Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial
ISSN: 1137-3601
revista@aepia.org
Asociación Española para la Inteligencia Artificial
España

Gonnet, Silvio; Leone, Horacio; Henning, Gabriela
Representing and capturing the experts´ knowledge in a design process
Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial, vol. 7, núm. 21, 2003, pp. 37-46
Asociación Española para la Inteligencia Artificial
Valencia, España

Representing and Capturing the Experts' Knowledge in a Design Process

Silvio Gonnet [a], Horacio Leone [a], Gabriela Henning [b]

[a] GIPSI- UTN / INGAR, Avellaneda 3657, 3000 - Santa Fe, Argentina

[b] INTEC, Güemes 3450, 3000 - Santa Fe, Argentina

{sgonnet,hleone}@ceride.gov.ar, ghenning@intec.unl.edu.ar

Abstract

An object-oriented framework to support the modeling and management of the design process is introduced. It naturally integrates the representation of both the design process itself, and the outcomes that are achieved as the result of the various design activities. The integral view of tracing that was adopted not only captures and manages the products being generated but also the activities that occurred, their associated context and the adopted decisions. The Version Administration System introduced in this paper provides an explicit mechanism to manage the different model versions being generated during the course of a design project as design activities are executed.

Keywords: Design rationale, Change management, Modeling languages, Design support systems.

1. Introduction

Development of products in many engineering disciplines is a challenging task. Even for quite different types of products, development processes have strong common characteristics and features, such as the ones listed below:

- Design problems are inherently ill defined; therefore, the structure of the design process is not known in advance. It starts with a small set of requirements that include goals and constraints and evolves through subsequent stages of increasing complexity in a non-linear manner. In most cases there is a lack of a fully articulated methodology, so, there is no clear distinction between solution stages. Furthermore, there might be a need for a backtracking process to change previously adopted decisions.

- During a design process, various models of the artifact being designed are generated. They differ in granularity, complexity and associated assumptions; therefore, there is an explicit need to properly manage model versions.

- Once a design project is finished, the things that remain are mainly "design products" such as the models that were generated, detailed specifications of the resulting artifact, drawings, sketches, etc. However, there is no explicit representation of how they were obtained. More specifically, there is no trace of:

  - Which activity/ies originated a given product?

  - Which requirements were imposed?

- Which actors performed a given activity?
- Which is the underlying rationale behind a decision-type activity?

- Due to their size and complexity or specific needs of expertise, design problems are rarely tackled by individuals, design teams are the usual coin. Thus, human experts along with computer-aided tools are the ones that by interacting cooperatively, sharing resources of various types and design products, solve complex problems.

As a consequence of the features pointed out above there is a real need for support tools that could capture and efficiently manage the solution process. By having such tools, the tracking and tracing of the development process would be possible as well as the analysis of its rationale. In this way, the experts' knowledge could be captured, thus providing the foundations for learning and training activities and for future reuse. It should be adopted a holistic or integral view of tracing, that not only captures and manages the products being generated (e.g. model traceability) but also the activities that occur, their associated context, the adopted decisions as well as the different roles that the distinct design actors assumed during the solution process.

On the other hand, product management systems have been around for a long time, and they are widely used in practice. This is not surprising because they respond to a very basic demand: the products of development processes have to be recorded and organized (Westfechtel, 1998). These management systems and software configuration management systems focus on products, but neglect the tracing of the design processes. In consequence, they do not satisfy the need for keeping consistency and navigability among models (and model's components) identified along the design process.

Depending on the domain being tackled and on the problem at hand, design methodologies can vary. Boyle (1989) suggests a classification that identifies three main categories: analytic, procedural and experimental design. Underneath this characterization there are concepts such design objects, object attributes, operations on objects, as well as the different roles that are assigned to humans and equipment in these classes of design methodologies. This work focuses on procedural design, which is the most frequent one in engineering disciplines. Design is considered as an iterative process that operates under the generate-test-analyze-suggest-modify paradigm. During this development process the artifact being designed is checked against objectives. In general, the design process does not follow a predefined workflow, and cannot be predicted beforehand. Moreover, certain complex tasks are executed interactively by humans/design teams by interplaying with computer support systems.

This contribution tries to address the issues raised above. It is organized as follows. In the next section, the main concepts related with the representation and capture of the design process are presented. Moreover, two different granularity contexts (activity and operation contexts) are first introduced. In section 3, the activity context is described in more detail and in section 4 the operation context is discussed, presenting also a version administration system. Finally, section 5 presents conclusions.

2. Modeling Elements. Representing How The Design Process Is Performed

Design knowledge still rests in the minds of experienced designers, but it is desirable to make it part of a computer support environment. Therefore, it is necessary to have a model of the design process that allows to capture how it has been performed.

Designers react contextually according to the domain knowledge they acquire. Then, the process modeling approach proposed here aims at strongly relating the context where an activity is performed to the activity itself, otherwise some information about the activity would be lost. This approach aims at capturing not only activities performed during the design process but also why and when (the activity context) these activities were done and whom (the actor/s) executed them. On the other hand, activities operate on the results or products of the design process, called *design objects*, that include requirements, the representation of the design artifact itself, and arguments.

In consequence, the design process model has to handle different levels of granularity of contexts. There is an *activity context*, that requires exploring decision making alternatives, and an *operation context* which implements a given decision through the execution of operations which transform the product under development. This originates new contexts, which are themselves subjects of decisions (See Figure 1).

The object-oriented paradigm is used for representing the process model. Furthermore, both the object-oriented paradigm and the situational calculus are employed for modeling the evolution of the design objects. In Figure 1, six main concepts used to model the design process are shown. They

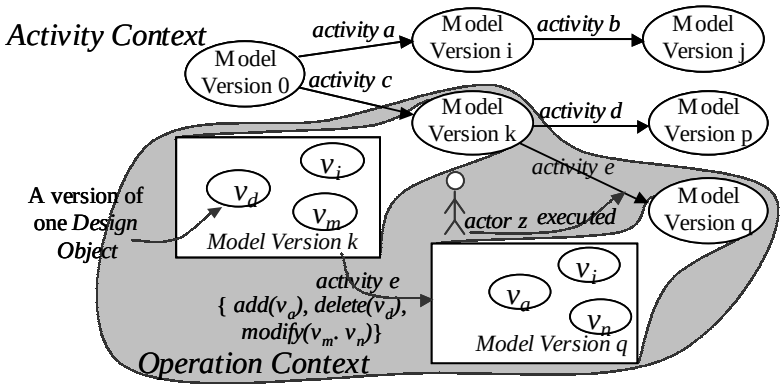

Figure 1. Capturing the design process

are: model version, requirement, design object, activity, operation and actor.

- *Model versions* represent the activity context. They supply a snapshot description of the state of the design process, including the artifact being designed.

- *Requirements* specify the functional and non-functional characteristics that a product must satisfy. They are represented as *design objects*.

- *Design objects* model the different products of the design activities. They evolve as the design process proceeds and their versions form part of one or more *model versions*.

- *Activities* that carry out the design process.

- *Operations* that perform the actual transformation of design objects. Each *activity* is materialized by a set of *operations*.

- *Actors*, whom perform the activities and operations.

3. Activity Context

As it pointed out in the introduction, once a design project is finished, those things that remain are mainly "design products" (e.g. generated models, detailed specifications of resulting artifacts, drawings, sketches, etc.). However, there is no explicit representation of how they were obtained. More specifically, there is no trace of which activity/ies originated a given product, which requirements were imposed, which actors performed a given activity and which was the underlying rationale behind a decision-type activity. Regarding activity types, though it is not within the scope of this paper, it is assumed that activities are identified and classified according to the different types that were presented in Eggersmann et al. (2003). In this section, a process model that allows us to capture a design process and answer the questions previously posed is presented.

3.1. Which activity/ies originated a given product?

The design process is carried out by a set of *activities*, which may be described at various abstraction levels. An *activity* may be decomposed into a set of *sub-activities*, they may be organized according to a schedule or they may be performed without a previous order. The relationships between an *activity* and its *sub-activities* are captured by an aggregation link. In Figure 2, this is depicted using the UML object-oriented paradigm notation (Booch et al., 1999).

The aggregation relationship is transitive; then, using first order logic the transitivity property is expressed as follows:

$$(\forall\, a_1, a_2, a_3)\ subActivityOf(a_1, a_2) \wedge subActivityOf(a_2, a_3) \Rightarrow subActivityOf(a_1, a_3) \quad (1)$$

Where $subActivityOf(a_i, a_k)$ is a predicate that means that a_i is a sub-activity of a_k.

The following two axioms state that an activity cannot be a sub-activity of itself, and it is never the case that an activity is a sub-activity of another activity which, in turn, is a sub-activity of the first one. This shows that the relation *subActivityOf* is non-reflexive and anti-symmetric:

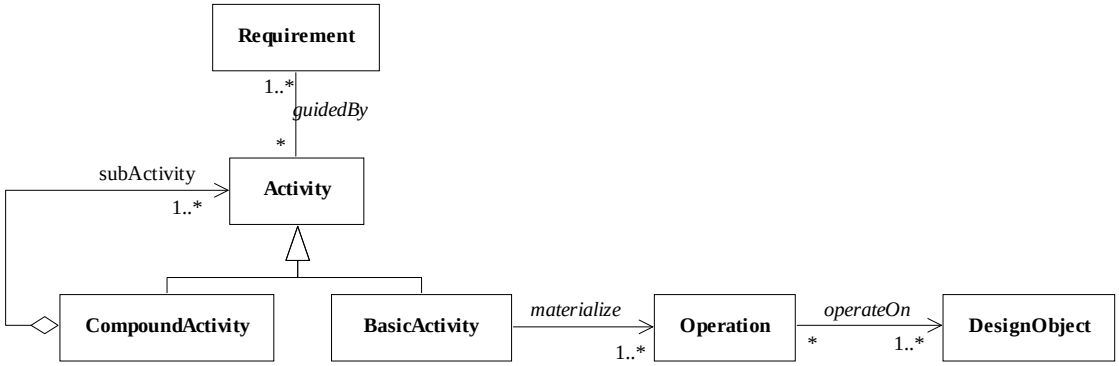

Figure 2. Activity decomposition and relationship with design products by means of operations

$$(\forall\, a)\ \neg subActivityOf(a, a) \qquad (2)$$

$$(\forall\, a_1, a_2)\ subActivityOf(a_1, a_2) \Rightarrow \neg\, subActivityOf(a_2, a_1) \qquad (3)$$

Moreover, an activity cannot be a sub-activity of two or more distinct activities that are not sub-activities of each other:

$$(\forall\, a_1, a_2, a_3)\ subActivityOf(a_1, a_2) \wedge subActivityOf(a_1, a_3) \Rightarrow a_2 = a_3 \vee subActivityOf(a_2, a_3) \vee subActivityOf(a_3, a_2) \qquad (4)$$

The recursive decomposition of *sub-activities* leads to an overall activity structure. Taking into account the *subActivityOf* relationship, activities are classified into *basic* and *compound activities*. The activity structure bottoms out in activities that are not further decomposed and are, therefore, called *basic activities* (See Figure 2).

$$(\forall\, a)\ basicActivity(a) \Leftrightarrow \neg(\exists\, a')\ subActivityOf(a', a) \qquad (5)$$

Basic activities are materialized in a sequence of operations ϕ. This fact is represented by the predicate *materialize*(ϕ, *a*).

$$(\forall\, a)\ basicActivity(a) \Leftrightarrow (\exists\, \phi)\ materialize(\phi, a) \qquad (6)$$

An *operation* is the basic transformational action primitive which represents actions on *design objects*. Operations prescribe how the design domain is changed. They are specified in more detail in the section entitled operation context.

A compound activity cannot be a leaf node in the activity hierarchy; thus any activity that is regarded as a compound activity is not a basic one.

$$(\forall\, a)\ compoundActivity(a) \Leftrightarrow \neg\, basicActivity(a) \qquad (7)$$

3.2. Which requirements were imposed?

The design process may be interpreted as a series of activities guided by *requirements*, specifying the functional and non-functional characteristics that a product must satisfy. Furthermore, *requirements* may prescribe constraints on the design process. Generally, they are specified as goals or constrains. Often *requirements* may not be stated explicitly or in sufficient detail at the beginning of the design process (Brown and Chandrasekaran, 1989). They are refined and specified more precisely as greater comprehension of the design problem is reached (Boyle, 1989; Goel, 1994). Then, it is very important to represent how *requirements* evolve during a project execution. This is analyzed in section 4, where a *requirement* is represented as a *design object*. In the activity context it is possible to recognize which requirements guided an activity through the relationship *guidedBy* (Figure 2). Conversely, this relationship allows us to know which activities were performed with the aim of satisfying a given *requirement*.

3.3. Which actors performed a given activity?

As it was previously mentioned, activities are performed by actors with a goal in mind. The

process model presented in this paper extends the actor model introduced by Eggersmann et al. (2001). This extension is made with the aim of answering the question "Which actors performed a given activity?". Indeed, each *activity* is related to an *actor* who executes it (*execution* relationship in Figure 3). An *actor* may be either an *individual* (a human or computational program) or a *team*. *Teams* are composed of *actors*, *individuals* and/or other *teams*. *Teams* allow to represent compound *skills* that are needed for performing activities. They are not organizational units, because the present work focuses on process support/process tracing.

Each *actor* may have *goals*, called *actor's goals*, which express the *actor's* intentions and desires. These goals may usually be described in terms of the desired product, but there is a part of them harder to describe because it is not directly related to the outcome of an activity. It may, for example, deal with deadlines or even fuzzy qualifications such as "as fast as possible". Therefore, this part is often represented as text. Then, an *actor's goal* may be modeled as an aggregation of goals about products and fuzzy goals. So, an *actor's goal* may be decomposed into a set of subgoals.

The actor's decision of executing a given activity for reaching one or more goals with the final aim of satisfying a set of requirements is represented by the *promote* links among *activity*, *actor's goal* and *requirement* (Figure 3). These links reflect the actor's intention and can be used to evaluate whether the work done did really satisfy the goal, at least partially. Moreover, the model shown in Figure 3 allows to represent the fact that *activities* are executed by those *actors* having the necessary *skills* to carry them out. As seen, the *actor*, *activity* and *skill* classes are connected to each other. The link between *activity* and *skill* represents the skills required to execute the activity. The *activity – actor* association models who performed a given activity. Finally, the *individual actor – skill* link represents the know how of a particular actor.

3.4. Which is the underlying rationale behind a decision-type activity?

With the aim of representing the rationale associated with the execution of a given activity, the IBIS model (Rittel and Kunz, 1970) is refined in this paper. The IBIS model focuses on articulating key design *issues*. An *issue* is a question to be answered and a *position* is an alternative which exists for solving such issue. We refine this view by introducing *requirements*, which specify *issues*, and also by decomposing *positions* into *artifacts*, *attributes*, and *values* (see Figure 4). An *artifact* represents the product that it is being designed, and *attributes* and *values* characterize the *position*. Then, the different alternative products that arise in the design process are represented by the *position* concept. A *position* is qualified by one or more *arguments* and addresses at least one *requirement*. An *argument* either supports or objects a *position*. It allows to test whether the *position* is capable of fulfilling the prescribed *requirements* by the *answer* relationship.

Positions, *artifacts*, *attributes*, *values* and *arguments* evolve during the execution of a design project and their various states are fundamental for representing the different contexts where an *activity* is performed. Then, they are represented as *design objects* (see Figure 5), and the operation context section describes how their evolution is represented during the design process.

Activities have the goal of designing a product that is specified by a set of *requirements*. These activities generate *artifacts* which are part of *positions* and which may be missing relevant information. A *position* encompasses a design *artifact* (such as a chemical reaction pathway, a flowsheet structure, a mathematical model, etc.), its *attributes* and corresponding *values*. Activities performed after synthesis activities (Eggersmann et al., 2003) which have generated *positions*, allow the enlargement of such *positions* by refining the

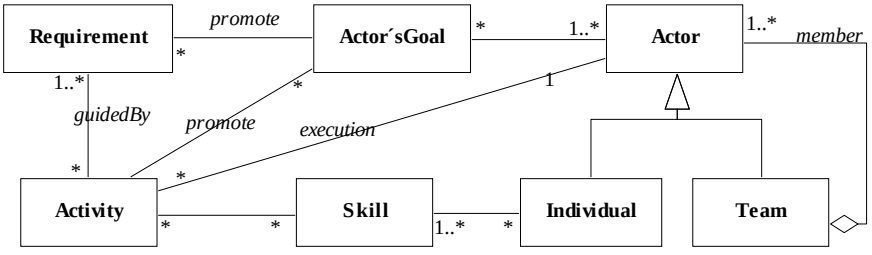

Figure 3. Actor's Model

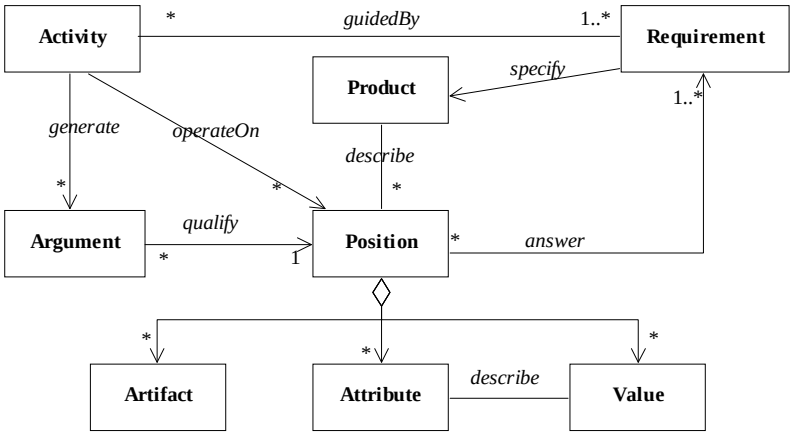

Figure 4. Model that captures the relationships among the various Design Objects, allowing to uncover the rationale behind a decision-type activity

artifacts and providing *attributes* and *values* in order to have enough information to carry out *decision activities*, which in turn use *requirements* and *arguments* to select *positions*. Requirements are also used during other types of activities to indicate the most important aspects to focus on. As seen, requirements are used in every activity, but with a different purpose or aim. Thus, some activities attempt to answer the question: "How can requirements be fulfilled?" and other ones supply data that would allow to check whether requirements are met or not. Finally, *decision activities* weigh up requirements, establish which are the most important ones and test if these requirements are indeed met. Furthermore, as indicated by Eggersmann et al. (2003) requirements can be generated during the design process by any of the activities that are executed.

4. The Operation Context

Activities operate on the outcomes or products of the design process, called *design objects*. A *design object* (Figure 5) represents any entity that can evolve during a design project. It is represented in 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 due to model evolution during a design project. This object is called *versionable object* (*o*).

Furthermore, relationships among the different *versionable objects* are maintained in the *repository*. These relationships correspond, according to the notation being used, to the rules that allow associating objects to form syntactically valid models. This is captured by the *association* predicate and the link named *association* in the object model shown in Figure 6. Between any two *versionable objects* o_i and o_j, *association*(o_i, o_j, r_k) means that o_i is linked to o_j by the relationship r_k.

On the other hand, the *versions' level* keeps the different versions of each *design object*. These are called *object versions* (*v*). The relationship between a *versionable object* and one of its *object versions* is represented by the predicate *version*. Thus, *version(v, o)* means *v* is a version of *o*. Therefore, a given *design object* keeps a unique instance in the *repository* and all *versions* it assumes in different *model versions* belong to the *versions' layer*.

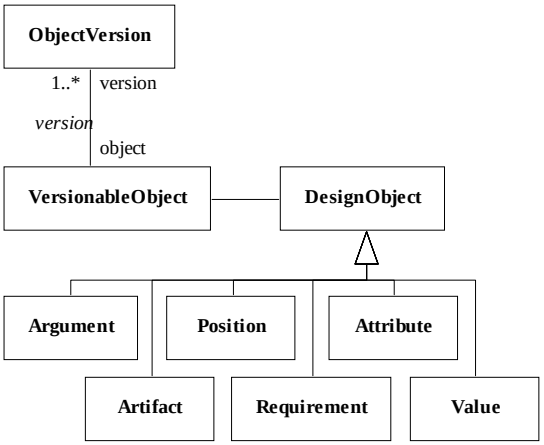

Figure 5. Design Objects´ Model

At a given stage during the execution of a design project, the states assumed by the set of relevant *design objects*, from now on called *model version*, supply a snapshot description of the state of the design process, including the artifact being designed. Since the situational calculus (Reiter, 2001) is used for modeling the version generation process, the model evolution is posed as a history made up of discrete situations. A new *model version* m_n is generated when one activity *a* (a basic activity) is executed. Activity *a* is materialized by a sequence of operations ϕ (expression 6) and the new *model version* m_n is the result of applying such sequence ϕ to the components of a previous *model version* m_p. This is achieved by performing the following evaluation: $apply(\phi, m_p) = m_n$. The *apply* function is defined as follows:

$$apply: \Phi \times M \rightarrow M \qquad (8)$$

Where Φ is the set of all possible operation sequences ϕ, and *M* is the set of possible *model versions m*. A sequence of operations ϕ is defined as follows:

$$\phi = \begin{cases} \lambda, \text{ empty sequence} \\ o \bullet \phi, \text{ where } o \text{ is an operation} \end{cases} \qquad (9)$$

Then, the inductive definition of the *apply* function is given by:

$$\begin{gathered} apply(\lambda, m) = m \\ apply(o \bullet \lambda, m) = m', m \neq m' \\ apply(o \bullet \phi, m) = apply(\phi, apply(o \bullet \lambda, m)) \end{gathered} \qquad (10)$$

The primitive operations first proposed to represent the transformation of *model versions* are *add*, *delete*, and *modify*. 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 exists 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 the *model version* that arises after applying to *model version m* the sequence of operations ϕ, if and only if:

(i) *v* is added when the new *model version* is created ($add(v) \in \phi$ or $modify(v_p, v) \in \phi$);

or

(ii) *v* belonged to the previous *model version m* and it is not deleted when ϕ is applied ($delete(v) \notin \phi$ or $modify(v, v_s) \notin \phi$).

From these definitions and by using the format of successor state axioms proposed by Reiter (2001), it is presented a formal specification of the cases in which an *object version* belongs to a *model version*. In the next expression, the predicate *belong(v, m)* is true when the *object version v* belongs to the model version *m*.

$$\begin{gathered} (\forall \phi, v, v_p, v_s, m)\ belong(v, apply(\phi, m)) \\ \Leftrightarrow \\ (add(v) \in \phi \vee modify(v_p, v) \in \phi \vee \\ belong(v, m)) \wedge \\ (delete(v) \notin \phi \wedge modify(v, v_s) \notin \phi) \end{gathered} \qquad (11)$$

From this expression, the *object versions* that belong to a *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.

$$\begin{gathered} m_{i+1} = apply(\phi_i, m_i) \\ m_i = apply(\phi_{i-1}, m_{i-1}) \\ \ldots \\ m_1 = apply(\phi_o, m_o) \\ \\ m_{i+1} = apply(\phi_i, apply(\phi_{i-1}, apply(\ldots \\ apply(\phi_o, m_o)\ldots))) \\ \\ m_{i+1} = apply(\phi_o \bullet \ldots \bullet \phi_{i-1} \bullet \phi_i, m_o) \end{gathered} \qquad (12)$$

Where $\phi_i \bullet \phi_j$ is the concatenation of sequences ϕ_i and ϕ_j.

Then, the relationship existing between the two levels (*repository* and *versions' level*) and a *model version* may be expressed by the following definition: each *versionable object* has one or more *object versions* associated to it at a given time instant, but at most, one *object version* associated to a given *versionable object* can belong to a particular *model version*.

Once the object versions conforming a model version are defined, the relationships existing among them have to be specified. It should be noted that in this proposal, object versions belonging to a model version are not explicitly associated to other object versions of the same model version. On the other hand, as it was previously mentioned, the objects of the repository store the information of all the objects they have been related to during the various model versions ($association(o_i, o_j, r_k)$: o_i is linked to o_j by the relationship r_k.). Consequently, the link existing between two object versions must be inferred from

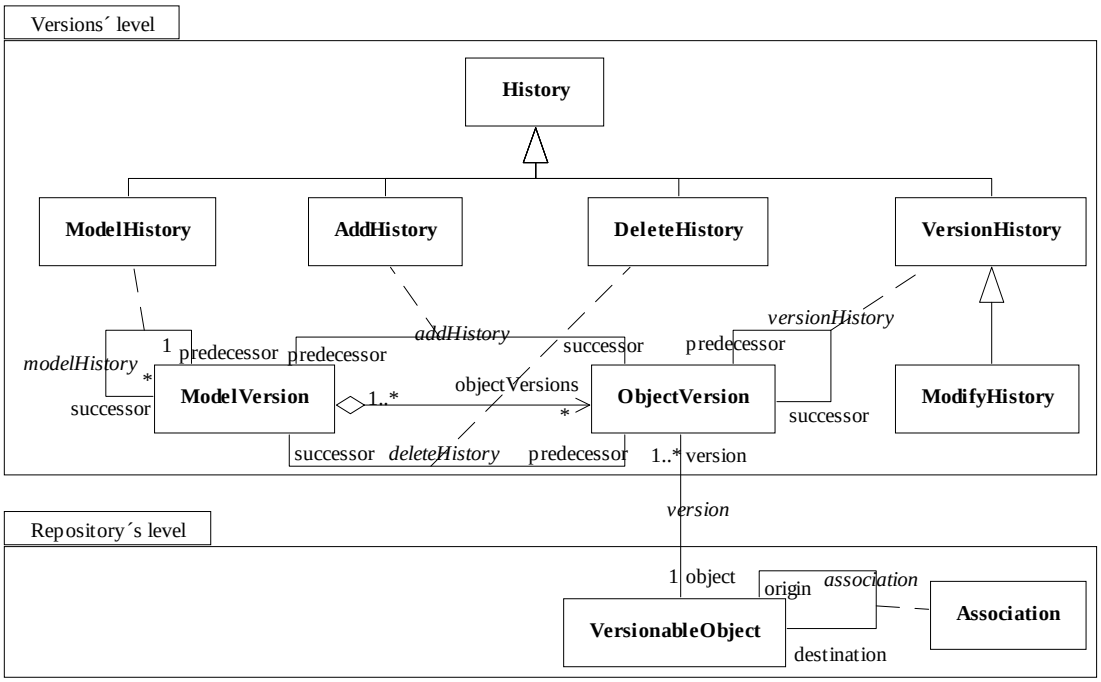

Figure 6. Version Administration Metamodel

the relationship established between the objects that have been versioned by them. Thus, an association r_k among two *object versions* v_1 and v_2 exists in the model version m_i (*associationInf*(v_1, v_2, m_i, r_k)), if and only if there exists the association r_k among the versionable objects of which v_1 and v_2 are version of (*association*(o_1, o_2, r_k), *version*(v_1, o_1) and *version*(v_2, o_2)). This fact is presented by expression 13.

$$(\forall\, v_1, v_2, r_l, m_i)\ associationInf(v_1, v_2, m_i, r_l) \Leftrightarrow (\exists\, o_1, o_2)\ belong(v_1, m_i) \wedge belong(v_2, m_i) \wedge version(v_1, o_1) \wedge version(v_2, o_2) \wedge association(o_1, o_2, r_l) \quad \textbf{(13)}$$

Having introduced the representation of the set of *versions* of a model, we will explain the way in which *navigability* capacity is provided. The proposed scheme uses the situational calculus to represent whether a version *v* belongs to a certain *model version* or not and to allow for the reconstruction of a particular *model version*. In addition, this scheme is strengthened by the object-oriented paradigm, which models the relationships existing among *object versions* of different *model versions*, allowing navigation along the history of the *object versions* constituting a given *model version*.

The relationships among *object versions* are represented by means of explicit links at the *Versions' level*, named *add history*, *delete history* and *version history* associations (Figure 6). Each transformation operation that is applied to a *model version* incorporates the necessary information to trace the model evolution. This information is represented by relationships between the *object versions* the operation is applied to and the new ones arising as a result of its execution.

4.1. Which operation/s originated a given version?

There are other frequent *operations* that must be represented if the history of the changes performed in the *model version* is to be maintained. For example, a frequent operation in modeling is the one that allows decomposing an entity into one or more entities. In this context, such operation is called *refine* and it allows the object versions (ψ: set of *object versions*) that implement the refining to appear in the new *model version*. The reverse operation is *simplify*. By means of this operation, a structure of *object versions* (ψ) becomes an *object version* (v). To start with, these two operations could be defined in terms of *add* and *delete*. However, it is necessary to make a distinction with the aim of keeping the history of the changes carried out in the *model version*.

The *refine(v,ψ)* operation is expressed in terms of a series of *add* and *delete* operations according to the following expression.

$$\begin{gathered}\phi = \phi_1 \bullet refine(v, \psi) \bullet \phi_2 \\ \Rightarrow \\ (\forall\, v_r \in \psi,\ add(v_r) \in \phi_3 \wedge delete(v) \in \phi_3) \wedge \\ (\phi = \phi_1 \bullet \phi_3 \bullet \phi_2)\end{gathered} \quad \textbf{(14)}$$

The possibility of expressing compound operations in terms of basic operations, as proposed in expression (14), allows keeping the successor state axiom presented in (11) valid, without having to bring it up to date when adding each new operation.

Expression (14) fails to express the relationship that exists between the *object version v* and the *object versions* v_r belonging to ψ. This relationship is modeled through the object-oriented paradigm. An *object version* is associated with one or more *predecessor object versions* and one or more *successor object versions*. This association is called *version history* (Figure 7) and it is the association to be specialized to define the different operations. In the case of the *refine* operation, *version history* specializes in *refine history*, where its *predecessor* is an *object version* and its *successors* are one or more *object versions*.

Analogously to the *refine* operation, the *simplify(ψ,v)* operation is defined, which is expressed in the following expression.

$$\begin{gathered}\phi = \phi_1 \bullet simplify(\psi, v) \bullet \phi_2 \\ \Rightarrow \\ (\forall\, v_s \in \psi,\ delete(v_s) \in \phi_3 \wedge add(v) \in \phi_3) \wedge \\ (\phi = \phi_1 \bullet \phi_3 \bullet \phi_2)\end{gathered} \quad \textbf{(15)}$$

As it can be seen, the successor state axiom presented in expression (11) still remains valid. Expression (15) has the same weakness pointed out for expression (14) since it does not express the relationship that exists between the *object versions* v_s belonging to ψ and the *object version v*. In this case, *version history* specializes in *simplify history*, where its *predecessors* are one or more *object versions* and its *successor* is an *object version* (see Figure 7). Gonnet and Leone (2001) have presented an *operation model* that implements the basic operations using the Command design pattern (Gamma et al., 1995) and allows the extension of the set of operations in a flexible form, without having to change the existing classes. The *operation model* implements the semantics of the basic operations (*add*, *delete*, *modify*) employed in the successor state axiom that was defined in expression 11, and defines an abstract operation that must be specified when a new operation is added. The specification is made in terms of the basic operations that were defined. Therefore, the operations *refine* and *simplify* specialize the *operation model*.

In a similar way to the *refine* and *simplify* operations, other operations identified in the design process can be defined. For instance, the possible operations for decision-type activities (where a *position* is selected, rejected, or kept in mind as a possible alternative), include *select*, *evaluate*, *justify*, and *request*. A *select* operation refers to the choice of one or more design products from a number of possible alternatives. Before, some information generated during previous activities is compared with the requirements to fulfill. Thus, an *evaluation* operation provides *arguments* to justify a decision. Similarly, *justify* offers a rationale for the selection of a certain alternative. The operation *request* solicits additional information (generates a new *requirement*), allowing a decision to be interactive.

These operations must be represented if the history of the changes and its rationale are to be maintained. For example, *evaluate*(v_a, v_p) generates a new argument v_a qualifying a certain position v_p. The *evaluate*(v_a, v_p) operation is expressed in terms of the *add* operation according to the following expression.

$$\begin{gathered}\phi = \phi_1 \bullet evaluate(v_a, v_p) \bullet \phi_2 \\ \Rightarrow \\ add(v_a) \in \phi_3 \ \wedge (\phi = \phi_1 \bullet \phi_3 \bullet \phi_2)\end{gathered} \quad \textbf{(16)}$$

Evaluate history specializes *version history*; its predecessor is an *object version* (v_p) and its *successors* are two *object versions* (v_a, v_p).

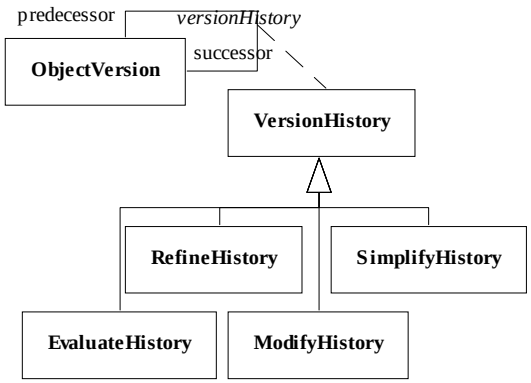

Figure 7. Specializing Version History with the various operations

5. Conclusions

This work proposes a framework for representing and capturing the design process. This is a fundamental phase for developing computational tools to support the design process and to guide designers in the different activities of a design project. The framework is defined in terms of metamodels that allow the representation of the executed design process and the evolution of the different design objects that participated in it. Design objects may be design products as well as the requirements that specified them, or argumentations and goals posed by actors when they performed a given activity.

Metamodels can be specialized according to the particular domain being tackled. It can be done in terms of the different *operations* that are applied to the distinct *design objects*, and in terms of the different *design objects* that participate in the design process. For example, a *user's goal* may be specialized, as in Eggersmann et al. (2001), with the aim of representing complex goals and their decomposition. Another possible specialization of a *design object* is the one of *requirements*, to represent their structure, as it is proposed in Lin et al. (1996).

Situational calculus in conjunction with the object-oriented paradigm let us represent experts' knowledge and their particular rationale in relation to a given operation they applied. On the other hand, the extension of the IBIS model allowed us to model, at a higher level, the rationale behind a decision taken during the design process. Thus, the proposed tools allow the tracing of the design process and its resulting products, as well as the analysis of the reasoning line employed during such process, setting the grounds for learning and future reuse.

References

1. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, Addison Wesley (1999).

2. Boyle, J-M.: Interactive engineering system design: a study for artificial intelligence applications. Artificial Intelligence in Engineering, 4, (1989) 58-69.

3. Brown, D. and Chandrasekaran, B.: Design Problem Solving. Knowledge Structures and Control Strategies. Pitman (1989).

4. Eggersmann, M., Henning, G., Krobb, C., Leone, H. and Marquardt W.: Modeling of actors within a chemical engineering work process model, Proceedings International CIRP Design Seminar, Stockholm, Sweden, 6-8 June (2001) 203-208.

5. Eggersmann, M., Gonnet, S., Henning, G., Krobb, C., Leone, H. and Marquardt W.: Modeling and understanding different types of process design activities, Latin American Applied Research, 33, (2003) 167-175.

6. Gamma, E., Helm, R., Johnson, R., Vlissides, K.: Design Patterns. Elements of Reusable Object-Oriented Software, Addison Wesley (1995).

7. Goel, V.: A comparison of design and nondesign problem space. Artificial Intelligence in Engineering, 9, (1994) 53-72.

8. Gonnet, S. and Leone, H.: A Framework for Model Version Management in a Design Process, Proceedings 13th International Conference on Software Engineering and Knowledge Engineering SEKE´ 01, Knowledge Systems Institute (2001) 260-267.

9. Lin J., Fox M., Bilgic T.: A Requirement Ontology for Engineering Design, Concurrent Engineering: Research and Applications, 4, (1996), 279-291.

10. Reiter R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press, (2001).

11. Rittel, H.W.J., Kunz, W.: Issues as elements of information systems, Institute of Urban and Regional Development. Working Paper 131, Univ. of California, Berkeley (1970).

12. Westfechtel, B.: Models and Tools for Managing Development Processes, Lecture Notes in Computer Science Vol. 1646, Springer-Verlag (1998).

Acknowledgements

This work was sponsored by Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Universidad Tecnológica Nacional and Universidad Nacional del Litoral. Authors gratefully acknowledge help received from these institutions.