts are represented by *designObjectType(RHasResp)*, *domainRelationship(RHRComponent, RHasResp, Component)*, and *domain Relationship(RHRResponsibility, RHasResp, Responsibility))*. In addition, properties like *AssignedBy* and *Visibility* could be assigned to *RHasResp* design object type to represent who is the responsible actor that assigns that responsibility to a component and the visibility of the responsibility (public or private).

Binary predicates *isDesignObjectType* and *isAssociationType* complete the set of predicates for representing the types of products and associations that arise during a design process. Continuing with the example presented in Section 3.3, a *Domain level* is added (Figure 2). A set of facts represents the knowledge about the type of the versionable objects C_{ov} and S_{ov}, and the kind of domain relationship of the association A_{sc} that links them (Figure 2).

4.2. Defining operations suitable for SADP

The primitive operations *add*, *delete*, and *modify* are not enough for capturing and tracing a SADP execution. Although an architectural design decision can be materialized by a complex sequence of these primitives, its very low abstraction level precludes capturing the meaning of the decision made, thus the decision is lost. Consequently, the model previously introduced must be extended in terms of the suitable high-level operations for SADP design domain, which really capture the performed architectural decisions. The proposed operations should be applicable to the design objects included in the domain. Table 1 lists some examples of architectural operations for SADP domain. An extensive catalogue of architectural design operations with their functional specifications can be found in Roldán (2009).

Table 1: *Possible operations for the software architecture design domain*

Basic operations	
addComponent	deleteComponent
addConnector	deleteConnector
addFunctional-Requirement	deleteFunctional-Requirement
addPort	deletePort
addCharacteristic	deleteCharacteristic
addQualityRequirement	deleteQuality-Requirement
addResponsibility	deleteResponsibility
addRole	deleteRole
addScenario	deleteScenario
Special operations	
refineComponent	delegateScenario
refineResponsibility	verifyScenario
delegateResponsibility	
Architectural patterns and tactics	
applyControlLoop	applyInsertIntermediary
applyClientServer	applyIncreaseComputa-tionalEfficiency

Such operations are grouped in categories that range from the most basic to more complex ones, in order to understand the operation abstraction level. *Basic* operations allow creating and deleting basic design objects (like *components* and *connectors*) (some functional specifications are given in Figure 3). *Special* operations are more complex and comprise design activities like object refinement or delegation (some specifications are given in Figure 4). With a higher level of abstraction, *Architectural patterns and tactics* operations are defined. They generate several design objects, which follow a configuration based on the applied architectural style (some specifications are given in Figure 5).

Figure 3 presents functional specifications for two of the basic operations defined in Table 1. Operations are defined in terms of primitive operations such as *add* and non-primitive ones, such as *addPort*. For example, the *addComponent*(s, nc, l_{Resps}, l_{Ports}, l_{props}) operation adds a component called nc as part of a system s. As it can be seen in Figure 3, this operation is defined by a series of commands. First, a version

Facts at Domain Level

designObjectType(*System*)

designObjectType(*Component*)

designObjectType(*PhysicalComponent*)

domainRelationship(*RSistPComp*, *System*, *PhysicalComponent*)

isDesignObjectType(C_{vo}, *PhysicalComponent*)

isDesignObjectType(S_{vo}, *System*)

isAssociationType(A_{sc}, *RSistPComp*)

Figure 2: *Facts expressing partial knowledge about SADP domain and the types of the design objects generated in during a project and their associations.*

```
addComponent(s: System, nc: String,            deleteComponent(c: Component)
lResps: Collection[String], lPorts: Collection[String],   lPorts = get(Port, c)
lprops: Collection[PrimitiveDataType])            for each p in lPorts
  c:= add(nc, Component, lprops)                    deletePort(p)
  for each nr in lResps                           end for
    addResponsibility(c, nr, [ ])                 lResp = get(Responsibility, c)
  end for                                         for each r in lResp
  for each np in lPorts                             deleteResponsibility(r)
    addPort(c, np, [ ])                           end for
  end for                                         delete(c)
  addAssociation(s, c, RSistComp)               end
end
```

Figure 3: *Specification of two basic operations.*

of component *c* is added (*add(c, Component, l_{props})*). After that, a set of responsibilities (whose names are given by the l_{Resp} collection) and ports (whose names are provided by the l_{Ports} collection) are added.

Finally, an association between the new component *c* and an existing system *s* is set. This last operation is performed by an auxiliary function *addAssociation* whose effect is described by the *association* predicate introduced in Section 3 (12). It should be noted that when performing an *add* operation both an object version and a versionable object are included, at *versions* and *repository* levels, respectively. The versionable object maintains the references to all versions it assumes during a SADP, which belong to different model versions (fact represented by means of *version* predicate introduced in Section 3). In addition, *associations* are set among versionable objects (*repository* level).

Although functional specifications of operations (Figure 3) are expressed by using non-primitive operations or other operations already defined in the domain, they can be translated in a sequence of primitive operations *add*, *delete*, and/or *modify*. From this, it is possible to express these more complex operations in terms of *added* and *deleted* predicates introduced in (7) and (8). For illustration purposes, let us consider again the *addComponent*(s, c, l_{Resps}, l_{Ports}, l_{props}) operation. If it is applied to a model version m, then a version of a component c (and its ports and responsibilities), will belong to the successor model version (*apply*(ϕ, m)), as it is defined in (22).

$$(\forall\phi, s, c, l_{Resps}, l_{Ports}, m)addComponent(s, c, l_{Resps}, l_{Ports}, l_{props}) \in \phi \Rightarrow added(c, apply(\phi, m)) \wedge (\exists asc)inferredAssociation(asc, s, c, apply(\phi, m)) \wedge ((\forall r \in l_{Resps})added(r, apply(\phi, m)) \wedge (\exists acr)inferredAssociation(asr, r, c, apply(\phi, m))) \wedge ((\forall p \in l_{Ports})added(p, apply(\phi, m)) \wedge (\exists acp)inferredAssociation(asp, p, c, apply(\phi, m))). \quad (22)$$

Therefore, the definition of new operations allows enlarging the set of operations, without modifying the successor state axiom defined in (9), maintaining the coherency of the proposed model.

The precondition for applying the *addComponent* operation is specified in (23), where the *poss*(op, m) predicate expresses that an operation op is applicable to a given model version m.

$$(\forall s, c, l_{Resps}, l_{Ports}, m)poss(addComponent(s, c, l_{Resp}, l_{Ports}, l_{props}), m) \Leftrightarrow belong(s, m) \wedge \neg belong(c, m) \wedge (\forall r \in l_{Resps})\neg belong(r, m) \wedge (\forall p \in l_{Ports})\neg belong(p, m). \quad (23)$$

Similarly to definition of basic operations, it is possible to define the special ones. Figure 4 presents an example of special operations. The *delegateResponsibility*(r, c_1, c_2) operation enables delegating a responsibility r from a component c_1 to a component c_2. Responsibility delegation generally occurs when a component is removed from a model version as a consequence of a refinement activity, and a new (set of) component (*s*) assumes (assume) the responsibilities of the original component. If r is a responsibility of a component c_1 that belongs to model version m, and *delegateResponsibility*(r, c_1, c_2) operation is included in the sequence of operations applied to m, then, the responsibility r is assigned to c_2 in the resulting model version.

The functional specification of *delegateResponsibility* in Figure 4 presents the commands that compose this operation. The arguments of *delegateResponsibility* are the

```
delegateResponsibility(r: Responsibility, c1: Component, c2: Component)
    hr1 := get(RHasResp, r)
    hr2 := add(null, RHasResp, [])
    addAssociacion(hr2,c2, RHRComponent)
    addAssociacion(hr2, r, RHRResponsibility)
    delete(hr1)
end
```

Figure 4: *Specification of delegateResponsibility special operation.*

```
applyControlLoop(c: Component)
    s := get(System, c)
    diagnos :=addComponent(s, 'Diagnosis', ['Diagnose'],['Diag-P1', 'Diag-P2'],[])
    polMgr := addComponent(s, 'PolicyManager', ['PoliciesDefinition'],['PolMgr-P1', 'PolMgr-P2'],[])
    reactor := addComponent(s, 'Reactor', ['ActionExecution'], [Reac-P1, Reac-P2],[])
    // Symbol • represents a predefined function for concatenating collections
    cDgnPMgr := addConnector(s, 'CDgnPMgr', [CDgnPMgr-R1, CDgnPMgr-R2],
                    get(Port, diagnos(0)) • get(Port, polMgr(0)),[])
    cPMgrReact := addConnector(s, 'CPMgrReact',
                    [CPMgrReact-R1, CPMgrReact-R2],
                    get(Port, polMgr(0)) • get(Port, reactor(0)),[])
    // selection of the resps that are going to be delegated to diagnosis (user interactive)
    $l_{resps}$ = select(get(Responsibility, c))
    for each r in $l_{resps}$
      delegateResponsibility(r, c, diagnos(0))
    end for
    // selection of the resps that are going to be delegated to polMgr (user interactive)
    $l_{resps}$ = select(get(Responsibility, c))
    for each r in $l_{resps}$
      delegateResponsibility(r, c, polMgr(0))
    end for
    // selection of the resps that are going to be delegated to reactor (user interactive)
    $l_{resps}$ = select(get(Responsibility, c))
    for each r in $l_{resps}$
      delegateResponsibility(r, c, reactor(0))
    end for
    // To reconfigure connections between existent connectors and new components
    $l_{cp}$ := get(Port, c)              // refined component's ports
    $l_{ap}$ := get(Port, null)           // a null argument value indicates that all ports in current model version are wanted
    for each p in $l_{cp}$
      np := select($l_{ap}$)              // ask the user a port for reconnecting
      r  := get(Role, p)                  // get the role to which the original component is connected
      addAssociation(np, r, RAttachment)
    end for
    deleteComponent(c)
end
```

Figure 5: *Specification of applyControlLoop operation.*

responsibility to be delegated (r), the component to what r is currently assigned (c_1), and the new component (c_2) to what r is going to be delegated. The delegation works by adding a new *RHasResp* version ($hr2$), which represents a reified association between c_2 and r. Finally the version ($hr1$), which represents that r was assigned to c_1, is deleted.

In the same way as in (22) for *addComponent* operation, *delegateResponsibility* operation can be described in terms of added (7) and deleted (8) predicates. Expression (24) presents *delegateResponsibility* specification using first-order logic.

$$\begin{aligned}&(\forall\phi, r, c_1, c_2, m)\mathit{delegateResponsibiliy}\\ &(r, c_1, c_2) \in \phi\\ &\quad\Rightarrow (\exists hr_2)\mathit{added}(hr_2, \mathit{apply}(\phi, m)) \wedge\\ &\quad(\exists hr_1)\mathit{deleted}(hr_1, \mathit{apply}(\phi, m)) \wedge\\ &\quad(\exists a_{c2-hr2})(\mathit{inferredAssociation}(a_{c2-hr2},\\ &\quad c_2, hr_2, \mathit{apply}(\phi, m)) \wedge (\exists a_{hr2-r})(\mathit{inferred}\\ &\quad\mathit{Association}(a_{hr2-r}, hr_2, r, \mathit{apply}(\phi, m))\end{aligned} \tag{24}$$

In addition, expression (25) indicates the preconditions of *delegateResponsibility*, which have to be satisfied to allow the designer performs the operation. The precondition axiom indicates that components c_1 and c_2, and the responsibility to delegate (r) have to belong to the model version m, and in such a model version, responsibility r is assigned to c_1, by means of hr_1 version. Also, in model version m there is not association among r and c_2 representing that r is a responsibility of c_2.

$$\begin{aligned}&(\forall\phi, c_1, c_2, r, m)\mathit{poss}(\mathit{delegateResponsibiliy}\\ &(c_1, c_2, r), m)\\ &\quad\Leftrightarrow \mathit{belong}(c_1, m) \wedge \mathit{belong}(c_2, m) \wedge\\ &\quad\mathit{belong}(r, m) \wedge (\exists hr_1)\mathit{belong}(hr_1, m) \wedge\\ &\quad(\exists a_{c1-hr1})\mathit{inferredAssociation}(a_{c1-hr1}, c_1,\\ &\quad hr_1, m) \wedge (\exists a_{hr1-r})\mathit{inferredAssociation}\\ &\quad(a_{hr1-r}, hr_1, r, m) \wedge (\neg\exists hr_2)\mathit{belong}(hr_2, m)\\ &\quad\wedge(\neg\exists a_{c2-hr2})\mathit{inferredAssociation}(a_{c2-hr2},\\ &\quad c_2, hr_2, m) \wedge (\neg\exists a_{hr2-r})\mathit{inferredAssociation}\\ &\quad(a_{hr2-r}, hr_2, r, m)\end{aligned} \tag{25}$$

Operations that apply an architecture pattern refine an existent component in a new set of components and connectors that are instantiated from the elements types provided by the pattern. They interact with the designer asking for the responsibilities to be delegated, the way in which connections of the external components should be attached to the new ones. An example of pattern applying operation is defined in Figure 5. In this case, *applyControlLoop* operation is specified.

The *applyControlLoop* pattern comes from the process control paradigm and defines an architecture, where

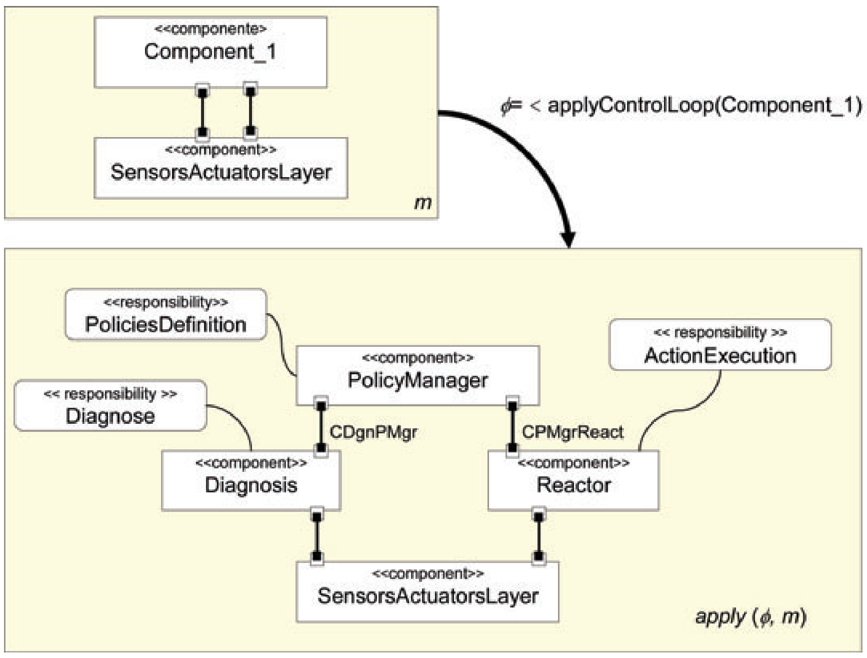


Figure 6: *Applying of applyControlLoop operation.*

monitoring policies are activated in response to different events produced by several sensors (Shaw, 1994). When applying these policies, they may also generate new events or actions. The *applyControlLoop*(c) operation consists on refining a component c. Such a component has several responsibilities and is located on top of a layer (a type of component) named *SensorsActuatorsLayer*. This layer acts as an interface to the various devices that operate on the environment to be controlled. To apply the *control loop* pattern, *applyControlLoop* operation performs a series of *addComponent* adding the main elements of the solution: the *Diagnosis* component is the responsible of monitoring the state of the devices and their action rules; the *Reactor* component encapsulates actions predefined for certain situations requiring immediate attention; and the *PolicyManager* component provides the features for defining the possible actions to respond to the possible situations informed from the sensors. Then, *addConnector* operation is employed to add two connectors to create the pattern configuration. The roles of the connectors are attached to the respective ports of the previously added components, which are provided in an argument collection of *addConnector* command. The next step is to delegate the responsibilities of the original component c to each new component. This is made by means of the *get* auxiliary commands, to obtain a list of responsibilities, and the *select* auxiliary command, to select just the ones to delegate. Then the *loop* auxiliary command (translated in this syntax as *for each – in*) is using for iterating over the list of selected responsibilities (l_{resps}) and delegating each one to a child component. Afterwards, it is necessary to reconfigure the attachments among the components that existed before executing the *applyControlLoop* operation. This is made, firstly, by getting the ports of the c component (*get* command). Also, the rest of the ports belonging to the model version are obtained and assigned to l_{ap}. Finally, by iterating over the collection of ports of the original component, the attached role of a port of the original component is recovered and the designer selects a port from l_{ap} (all ports in model version) to attach to p.

Figure 6 shows the evolution from one model version to other one, after applying a sequence of operations that contains *applyControlLoop*(c) operation. It represents two successive versions of the architectural model by using a Component&Connector view. It is described by means of a notation similar to ACME, in which the relationships between versions are inferred. More operations for architectural patterns applications are detailed in Roldán *et al.* (2006) and Roldán (2009).

Other design operations that apply architectural patterns, called *tactics* by Bass *et al.* (2003), do not modify the structure of the architectural model, but modify some properties of the affected design objects. Examples of these operations are *applyIncreaseComputationalEfficiency* and *applyIntroduceConcurrency* (Figure 7). The first one, which was introduced in the example of Section 3.3 works by generating a new version of component c (by means of modify operation). This new version of c has a better processor since its *Processor* property (introduced in Figure 1) has the value provided by the *CPUspeedValue* argument, whereas the rest of the properties keeps the same values as the previous version). The second tactic *applyIntroduceConcurrency* works by adding a characteristic denominated *Concurrency* to the component c (by means of *addCharacteristic* operation) and adding a new responsibility to c for achieving *load balancing* features (by means of *addResponsibility* operation).

5. Case study

The proposed model was implemented by adopting the O-Telos language (Jeusfeld *et al.*, 2010) and their implementation on ConceptBase (Jeusfeld *et al.*, 2010), a deductive object-oriented database manager. ConceptBase integrates techniques from deductive and object-oriented databases in the logical framework of the O-Telos' data model, a dialect

```
applyIncreaseComputationalEfficiency (c: Component, CPUspeedValue: String)
  modify(c, [null, null, null, null, CPUspeedValue])
end
applyIntroduceConcurrency (c: Component)
  addCharacteristic(c, Concurrency)
  addResponsibility(c, LoadBalancing)
end
```

Figure 7: *Specification of two tactics.*

of Telos. Telos is a conceptual modelling language for representing knowledge about information systems based on the core concepts of object-oriented technology, integrity constraints, and deductive rules. Therefore, the O-Telos language was employed to formally specify the axioms of the proposed model. In association, the introduced concepts are translated in an object-oriented model by implementing them as classes in ConceptBase. Since ConceptBase is a deductive object base manager, it enables to check and validate the proposed model.

The classes in Figure 8 express the double representation of a design object as *VersionableObject* and *ObjectVersion.* While *VersionableObject* represents a unique entity for each design object that has been created and/or modified due to model evolution during a design project, *ObjectVersion* represents the different versions of each design object. The relationship between a versionable object and one of its object versions is represented by the *version* relationship. This portion of the object-oriented model in ConceptBase is associated to the axioms (1)–(3) that were introduced in Section 3. In addition, *ModelVersion* class supplies a snapshot description of the state of the design process at a given point on a design project. The *belong* relationship in Figure 8 is an inferred association from successor state axiom *belong* (9). Furthermore, the model implemented in ConceptBase specifies the relationships existing among object versions. First, it should be noted that from axiom (12), object versions belonging to a model version are not explicitly associated with other versions belonging to the same model version. These links are represented at the repository level (*association* relationship). *ModelVersion* and *ObjectVersion* concepts are generalized by *AbstractVersion.* The model is enriched with relationships among object versions of different model versions, which enables the navigation along the history of the object versions that comprise a given model version. The relationships among object versions are represented by means of explicit links, named *versionHistory*. Each transformation operation (*Operation* class) that is applied to a model version incorporates the necessary information in terms of the previous link to trace the model evolution. The *modelHistory* relationship represents the sequence of operations applied that gave rise a new model version (which is associated to the meaning of axioms (4)–(6)) and makes it possible the definition of attributes oriented towards the characterization of

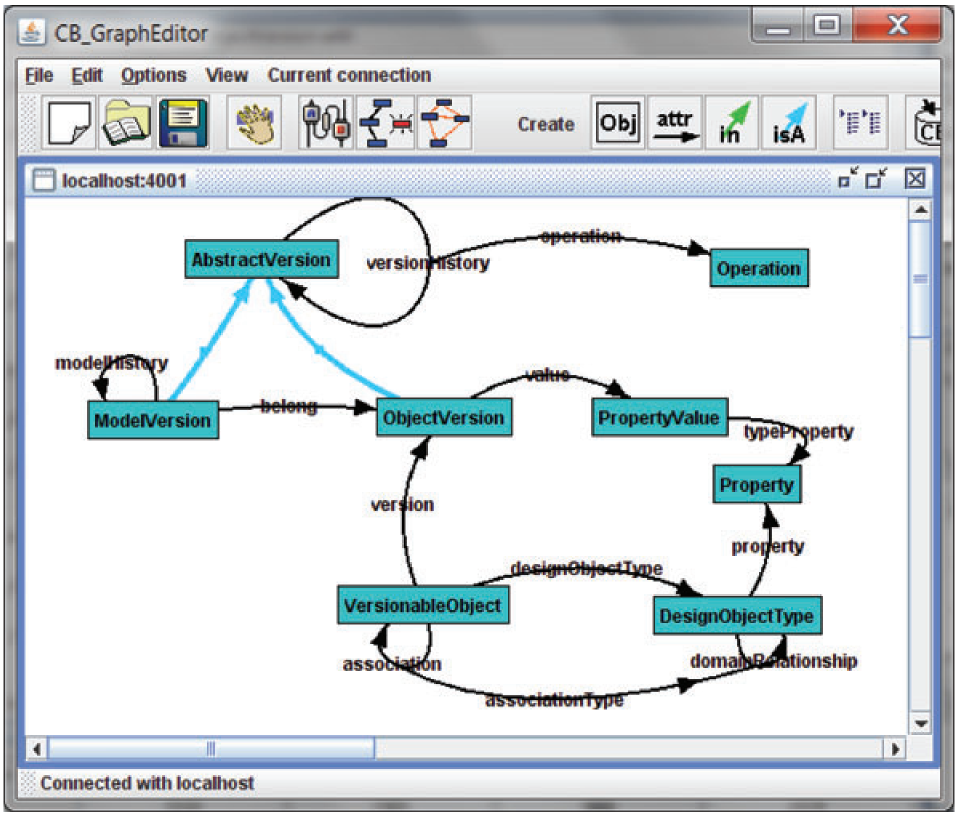
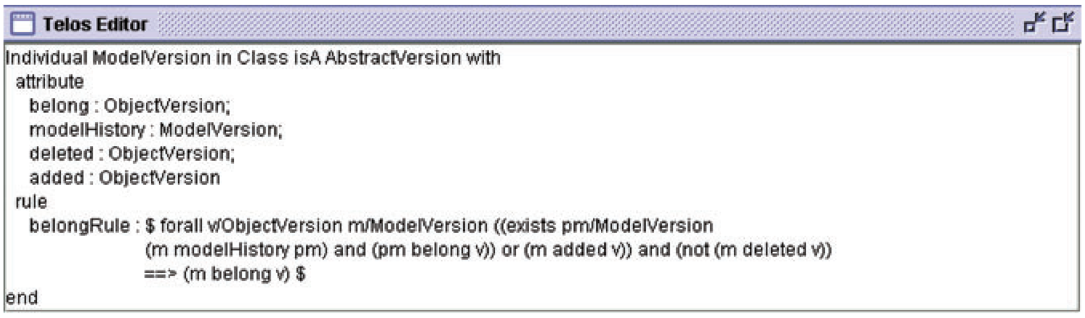


Figure 8: *Implementation of the proposed model in ConceptBase.*



```
Individual ModelVersion in Class isA AbstractVersion with
 attribute
   belong : ObjectVersion;
   modelHistory : ModelVersion;
   deleted : ObjectVersion;
   added : ObjectVersion
 rule
   belongRule : $ forall v/ObjectVersion m/ModelVersion ((exists pm/ModelVersion
                    (m modelHistory pm) and (pm belong v)) or (m added v)) and (not (m deleted v))
                    ==> (m belong v) $
end
```

Figure 9: *Successor state axiom (9) using O-Telos.*

the history of the executed operations. Such a portion of the model implements the semantics of axiom (10).

On the other hand, the types of design objects whose evolution wants to be kept are represented by *DesignObjectType* concept. These design object types specify the properties that characterize them (*Property*), allowing the designer to define object versions with specific values for such properties (see *PropertyValue* concept, which is related to *ObjectVersion* through the *value* relationship). Also, the possible relationships that can exist among these types are defined by *domainRelationship* relationship. This portion of the model is relative to the domain of the design process and is associated to the predicates *designObjectType*, *domainRelationship*, and *property* introduced in Section 4.1 to define a SADP domain.

Figure 9 shows a specification of the cases in which an *object version belongs* to a *model version* (9), using the O-Telos language. It is introduced as a rule of *ModelVersion* (*belongRule* in Figure 9).

5.1. Developing the architecture of a monitoring system

To validate the proposed model, the design of the software architecture of a monitoring system for an industrial process (see Figure 10) has been carried out. It is based on classical case studies addressed in other contributions (Shaw *et al.*, 1995; Bass *et al.*, 2003). Monitoring activities are focused on two core distillation columns: an extractive distillation column and a solvent stripping one, both working together in a highly integrated manner. The system should monitor control loops and temperature sensors, by continued acquisition of real-time process data, tracking set-point values, alarm conditions and outputs of valves, and comparing them with normal pattern behaviour. The system should also monitor the process state, using real-time process data previously processed in combination with expert knowledge in order to maintain process stability and performance. In order to meet all these functional requirements, the system should be connected to input and output devices. The main functions considered in designing the monitoring system include administration of users and permissions, configuration of input/output devices, priority-based event management, process diagnosis, and specification of warning and process protective actions.

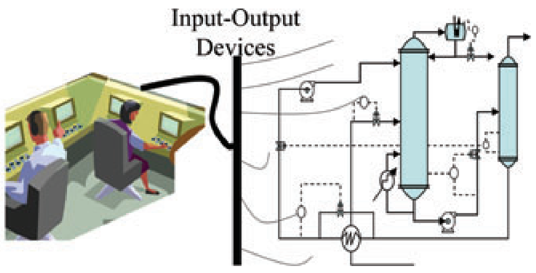


Figure 10: *Monitoring system for an industrial process.*

Figure 11 portrays a partial view of the design process under study. As it is illustrated in the figure, each model version results of applying a sequence of operations on the design objects belonging to a previous model version, allowing their evolution to a new model version, which contains the resulting object versions.

The first model version (*ModelVersion*_1) is the result of a synthesis design decision. This design decision is materialized by the sequence of operations *phi*1, which is applied on the initial model version *InitialModelVersion*.

$$phi1 =< addSystem(\text{‘}MonitoringSystem\text{’}) >$$

The sequence *phi*1 comprises a single operation, *addSystem*(‘*MonitoringSystem*’), represented by *AddSystemMonitoringSystem* operation in Figure 11. As a consequence, a first object version of *MonitoringSystem* is added at the version level (see *ObjectVersion*_1 in Figure 11), and a versionable object (see *VersionableObject*_1 in Figure 11) is added on repository. On *ModelVersion*_1, a new sequence of operations *phi*2 is applied, which comprises a series of *addQualityRequirement* and *addFunctionalRequirement* operations:

$$phi2 =< addQualityRequirement(Object$$

$$Version\_1, \text{‘}QRModifiability\text{’}, []),$$

$$addQualityRequirement(ObjectVersion\_1,$$

$$\text{‘}QRPerformance\text{’}, []),$$

$$addFunctionalRequirement(ObjectVersion\_1,$$

$$\text{‘}FRProcessDataAdquisition\text{’}, []),$$

$$addFunctionalRequirement(ObjectVersion\_1,$$

$$\text{‘}FRSetPointValuesTracking\text{’}, []),$$

$$addFunctionalRequirement(ObjectVersion\_1,$$

$$\text{‘}FRAlarmConditionsTracking\text{’}, []),$$

$$addFunctionalRequirement(ObjectVersion\_1,$$

$$\text{‘}FRProcessStateMonitoring\text{’}, []),$$

$$addFunctionalRequirement(ObjectVersion\_1,$$

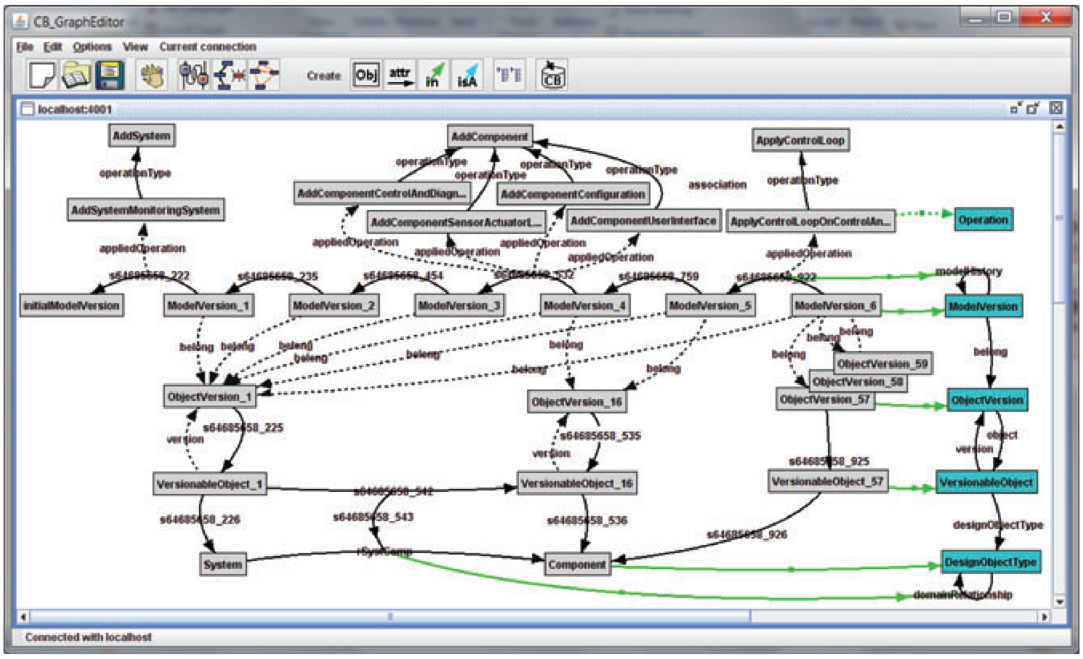


Figure 11: *Design decisions view and model version structure in ConceptBase.*

'*FRUsers&PermissionsAdministration*', []),

addFunctionalRequirement(*ObjectVersion_1*,

'*FRIODevicesConfiguration*', []),

addFunctionalRequirement(*ObjectVersion_1*,

'*FRWarning&ProtectiveActionsSpecif*', []) >,

where *ObjectVersion_1* is the object version of *MonitoringSystem* that makes explicit in the architectural model the requirements to be achieved. By this applying, *ModelVersion_2* is obtained.

The next design activity consists on defining a set of quality scenarios to make the quality requirements more concrete, to make their assessment easier. It is materialized and captured in a sequence of *addQualityScenario* operations applied on *ModelVersion_2*, which gives rise *ModelVersion_3*:

phi3 = < *addQualityScenario*

(*V1QRModifiability*, '*SceModifiability1*',

['*A new sensor is attached. The system has to*

get it without modifications']),

addQualityScenario(*V1QRModifiability*,

'*SceModifiability2*', ['*A process operator*

should easely configure 70% of the system

functionality']),

addQualityScenario

(*V1QRModifiability*, '*SceConfigurability*',

['*New events to monitor must be defined and*

priorities assigned to existent events change.

These modifications have a minimal impact

on the system']),

addQualityScenario

(*V1QRFlexibility*, '*ScePerformance1*',

['*An abnormal situation is detected by*

sensors. The system should start corrective

or alarm action in response to it in less

than 1 second.'], *addQualityScenario*

(*V1QRFlexibility*, '*ScePerformance2*',

['*The average time taken to provide a*

response to any kind of event should not

be longer than 5 seconds.']) > .

It should be noted that the set of object versions generated by the sequences of operations applied on *ModelVersion_1* and *ModelVersion_2* are not shown for clarity reasons. Consequently, the four components are added to the *ModelVersion_4* as a consequence of executing the following sequence of operations:

phi4 = < *addComponent*(*ObjectVersion_1*,

'*Control&Diagnosis*', ['*RDiagnosis*',

'*REventManaging*', '*RIODevicesConfig*',

'*RWarning&ProtectiveActionsLaunching*'],

['*C&DPort1*', '*C&DPort2*', '*C&DPort3*'], []),

addComponent(*ObjectVersion_1*, '*Sensor*

ActuatorLayer', ['*RReceivingSensorsData*',

'*RSendingCommandToActuators*'], ['*SALPort1*',

'*SALPort2*', '*SALPort3*'], []),

addComponent(*V1SysMonitoring*,

'*PConfiguration*', ['*RDevicesCfg*', '*RAlertsCfg*',

$$`RProtectiveActionsCfg'], [`ConfPort1', `ConfPort2', `ConfPort3'], []), addComponent(ObjectVersion\_1, `UserInterface', [`RParametersValuesInput', `RInfoVisualization', `RReports'], [`UIPort1'], []) >,$$

where *ObjectVersion_1* is the object version of *MonitoringSystem*. These operations add an object version of *Control&Diagnosis*, *SensorActuatorLayer*, *Configuration*, and *UserInterface* components at the versions level and their respective versionable objects (Figure 11 only shows *ObjectVersion_16* and *VersionableObject _16*, representing *Control&Diagnosis* component) on repository.

The next design decision addresses how these components are connected and the characteristics of the connections. It is expressed in the following sequence of operations:

$$phi5 = < addConnector(ObjectVersion\_1, `CC\&DSAL', [`CC\&DSALRole1', `CC\&DSALRole2'], [V1SALPort1, V1C\&DPort1], []), addConnector(ObjectVersion\_1, `CSALC\&D', [`CSALC\&DRole1', `CSALC\&DRole2'], [V1SALPort2, V1C\&DPort2], []), addConnector(ObjectVersion\_1, `CConfSAL', [`CConfSALRole1', `CConfSALRole2'], [V1SALPort3, V1ConfPort1], []), addConnector(ObjectVersion\_1, `CC\&AConf', [`CC\&AConfRole1', `CC\&AConfRole2'], [V1C\&APort3, V1ConfPort2], []), addConnector(ObjectVersion\_1, `CConfUI', [`CConfUIRole1', `CConfUIRole2'], [V1ConfPort3, V1UIPort1], []) >$$

This sequence is executed on *ModelVersion_4* and gives rise to *ModelVersion_5*. Now, at this point of the design process, the designer considers the applying of *control loop* architectural pattern, which leads to *ModelVersion_6*. In Figure 11, we can observe that the *ObjectVersion_16* (*Control&Diagnosis* component) belongs to the *ModelVersion_4* (represented by the link denominated as *belong*), but not belong to the model version *ModelVersion_6*. This fact is due to the *Control&Diagnosis* component is refined in three new components through the applying of the sequence of operations *phi6* on *ModelVersion_5* (see Figure 11):

$$phi6 = < applyControlLoop (ObjectVersion\_16) >$$

These three new components added are: *Diagnosis* (*ObjectVersion_57* in Figure 11), *PolicyManager* (*ObjectVersion_58* in Figure 11), and *Reactor* (*ObjectVersion_59* in Figure 11). As it was specified in Figure 5, the *applyControlLoop* operation asks the designer the necessary information for delegating responsibilities, and for attaching previous connectors to the components of the new configuration. By focusing the attention on the history link from *ModelVersion_5* to *ModelVersion_6* (*s64685658_922 modelHistory*), it is observed that the applied operation was captured by the *ApplyControlLoopOnControlAndDiagnosis* object.

Figure 12 shows the elements of an *object version*. In this case, it is illustrated a version of a *Diagnosis component* (*ObjectVersion_57*). The link *versionHistory* captures the information of the performed *operation* that generated it (*ApplyControlLoopOnControlAndDiagnosis*), its *argument* (*ObjectVersion_16*, the refined component), and its *results* (*ObjectVersion_57*, . . . ,59 represent components, *ObjectVersion_60* . . . 61 are connectors, and the other object versions are instances of responsibilities, ports, and roles generated by the *ApplyControlLoopOperation*).

5.2. Reasoning capabilities

The proposed model enables the capture of design process states that were reached during a SADP. This section shows how this proposal supports reasoning to recover captured knowledge and answer questions about how the design process took place. Queries are answered by navigating through the model history, by tracing the captured operations and their predecessor versions and successor versions. Such a tracing begins from a particular version and moves backwards or forwards depending on the intended information to recover. A set of queries is formulated with the aim of showing the reader the feasibility of extending the model in order to infer more information about the SADP from the knowledge base. So, the examples do not pretend to show how a designer would interact with the knowledge base. If this type of interaction were considered valuable, an appropriate and user-friendly interface would need to be designed.

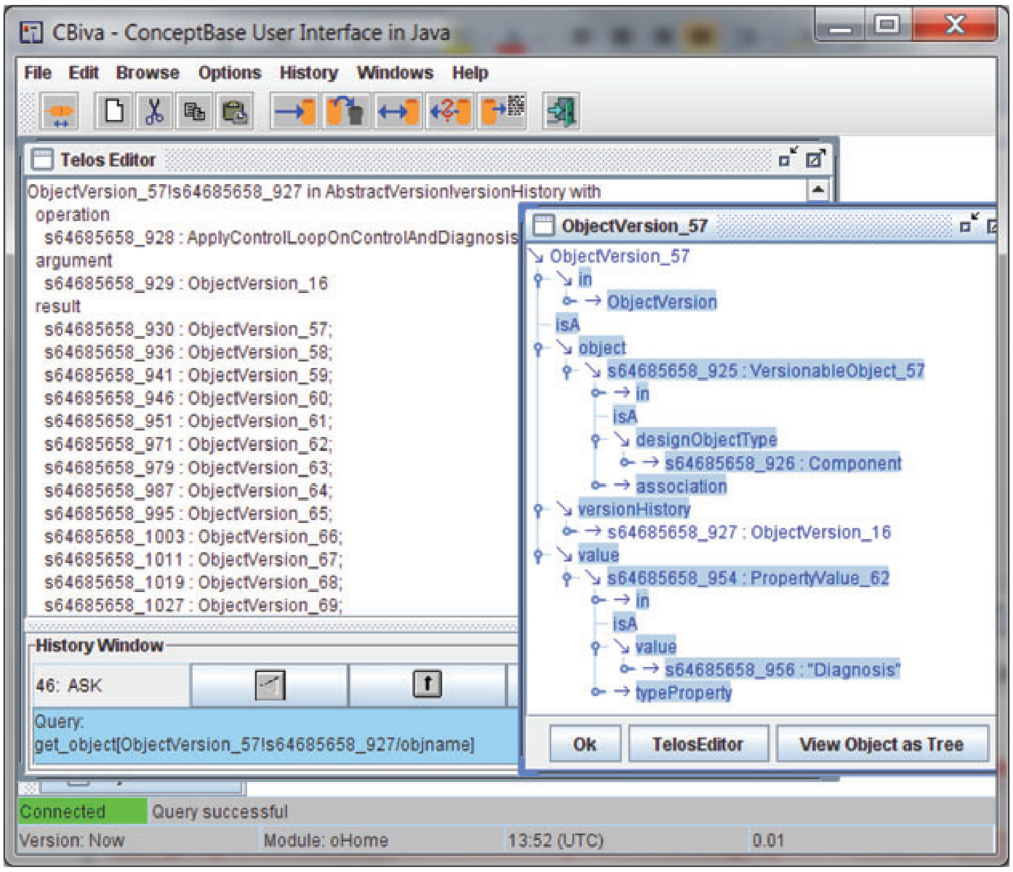
For example, consider the following queries:

(1) Which design operations originated a given design state?

A design state is represented by *model version* concept. As it was introduced in Section 2.2, it is possible to reconstruct a *model version* m_{i+1} by applying all the sequence of operations from the initial *model version* m_0. This fact is represented in expression (26). Therefore, the sequence of operations $\phi_1 \bullet \ldots \bullet \phi_i \bullet \phi_{i+1}$ answers which operations were applied to obtain the model version m_{i+1}.

$$\begin{aligned} m_{i+1} &= apply(\phi_{i+1}, m_i); \\ m_i &= apply(\phi_i, m_{i-1}); \ldots; \\ m_1 &= apply(\phi_1, m_0) \\ m_{i+1} &= apply(\phi_{i+1}, \\ &\quad apply(\phi_i, apply(\ldots apply(\phi_1, m_0)\ldots))) \\ m_{i+1} &= apply(\phi_1 \bullet \ldots \bullet \phi_i \phi_{i+1}, m_0). \end{aligned} \tag{26}$$

In the ConceptBase implementation of the proposed model, when a sequence of operation is performed, it is captured by means of a *modelHistory* predicate. Therefore, $\phi_1 \bullet \phi_2 \bullet \phi_3 \bullet \phi_4 \bullet \phi_5 \bullet \phi_6$ is the sequence of operations applied to the *InitialModelVersion* to obtain the *ModelVersion_6* model version. This is retrieved from the knowledge base using



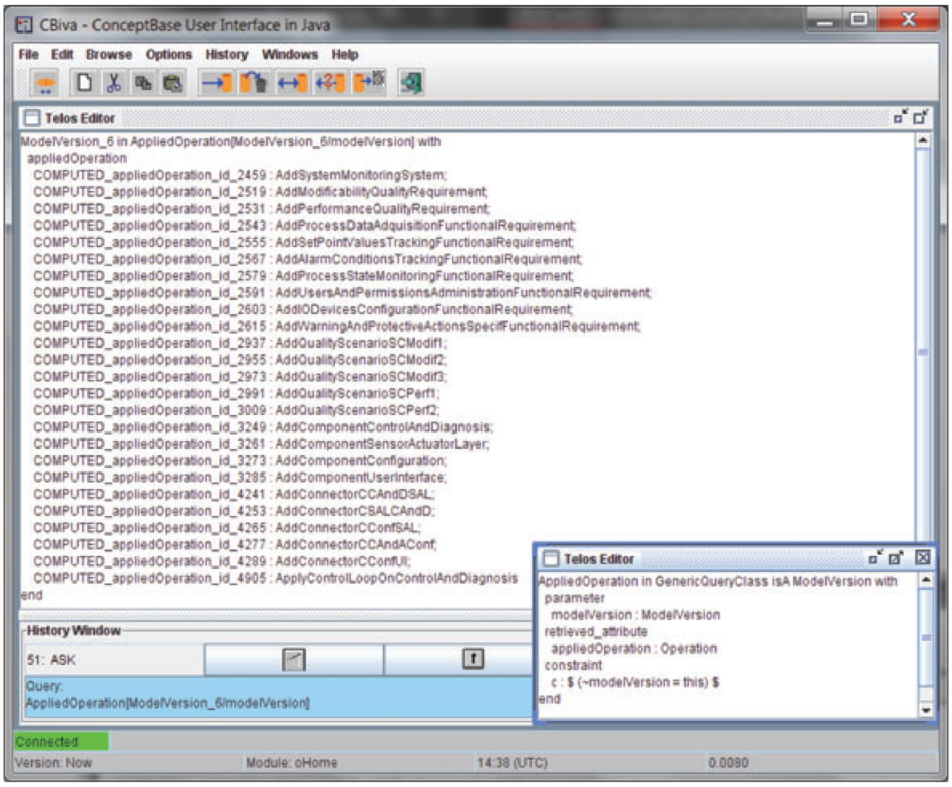
```
ObjectVersion_57!s64685658_927 in AbstractVersion!versionHistory with
 operation
  s64685658_928 : ApplyControlLoopOnControlAndDiagnosis
 argument
  s64685658_929 : ObjectVersion_16
 result
  s64685658_930 : ObjectVersion_57;
  s64685658_936 : ObjectVersion_58;
  s64685658_941 : ObjectVersion_59;
  s64685658_946 : ObjectVersion_60;
  s64685658_951 : ObjectVersion_61;
  s64685658_971 : ObjectVersion_62;
  s64685658_979 : ObjectVersion_63;
  s64685658_987 : ObjectVersion_64;
  s64685658_995 : ObjectVersion_65;
  s64685658_1003 : ObjectVersion_66;
  s64685658_1011 : ObjectVersion_67;
  s64685658_1019 : ObjectVersion_68;
  s64685658_1027 : ObjectVersion_69;
```

```
History Window
46: ASK
Query:
get_object[ObjectVersion_57!s64685658_927/objname]
```

```
ObjectVersion_57
  in
    ObjectVersion
  isA
  object
    s64685658_925 : VersionableObject_57
      in
      isA
      designObjectType
        s64685658_926 : Component
      association
  versionHistory
    s64685658_927 : ObjectVersion_16
  value
    s64685658_954 : PropertyValue_62
      in
      isA
      value
        s64685658_956 : "Diagnosis"
      typeProperty
```

Figure 12: *ObjectVersion view: object version of diagnosis component.*

the *GenericQueryClass* concept (Figure 13), an O-Telos construct. A generic query class in ConceptBase is defined as a specialization of a class whose instances may be part of the sought answer. Therefore, to ask about the operations applied to a model version, it is possible to formulate the query as a specialization of *ModelVersion* class. In this particular case, it is named *AppliedOperation*. In turn, the *modelVersion* parameter is defined to specify the particular model version to ask, in this case *ModelVersion_6* (*AppliedOperation[ModelVersion_6/modelVersion]* in Figure 13), and the

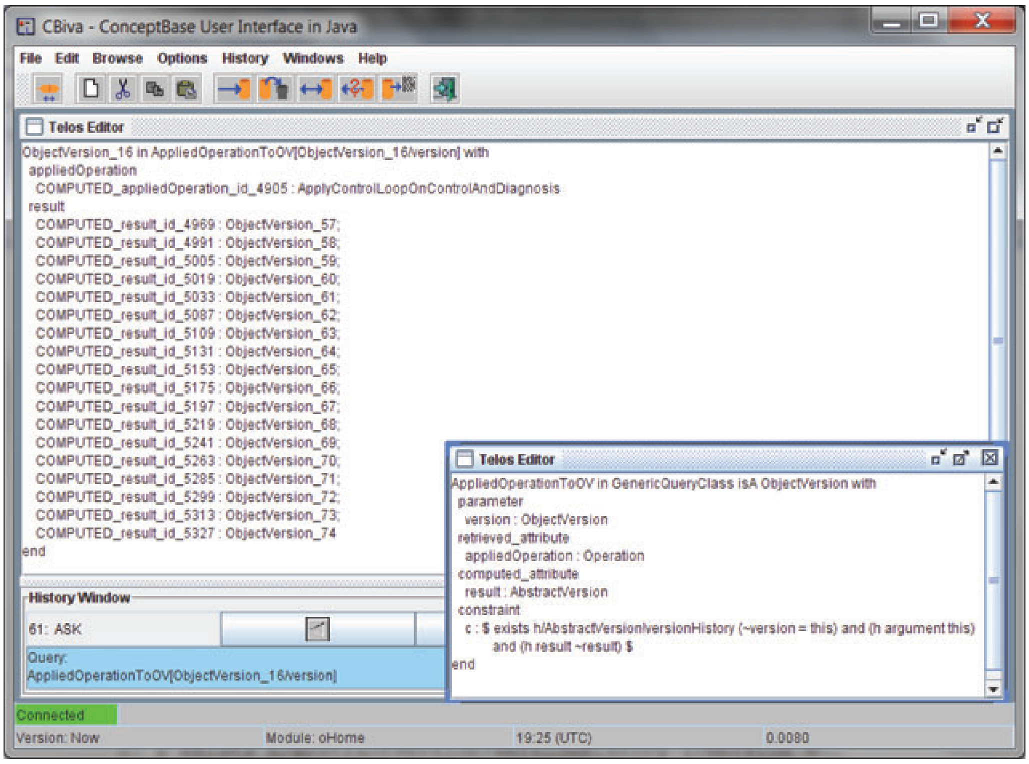

CBiva - ConceptBase User Interface in Java
File Edit Browse Options History Windows Help
Telos Editor
Telos Editor
Connected
Version: Now
Module: oHome
14:38 (UTC)
0.0080

```
ModelVersion_6 in AppliedOperation[ModelVersion_6/modelVersion] with
 appliedOperation
  COMPUTED_appliedOperation_id_2459 : AddSystemMonitoringSystem;
  COMPUTED_appliedOperation_id_2519 : AddModificabilityQualityRequirement;
  COMPUTED_appliedOperation_id_2531 : AddPerformanceQualityRequirement;
  COMPUTED_appliedOperation_id_2543 : AddProcessDataAdquisitionFunctionalRequirement;
  COMPUTED_appliedOperation_id_2555 : AddSetPointValuesTrackingFunctionalRequirement;
  COMPUTED_appliedOperation_id_2567 : AddAlarmConditionsTrackingFunctionalRequirement;
  COMPUTED_appliedOperation_id_2579 : AddProcessStateMonitoringFunctionalRequirement;
  COMPUTED_appliedOperation_id_2591 : AddUsersAndPermissionsAdministrationFunctionalRequirement;
  COMPUTED_appliedOperation_id_2603 : AddIODevicesConfigurationFunctionalRequirement;
  COMPUTED_appliedOperation_id_2615 : AddWarningAndProtectiveActionsSpecifFunctionalRequirement;
  COMPUTED_appliedOperation_id_2937 : AddQualityScenarioSCModif1;
  COMPUTED_appliedOperation_id_2955 : AddQualityScenarioSCModif2;
  COMPUTED_appliedOperation_id_2973 : AddQualityScenarioSCModif3;
  COMPUTED_appliedOperation_id_2991 : AddQualityScenarioSCPerf1;
  COMPUTED_appliedOperation_id_3009 : AddQualityScenarioSCPerf2;
  COMPUTED_appliedOperation_id_3249 : AddComponentControlAndDiagnosis;
  COMPUTED_appliedOperation_id_3261 : AddComponentSensorActuatorLayer;
  COMPUTED_appliedOperation_id_3273 : AddComponentConfiguration;
  COMPUTED_appliedOperation_id_3285 : AddComponentUserInterface;
  COMPUTED_appliedOperation_id_4241 : AddConnectorCCAndDSAL;
  COMPUTED_appliedOperation_id_4253 : AddConnectorCSALCAndD;
  COMPUTED_appliedOperation_id_4265 : AddConnectorCConfSAL;
  COMPUTED_appliedOperation_id_4277 : AddConnectorCCAndAConf;
  COMPUTED_appliedOperation_id_4289 : AddConnectorCConfUI;
  COMPUTED_appliedOperation_id_4905 : ApplyControlLoopOnControlAndDiagnosis
end
```

```
History Window
51: ASK
Query:
AppliedOperation[ModelVersion_6/modelVersion]
```

```
AppliedOperation in GenericQueryClass isA ModelVersion with
 parameter
  modelVersion : ModelVersion
 retrieved_attribute
  appliedOperation : Operation
 constraint
  c : $ (~modelVersion = this) $
end
```

Figure 13: *Query about sequence of operations applied to InitialModelVersion to obtain ModelVersion_6.*



```
ObjectVersion_16 in AppliedOperationToOV[ObjectVersion_16/version] with
 appliedOperation
  COMPUTED_appliedOperation_id_4905 : ApplyControlLoopOnControlAndDiagnosis
 result
  COMPUTED_result_id_4969 : ObjectVersion_57;
  COMPUTED_result_id_4991 : ObjectVersion_58;
  COMPUTED_result_id_5005 : ObjectVersion_59;
  COMPUTED_result_id_5019 : ObjectVersion_60;
  COMPUTED_result_id_5033 : ObjectVersion_61;
  COMPUTED_result_id_5087 : ObjectVersion_62;
  COMPUTED_result_id_5109 : ObjectVersion_63;
  COMPUTED_result_id_5131 : ObjectVersion_64;
  COMPUTED_result_id_5153 : ObjectVersion_65;
  COMPUTED_result_id_5175 : ObjectVersion_66;
  COMPUTED_result_id_5197 : ObjectVersion_67;
  COMPUTED_result_id_5219 : ObjectVersion_68;
  COMPUTED_result_id_5241 : ObjectVersion_69;
  COMPUTED_result_id_5263 : ObjectVersion_70;
  COMPUTED_result_id_5285 : ObjectVersion_71;
  COMPUTED_result_id_5299 : ObjectVersion_72;
  COMPUTED_result_id_5313 : ObjectVersion_73;
  COMPUTED_result_id_5327 : ObjectVersion_74
end
```

```
Query:
AppliedOperationToOV[ObjectVersion_16/version]
```

```
AppliedOperationToOV in GenericQueryClass isA ObjectVersion with
 parameter
  version : ObjectVersion
 retrieved_attribute
  appliedOperation : Operation
 computed_attribute
  result : AbstractVersion
 constraint
  c : $ exists h/AbstractVersion!versionHistory (~version = this) and (h argument this)
        and (h result ~result) $
end
```

Figure 14: *Query about operations applied to ObjectVersion_16.*

appliedOperation attribute is defined to capture the answer. Moreover, the instances of the *AppliedOperation* query are constrained by means of the *c* constraint, specified in Figure 13. This constraint will identify only the *modelVersion* specified by the input parameter.

(2) Which are the alternative successor model versions of a certain model version?

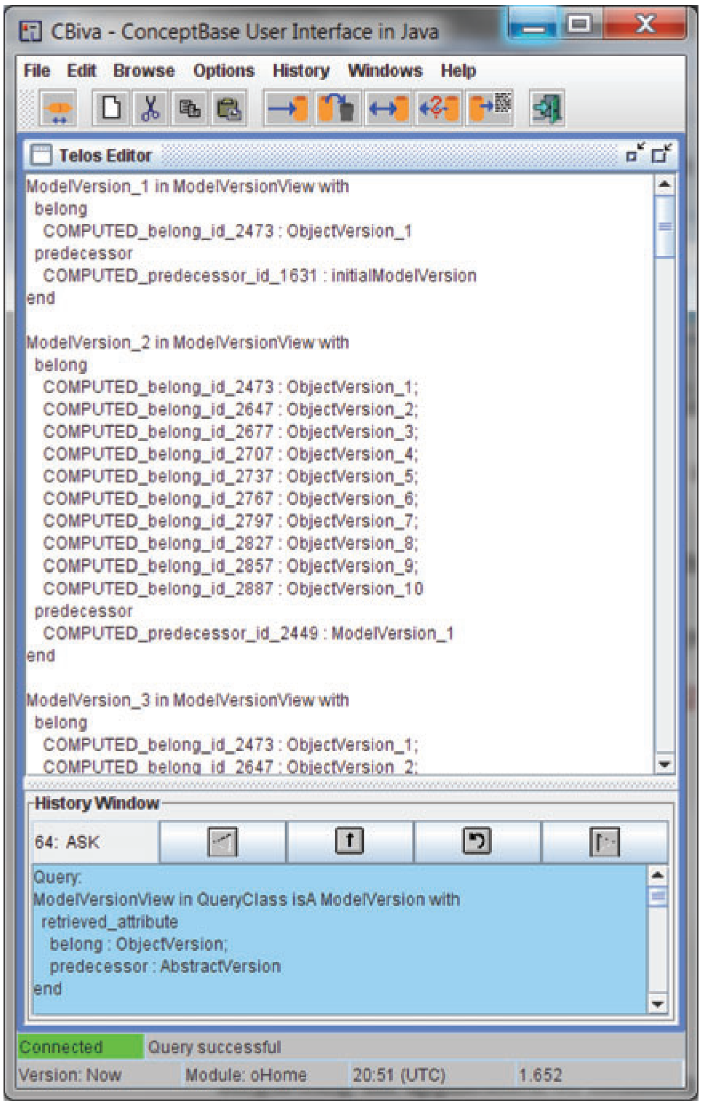
Given a model version, for example, *ModelVersion_5*, it is possible to apply several design operations evolving the SADP to different states. This fact is represented by the *model History* predicate, which link the predecessor model version with the successor model version. If m_s is the result of apply the sequence ϕ_s to the m_p model version ($m_s = apply(\phi_s, m_p)$), then m_p is the predecessor model version and m_s is the successor model version. In consequence, a model version has a unique predecessor model version (except the initial model version) and a model version can have zero, one, or more successor model versions. Each successor model version represents an alternative design state reached during the design process. Note that, in the case study, only a successor model version was proposed for *InitialModelVersion*, *ModelVersion_1*, *ModelVersion_2*, *ModelVersion_3*, *ModelVersion_4*, and *ModelVersion_5*. If ADD method is considered, architectural patterns (defined as design operation in this proposal) are chosen at given state (model version) to fulfil a set of requirements (*functional* and *quality requirements*). From the instantiation of an architectural pattern (execution of the design operation), several elements (like *components* and *connectors*) are included in the architectural model, evolving the design state a new model version. Therefore, at a given model version, as *ModelVersion_5*, it is possible to perform a sequence of operations including *ApplyControlLoop* operation, evolving the design to *ModelVersion_6* (Figure 11). This operation was performed to address the requirements about activate monitoring policies in response to different events produced by several sensors. If the designer selects another quality requirement to reach, then another model version is defined to represent the new design state.

(3) How did a design object change along the design process?

The versions of design objects are represented by *object version* concept. When an operation is applied, the effects of the operation can add new *object versions* (represented by the *added* predicate, expression (7)) or delete *object versions* (represented by the *deleted* predicate, expression (8)) to a *model version*. For example, when the *applyControlLoop* operation was applied to *ModelVersion_5*, the *ObjectVersion_16* was deleted (therefore, it does not belong to *ModelVersion_6*, Figure 11), and several object versions were added (*ObjectVersion_57–59* in Figure 11). In the ConceptBase implementation, these operation effects were captured by *versionHistory* relationship. Therefore, to answer this query, a generic query class is defined (Figure 14). Figure 14 shows that *ObjectVersion_16* was refined in several object versions, described by *result attribute*, by the application of *ApplyControlLoop* operation.

(4) Which were the products generated due to the execution of a sequence of operations?

This knowledge is retrieved from expression (9) that states which object versions belong to the version model as a result of applying the sequence of operations ϕ on model version m. Figure 15 shows the results of asking *ModelVersionView* query for retrieving all model versions and their belonging object versions. In the results window, *ModelVersion_1*, *ModelVersion_2*, and a partial *ModelVersion_3* from the developed case study are shown.


CBiva - ConceptBase User Interface in Java
File Edit Browse Options History Windows Help
Telos Editor
History Window
64: ASK

```
ModelVersion_1 in ModelVersionView with
 belong
  COMPUTED_belong_id_2473 : ObjectVersion_1
 predecessor
  COMPUTED_predecessor_id_1631 : initialModelVersion
end

ModelVersion_2 in ModelVersionView with
 belong
  COMPUTED_belong_id_2473 : ObjectVersion_1;
  COMPUTED_belong_id_2647 : ObjectVersion_2;
  COMPUTED_belong_id_2677 : ObjectVersion_3;
  COMPUTED_belong_id_2707 : ObjectVersion_4;
  COMPUTED_belong_id_2737 : ObjectVersion_5;
  COMPUTED_belong_id_2767 : ObjectVersion_6;
  COMPUTED_belong_id_2797 : ObjectVersion_7;
  COMPUTED_belong_id_2827 : ObjectVersion_8;
  COMPUTED_belong_id_2857 : ObjectVersion_9;
  COMPUTED_belong_id_2887 : ObjectVersion_10
 predecessor
  COMPUTED_predecessor_id_2449 : ModelVersion_1
end

ModelVersion_3 in ModelVersionView with
 belong
  COMPUTED_belong_id_2473 : ObjectVersion_1;
  COMPUTED_belong_id_2647 : ObjectVersion_2;
```

```
Query:
ModelVersionView in QueryClass isA ModelVersion with
 retrieved_attribute
  belong : ObjectVersion;
  predecessor : AbstractVersion
end
```

```
Connected    Query successful
Version: Now    Module: oHome    20:51 (UTC)    1.652
```

Figure 15: *Query about object versions belonging to model versions.*

6. Related work and discussion

Several approaches for representing architectural design decisions and architectural design rationale have been recently proposed. They assist software architects in their decision-making activities by capturing and characterizing architectural knowledge. Most of these tools are based on conceptual or semi-formal models, which provide a characterization and interpretation of SADP and make the emphasis on specifics activities of SADP. Tyree and Akerman (2005) have proposed a template of attributes to represent architectural design decisions. Such an approach allows the designers documenting some critical evolutions of SADP. The approach discussed in Capilla *et al.* (2007) is similar, but instead of providing a complete list of attributes to describe a design decision, they propose the use of mandatory and optional attributes that can be tailored according to different needs for making more agile the efforts of capturing a design decision. In addition, they include specific attributes and relationships aimed to support the evolution of design decisions. Archium is a tool that models design decisions and their relationships with resulting components (Jansen & Bosch, 2005). It is based on a conceptual model for representing architectural design decisions and their context, which allows keeping the evolution of an architecture design by keeping architectural deltas (changes). The perspective employed in this approach is different from ours since the design decisions are not explicitly captured; they remain as tacit knowledge. Hence, the tacit knowledge embedded in the captured design is used to trace back from the changes to the decisions they originated from (Jansen *et al.*, 2008). Contrary to this perspective, our approach captures the design decisions by materializing them in a sequence of operations that is applied on the current model version. In this way, the design decisions are captured during the SADP and not after. Ali Babar and Gorton (2007) propose another framework for the capture and recover of architecture knowledge called Process-centric Architecture Knowledge Management Environment (PAKME). PAKME uses a data model for characterizing architectural constructs (such as design decisions, alternatives, rationale, and quality attributes), their attributes and relationships. Each design decision is captured as a case along with rationale and contextual information using a template. The main function of this tool is to support general knowledge representation such as design patterns or tactics in order to facilitate architectural knowledge reuse, goal that is just partially achieved. The primary purpose of our proposal, besides supporting architectural knowledge representation, is to assist the architectural development process, providing the elements to define a proper domain model. Also, it is important to note that a tool based on the proposed model could be opened to allow integration with external knowledge sources, in order to support their conceptual models and operations.

Besides the valuable contribution to the capture, codification, and partially retrieving of the knowledge (generated or applied) in a design process, the models that underlie the aforementioned approaches lacks formalism. On the contrary, our approach is formalized on situation calculus and first-order logic, which allows us to define the model precisely. Such a formalization serves, in some way, as a specification for the design of computational tools for capturing the knowledge about SADP (Roldán *et al.,* 2010).

There are many contributions that use situation calculus for modelling dynamic domains. Albrecht *et al.* (2003) describe a situation calculus agent system for e-business. Agents' knowledge about actions and their effects is represented using the conventions of situation calculus. This knowledge enables the agents to keep track of the world and to deduce the effects of plans of action, all of which are applicable to e-business needs. Koubarakis and Plexousakis (2002) presented an application of situation calculus for enterprise and business process modelling. They propose a formal framework to represent knowledge about organizations and their business processes. The use of situation calculus permits them to verify specific correctness properties of the specifications.

In our proposal, situation calculus is not employed as a reasoning formalism to obtain a plan of action to guide the design activities. Instead, it is applied as a modelling formalism for representing design knowledge. Situation calculus is used for expressing without ambiguity when it is possible to perform an operation in a given instant of the design process. In addition, situation calculus allows us to determine which versions of the several objects, that have been generated, belong to that point of SADP (in other words, to a particular version model).

Since the proposed model is intended to record the evolution of a design process through the tracking of all the generated versions, it shares some similarities with other versioning approaches like software configuration management (SCM) systems (Westfechtel, 1999), database management systems, ontology management, and knowledge management frameworks (Maliappis & Sideridis, 2004). These approaches also provide support for the administration of design process products and their evolution (Westfechtel & Conradi, 2003). Particularly, as Westfechtel (1999) has pointed out, SCM systems have proved to be an indispensable aid in organizing the products generated along big development efforts. However, their underlying data models, to represent versions, are very simple. Indeed, SCM systems have been created just to focus on the products of development processes, neglecting the representation of the design activities, the decisions that were taken, the people and computerized tools that performed such activities, and the rationale that underlies the adopted decisions. Thus, they do not satisfy the need of capturing the design knowledge. On the contrary, once a design stage is complete, what remains is mainly the final artefact, but there is no explicit representation of how this product was obtained.

The proposed formalism can be specialized to a specific domain model that includes concepts of a particular software architectures domain. A domain model serves as a common language for communication among the actors that participate in a design process. Since all modelling concepts (design object types) are clearly defined and related to other concepts, the software architectural model is easier to understand.

Despite of the benefits of the present proposal on capturing and representing SADP as the process takes place, the approach has some limitations. The main restriction is inherent to the size of the repository that keeps all the design operations performed and versions generated during an architectural design process. A way of diminishing the problem is the definition of high-level operations, which encapsulate complex decisions (that could comprise dozens of basic operations in just one operation). It must be also stated, that in order to obtain advantages of the model, it should be integrated to other tools for supporting architectural design, thus reducing the overhead of performing design operations by including specific tool operations as part of the SADP domain.

It should be mentioned that in this proposal, the defined design operations do not cover all the activities of the software architecture life cycle (Hofmeister *et al.*, 2007; Tang *et al.*, 2010), and just representative analysis and synthesis design operations have been proposed. Mainly, the current operations are focused on eliciting requirements and manipulating (adding, modifying, deleting, and refining) architectural elements like components, connectors, or responsibilities. However, the set of operations can be easily extended to design operations related to other activities of the architecture life cycle, like the set of operations for evaluation proposed in Roldán (2009).

The proposed formal model set the grounds for developing a tool that supports SADP, which makes possible the representation of types and relationships of specific domains as well as the definition of operations to manage the design products. The model provides the necessary mechanisms to capture and represent the performed sequences of operations as design decisions, along with their resulting products, thus mitigating the problem of overloading the designer's work with documenting approaches that do not fit to the natural course of SADP. The representation capabilities of the model goes beyond the simple design decisions documentation, where information is just informally stored in textual, template-based, or graphical formats.

The next step in domain operations definition is transforming the existent operations in requirements or goal driven operations. It consists in adding an extra argument (or term) to an operation for indicating which is the goal to be reached when applying such an operation, and including special relationships with 'achieves' or 'satisfies' semantic to link the results and the predecessor versions.

Some research in collaborative design has been carried out by the authors (Gonnet *et al.*, 2007) exploring the utilization of situation calculus for conflicts detection and resolution. This work needs being extended in the field of software architectures design process.

7. Conclusions

The model proposed in this paper captures the operations that generate each design product during the SADP, and therefore, it enables the designer to get a better understanding of the information on how the design objects have been obtained. It also offers an explicit mechanism to manage the different model versions generated during the SADP. Thus, it allows the tracing of the SADP and its resulting products. The proposed formalism records the design decisions as they are made (by means of capturing the executed operations) along with their impact on the architectural model (the results of the operations). This is a fundamental step towards the development of a knowledge base to support the SADP and to guide designers in the different activities of a design project, setting the grounds for learning, future reuse, and for proposals of formal means for detecting potential conflicts.

Situation calculus, the formal background of the proposed framework, allows us to represent the evolution of a SADP by means of the specification of a set of actions, fluents, state successor axioms, and action preconditions. The situations are represented by the *model version* concept. Actions are *add*, *delete*, and *modify* primitive operations, which conform the several sequences of operations that are applied to generate a new situation (model version). The proposed model also includes complex operations (*design operations*) made of simpler ones. Action preconditions axioms allow us expressing without ambiguities when an operation can be performed in a specific state of the design process. *belong* is a fluent, which specifies the object versions that are part of a model version. This fact is represented by a very simple successor state axiom (9), using *added* and *deleted* predicates. In addition, the representation of a design object at two levels, *versionable object* and *object version*, allow us to maintain just one fluent (*belong*) in the model. Associations among design objects are represented at the repository level, and as a result, they are not defined as fluents. Therefore, our proposal requires only one successor state axiom (9), which is a valuable contribution in order to obtain a compact and simple model.

Furthermore, this contribution uses an operational perspective where design decisions can be modelled by means of design operations. This approach is employed in other contributions like Bass *et al.* (2003). The structure of the conceptual framework allows the easy definition of specific design operations, like *applyControlLoop* and *applyIncreaseComputationalEfficiency*, without modifying the successor state axiom (9). Similarly, other design operations, like the architectural patterns and tactics defined by Bass *et al.* (2003), can be specified.

The model offers an explicit mechanism, based on situation calculus, to represent the different design process states that are reached during a SADP. Using this feature, it is possible to retrieve the history of a given model version or a specific version of an architectural element. A computational implementation of this capability will show a view the architectural decisions in a chronological order, to better understand the decision-making process.

Acknowledgements

The authors wish to acknowledge the financial support received from CONICET, Universidad Tecnológica Nacional and Agencia Nacional de Promoción Científica y Tecnológica (PAE PICT 02315).

References

Albrecht, C., D. Dean and J. Hansen (2003) Using situation calculus for e-business agents, *Expert Systems with Applications*, **24**, 391–397.

Ali Babar, M. and I. Gorton (2007) A tool for managing software architecture knowledge, in *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI '07)*, Washington, DC: IEEE Computer Society, 11–11. DOI = 10.1109/SHARK-ADI.2007.1.

Bachmann, F., L. Bass and M. Klein (2003) Preliminary design of ArchE: a software architecture design assistant, *Technical Report CMU/SEI-2003-TR-021*, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Bass, L., P. Clements and R. Kazman (2003) *Software architecture in practice*, 2nd edn, MA: Addison-Wesley.

Capilla, R., F. Nava and A. TANG (2007) Attributes for characterizing the evolution of architectural design decisions, in *Proceedings of the Third International IEEE Workshop on Software Evolvability*, IEEE CS, Paris, France, 15–22.

Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. STafford (2010) *Documenting Software Architectures: Views and Beyond*, 2nd edn, MA: Addison Wesley.

Dingsøyr, T. and H. Van Vliet (2009) Introduction to software architecture and knowledge management, in M. Ali Babar, T. Dingsøyr, P. Lago and H. van Vliet (eds), *Software Architecture Knowledge Management, Theory and Practice*, Heidelberg: Springer, 1–17.

Garlan, D., R. T. Monroe and D. Wile (2000) Acme: architectural description of component-based systems, in G. T. Leavens and M. Sitaraman (eds), *Foundations of component-based systems*, New York: Cambridge University Press, 47–68.

Gonnet, S., H. Leone and G. Henning (2007) A model for capturing and representing the engineering process, *Expert Systems with Applications*, **33**, 881–902, DOI = 10.1016/j.eswa.2006.07.004.

Hofmeister, C., P. Kruchten, R. L. Nord, H. Obbink, A. Ran and P. America (2007) A general model of software architecture design derived from five industrial approaches, *Journal of Systems and Software*, **80**, 106–126.

IEEE (2000) *IEEE 1417 Recommended Practice for Architectural Description of Software-Intensive Systems*, New York: IEEE Press.

Jansen, A. and J. Bosch (2005) Software architecture as a set of architectural design decisions, in *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*, Washington, DC: IEEE Computer Society, 109–120.

Jansen, A., J. Bosch and P. Avgeriou (2008), Documenting after the fact: recovering architectural design decisions, *Journal of Systems and Software*, **81**, 536–557.

Jeusfeld, A., C. Quix and M. Jarke (2010) ConceptBase V7.2 User Manual, Aachen: RWTH Aachen, Informatik 5.

Koubarakis, M. and D. Plexousakis (2002) A formal framework for business process modelling and design, *Information Systems*, **27**, 299–319.

Kruchten, P., P. Lago and H. Van Vliet (2006) Building up and reasoning about architectural knowledge, *Quality of Software Architecture (QoSA)*, Heidelberg: Springer-Verlag, 43–58.

Maliappis, M. T. and A. B. Sideridis (2004) A framework of knowledge versioning management, *Expert Systems*, **21**, 149–156.

McCarthy, J. (1963) *Situations, Actions, and Causal Laws*, Memo 2, Stanford, CA: Stanford University Artificial Intelligence Project.

Medvidovic, N., D. Rosenblum, D. Redmiles and J. Robbins (2002) Modeling software architectures in the Unified Modeling Language, *ACM Transaction on Software Engineering and Methodology*, **11**, 2–57.

Medvidovic, N., E. Dashofy and R. Taylor (2007) Moving architectural description from under the technology lamppost, *Information and Software Technology*, **49**, 12–31.

Reiter, R. (2001) *Knowledge in Action: Logical Foundation for Describing and Implementing Dynamical Systems*, Cambridge, Massachusetts: The MIT Press.

Roldan, M. L. (2009) *Un modelo para la representación de conocimiento y razonamiento en el proceso de diseño de arquitecturas* *de software*, PhD Dissertation, Universidad Tecnológica Nacional, Facultad Regional Santa Fe, Argentina.

ROLDAN, M. L., S. GONNET and H. LEONE (2006) A model for capturing and tracing architectural designs, in *Advanced Software Engineering: Expanding the Frontiers of Software Technology, IFIP International Federation for Information Processing*, Boston: Springer, Vol. 219/2006, 16–31.

ROLDAN, M. L., S. GONNET and H. LEONE (2010) TracED: a tool for capturing and tracing engineering design processes, *Advances in Engineering Software*, **41**, 1087–1109. DOI = 10.1016/j.advengsoft.2010.06.006.

SCHERL, R. and H. LEVESQUE (2003) Knowledge, action, and the frame problem, *Artificial Intelligence*, **144**, 1–39.

SHAW, M. (1994) *Beyond objects: a software design paradigm based on process control*, *Technical Report CMU-CS-94–154*, Carnegie Mellon University, Pittsburgh.

SHAW, M., D. GARLAN, R. ALLEN, D. KLEIN, J. OCKERBLOOM, C. SCOTTT and M. SCHUMACHER (1995) *Candidate Model Problems in Software Architecture*, Pittsburgh: The Software Architecture Group Computer Science Department, Carnegie Mellon University

TANG, A., P. AVGERIOU, A. JANSEN, R. CAPILLA and M. ALI (2010) A comparative study of architecture knowledge management tools, *Journal of Systems and Software*, **83**, 352–370.

TAYLOR, R. and A. VAN DER HOEK (2007) Software design and architecture: the once and future focus of software engineering, in *International Conference on Software Engineering, Future of Software Engineering (FOSE '07)*, 226–243.

TYREE, J. and A. AKERMAN (2005) Architecture decisions: demystifying architecture, *IEEE Software*, **22**, 19–27.

WEINREICH, R. and G. BUCHGEHER (2011) Towards supporting the software architecture life cycle, *Journal of Systems and Software*, **85**(3), 546–561.

WEINREICH, R. and G. BUCHGEHER (2011) Towards supporting the software architecture life cycle, *Journal of Systems and Software*, In Press, Corrected Proof, Available online 7 June 2011.

WESTFECHTEL, B. (1999) Models and tools for managing development processes, *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer, 1646.

WESTFECHTEL, B. and R. CONRADI (2003) Software architecture and software configuration management, *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer, Vol. **2649**, 24–39.

The authors

María Luciana Roldán

María Luciana Roldán received her Information Systems Engineering degree from 'Universidad Tecnológica Nacional' (UTN), Santa Fe, Argentina, in 2002. She also obtained her PhD degree in Engineering from 'Universidad Tecnológica Nacional' in 2009. She also has a Postdoctoral Research Fellowship from the National Council for Scientific and Technical Research of Argentina (CONICET), to work at 'Instituto de Desarrollo y Diseño' (INGAR). Her research interests focus on software architectures, specially in models and methods for capturing the design process and its rationale. Additionally, she works as an Assistant Professor at Universidad Tecnológica Nacional.

Silvio Gonnet

Silvio Gonnet received an Engineering degree in Information Systems from 'Universidad Tecnológica Nacional' (UTN), Santa Fe, Argentina, in 1998, and also obtained his PhD degree in Engineering from 'Universidad Nacional del Litoral' (UNL) in 2003. He currently holds a Researcher position at the National Council for Scientific and Technical Research of Argentina (CONICET), to work at 'Instituto de Desarrollo y Diseño' (INGAR). Also, he works as an Assistant Professor at Universidad Tecnológica Nacional. His research interests are in models to support the design process, software architectures, and semantic web.

Horacio Leone

Horacio Leone is a full Professor at the Department of Information Systems Engineering of the 'Facultad Regional Santa Fe, Universidad Tecnológica Nacional' (Santa Fe, Argentina), where he is currently the Department Head. He also holds a Researcher position at the National Council for Scientific and Technical Research of Argentina (CONICET), working at 'Instituto de Desarrollo y Diseño'. He obtained his PhD degree in Chemical Engineering from 'Universidad Nacional del Litoral' (Santa Fe, Argentina) in 1986 and was a Postdoctoral Fellow at the Massachusetts Institute of Technology (1986–1989). His current research activities focus on software architectures, models for supporting the design process, semantic web applications to supply chain information systems, and enterprise modelling. He has supervised several PhD students.