Advances in Engineering Software 41 (2010) 1087–1109

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

TracED: A tool for capturing and tracing engineering design processes

María Luciana Roldán *, Silvio Gonnet, Horacio Leone

CIDISI – INGAR/UTN – CONICET, Avellaneda 3657, 3000 Santa Fe, Argentina

ARTICLE INFO

Article history:
Received 14 November 2008
Received in revised form 25 June 2010
Accepted 28 June 2010
Available online 31 July 2010

Keywords:
Design process support
Design history
Software architectures
Knowledge acquisition
Architectural design decisions
Operational approach

ABSTRACT

The design of products or production processes in many engineering disciplines such as chemical, or software engineering, involves many creative and sometimes unstructured activities, with an opportunistic control flow. During these design processes, several models are generated, which have different levels of abstraction of the object being designed. Given the difficulties in dealing with this complexity using an improvised way, there is an urgent need for tools that support the capture and tracing of this process. In this proposal, TracED, a computational environment to support the capture and tracing of engineering design process is presented. It allows defining a particular engineering design domain and supporting the capture of how products under development are transformed along an engineering design process. Particularly, in this work, we consider software architectures design domain. As in any complex process, the support of computational tools is required for enabling its capture.

1. Introduction

Product and production process design in many engineering disciplines such as chemical, or software engineering is a challenging task. Over the last 30 years, the engineering design process has been transformed by the introduction of massive computing power, where problem-solving environments (PSEs) play a main role. PSEs aim to assist engineers in solving several complex tasks. A design PSE must be able to handle all requirements of a design team and to integrate with the stages of the design process. Moreover, it must be configurable by the design team itself to suit each new problem as it is tackled [1]. The design PSE should be flexible and should not constrain the task of the designer in an unnatural manner [2]. In order to assist engineers during the design process, it is necessary to understand such a process and to have a computer representation of it. However, design problems are ill-structured, because the designer does never have enough information in the initial state and the properties of the goal state are never fully specified in advance. Therefore, many different goal states are conceivable and acceptable [3,4]. All this become designing in a nondeterministic process, which is difficult to model and even more difficult to prescribe [5]. As a consequence of the previously pointed out features, there is a real need of supporting tools that could capture how an engineering design process was carried out. By having such tools, the tracking and tracing of the design process would be possible, as well as the analysis of its rationale. In this way, the design experts' knowledge could be captured, thus providing the foundations for learning and training activities and future reuse.

Several attempts to provide support to the design process in different engineering domains have been reported [6–8]. Some tools are based on design reasoning capture by means of the concepts proposed by the IBIS model [9]. Another line of research related to design is the management of development processes products (where products are models, data, diagrams, etc.). For some time now, there have been widely used systems for managing products and their versions [10]. This practice responds to the basic need of storing and organizing the products of a development process [11]. These management systems, like software configurations-managing systems, are focused on products, and they do not consider the design process tracing. Consequently, these tools do not satisfy the need of capturing the design process together with its reasoning.

In this paper, *TracED*, a computational environment to support the capture and tracing of engineering design process is proposed. The environment is based on a generic model for capturing the design process in terms of the operations applied to the design objects [12]. Its goal is the capture of the developed model versions during a design process; in other words, it represents the design states and how they were obtained. The environment was designed with the requirement of supporting various design domains, therefore, *TracED* could be adapted to each new design problem, according to the particular concepts of a given design domain and the possible operations that can be applied over the instances of those concepts. For example, regarding software engineering, one of the most important stages is the software architecture design process

* Corresponding author. Tel.: +54 342 4534451; fax: +54 342 4553439.
E-mail addresses: lroldan@santafe-conicet.gov.ar (M.L. Roldán), sgonnet@santafe-conicet.gov.ar (S. Gonnet), hleone@santafe-conicet.gov.ar (H. Leone).

0965-9978/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2010.06.006

(SADP), to which research and industry communities have paid special attention in the last decade [13]. The software architecture of a computing system is the structure of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [13]. It is the first artefact that may be analyzed to determine how well software quality attributes (performance, modifiability, availability and so on) are being achieved. In addition, the software architecture serves as mean of communication, is the manifestation of earliest design decisions, and is a re-usable abstraction that can be transferred to new systems. It gives a high return on investment because decisions made for the architecture have substantial downstream consequences and because checking and fixing a software architecture is relatively inexpensive. As other complex engineering design processes, SADP involves several activities such as exploration, evaluation, and composition of design alternatives [14,15]. In order to address these activities, the research community has been working intensively in the achievement of modelling languages [16,17], design methods [13] and computer environments for architect assistance [18–20]. Those tools are focused on assisting designers in generating a software architecture design to satisfy a set of requirements. However, regular design tools often left out documentation of associated rationale, design decisions and applied knowledge. These omissions stem from the fact that such information may be intuitive or obvious to the architects involved in the design process, or from the lack of adequate computer-aided environments that allow them to support the design process and the software maintenance (to fix bugs, to implement new functionalities, etc.). Thus, most architectural design knowledge and architectural design decisions made through SADP remain in the minds of experienced designers, and are lost over time. Consequently, capturing design decisions is highly important to capitalize previous designs. To this aim, this contribution assumes an operational perspective where design decisions are modelled by means of design operations.

The remainder of this article is organized as follows. In Section 2, a conceptual and generic model for capturing and tracing a design process is presented. The proposed model employs a versioning administration approach based on an operational perspective. It is not intended for a specific domain; on the contrary, it can be applied to different domains such as software [21] and chemical engineering [12]. Then, this section introduces suitable extensions for making it applicable to the software architecture design process. Therefore, the model is outlined in an object-oriented approach to provide the foundations for developing a computational tool that enables capturing and tracing a design process; particularly the definition of the necessary concepts and operations for SADP domain are included. Afterwards, the core tools of TracED are presented in Section 3. This section describes how a particular engineering design domain can be defined on TracED, and then, by using it, the several products of a design process can be captured and traced. As a conducting example, we present a case study on the software architectures domain, in order to illustrate the approach. In Section 4, related research is analyzed. Finally, conclusions and an outline for further work round out this article in Section 5.

2. A model to capture and trace the engineering design process

As it was introduced in the previous section, it is necessary to have a computational environment to support the capture and tracing of engineering design processes. As a result, it would be possible to think about the reuse of parts of previous projects to modify, extend, and integrate it according to new needs, as stated by Concheri and Milanese [22]. They also denote that in order to provide such features, all the information involved and used in the design process should be formalized and modelled in a suitable format for automatic data processing. Therefore, the scheme to capture and trace an engineering design process proposed in this paper is a mixed approach that combines object-oriented technology and situational calculus [23,24]. This choice is justified by the following reasons: (i) object-oriented approaches make the knowledge representation task much simpler because they reflect a more natural view of the domain to be modelled, and the model extension requires no strategic changes in the structure of the knowledge base itself; (ii) by means of the situation calculus, the evolution of the products of a design project is represented. The situation calculus is a first-order language for representing changes, sometimes enriched with some second-order features [24]. The basic concepts are *situations*, *actions*, and *fluents*. Briefly, actions are what make the dynamic world change from one situation to another. Fluents are situation-dependent functions used to describe the effects of actions. Possible world histories, which are sequences of actions, are represented by first order terms called situations. Situation calculus is perfectly suitable to model dynamic worlds, which, in this case, is the evolution of a model in an engineering design process. This evolution is specified by using successor state axioms, first order formulas that encode the axioms about how the products of a design project evolve. Therefore, situation calculus is used to formalize the evolution of a design process and object-oriented technology to represent the main concepts to enable the construction of computational tools that implement the formulated specification.

The proposed scheme considers the design process as a sequence of activities that operate on the products of the design process, named *design objects*. Typical *design objects* are models of the artefact being designed (i.e. an information system, an industrial piece of equipment, or a chemical process plant), specifications to be met (i.e. stream purity specs, products' throughput for process system engineering, quality attributes such as modifiability or performance for software engineering), variable values (i.e. reflux ratios, number of stages of a separation unit, operating temperatures and pressures, etc.). Naturally, these objects evolve as the design process takes place, giving rise to several versions that must be kept. These versions may be considered as snapshots taken to design objects at a given point of time; and the set of those versions conform a *model version*. A *model version* describes the state of the design process in that time, including the artefact being designed. For example, Fig. 1 partially shows two model versions, m_k and m_q. Both model versions are the result of a fragment of a SADP and include the structure of the artefact being designed. The example corresponds to the case study described in Section 3.

In this scheme, each *model version* is generated by applying a *sequence of operations* on a *predecessor model version*. The sequence of operations may include the elimination, creation, and modification of *versions* that constitute the *predecessor model version*. As it is exemplified in Fig. 1, the model version m_q is obtained from the model version m_k by deleting the *WebApplication* component and adding the components *View*, *Model*, and *Controller*, with their ports and connectors. Therefore, all *model versions* may have zero or more *successor model versions* and must have only one *predecessor model version* (except for the *initial model version*, which does not have a *predecessor model version*). Consequently, the representation scheme of versions is a tree structure, where each *model version* is a node and the root is the *initial model version*. This tree structure is illustrated in Fig. 1. There, it is possible to see the several *successor model versions* that have been proposed (*Model Version* m_i, *Model Version* m_k) from the *initial model version* (*Model Version* m_0), by applying the sequence of operations ϕ_i and ϕ_k, respectively. This model evolution is posed as a history made up of discrete situations. The situation calculus [23,24] is adopted

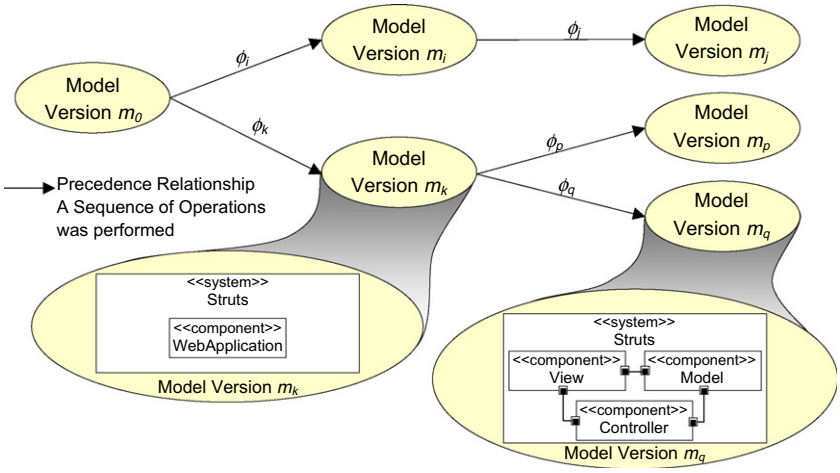

Fig. 1. Tree-structured representation of a design process.

for modelling such version generation process. In the context given by the design process, it is possible to assimilate each new generated *model version* with a *situation*, and each *action* with a *sequence of operations*, which is applied on a predecessor model version. Therefore, the new model version m_q is achieved by performing the following evaluation: $apply(\phi_q, m_k) = m_q$, where ϕ_q represents the sequence of operations applied on the predecessor model version m_k.

The general scheme used in such approach represents a *design object* at two levels, the *repository* and the *versions level*. Each *model version* is generated from views of a *repository* that keeps all the objects that have been created and modified due to the model evolution during a design project. The elements constituting the *repository* are called *versionable objects*. A *versionable object* (*VersionableObject* in Fig. 2) represents the artefact that can evolve during a design project, whose history is desirable to be kept during the modelling process. Furthermore, relationships among the different objects are maintained in the *repository* (represented by *Association* association class, Fig. 2).

At the *versions level*, the evolution of *versionable objects* contained in the *repository* is explicitly represented. A *model version* consists of a set of instances of *object versions* (*ObjectVersion* in Fig. 2), which represent the versions of the objects that compose a given model at a time point. Therefore, a *versionable object* keeps a unique instance in the *repository*, and the *versions* that it assumes in different *model versions* belong to the *versions level*. The class diagram [25] illustrated in Fig. 2 shows the main concepts. The relationship between a *versionable object* and one of its *object versions* is represented by the *version* relationship. Each transformation operation applied to a *model version* incorporates the necessary information to trace a model evolution. This information is represented by *VersionHistory* relationships between the *object versions* to which the *operation* is applied and the ones arising as the result of its execution (Fig. 2). In order to represent engineering design evolution, a model version has zero or more successor model versions (noted by $*$ cardinality at *successor* role of *ModelHistory* association shown in Fig. 2).

This domain-independent model for version administration offers primitive operations to represent the transformation of *model versions*: *add*, *delete*, and *modify*. By using the *add*(v) operation, an *object version* v, that did not exist in a predecessor *model version*, can be incorporated into a successor *model version*. Conversely, the *delete*(v) operation eliminates an *object version* v that existed in the predecessor *model version*. In addition, 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 is *added* after applying the sequence of operations φ to *model version* m when the new version v is created by means of an *add* or *modify* operation (Expression 1). On the other hand, Expression 2 represents the fact that an *object version* v is *deleted* after applying the sequence of operations φ to *model version* m when the version v is deleted by the *delete* or *modify* operation.

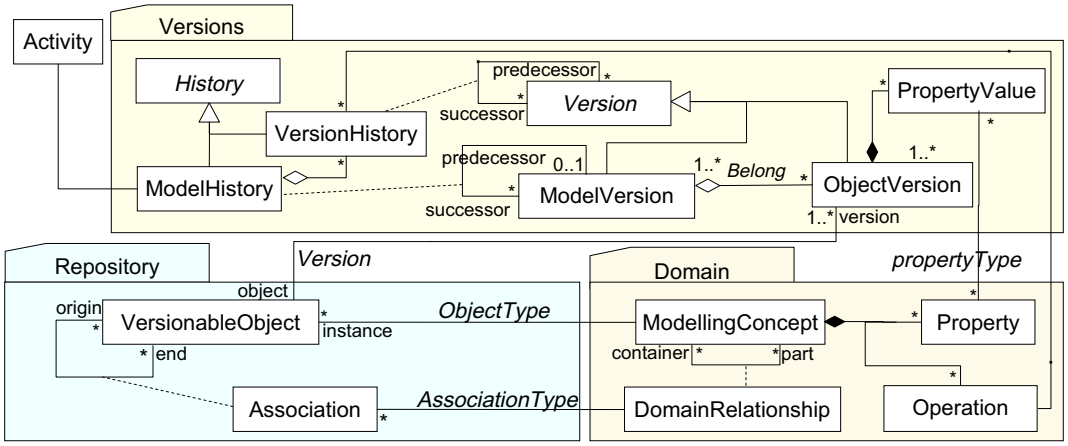

Fig. 2. Version administration model.

$$(\forall \phi, v, m)(add(v) \in \phi \vee (\exists v_p) modify(v_p, v) \in \phi) \Rightarrow added(v, apply(\phi, m)) \quad (1)$$

$$(\forall \phi, v, m)(delete(v) \in \phi \vee (\exists v_s) modify(v, v_s) \in \phi) \Rightarrow deleted(v, apply(\phi, m)) \quad (2)$$

From these definitions, and using the format of successor state axioms proposed by Reiter [23], a formal specification of the cases in which an *object version* belongs to a *model version* is presented. In Expression 3, the predicate *belong*(*v, m*) is true when *object version v* belongs to model version *m*. 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: (i) *v* is added when the new *version* is created; (ii) *v* already belonged to the predecessor *model version m* (*belong*(*v, m*)) and it is not deleted when φ is applied. From this expression, the *object versions* belonging to a *model version* can be specified. Then, it is possible to reconstruct a *model version* m_{i+1} by applying all operation sequences from the initial *model version* m_0.

$$(\forall \phi, v, m) belong(v, apply(\phi, m)) \iff (belong(v, m) \vee added(v, apply(\phi, m))) \wedge (\neg deleted(v, apply(\phi, m))) \quad (3)$$

Once the versions belonging to a *model version* are defined, the relationships existing among *object versions* have to be specified. In this proposal, *object versions* belonging to a *model version* are not explicitly associated to other *object versions* belonging to the same model version. These links are represented at the repository level. Consequently, the relationship existing between two object versions must be inferred from the relationship established between the versionable objects that have been versioned. This inference is represented by Expression 4, 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)).

$$(\forall v_1, v_2, m, a_k) inferredAssociation(a_k, v_1, v_2, m) \iff (\exists o_1, o_2) belong(v_1, m) \wedge belong(v_2, m) \wedge version(v_1, o_1) \wedge version(v_2, o_2) \wedge association(a_k, o_1, o_2) \quad (4)$$

On the other hand, it is important to consider that operations have *preconditions*. Precondition axioms allow specifying the conditions under which an action can be performed. Thus, it is necessary 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 Expression 5.

$$(\forall \phi, m) poss_{so}(\phi, m) \iff (\forall op_i, op_i \in \phi) poss_o(op_i, \phi, m) \quad (5)$$

The $poss_o(op_i, \phi, m)$ predicate expresses that 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 *i*-th operation belonging to ϕ (Expression 6).

$$(\forall \phi, m)(\forall op_i, op_i \in \phi, \exists \phi_1, \phi_2, \phi = \phi_1 \cdot op_i \cdot \phi_2) poss_o(op_i, \phi, m) \iff (\forall m_i, m \leqslant m_i < apply(\phi_1 \cdot op_i, m)) poss(op_i, m_i) \quad (6)$$

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. 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:

- Operation *add*(*v*) can be applied to model version *m* if the object version *v* does not belong to *m* (Expression 7);
- Operation *delete*(*v*) can be applied to model version *m* if the object version *v* belongs to *m* (Expression 8) ;
- Operation *modify*(v_i, v_j) can be applied to model version *m* if the object version v_i belongs to *m* and object version v_j does not belong to it (Expression 9);

$$(\forall v, m) poss(add(v), m) \iff \neg belong(v, m) \quad (7)$$

$$(\forall v, m) poss(delete(v), m) \iff belong(v, m) \quad (8)$$

$$(\forall v_i, v_j, m) poss(modify(v_i, v_j), m) \iff belong(v_i, m) \wedge \neg belong(v_j, m) \quad (9)$$

In this way, situation calculus allows formalizing the evolution of a design process 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 given by *add*, *delete*, and *modify* primitive operations, which conform the several sequences of operations that are applied to generate a new situation (model version). *Belong* is a fluent which specifies the object versions that belong to a model version. This fact is represented by a successor state axiom (Expression 3), where the effects of operations are expressed. These effects were defined by *added* and *deleted* predicates. Finally, preconditions axioms make possible to maintain the model consistence. With these elements, situation calculus represents change and evolution in an engineering design process.

Fig. 3 illustrates the described schema to express the model versions. In this case, a fragment of a SADP is shown. The example presents the two model versions illustrated in Fig. 1, where the *model version* m_q is generated from the *model version* m_k by the application of a *sequence of operations*. In this simple case, *design objects* are instances of the structural view concepts of a software architectural description languages such as ACME [16], i.e. components, connectors, ports, so on; and the set of primitive operations (composed by *add*, *delete* and *modify*) has been extended by operations as *applyMVC* (*applyMVC* operation will be explained in Section 2.1.2). This example shows a sequence of operations composed by only one operation, *applyMVC*. This operation refines the *WebApplication* component (*model version* m_k) on *View*, *Model*, and *Controller* components, with their ports and connections (*model version* m_q).

In Fig. 3, third level is also presented, the *inferred models level*. The third level is inferred from views produced by the *version level* on the *repository*. The *Struts* system belongs to both inferred model versions (*k* and *q*) and it is obtained by viewing the repository from the model versions m_k and m_q. At repository level, the *Struts* system is represented by *S*, an instance of *versionable object*. At this level, *S* is linked with the versionable objects that represent *WebApplication*, *View*, *Model*, and *Controller* components (*W*, *V*, *M*, and *C* versionable objects, respectively). As it is illustrated in Fig. 3, *S* has an object version *V1S* that belongs to *model version* m_k and m_q (illustrated in the intersection of both model versions). In addition, the *WebApplication* component only belongs to the *k* inferred model version. At versions level, this component has an object version, *V1W*, which belongs to model version m_k but it does not belong to model version m_q. This object version was refined in a set of object versions by applying an *applyMVC* operation. Then, *V1W* was deleted from the successor model version (m_q); and the object versions representing the model, view and controller components were added to m_q.

2.1. Instantiating the generic model with a particular engineering design domain

The primitive operations *add*, *delete*, and *modify* are not enough to capture and trace an engineering design process like SADP or

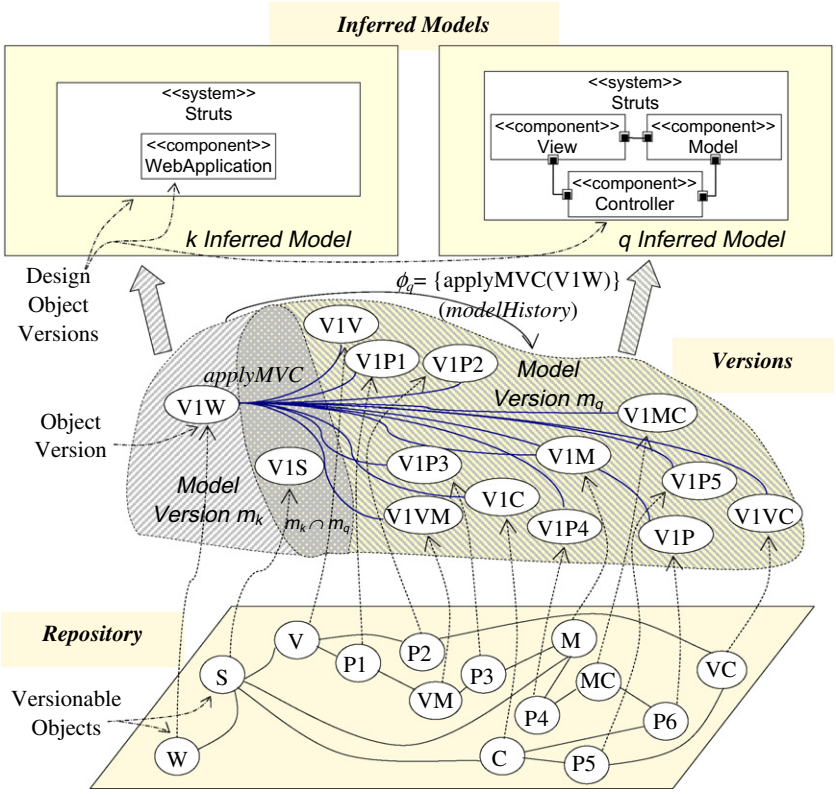

Fig. 3. Representation scheme of model versions.

process system engineering [26]. For that reason, one important contribution of this proposal is allow for representing the particular operations of a design domain. The strategy consists of, first of all, identifying the several design object types (modelling concepts in Fig. 2) that participate in the design process, such as *System*, *Component*, *Connector*, *Responsibility*, and *Scenario* for SADP, or *Stream*, *Reactor*, *Distillation column* for process system engineering. Then, the specific operations of that engineering design domain should be added to the model. This model extension is tackled in the following sections.

2.1.1. The domain model

In order to capture the versions generated during a design process, a generic and flexible computational tool for their capturing and tracing has to allow the definition of the particular modelling concepts (design object types) according to the working domain. For this purpose, the *Domain* package shown in Fig. 2 enables the definition of these concepts. *Domain* package gives the concepts for the specification of the particular design domain where the structure of the design objects, whose versions will be captured, is detailed. Thus, these design objects are the building blocks to represent the design artefact, requirements, and other design products such as the arguments of made decisions.

For each design object type, whose history has to be maintained, an instance of *ModellingConcept* (Fig. 2) must be generated for which its versionable properties are specified by a set of instances of *Property* class. Furthermore, the relationships among those concepts will be instantiated from *DomainRelationship* in *Domain* package (Fig. 2).

2.1.1.1. A domain model for software architectures design process (SADP). By considering SADP, the architect (designer) needs to capture and trace the evolution of design objects like component, connector, port, responsibility, etc. Theses concepts are found in numerous Architectural Description Languages (ADL) [17]. Additionally, the architect can extend the modelling concepts set with concepts that come from architectures design methods, such as Attribute Driven Design (ADD) [27], which considers quality requirement, scenario, and assessment. On the other hand, design rationale related concepts (constraint, assumption, argumentation, etc.) should be also considered.

This proposal defines the domain with design objects taken from the Attribute Driven Design method (ADD) [27], and the architectural description language ACME [16]. The class diagram shown in Fig. 4 introduces these concepts and their relationships. This model is implemented by the instantiation of the classes of *Domain* package (Fig. 2). The classes presented in Fig. 4 are going

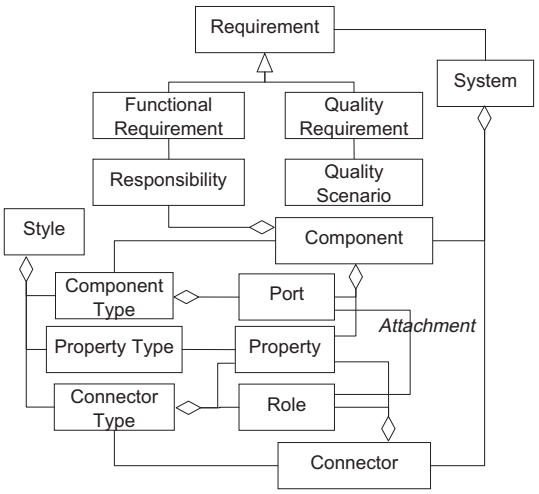

Fig. 4. A SADP domain model.

to be instances of *ModellingConcept* and their properties are going to be instances of *Property*. Finally, the relationships in Fig. 4 will be instantiated from *DomainRelationship* in *Domain* package.

The ADD method is based on a decomposition process where architectural patterns (or *styles*) are chosen at each stage to fulfil a set of *quality scenarios*. Then, *component* and *connector types* provided by architectural patterns are instantiated and functionality is allocated to them. The input to ADD is a set of requirements (*functional* and *quality requirements*). *Quality requirements* are expressed as a set of system specific *quality scenarios*, and *functional requirements* are translated into a set of *responsibilities*. Quality scenarios and responsibilities can be delegated to other components when the original component is refined. At the end of a method iteration, the designer verifies how well the architectural design achieves the scenarios and sets an *assessment* (this concept is included in Fig. 5). In ADD, the different model versions can be represented using various types of views. In the scope of this work, the component view is employed to describe the architecture, together with some design rationale concepts. Accordingly, ACME [16] has been chosen as the architectural description language. ACME defines a *component* as a computational element and data store of a system. A *component* may have multiple interfaces, named *ports*. *Connectors* represent interactions among *components* and have interfaces that are defined by a pair of *roles*. The *systems* comprise *components* and *connectors*, establishing *attachments* between *roles* and *ports*. In Fig. 4, the *attachment* concept is not considered as a *modelling concept* but as a *domain relationship*. Moreover, ACME proposes elements to document extra-structural properties of a system's architecture, such as *properties*. Furthermore, it is possible to attach *constraints* to design elements (this concept is included in Fig. 5). With the aim of providing a more powerful language, ACME defines *component*, *connector*, and *property type* building blocks. Based on these modelling concepts, it is possible to define *families* or *styles*. They are composed by a set of *property*, *component* and *connector types* and a set of *constraints*.

Besides already described concepts, other modelling concepts that allow capturing the design rationale applied during the design process can be regarded, as the set shown in Fig. 5. These new modelling concepts are included in the domain model and related to those that were already shown in Fig. 4, which are generalized by the *architectural element* abstract concept. Some of these design rationale modelling concepts are: *assessment* (an appreciation of an architect about the achievement of a given *scenario*); *assumption* (a kind of constraint, which is known in advance); *alternative* (different solutions, or directions for the intended system); *argument* (an assertion in which reasons are advanced for or against some alternative or proposal), and *constraint* (a rule or condition that restrict the design process).

Due to the fact that instances of the concepts usually need to be associated to other elements (i.e. software architecture structural modelling concepts with design rationale modelling concepts), it is valuable for the designer to count on another group of modelling concepts that allow the linking of both, which represent reified associations (or relationships). For example, an object version whose type is *RSystAssump* could express that a *system* is restricted by a given *Assumption*-type object version (Fig. 5). Another modelling concept, *RTradeOff* allows expressing situations such as the existence of tradeoffs between two quality requirements, when the achievement of one quality requirement affects the achievement of another, and vice versa. For instance, an association *RTradeOff* may relate *modifiability* quality requirement and *performance* quality requirement objects, indicating that the applying of modifiability tactics such as separation of concerns causes a low performance, whereas a good performance level precludes the facility in changing the future implementation of the architecture's design. Additionally, *RConstraintsTo* represents the link among a *Constraint*-type object version and one or more object versions of architectural elements. *RSupportsArg* and *RRejectsArg* modelling concepts allow linking a set of architectural elements with the argument that justify or refuse their presence in a given model version. In addition, *RSystAlt* modelling concept establishes a one to one relationship between an *Alternative* object and a given *System* object (which aggregates a whole architectural solution). Finally, *ROppositeTo* and *RSimilarTo* express the existence of two opposite or similar explored alternatives. It should be noted that these domain relations were represented as modelling concepts instead of ordinary domain relationships. This was done to allow the versioning of their instances.

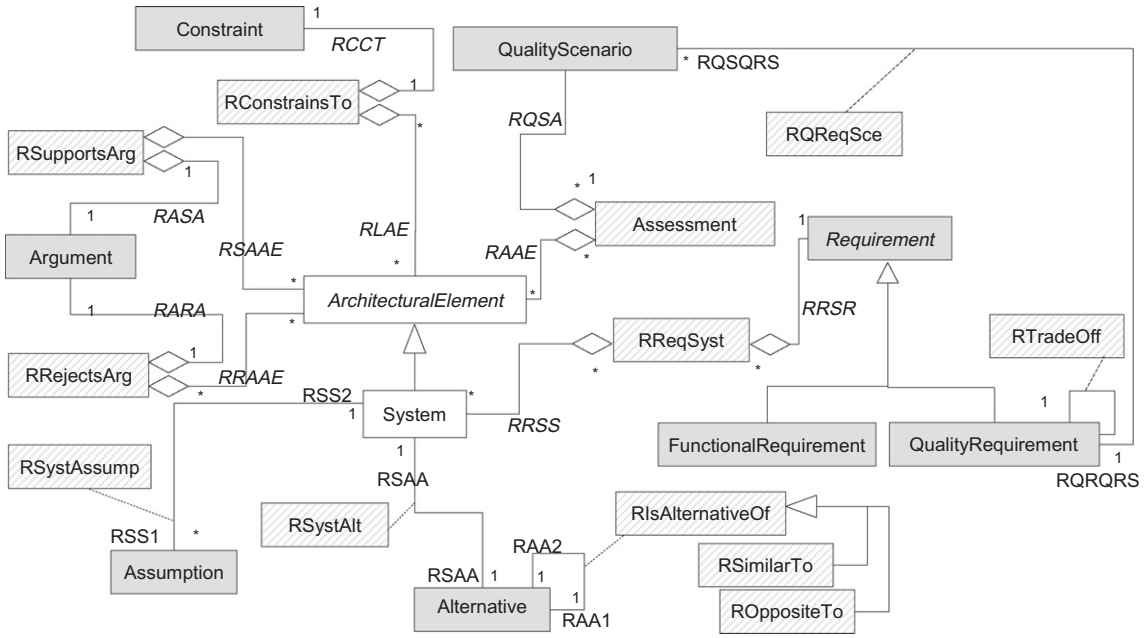

Fig. 5. Incorporating design rationale modelling concepts to a domain model.

Another way of adding design rationale to a model is including in the model the intentions and goals of the designer when an operation is performed. This is represented by linking the results of certain design decisions to the intended goals of those decisions. Regarding SADP, a "goal" could be a requirement to be met, a scenario to be verified, or a constraint that must be satisfied, among others. To achieve that, it is necessary to include modelling concepts that represent the relationship to be captured. For example, *RPossibleScenarioSolution* modelling concept (Fig. 6), which links a *QualityScenario* with a set of *ArchitecturalElements*, allows expressing that it is probably that these object versions achieve the quality scenario with an acceptable level of satisfaction. Additionally, Fig. 6 includes *RConsidersConstraint* modelling concept to express that a design (set of various interrelated *ArchitecturalElements*) takes into account a particular *Constraint* that was imposed on a system or part of a solution. It is important to note that *RConstrainsTo* (Fig. 5) and *RConsidersConstraint* (Fig. 6) are different concepts. The former permits to express an identified constraint that must be satisfied by an architectural model (it is imposed), and the later allows representing a case where particular set of object versions were incorporated to, or eliminated from, a given model version because a given *Constraint* was taken into account.

2.1.2. The operations model

The primitive operations *add*, *delete*, and *modify* have to be extended with suitable operations for a specific engineering design domain such as *applyMVC* operation employed in Fig. 3. In order to provide the foundations for computational tools, this contribution proposes an object-oriented operations model, which is flexible enough for specifying the domain's necessary operations. For each *modelling concept*, a set of possible *operations* is defined. To implement *operations*, the well-known Command design pattern was used [28]. Therefore, a *command* abstract class is introduced into the *Operations* package illustrated in Fig. 7. An *operation* is defined as a *macro command* (*MacroCommand* class), a subclass of *command* that simply executes a sequence of commands. Therefore, when an operation is specified, it is necessary to define both the *arguments* and the *body* of the operation. The body of a *macro*

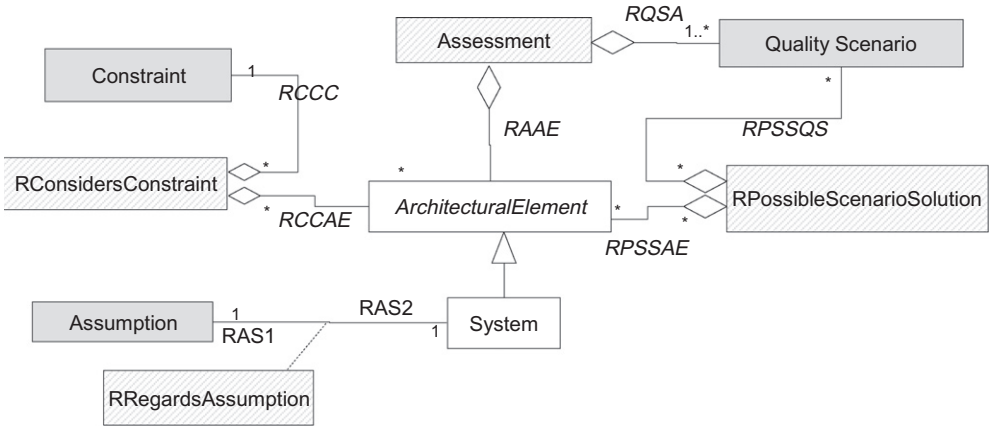

Fig. 6. Modelling concepts related to design rationale for expressing intentions and pursued objectives.

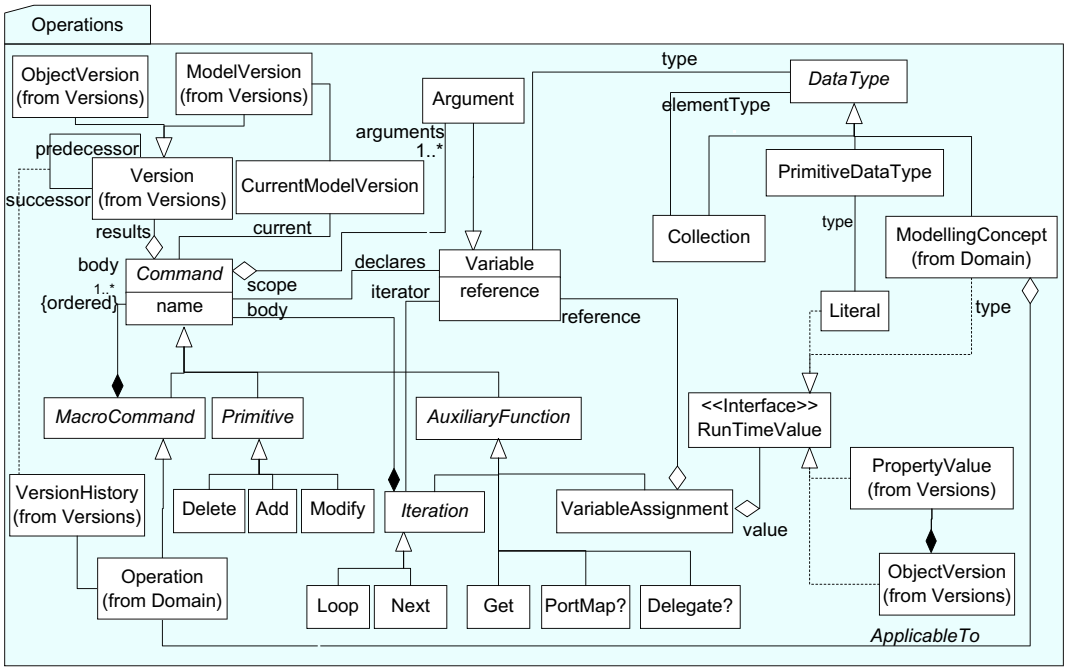

Fig. 7. Operations package.

command contains one or more already defined commands that are available for being used in other operation specifications. They can be *primitives* operations (such as *add*, *delete*, or *modify*), *auxiliary function* commands (like *iteration* and *variable assignment*), or previous defined operations. *AuxiliaryFunction* abstract class represents special predefined commands offered by the operation model, which can be used as building blocks for defining complex operations. *Iteration* class represents a command, which has a repetitive behaviour and is related to a collection of elements. It is specialized in *Loop* and *Next*, which have a slightly difference among them. The first one executes a body (or commands sequence) for each element in the collection. The second one, *Next*, works as a pointer that goes through the collection of elements, and allows access to each element. Frequently, *Next* is used as part of a body of *Loop* command. Another valid auxiliary function is *VariableAssignment* that represents the assignment of a value to a variable of a given type. As subclasses of *AuxiliaryFunction*, additional commands have been defined in [29].

As shown in Fig. 7, every *command* has one or more data typed *arguments*. *Arguments* are considered as a kind of *variable*. A *variable* can be also declared and used in the body of an *operation* and has a given type. The abstract class *DataType* generalizes the types described by the model and it is based on the data types defined by the Abstract Syntax Kernel Metamodel for Expressions defined by the UML 2.0 OCL Specification [30]. Therefore, *DataType* subclasses are *PrimitiveDataType* and *Collection*. *PrimiteDataType* includes *Integer*, *Float*, *String* and *Boolean* types. *Collection* describes an ordered list of elements of a particular given type, without duplicates, and it is parameterized with an *element type*. Furthermore, *ModellingConcept* is included as a data type to enable specifying arguments that make the type of an expected object version explicit to be added during the execution of an *add* primitive.

As regards *VariableAssignment*, it denotes the mapping between a *Variable* and a *RunTimeValue*. The last one is an interface that represents the run-time values during the execution of an operation. *RunTimeValue* can be realized by different values like *literal*, *object version*, *modelling concept*, or *property value* (value of a property of an object version, Fig. 7), depending on the variable type.

2.1.2.1. An operations model for software architecture design process (SADP). In this section, the primitive operations *add*, *delete*, and *modify* are extended with suitable operations for SADP domain like the ones listed in Table 1. Table 1 classifies them as: (a) operations related to structural concepts, or (b) operations related to design rationale concepts. The first category is grouped in several complexity levels:

(i) *Basic*: operations that allow creating and deleting basic design objects (like components and connectors).
(ii) *Special*: more complex operations that involve object refinement or delegation.
(iii) *Styles/mechanisms* application: operations that generate a new set of design objects, which have a configuration based on an architectural style; or even if they do not modify the model structure, they affect some design objects properties.

Additionally, it is important to count on operations that are applicable to architectural design rationale concepts. These operations embody those designer's decisions, analysis activities, or tradeoffs evaluations, which have a fundamental impact on the architecture, and which have to be documented in order to enable the architecture's future evolution. In other words, the capture of design rationale provides a valuable tool for understanding a system's architectural design. For example, an operation like *setAssessment* conveys the action of assigning a value that indicates how well a *scenario* is satisfied by a group of design objects with a given configuration and a set of properties. Operation *addArgument* is intended for making explicit the reason that supports or rejects a position or decision, by adding it as an object version. In this case, an instance of a relationship concept indicates the argument that is rejecting or supporting a decision. The *addAssumption* operation is also specified using the identified concepts and the primitives operations. This operation considers the sequence of actions for adding an object version whose design object type is *assumption*, and adding the relation between an element (or set of elements). The *addAssumption* operation is applied when the designer wants to express facts that were beforehand considered on the design of system architecture.

Fig. 8 presents functional specifications for some of the basic operations defined in Table 1. The rest of the operations are defined in a similar way, using primitive operations like *add*(c), and non-primitive ones, like *addPort*(c, p). For example, the *addComponent*(s, c, l_{Resps}, l_{Ports}) operation allows adding a component c to a system s. As it can be seen in Fig. 8, this operation is carried out by a series of operations. First, a version of component c is added (*add*(c)). After that, a set of responsibilities (specified by list l_{Resp}) and ports (de-

Table 1
Possible operations for the software architecture design domain.

Structural modelling concepts' operations		
Basic operations		Special operations
addComponent	deleteComponent	refineComponent
addConnector	deleteConnector	refineResponsibility
addFunctionalRequirement	deleteFunctionalRequirement	delegateResponsibility
addPort	deletePort	delegateScenario
addProperty	deleteProperty	verifyScenario
addQualityRequirement	deleteQualityRequirement	assignPossibleScenarioSolution
addResponsibility	deleteResponsibility	Styles/Mechanisms application
addRole	deleteRole	applyIntermediaryBlackboard
addScenario	deleteScenario	applyControlLoop
addTypeComponent	deleteTypeComponent	applyRuleEngine
adTypeConnector	deleteTypeConnector	applyClientServer
aetAttachment	deleteAttachment	applyPoolOfConnections
		applyMVC
		applyMVC-go
Architectural design rationale concepts' operations		
setAssessment	addArgument	addAlternative
modifyAssessment	supportDecision	acceptAlternative
addContraint	opposeDecision	rejectAlternative
satisfyContraint	addAssumption	

```
addComponent(s,c, l_Resps,l_Ports)
add(c)
for each r in l_Resps
   addResponsibility(c,r)
end for
for each p in l_Ports
   addPort(c, p)
end for
addRelationship(s, c)

addPort(c, p)
add(p)
addRelationship(c, p)

addResponsibility(c, r)
add(r)
addRelationship(c, r)
```

```
deleteComponent(s, c)
l_Ports = getPorts(c)
for each p in l_Ports
   deletePort(c, p)
end for
delete(c)

deletePort(c, p)
// port deletion implies
// deletion of the connector
// attached to it
deleteConnector(getConnector
                (getRol(p)))
delete(p)

deleteResponsibility(c, r)
delete(r)
```

Fig. 8. Specifications of basic operations.

tailed by list l_{Ports}) are inserted. Some of these design objects can have associations at repository level. Such associations are asserted as facts, and, if possible, they are inferred at version level, as it was specified by Expression 4. For clarifying purposes, in operations' functional specifications, a pseudooperation *addRelationship* is used, which simulates the addition of an *association* into the repository (just between *versionable objects*). Functional specifications give an outline of how the operations could be configured using a computational tool. They are useful as an example of the syntax provided by a computational tool based on the proposed model. Section 3 shows an example of how TracED allows the definition of these operations by using similar elements.

Just as defining basic operations, it is possible to define the special operations. Fig. 9 presents some examples. The *delegateResponsibility*(c_1, c_2) operation enables delegating a responsibility of component c_1 to component c_2. Thus, if a given responsibility is assigned to a component c_1 in a model version m and a *delegateResponsibility*(c_1, c_2) operation is included in the sequence of operations applied to the model version m, then the resulting model version shows that the responsibilities delegated to c_2 will not be assigned to c_1. In a similar way, the operation *delegateScenario* proceeds with components and scenarios. A function with a question mark ('?') at the end indicates that it is an interactive function, thus the user is asked about how to proceed. For example, *delegate?* function is repeatedly executed into a loop of the *delegateResponsibility* operation in order to ask the user about assigning a given responsibility to another component or not. It should be noted that *delegate?* should be included in operations model as a subclass of *AuxiliaryFunction* (Fig. 7).

The operations that apply an architecture style refine a component instance with a new set of components and connectors that are instantiated from a preexisting architecture style. They interact with the designer asking how to delegate responsibilities and scenarios, as well as how to map connectors between external components and refined components. For example, considering the client–server style defined in Fig. 10, the *applyClientServer*(c) operation refines the component c in two new components: *Server* and *Client*. The type of *Server* component is *TServer* modelling concept, whereas the type of *Client* component is *TClient* modelling concept. Both *TServer* and *TClient* are *ModellingConcept* instances, which extend *Component* modelling concept. These new components have some predefined responsibilities and ports. Furthermore, a connection between them that meets the properties imposed by the style is added. Finally, the designer is asked for responsibilities delegation from the original component to *Server* and *Client* components.

In a similar way, *applyMVC* operation is defined. In this case, the operation aims to achieving separation of concerns in a software application, by using model-view-controller (MVC) [28] architec-

```
delegateResponsibility(c_1, c_2)
l_Resps = getResponsibility(c_1)
for each r in l_Resps
  if (delegate?(c_2, r))
      addRelationship(c_2, r)
  end if
end for

setAttachment(role, port)
addRelationship(role, port)
end for
```

```
delegateScenario(c_1, c_2)
  l_Scens = getScenario(c_1)
  for each s in l_Scens
    if (delegate?(c_2, s))
      addRelationship(c_2, s)
    end if
  end for
```

```
assignPossibleScenarioSolution(pss, sce, versionsList)
add(pss)
addRelationship(sce, pss)
for each v in versionsList
   addRelationship(pss, v)
end for
```

Fig. 9. Specifications of special operations.

```
applyClientServer(c)
  s = getSystem(c)
  addComponent(s,{Browser, TClient}, [P1])
  addComponent(s,{WebServer, TServer}, [P2])
  addConnector(s,{InternetConn,THTTPConnector},[R1,R2])
  delegateScenario(c,Browser)
  delegateScenario(c,WebServer)
  delegateResponsibility(c, Browser)
  delegateResponsibility(c, WebServer)
  setAttachment(R1, P1)
  setAttachment(R2, P2)
  deleteComponent(s, c)
```

```
applyMVC(c)
  s = getSystem(c)
  addComponent(s,{View, TView}, [P1,P2])
  addComponent(s,{Controller, TController}, [P3, P4])
  addComponent(s,{Model, TModel}, [P5,P6])
  addConnector(s,{ConnModView,TCModView},[R1,R6],[P1,P6])
  addConnector(s,{ConnViewCtrlr,TCViewCtrlr},[R2,R3],[P2,P3])
  addConnector(s,{ConnModCtrlr,TCModCtrlr },[R4,R5],[P4,P5])
  setAttachment(R1, P1)
  setAttachment(R2, P2)
  setAttachment(R3, P3)
  setAttachment(R4, P4)
  setAttachment(R5, P5)
  setAttachment(R6, P6)
  delegateScenario(c,Model)
  delegateScenario(c,View)
  delegateScenario(c,Controller)
  delegateResponsibility(c, Model)
  delegateResponsibility(c, View)
  delegateResponsibility(c, Controller)
  l_p = getPorts(c)
  for each p in l_p
    np = PortMap?(p) // Ask the user the port to map
    r = getRol(p)
    addRelationship(np, r)
  end for
  deleteComponent(s, c) //Predecessor object version is removed
```

```
applyMVC-go(c, g)
  // g (goal): receives the quality requirement that is
  // intended to reach when the operation is executed
addedVersions := applyMVC(c)
sce := select(getScenarios(g))
assignPossibleScenarioSolution(pss, sce, addedVersions)
```

```
applyIntermediary(BB, lConn)
addComponent({BB, TBlackBoard})
for each Cn in lConn
  addPort(BB, PBB1).
  addPort(BB, PBB2).
  LRol = GetRole(Cn)
  R1 = LRol(0)
  R2 = LRol(1)
  P1 = GetPort(R1)
  P2 = GetPort(R2)
  addConnector(C1, [R1, RCI1])
  addConnector(C2, [R2, RCI2])
  deleteAttachment(P1,R1)
  deleteAttachment(P2,R2)
  setAttachment(RCI1, PBB1)
  setAttachment(RCI2, PBB2)
  deleteConnector(Cn)
end for
```

Fig. 10. Specification of some architectural style application operations.

tural pattern. The *Model* provides access to the necessary business data as well as the business logic needed to manipulate that data. The *Model* typically has some means to interact with persistent storage — such as a database — to retrieve, add, and update the data. The *View* is responsible for displaying data from the *Model* to the user. This layer also sends user data to the *Controller*. The *Controller* handles all requests from the user and selects the view to return. When the *Controller* receives a request, it forwards the request to the appropriate handler, which interprets what action to take according to the request. The *Controller* calls on the *Model* to perform the desired function. After the *Model* has performed the function, the *Controller* selects the *View* to be sent back to the

user according to the state of the *Model*'s data. Each of these components is generated by *applyMVC* execution, along with their responsibilities, ports, and connectors. Additionally, the assigned responsibilities and the scenarios to be satisfied by the original component *c* are delegated to new components (*delegateResponsibilites* and *delegateScenarios* operations, Fig. 9), by interacting with the actor who executed the operation.

Fig. 10 also presents an alternative for materializing the MVC pattern in a design operation, called *applyMVC-go*. The suffix *go* stands for "goal oriented", thus indicating the designer specifies and executes the operation thinking in addressing an specific goal or objective. The *applyMVC-go* operation besides appling the MVC pattern, it captures why the designer executes it. To achieve that, *applyMVC-go* includes an extra argument to indicate the quality requirement that the designer is addressing. At the end of the *applyMVC-go* specification, additional commands are included to prescribe how the added object versions have to be linked to a particular object version that represents a quality scenario to make explicit the intention of the designer. As the passed argument value is a quality requirement, the designer is asked to select one quality scenario relative to such a requirement, which is going to be specifically addressed (this action is embedded in the *assignPossibleScenarioSolution* operation, which was specified in Fig. 9).

Finally, Fig. 10 presents *applyIntermediary* operation which materializes in the present model, a well known tactic for breaking the dependency chain among components in a system. In this case, the intermediary is a blackboard (an object version whose type is *TBlackBoard* modelling concept).

In spite of the complexity inherent to style application operations, note that each of them can be translated into a sequence of primitive operations *add*, *delete*, and/or *modify* that are applied to a predecessor model version, which results in a new model version. From this, it is possible to express these operations in terms of *added* and *deleted* predicates introduced in Expressions 1 and 2. For illustration purposes, let us consider again the *addComponent*(s, c, l_{Resps}, l_{Ports}) operation. If it is part of a sequence of operations ϕ, which is applied to a model version m, then a version of a component c that have a set of *responsibilities* r_i (l_{Resps}) and *ports* p_i (l_{Ports}) will belong to the successor model version (*apply*(ϕ, m)), as it is defined in Expression 10.

$$\begin{aligned}&(\forall \phi, s, c, l_{Resps}, l_{Ports}, m)\, addComponent(s, c, l_{Resps}, l_{Ports}) \in \phi \\ &\quad \Longleftrightarrow added(c, apply(\phi, m)) \\ &\quad \wedge (\exists r_c)\, inferredAssociation(r_c, s, c, apply(\phi, m)) \\ &\quad \wedge ((\forall r \in l_{Resps})\, added(r, apply(\phi, m)) \\ &\quad \wedge (\exists r_r)\, inferredAssociation(r_r, c, r, apply(\phi, m))) \\ &\quad \wedge ((\forall p \in l_{Ports})\, added(p, apply(\phi, m)) \\ &\quad \wedge (\exists r_p)\, inferredAssociation(r_p, c, p, apply(\phi, m))) \end{aligned} \tag{10}$$

It is important to point out that *inferredAssociation*(a, v_1, v_2, s) predicate is just an relationship inferred from a repository association between versionable objects o_1 and o_2, which are associated with v_1 and v_2 by *version* predicate. The association between versionable objects was asserted as a fact (*association*(a, o_1, o_2)) (Expression 4).

In this way, the definition of new operations allows enlarging the set of operations, without modifying the successor state axiom (Expression 3). Furthermore, it is possible to define the precondition axioms of the proposed operations. For instance, the precondition for applying the *addComponent* operation is specified in Expression 11, where the *poss*(op, m) predicate expresses that a given version s, which represents the system where the new component is going to be incorporated, must belong to model version m, and component version c as well as their responsibilities and ports must not belong to model version m.

$$\begin{aligned}&(\forall s, c, l_{Resps}, l_{Ports}, m)\, poss(addComponent(s, c, l_{Resp}, l_{Ports}), m) \\ &\quad \Longleftrightarrow belong(s, m) \wedge \neg belong(c, m) \\ &\quad \wedge (\forall r \in l_{Resps}) \neg belong(r, m) \wedge (\forall p \in l_{Ports}) \neg belong(p, m) \end{aligned} \tag{11}$$

It is important to note that each functional specification of an operation proposed in this section implies the instantiation of the operation model. The following section details how the computational environment allows such instantiation, by providing graphic user interfaces with features for operations definition.

3. TracED

TracED is a research prototype for validating the proposed model for capturing and tracing engineering designs. Such a prototype has been developed using Java language, and MySQL database [31].

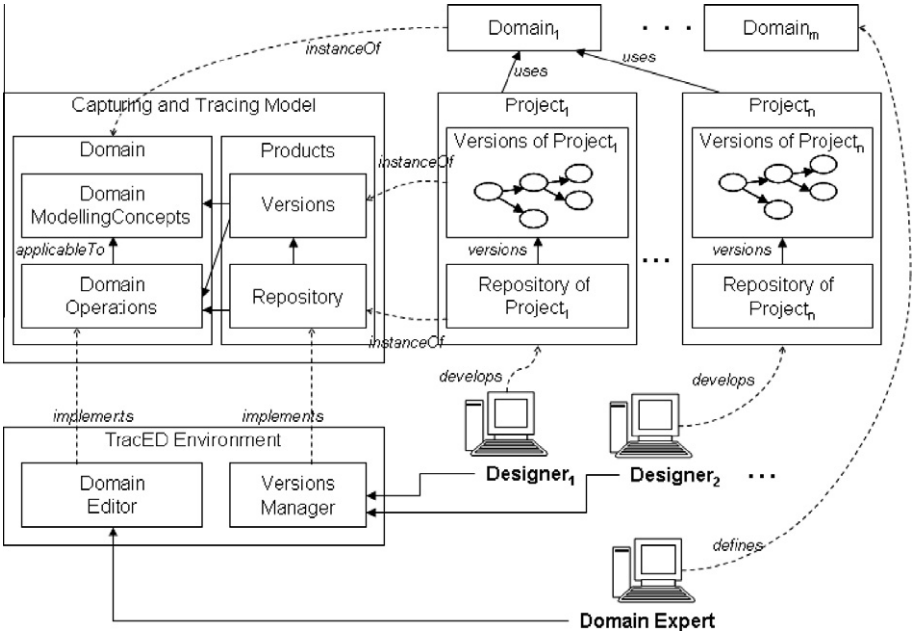

Fig. 11. A global picture of TracED environment.

Hibernate framework [32] has been used in order to achieve objects persistency on a relational database, providing agile objects saving and recovering.

The major components are *Domain Editor* and *Versions Manager*. Both of them were developed following the object-oriented models proposed in the previous section. The *Domain Editor* allows the definition of *modelling concepts* along with their suitable *operations*. The *Versions Manager* offers features for users to carry on engineering design model evolution. User views of both modules were obtained by means of an Observer design pattern implementation [28]. In Fig. 11, a global picture of TracED environment is presented. *Domain Editor* and *Versions Manager* components implement the proposed model. Making use of *Domain Editor*, a domain expert defines one or more engineering design domains. Then, by working with the *Versions Manager*, designers can choose an existent domain and use the modelling concepts and operations defined in it to develop several design projects.

3.1. Domain editor: specifying modelling concepts and operations

Domain editor is the tool that enables the definition of an engineering design domain. This is achieved by implementing the domain package proposed in Fig. 2. Therefore, the modelling concepts included in Figs. 4–6 can be defined, by using *Domain Editor*. A partial view of SADP domain specified in TracED is visualized in Fig. 12.

Modelling concepts are organized hierarchically in a tree structure. On the upper-left corner of Fig. 12, the modelling concepts navigation tree shows the concepts' tree. Each concept is able to have zero or more descendents and a unique parent. This structure is obtained by specializing *modelling concept* (Fig. 2) in *abstract modelling concept* and *concrete modelling concept*. An *abstract modelling concept* can not be instanced in any project. Abstract concepts are useful because they define general design objects which may be specialized later. This enables generalizing common properties and relationships used by a set of design objects. For example, the *Requirement* abstract modelling concept has been added in Fig. 12, which generalises *Quality Requirement* and *Functional Requirement* design objects. An *abstract modelling concept* is illustrated in the domain editor with its name in italics.

In the modelling concepts navigation tree shown in Fig. 12, the name of the selected concept appears highlighted, in this case *component*. In the frame below, the properties or attributes of the selected concept are listed. Each attribute is a pair of an attribute name and an attribute type. There are four defined primitive types: *String*, *Integer*, *Boolean*, and *Float*. In the example, there are two attributes defined as *String*, *name* and *type*. The first has an obvious meaning, and the second one can indicate if the component is a module, a filter, a data storage, etc. Properties are defined and edited working with the properties tab of the modelling concept specification window. There, the designer can indicate whether a modelling concept is abstract or concrete, assign a concept description, create or modify properties (instances of *Property* class, Fig. 2), and set visibility of properties. Additionally, it is possible to define the modelling concept as a relationship (representing an *association* reification), which is useful when it is necessary to keep versions of such a relationship. This is the case of *hasResp* modelling concept in Fig. 12, which defined in this way allows tracing how responsibilities were delegated from one component to another.

The designer can create binary relationships between modelling concepts. These relationships are instances of *DomainRelationship* (Fig. 2). One relationship's end assumes the role of *container*, and

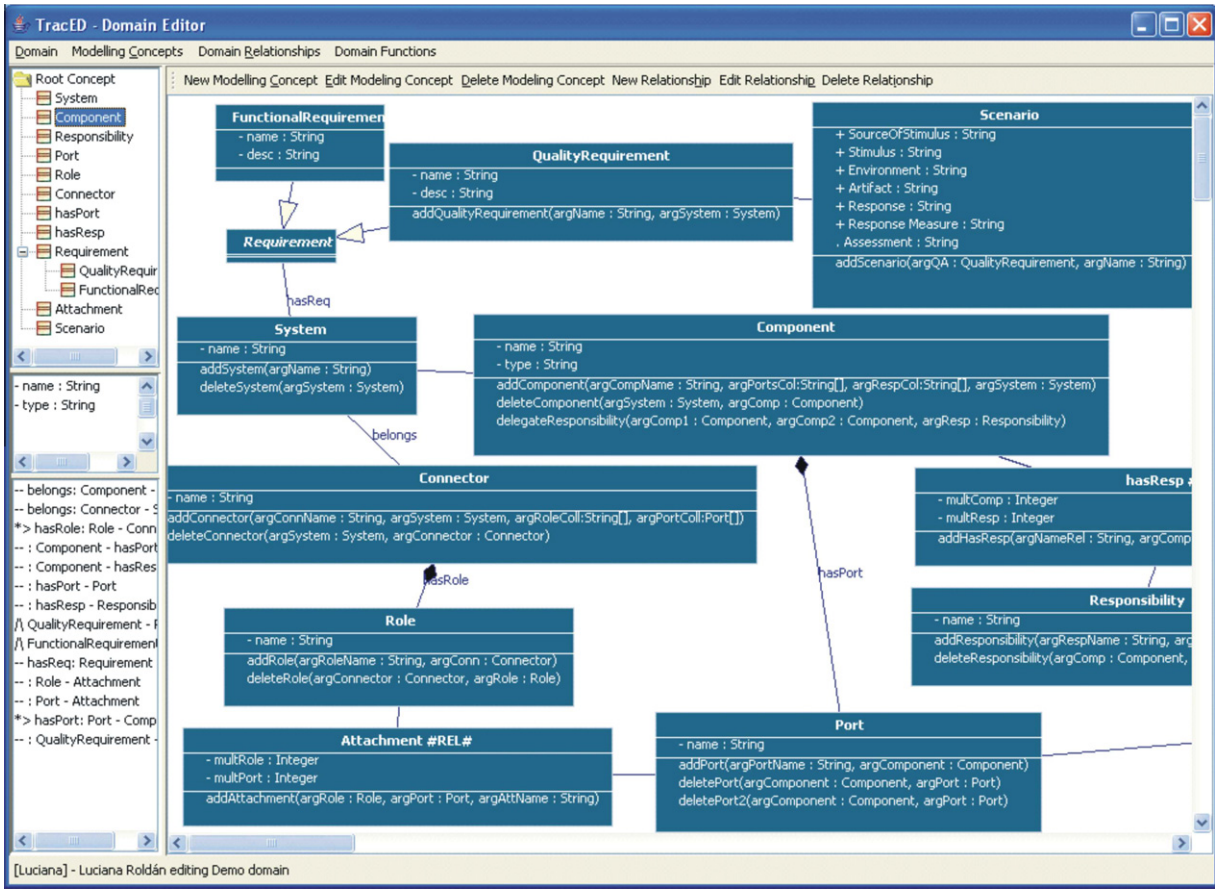

Fig. 12. Partial view of SADP domain model specified in TracED.

the other one, the role of *part*. Both relationship roles are qualified by the multiplicity attributes (*contMultiplicity* and *partMultiplicity*). Furthermore, it is possible to define the relationship as *composition*, *aggregation*, or *association*. They have the same semantics as their respective relationships in UML [25].

The Domain Editor provides features for specifying operations, whose implementation is based on the *Operations Model* (Fig. 6) presented in the previous section. Following, the way in which a domain operation can be defined by an instantiation of the *Operations Model* is depicted. Fig. 13 shows the Operations Tab of *Component* modelling concept specification window. As it can be observed, three operations have been defined for it: *addComponent*, *deleteComponent*, and *delegateResponsibility*. The Domain Editor allows the specification of these *macro commands* and their input *arguments*, by means of a special window that is presented in Fig. 14. There, the definition of *addComponent* macro command is shown, which follows the functional specification that was presented in Fig. 8. In this window, the *body* of a *macro command* is detailed. For accomplishing this, the actor can select one or more of the existing operations, which are listed in a combo box. Additionally, Fig. 14 explains how the *VariableAssignment* described in *Operations Model* is carried out using TracED. It is achieved by establishing a series of mappings among different arguments. In this way, the binding of arguments (variables) and their values (which are unknown at the moment of the specification) are set beforehand the execution of the *macro command*.

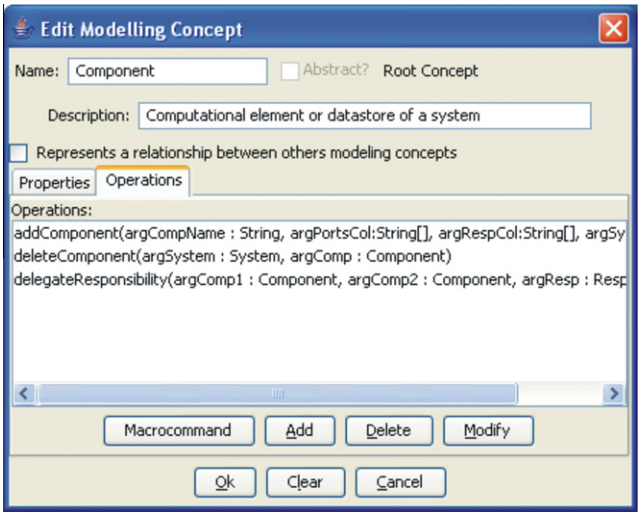

Fig. 13. Editing a modelling concept.

As stated in Fig. 7 (*Operations package*), the tool offers pre-built operations (*AuxiliaryFunction*) such as *Loop* and *Next*. *Loop* can be used in the body of macro commands as a repetitive control structure, in order to move the control along a collection variable; and it performs a series of actions for each element. On the other hand, *Next* is useful to get in sequential order the next element of a collection. *Next* does not have a body, it only saves a pointer to the last accessed element, and moves forward each time it is executed, returning as a result the element on the current position. Fig. 15 shows the loop over a responsibilities collection, which assigns each responsibility to a component by mean of the *addResponsibility* operation. As shown in Fig. 15, the body of a *Loop* (the repetitive sequence of actions) is defined in a separated window.

Moreover, in some macro command specifications, it is necessary to obtain object versions related to a given object version. For example, it can be considered an operation for deleting a component (*deleteComponent* operation), which has an object version component as input argument. The operation requires to know the component to be remove and the ports of that component, in order to delete them too. In such a case, the *deleteComponent* operation body has to include an instance of *Get* command, in order to obtain the ports associated to the component. This kind of commands does not constitute domain operations; they are just auxiliary commands (or functions) for querying about object versions (*AuxiliaryFunction*, Fig. 7). TracED provides this feature by selecting the radio button "*function*" in the operation specification window.

Aforementioned commands along with primitives and previously defined domain operations can be part of new domain oper-

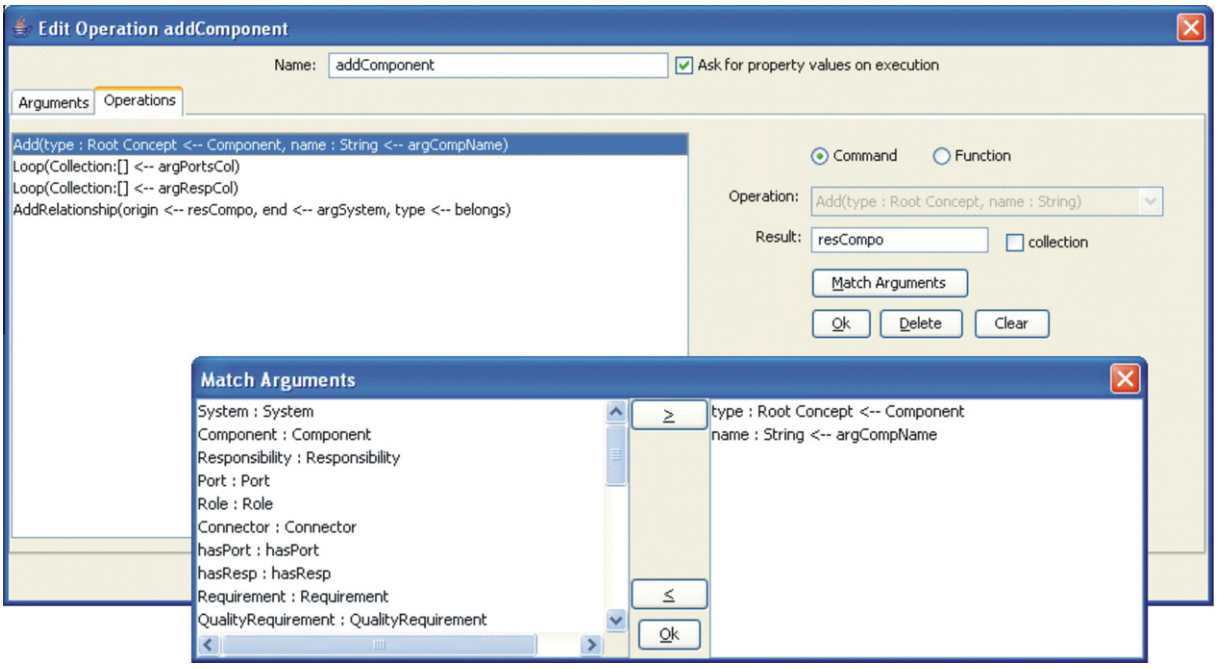

Fig. 14. Specification of *addComponent* macro command.

ations or macro commands. As it can be seen, macro commands have a higher abstraction level than the underlying basic operations. In this way, the higher the abstraction level of an operation, the freer is the designer of knowing the involved actions by the performed operation.

3.2. Versions manager

The versions manager is the core of *TracED*. It enables the execution of a design project. When a new design project is created, an existing domain has to be selected for it. Thus, version manager allows the developing of the project, by considering modelling concepts and operations defined in such a work domain. Therefore, the evolution of a project is based on the execution of these domain operations and the instantiation of these modelling concepts. Additionally, versions manager's features make TracED able to keep information about: (i) predecessor and successor model versions (if any exists) of each model version; (ii) history links which save traces of applied sequences of operations, which originated new model versions and (iii) references to the set of object versions that arose as a result of each operation execution. Furthermore, TracED implements the successor state axiom specified in Expression 3. Therefore, it is possible to reconstruct a *model version* m_{i+1} by applying all operation sequences from the initial *model version* m_0. Fig. 16 presents a sequence diagram that depicts an implementation of the successor state axiom proposed in the model (Expression 3), which aims to know the objects versions that belong to a given *ModelVersion_i* model version. As Expression 3 points out, in order to find out the objects versions that belong to a given model version it is necessary, first, to obtain the object versions that are present in the predecessor model version (*predecessorObjectVersions := getObjectVersions*()). The sequence diagram shows how this interaction among instances takes place. *ModelVersion_i* sends a message to itself (*getPredecessorModelVersion*()) to know its predecessor model version (*ModelVersion_i* – 1). Then, it sends a message to such *ModelVersion_i* – 1 instance, by means of *getObjectVersions*() method. This method has a recursive definition and causes a cascade of *getObjectVersions*() methods executions, which stops when the *initial model version* is reached. Therefore, this method returns all object versions (*predecessorObjectVersions*) that belong to the predecessor model version (*ModelVersion_i* – 1). Second, as Expression 3 prescribes, it must be obtained the added object versions resulting from the execution of the sequence of operations applied on *ModelVersion_i* – 1 to generate *ModelVersion_i* (*addedObjectVersions := getAddedObjectVersions*()). Consequently, by sending the *getPreviousModelHistory*() message, *ModelVersion_i* obtains the sequence of operations that originated it from *ModelVersion_i* – 1. This is possible because each executed operation in the sequence of operations has been captured by means of *VersionHistory* objects, which keep information of each operation performed. In this way, it is possible to get a set of added object versions (*addedObjectVersions* in Fig. 16), by means of *getAddedObjectVersions*() message. Thus, *addedObjectVersions* collects all the object versions that arose from operation executions (for example, regarding the SADP domain in Fig. 3, the *Model*, *View* and *Controller component*-typed object versions, which arise from the execution of *applyMVC* operation). Finally, successor state axiom in Expression 3 suggests that object versions that were deleted by the applying of the sequence of operations on the predecessor model version (*deletedObjectVersions := getDeletedObjectVersions*()) are not present in the regarded model version. Thus, *ModelVersion_i*, by sending a *getDeletedObjectVersions*() message, asks to *VersionHistory* object the set of deleted object versions (*deletedObjectVersions* in Fig. 16). In this way, *getDeletedOV*() collects all the object versions that were removed from the model version to which the operation was applied (following the previous example, the *WebApplication component*-typed object version that was refined in a new set of components by the *applyMVC* operation). Therefore, as illustrated in Fig. 16, the inferred object versions set (*inferredObjectVersions*) is obtained by deleting the *deletedObjectVersions* and adding the *addedObjectVersions* to the resulting object versions from the initial *getObjectVersions*() sent to the predecessor model version (*ModelVersion_i* – 1). As a result, the set of object versions that belongs to a given model version is obtained (*inferredObjectVersions*).

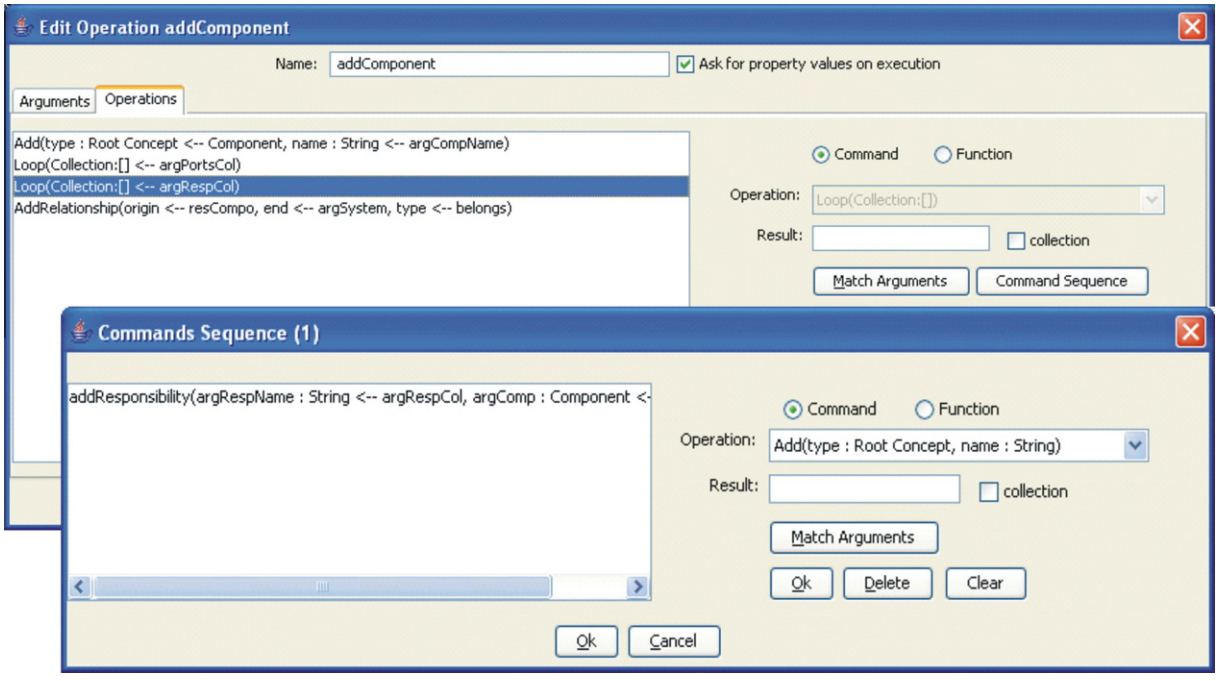

Fig. 15. Specification of an operations sequence in a *Loop*.

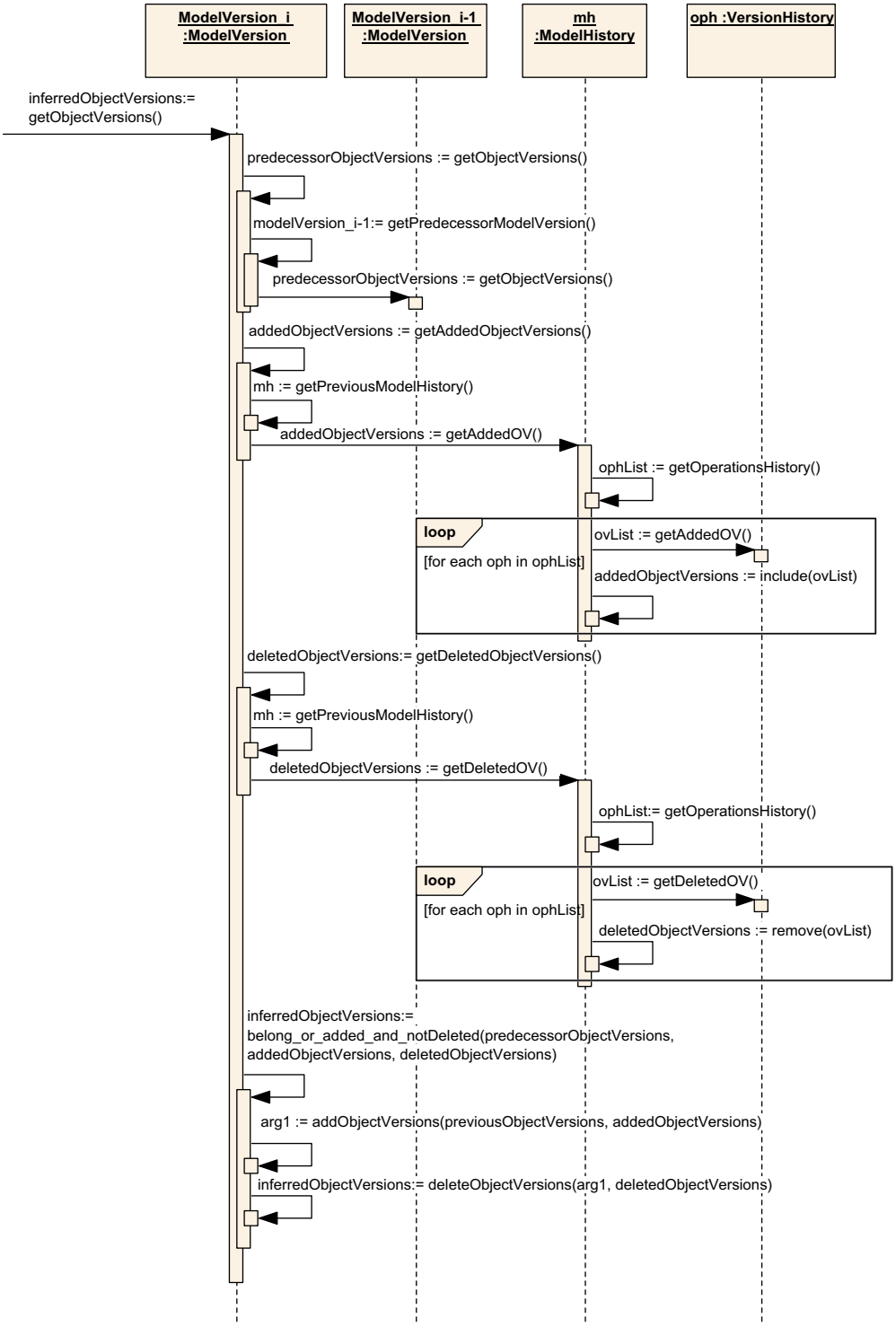

Fig. 16. Successor state axiom implementation on TracED.

3.2.1. Operation execution

This section depicts the execution of an *Operation* in the *Versions Manager* tool. Since an *Operation* is a *MacroCommand*, its execution consists of the execution of each of its subcommands. Thus, as it was introduced in Fig. 7, an operation has a commands body. A *macro command* has input *arguments*, whose values have to be provided before being executed. Each *command* in the commands sequence has its own *arguments*. In order to execute the operation, each *argument* must be linked to a given value. This value could be provided by the result of a previous command, or could be provided by an input argument defined for the operation itself. This means that in the moment of specifying the operation, the arguments values are not known, although it is known where can be get them. Thus, when the environment executes a given *operation*, it must recover the expected argument values.

Fig. 17 presents a sequence diagram, which models the dynamic aspects of an *operation* execution. This diagram involves instances of graphic user interface (*UserInterface*), *Operation*, *Command*, *Argument*, *ModelVersion*, *ModelHistory* and *VersionHistory* classes, which interchange messages among them. The first message of the sequence is triggered by some component of the *UserInterface* (it could be a menu option), which after obtaining the input argument values, dispatches an *execute*() message to the chosen operation instance. Before being performed the operation itself, its preconditions are checked (*verifyPreconditions*() method). Then, for each *command* in the *body*, it must be obtained and assigned the values for its arguments (by means of the *getLinkedArguments*(), *getValue*() and *setValue*() messages shown in Fig. 17) and then, with all this information ready, it must be triggered a new command execution (*successors := execute*()). When the cascade of executions of commands ends, an instance of *VersionHistory* is created to kept information about the execution of the operation. *VersionHistory* maintains the predecessors and successors object versions of the operation execution. This *VersionHistory* instance becomes part of the *ModelHistory* that links the predecessor model version with the successor model version.

It should be highlighted that, the first step in a macro command execution, is the checking of the operation's preconditions, which is implemented by the *verifyPreconditions*() method. As an example, the preconditions of *addComponent* operation were expressed by means of a *poss* predicate in Section 2.1.2. In case a precondition is not accomplished, TracED is capable of managing different policies. One outstanding aspect to consider is that before a delete operation execution, TracED controls if the object versions to be removed exist. In case of an attempt of eliminating an object version which has been just added to the current model version, the environment offers the following alternatives to the actor: (1) to undo the *add* operations that incorporated the objects that are now intended to be removed; (2) to create (automatically) a new model version where it is possible to execute the desired delete operations and (3) to cancel the execution. Option one is suitable when the designer involved in the process makes a mistake or intends to undo a given action; and option two is preferable when the actor really wants to keep a record of all past operations, in spite of not having consequences for the current model version.

3.2.2. A case study on the software architectures domain: Struts

To clarify the features of the *Version Manager*, a conducting case study is introduced, which considers the design process of Apache Struts, a free open-source framework for creating Java web applications [33]. The main considerations in the conception of Struts Project is that Web applications differ from conventional websites in that web applications can create a dynamic response. Thus, a web application can interact with databases and business logic engines to customize a response.

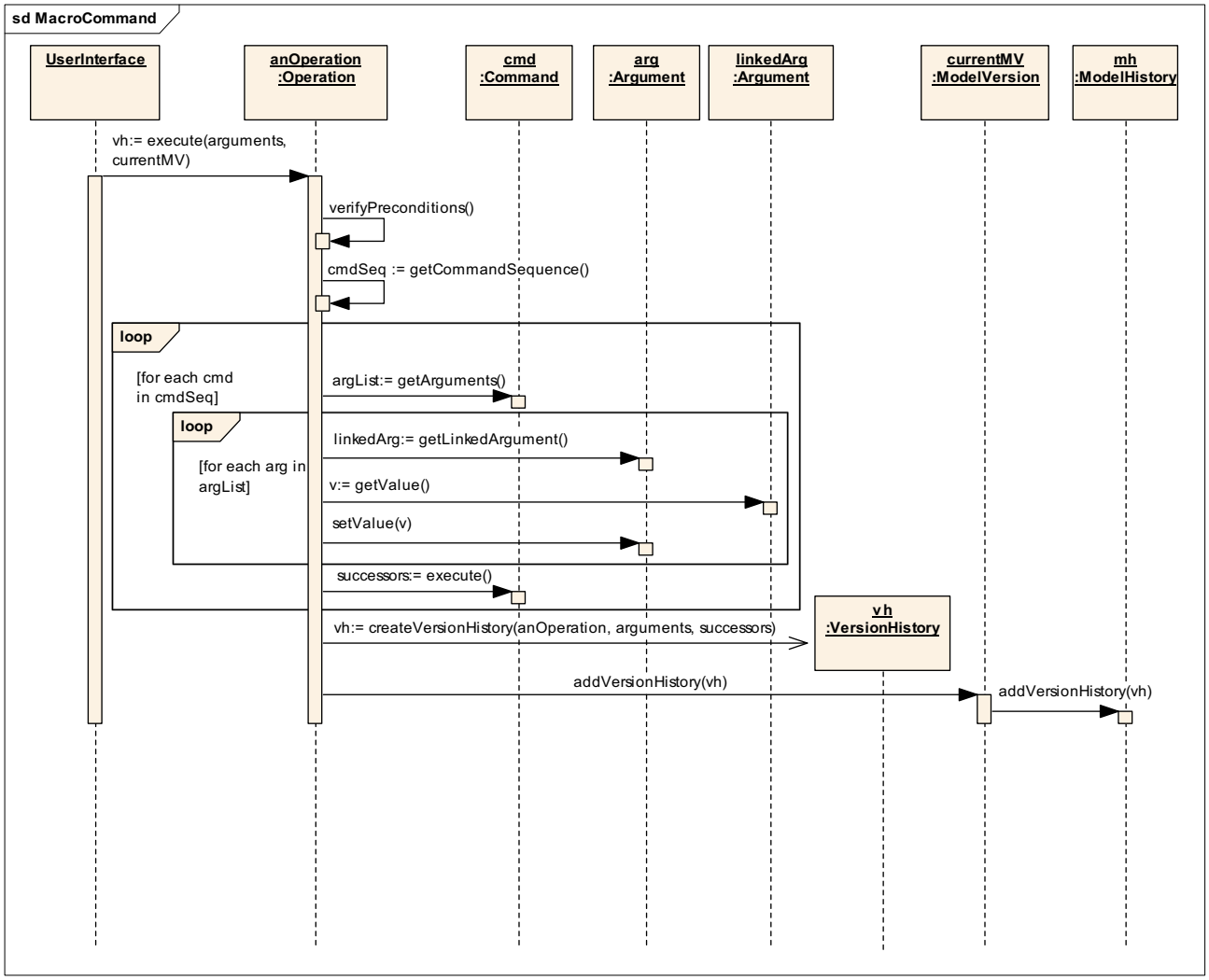

Fig. 17. Sequence diagram of an *Operation* execution.

Web applications based on Java Server Pages sometimes mix together database code, page design code, and control flow code. These large applications become difficult to maintain, unless these concerns are separated. One way to separate concerns, in a software application, is to use a model-view-controller (MVC) [28] architecture. In Fig. 10, a specification of this operation was proposed. The Struts framework has been designed to help developers to create web applications that use the MVC architecture. In this paper, the architectural design process of Struts is taken as a case study. Thus, this case study aims to resemble the work carried out by the architects who identified the need of separating concerns in java web developments, and starts working in the development of Struts framework.

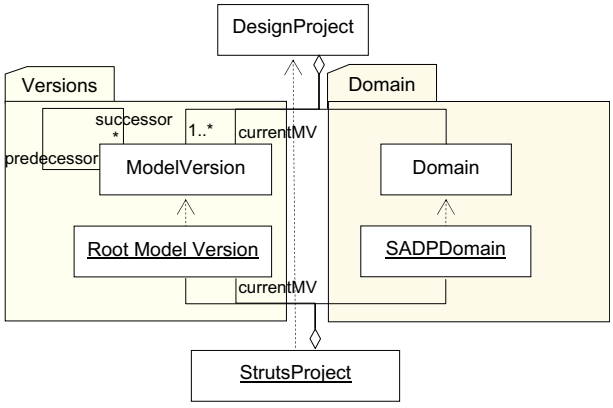

Fig. 18. Creation of a new design project.

Fig. 18 shows the instances [25] generated during the definition of a new project, called *StrutsProject*. An instance of *DesignProject* is created (*StrutsProject*), which is associated to the selected design domain, *SADPDomain* (which was partially defined in Section 3.1). Furthermore, the initial model version (*Root Model Version*) is generated, which is the root of the tree structure of the version management scheme. Once a design project is defined, it is possible to generate the model versions, capture the evolution of design objects, and navigate through their versions. Fig. 19 shows the user interface of version manager, where a snapshot of the first model version of Struts project is presented. There, created model versions are hierarchically organized like a tree. On the upper-left of the version manager window in Fig. 19 appears the "model versions tree" navigator. The initial model version is called *Root Model Version* that is the starting point to create all the new Model Versions. *Root Model Version* has no object versions and it cannot be edited by anyone.

To create a new model version, the designer must select a model version that will be the parent of the new one. The selected model version is called *current Model Version*. Parent model version is referenced as the predecessor model version. Some rules or restrictions must be satisfied when using this module. Firstly, when a successor model version has been created from a predecessor model version, it is not possible to execute more operations on the predecessor model version. The designer can chose an existing

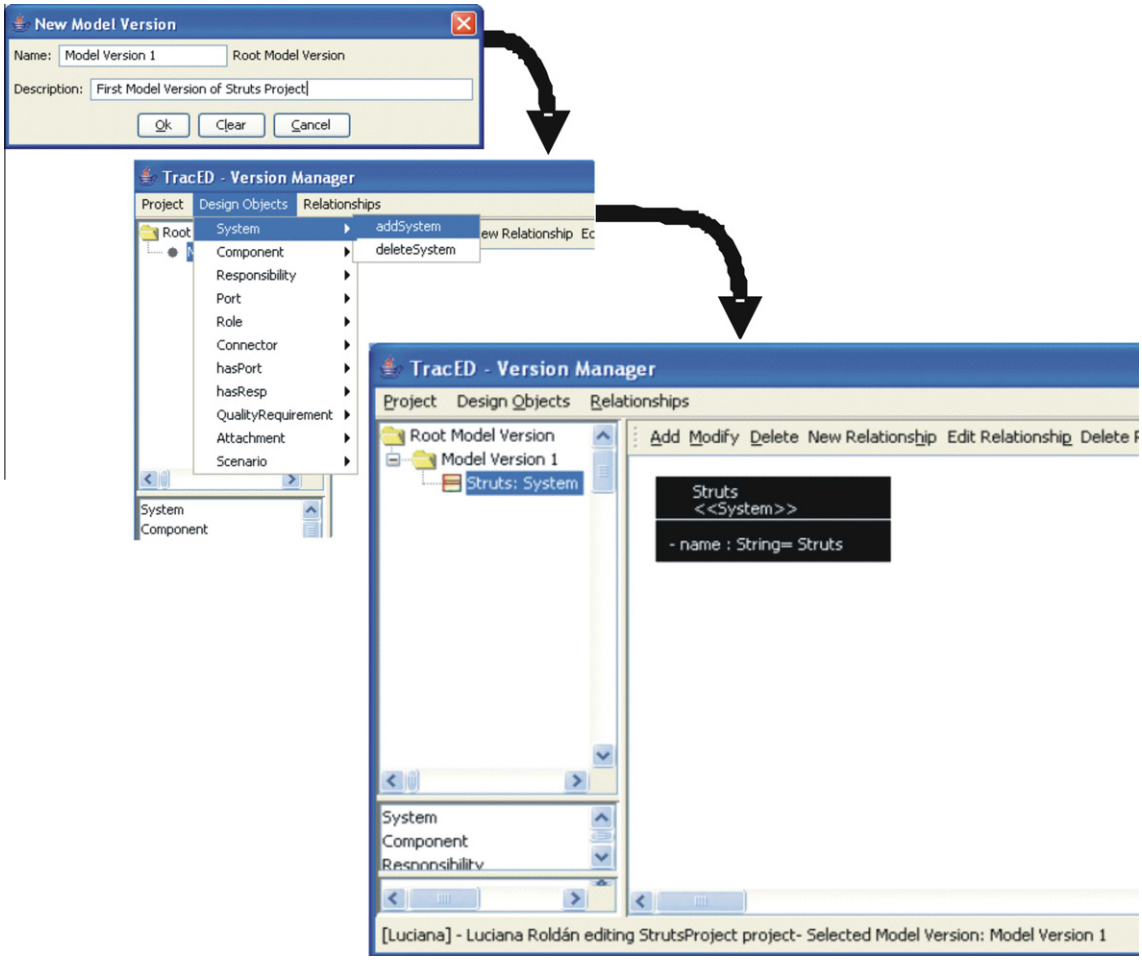

Fig. 19. Versions manager – creation of Model Version 1.

successor model version (a leaf model version), or can create a new model version from it.

3.2.3. Designing the Struts architecture

In order to guide the development of the case study on TracED, a SEI's design method for designing software architectures (ADD) was adopted, which was introduced in Section 2.1.1. Therefore, Struts' design process will be presented as a recursive decomposition process, which is guided by selected architectural drivers.

3.2.3.1. Creating the first model versions. In the context of a new project, the actor logged in TracED starts from the *Root Model Version*, which is empty. The first step is to create a model version (*Model Version* 1 in Fig. 19). At the model version's navigator panel, the user places the focus on this new model version, and executes one or more domain operations. In Design Objects menu, available operations for the current domain are arranged in submenus that correspond to each domain's modelling concept. For example, in order to select *addSystem* operation, system submenu must be accessed. The execution of *addSystem*(*Struts*) operation, which adds the first object version named *Struts* is shown in Fig. 19. Thus, a sequence of operations ϕ_1 consisting of a single *addSystem* operation must be applied on the root model version, in order to generate *Model Version* 1 (*apply*({*addSystem*(*Struts*)}, *Root Model Version*). The way of doing this by using TracED is slightly different from the one previously introduced because the actor must create the new model version first (providing a name, and an optional description, in this case, *Model Version* 1). After that, the designer executes successively each operation that forms part of the sequence of operations. Obviously, TracED is responsible for verifying the preconditions of each operation in the sequence. As the operations are executed on the predecessor version model, the sequence of operations applied to the predecessor model version is captured and the resulting (or successors) object versions populate the model version that was recently created.

At this point, the actor involved in developing *Struts* defines the architectural drivers for the intended architecture. They are *Modifiability* and *Testability*. *Modifiability* requirement aims to achieve *separation of concerns*, *loose coupling*, and *modularity* characteristics. Separation of concerns is desirable because there are many levels of functionality that need to be addressed. Modularity aspects allow functionalities developed in one project to be packaged independently and then be re-used across other projects; also allow developers to work independently and build upon each other's work. Finally, Struts requirement of *Testability* (which is related to loose coupling characteristics) aims to extending, enhancing, or correcting system features easier, by avoiding bugs caused by unexpected impacts of changes introduced in code, etc. Also refers to the easy way in which the software can demonstrate its faults through (typically execution-based) testing. Therefore, the architect incorporates the desired quality requirements for *Struts* system into the architectural design. This is done by executing a new sequence of operations $\phi_2 = \{addQualityRequirement(Modifiability), addQualityRequirement(Testability)\}$ on the predecessor model version (*Model Version* 1). For these operations, the precondition is that *Struts* object version exists in the predecessor model version (*Model Version* 1), which is true.

In order to proceed with the software architectures design process of *Struts*, *Quality Requirements* have to be translated into *Quality Scenarios*. *Quality Scenarios* provide a way of making a quality requirement concrete, and making the architectural design easier to be evaluated. These are the quality scenarios to be considered by the designer: (i) it is easy to add different view types such as HTML, WML, and XML (*ScModifiability*1); (ii) it requires just 1 day to incorporate into the project a new human resource, regarding training and "knowledge transferring" activities (i.e. a designer to work on the application view, a new programmer skilled in data access to work on the model, or another programmer to work in the business logic): the rest of the project is not affected (*ScModifiability*2); (iii) common style elements can be included by several

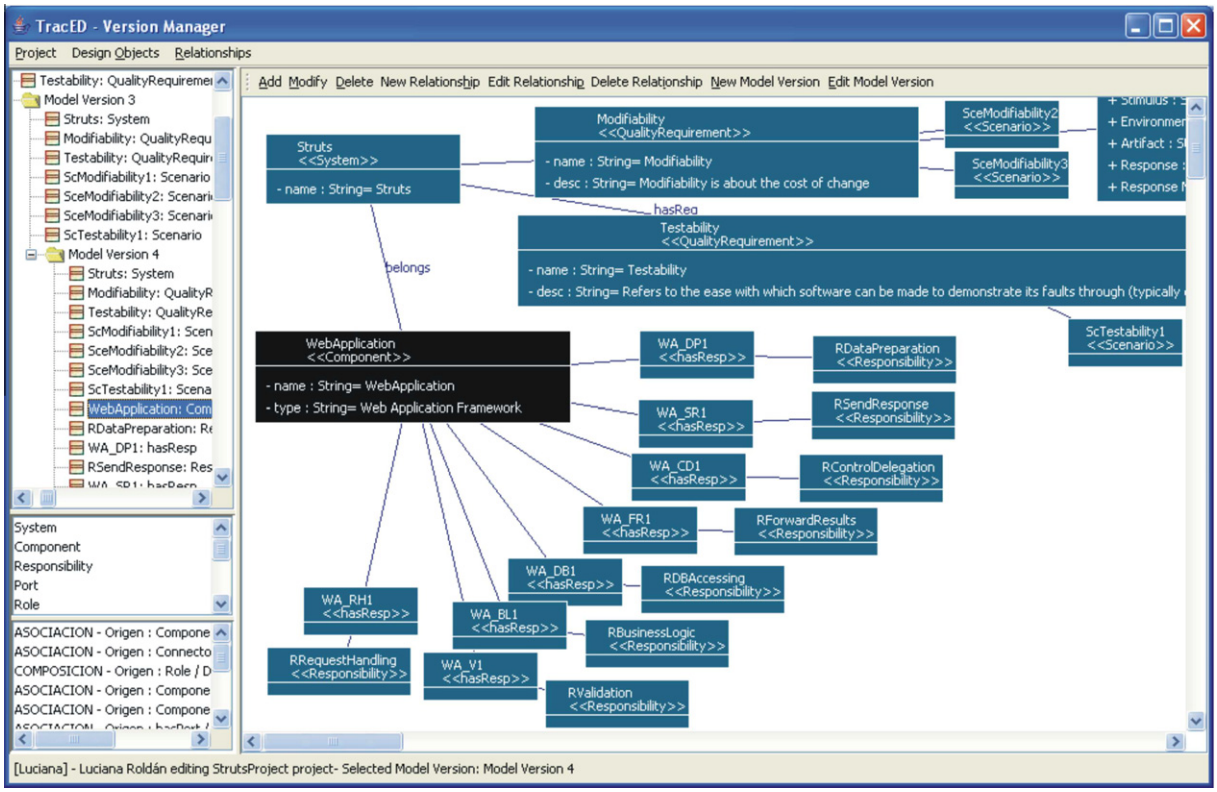

Fig. 20. Model Version 4 snapshot.

pages (*ScModifiability*3). The pages simply use the common definitions defined in a style sheet. If a style needs to be changed, that change can be made in one file rather than on each page individually; (iv) writing and running a unit test for a specific business logic component takes at most 1 person-hour (*ScTestability*1). The test methods exercise the class to be tested, and verify that such a class behaves as it is expected. Regarding this scenarios, a new model version *Model Version* 3 is attained by the execution of $\phi_3 = \{addScenario(ScModifiability1), addScenario(ScModifiability2), addScenario(ScModifiability3), addScenario(ScTestability1)\}$.

Next, the actor begins to build the architecture by incorporating structural elements. Therefore, the first component is added by executing an *addComponent* operation, which generates *WebApplication* object version. *WebApplication* represents the component that has the following responsibilities: handling of requests that come from browsers (*RRequestHandling*), validation making (*RValidation*), delegating of control on specific components (*RControlDelegation*), database interaction (*RDBAccessing*), processing of requested actions (*RBusinessLogic*), and updating of the state of the application and the user views (*RDataPreparation*, *RForwardResults*, *RGenerateResponse*). These responsibilities object versions are incorporated to the model version *Model Version* 4 as a consequence of applying *addComponent* operation, which includes in its body a loop over a responsibility named collection argument (see *addComponent* specification in Fig. 14). Fig. 20 illustrates the state of the resulting model version (*Model Version* 4). Note that the relation between the component and each responsibility has been reified as a *hasResp-type* object version. Additionally, it should be pointed out that all visualized associations are inferred from associations that exist at repository level (between versionable objects).

3.2.3.2. Applying model-view-controller style. At this point of the design process, the designer considers it is convenient to apply model-view-controller (*MVC*) style, in order to achieve the intended requirement of *modifiability*.

Having identified the quality requirements, the designer has in mind to perform the *applyMVC-go* operation (*applyMVC-go* specification was presented in Fig. 10) to the *WebApplication* component that belong to the current model version. Therefore, *applyMVC-go* belongs to the sequence of operation ϕ_5. As a result, a new model version, *Model Version* 5, is obtained where three new components *Model*, *View*, and *Controller* are present in *Struts* System (Fig. 21). Between *View* and *Controller* components, there is a connector named *ConnViewCtrlr*, which represents the interactions between them. Additionally, between *Model* and *Controller*, there is a connector in charge of informing changes to the state of the model (*ConnModCtrlr* connector in Fig. 21). There exists another connector (*ConnModView*) between *Model* and *View*, which represents the notifications sent by the *Model* to the *View* for updating. Then, as it is specified by the operation *applyMVC-go*, the responsibilities of the original component are delegated to each new component.

However, not just structural and behavioural elements are added by the operation execution. *ApplyMVC-go* operation also adds a set of object versions for representing design rationale. In fact, an explicit *RPossibleScenarioSolution* object versions is included, which provides the links between the new architectural elements and the scenario to be meet (in this case, *ScModifiability*2, related to *Modifiability* requirement). This version, labelled as *PSS1*, is shown in Fig. 21, where inferred repository associations are highlighted and labelled as *pss*. In this way, the captured object versions allow tracing the operation results (the impact of the decision on the architecture model) and the intended goals of the executed operation.

The case study goes on by evolving the architectural model in more complex model versions. The Web adds a constraint to software developers, given by the stateless connection between the client and the server, which made difficult for the *Model* to notify changes to the *View*. On the Web, the browser has to re-query the server to discover modifications to the application's state. For this reason, the designer decides to relax MVC pattern by deleting the existing connector between *Model* and *View* component. Thus,

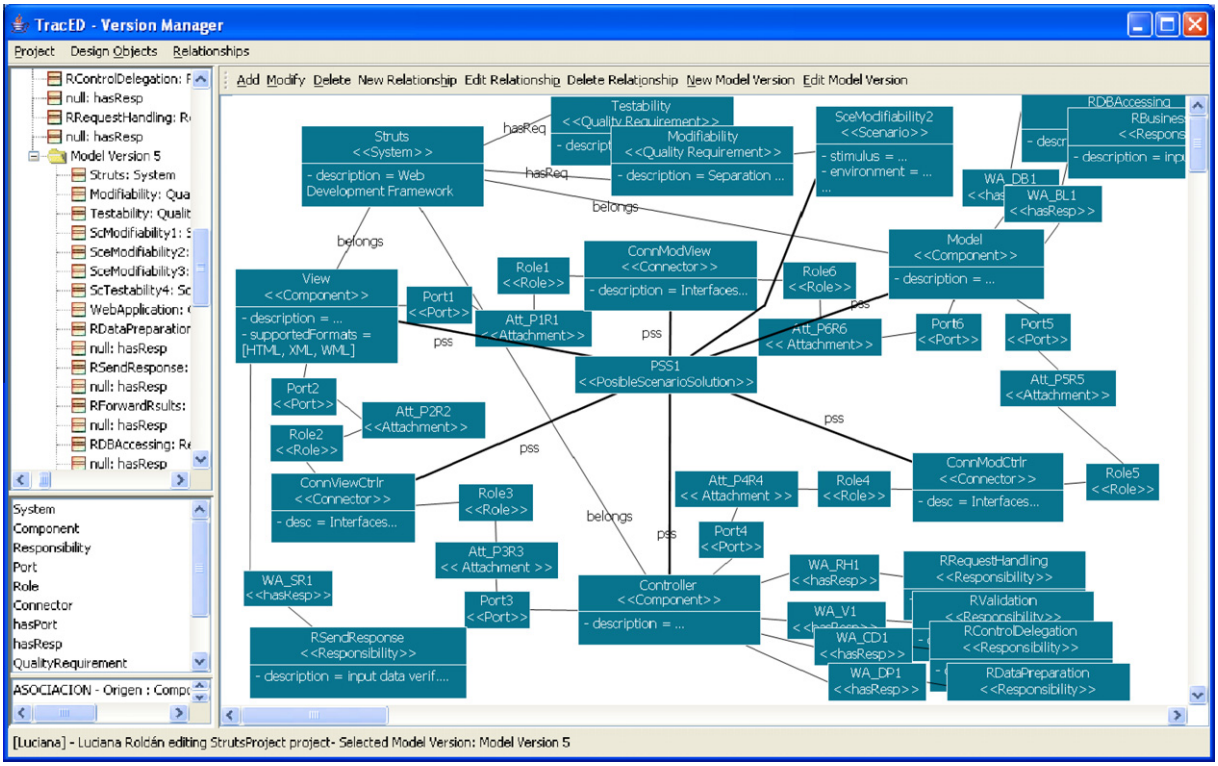

Fig. 21. Partial view of Model Version 5, after *applyMVC* execution.

Model Version 6 is reached as a consequence of applying ϕ_6 = {*deleteConnector*(*ConnModView*)}. Such a change means that the *Controller* is responsible for managing the control flow between the *Model* and the *View*. Fig. 22 shows the new model version that is obtained (*Model Version* 6).

At this point of the design process, by analyzing the last model version, the designer is able to verify if the scenarios that were set at the beginning of the design process have been fulfilled. Thus, the designer may execute *setAssessment* operations, incorporate design rationale information to the next model version, and evaluate if it is necessary to continue the design process.

3.2.4. Recovering information about an engineering design process

In general, the need of recovering how an engineering design project has evolved is a fact. For instance, in a software development project, it would be very useful to know how elements have evolved from the first stage (where requirements were specified) to implementation stage (where programming related objects are generated). All steps and components of the project should be clearly shown to the actor or user, thus allowing him to reconstruct and analyze the design process.

As it was previously pointed out, a *version history* link is created for each executed *operation*. Thus, it is possible to reconstruct the history of a given model version by beginning from the *Root Model Version*. Fig. 23 shows how TracED presents the history information.

TracED allows the user to see the history of a given model version. He can see which the predecessor was. By selecting a model version from model version's navigator panel, it is possible to see what happened through the time. Fig. 23 shows the History Window, which informs all operations that have been applied from the root to the selected model version. It shows the sequence of operations form by *addSystem* which has been applied on *Root Model Version*. In this window, it is possible to see detailed information about each applied operation. For instance, the time in which an operation was applied, who the actor involved was and the names assigned to the new object versions (successors object versions). At the top of *Model Version* 1, the sequence of operations that generates *Model Version* 2 is depicted. In this case, it involves a series of *addQualityRequirement* operations. Similarly, *Model Version* 3 is the result of applying a series of *addScenario* operations on *Model Version* 2. Finally, moving the focus to *Model Version* 4, the history window shows that an *addComponent* operation was executed at the top of *Model Version* 3. For each executed operation, the history window lists all the successor object versions that were generated because of such execution.

The History Window is just a way of presenting information about a design process.

However, the model allows generating several queries to exploit the information kept by all the captures. Based on the design process that was captured during the present case study, the model can answer questions like the followings:

3.2.4.1. By means of which design operations Model Version 5 was obtained?. This can be answered by concatenating all operation sequences from the initial model version (*root model version*) or from a given predecessor model version.

> Sequence of operation $\phi_1 \cdot \phi_2 \cdot \phi_3 \cdot \phi_4 \cdot \phi_5$ = {*addSystem*(*Struts*), *addQualityRequirement*(*Modifiability*), *addQualityRequirement*(*Testability*), *addScenario*(*ScModifiability*1), *addScenario*(*ScModifiability*2), *addScenario*(*ScModifiability*3), *addScenario*(*ScTestability*1), *addComponent* (*Struts*, *WebApplication*, [*RRequestHandling*, *RValidation*, *RControlDelegation*, *RDBAccessing*, *RBusinessLogic*, *RDataPreparation*, *RForwardResults*, *RGenerateResponse*], []), *applyMVC-go*(*WebApplication*, *Modifiability*)}

3.2.4.2. Which are the alternative successor model versions of Model Version 4?. In the case study, just one possible state was explored; this is achieved through the application of MVC architectural pat-

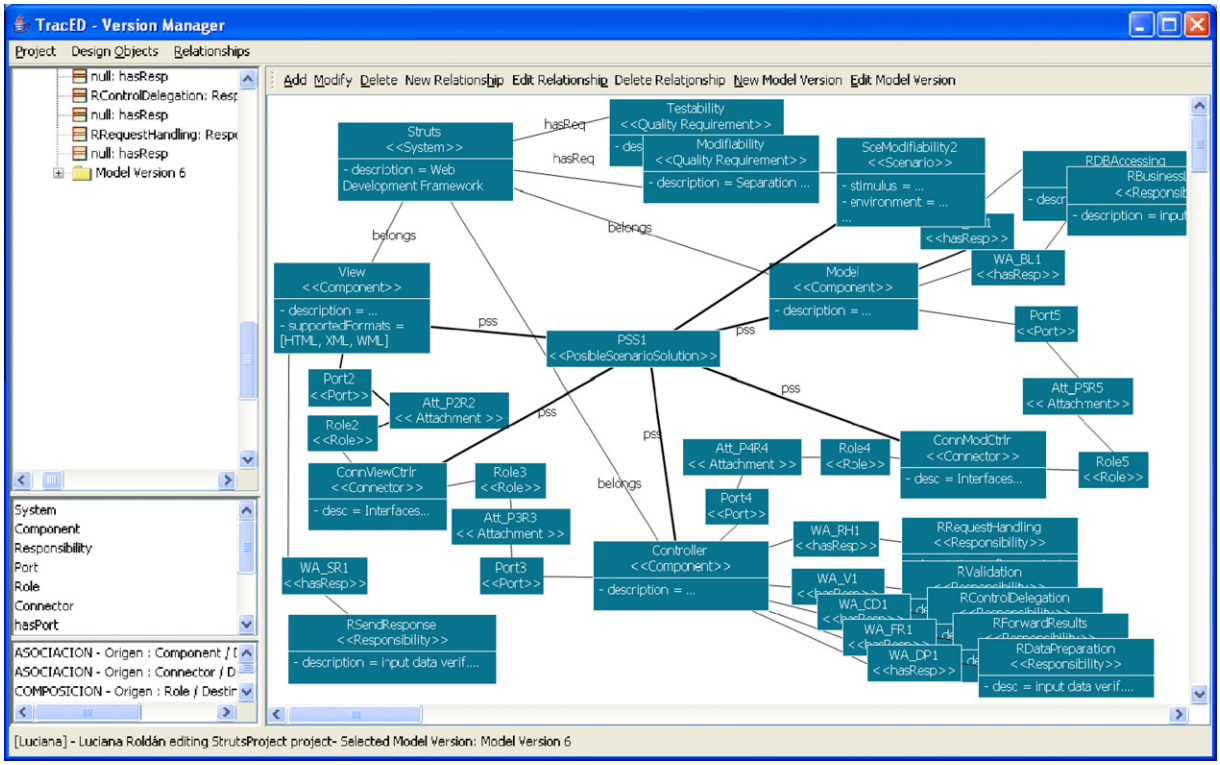

Fig. 22. Relaxing the *MVC* style in order to make the architecture suitable for a Web environment.

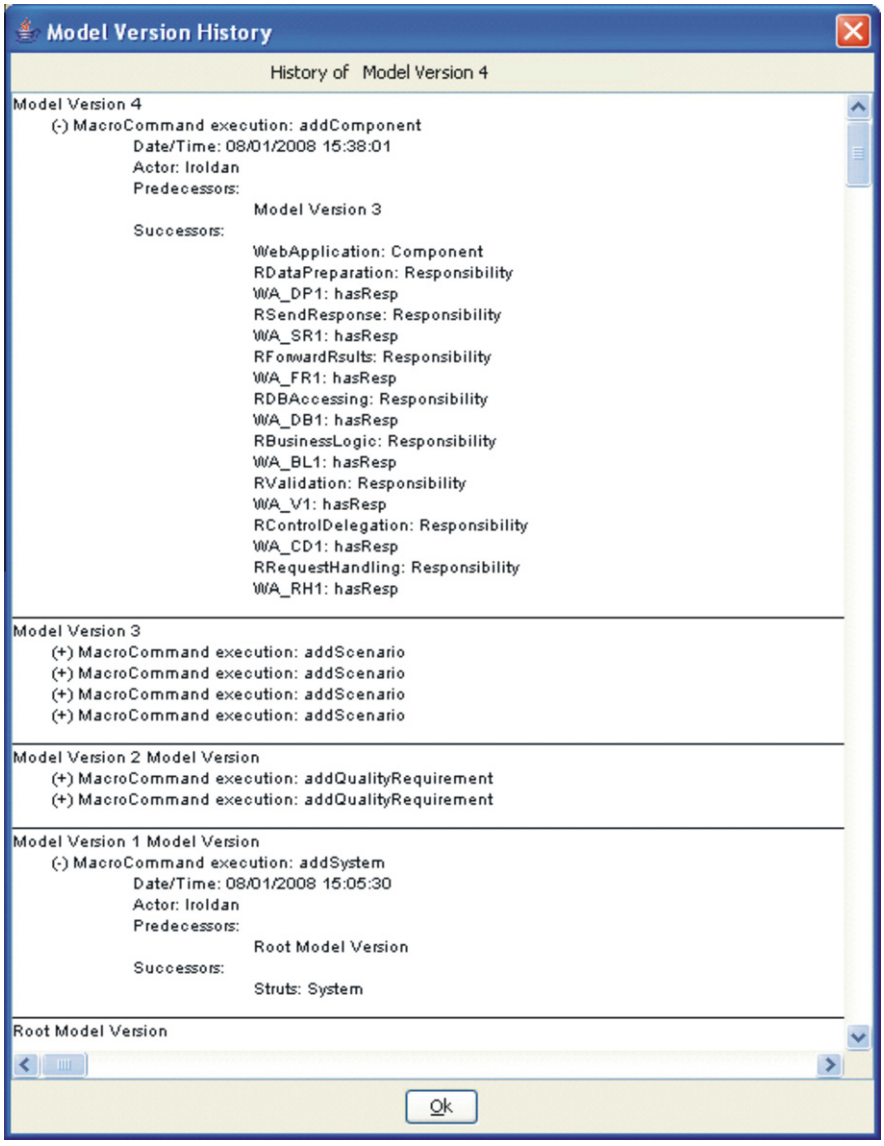

Fig. 23. Consulting the history of Model Version 4.

tern (made concrete by applyMVC-go operation), which obtains Model Version 5. However, the designer could have considered another solution, for instance, to apply a two-tiered pattern, which would generate an alternative successor model version. In this case, Model Version 4 would be the predecessor of the two alternative model versions.

3.2.4.3. How did a WebApplication component change along the design process?. This information is captured by the history links. It is possible to know that the *WebApplication* component was added by the execution of the sequence of operations ϕ_4. Furthermore, this component was refined by the execution of the *applyMVC-go* operation when the sequence of operation ϕ_5 was carried out. The component was refined in a set of three interrelated components (*View*, *Model* and *Controller*). In order to make the case study shorter, it does not show details about changes in particular properties of *WebApplication* component. This type of change would arise several object versions of it (materialized by executions of *modify* operations).

3.2.4.4. Who were the actors involved in the generation of WebApplication object version?. Despite it not having been detailed deeply in this contribution, *history* links also captures information related to the management of the design process. For instance, the history link that captured the *addComponent* operation that created *WebApplication* (as part of the sequence of operations ϕ_4) maintains that the user *"Luciana Roldán"* was the responsible of their execution (actor lroldan in Fig. 23).

3.2.4.5. Which design operations have designers applied with the intention of addressing modifiability requirement?. From a requirement (*modifiability* in this case) and a given model version, it is possible to obtain all the object versions whose type is *RPossibleScenarioSolution*. These object versions are linked to the scenarios that make assessable the intended requirement. Therefore, by navigating through these object versions is possible to know the operations that generated such object versions.

3.2.4.6. Which were the new products generated because of the execution of applyComponent in sequence of operations ϕ_4?. The results are the new component, its responsibilities, and ports. They are the following object versions: *WebApplication* (*Component*), *RDataPreparation* (*Responsibility*), *WA_DP*1 (*hasResp*), *RSendResponse* (*Responsibility*), *WA_SR*1 (*hasResp*), *RForwardRsults* (*Respon-*

sibility), *WA_FR*1 (*hasResp*), *RDBAccessing* (*Responsibility*), *WA_DB*1 (*hasResp*), *RBusinessLogic* (*Responsibility*), *WA_BL*1 (*hasResp*), *RValidation* (*Responsibility*), *WA_V*1 (*hasResp*), *RControlDelegation* (*Responsibility*), *WA_CD*1 (*hasResp*), *RRequestHandling* (*Responsibility*), and *WA_RH*1 (*hasResp*).

3.2.4.7. What were the architectural approaches applied to reach the Model Version 6?. The architectural approaches [34] are represented by styles application operations defined in the design domain. If we search operations of this type on the sequence of operations resulting from the concatenation of the sequences that derived on Model Version 6, thus $\phi_1 \cdot \phi_2 \cdot \phi_3 \cdot \phi_4 \cdot \phi_5 \cdot \phi_6$, we find out that *applyMVC-go* was performed. Therefore, the MVC pattern was employed.

4. Related work

Tracing of engineering design processes has been addressed in different engineering design domains. An example is the KDBS design support system for conceptual design of chemical processes [35]. The KBDS environment makes the representation of the designer's intentions possible and allows model traceability, reflecting to a certain extent the design history. The authors extended KBDS [8] by incorporating a stuff of concepts to relate the representation of argumentation with the generated products that were recorded in design process history. As Bañares-Alcántara and King [8] argue, the extension is conceived to record the design rationale after decisions are made but not while they are being made. It is responsibility of the designers (or design teams), to build the IBIS network that represents the design rationale in a explicit way. Furthermore, operations of the design process such as the request of additional information, or the inclusion of explicit arguments to justify a decision, are not captured while the design process is carried out. Therefore, several products of process are not recorded at they are generated/modified, missing the links among them and the design process itself.

Pohl and collaborators [36] proposed an approach called PRIME that has been applied in both software and mechanical engineering, which supports method-driven trace capture. We agree with them in that traceability is a prerequisite for managing evolution of systems, but sometimes it becomes expensive and labour intensive. They argue that in order to minimize the information to be recorded and to reduce additional costs, the types of trace information to be captured should be adjusted to project specific needs. Therefore, a project manager must be supported in the definition of the trace information and the trace steps required for recording this information. An unclear point of the proposal is how the products and actions of interest can be defined for a project, something that TracED allows by means of the domain and operations models. In addition, several approaches [37,38] have been proposed to support architectural design processes. The ideas presented in these proposals are certainly transferable to other design domains. Argo [37], for instance, supports the design process through the use of critics, task organization and priorization. Particularly, critics consist on agents that watch for specific conditions in the partial design as it is being constructed, and notify the designer when those conditions are detected. Critics can be used to advice to designers about the implications of, or the alternatives to, a design decision. In Argo, critics place their feedback in the designer's "to do" list. Each "to do" item remains the designer to address an open design issue. Argos's process modelling extension allows modelling the task involved in a typical design process. The process model is represented like a connected graph of design materials. In the same research line and adopting an operational perspective similar to the employed in this work, Diaz-Pace and Campo [38] propose a hierarchical planning approach, named Designbots, to explore software architecture alternatives. Both Argo and Designbots approaches are concerned in guiding a designer through the design process, which is different than, but complementary to, the focus of this paper. In fact, the knowledge captured by tools like TracED forms the foundation for design-assistance tools such as Argo and DesignBots.

Additionally, the recent interest in design decisions stimulated the development of several decision-based architectural tools [39]. In spite of the fact that most of them support the notion of design decisions, none of them represents design decisions as concrete executions of design operations as our approach does. In addition, the versioning administration model we propose provides the elements to capture the operations together with their results (successor object versions). This integrated capture of products and operations avoids the designer the need of setting explicitly the relation between architectural elements and architectural decisions.

5. Conclusions

In this paper, TracED, a tool for capturing and tracing engineering design processes is presented. The approach provides the mechanisms to capture the several products generated during a engineering design process, as well as the elements for specifying, executing, and capturing the operations that generated them. Based on an extensible model, TracED allows the definition of a particular domain and the operations suitable for it. Possible engineering design processes that could be managed using TracED are chemical engineering processes, or software engineering processes. The extension proposed in this contribution focus on software architectures design process. In order to support such a process, a suitable work domain was proposed, which was used in an example design project to design the architecture of a well-known development framework. The proposed model has been successful applied to another specific software engineering domain: mobile systems architectures [29]. Thanks to the flexibility of the model, modelling concepts and operations suitable to the design of software architectures for mobile systems were defined and a case study was carried out.

Situation calculus provides a formal framework to express which effects of the execution of certain operation are and when it can be applied. In addition, it allows the definition of a formal model for conflict detection in collaborative design. Taking this advantage into account, a future extension of the presented approach consists of supporting the conflict management process, developing features for conflict detection and resolution tasks. To overcome such troubles, the tool must enable the capture of the operations and actors that originated each product. This capture need to be based on the explicit mechanism to manage different versions of the design products proposed in the present contribution. A first application of the present proposal for collaborative environments has been introduced in [12].

Although the valuable contributions of the computational tool presented in this paper, it has some limitations because of its prototypical status. One of its weak points is the lack of a suitable graphical interface, which allows describing model versions by using various views. Regarding software architecture domain, even though an architect can specify several modelling concepts relative to any view type by means of domain editor (i.e. *Process*, *Resource* and *Thread* modelling concepts regarding a process view), Versions Manager always shows all the object versions that belong to a given model version. It is more valuable for the designer to display just those object versions of interest, according to a given view type.

Another aspect to improve is TracED's user interface. Textual outputs are not user friendly, thus, the more numerous and complex the applied operations are, the more illegible the textual report becomes. Another form of presentation consists on allowing bidirectional navigation between predecessor and successor (resulting) object versions, displaying in a contextual window the applied operation. It would be useful to provide an animated feature to replay a design process (or a fragment of a design process) using a graphical representation of the models, so the designer can actually see the changes occurred between model versions. Furthermore, the information captured could be represented using similar approach employed by SEURAT [40] or [41] to support the visualization and use of design decisions and their rationale. SEURAT [40] displays the decisions by hierarchical tables. In [41], tabular list and graphs are used to visualise the design decisions. Despite these limitations, current prototype of TracED is robust enough for experimental usage.

In addition, besides the implementation decisions made to build the prototype, other viable alternatives exist. One possibility is the integration of TracED with a CASE tool intended to support design activities. In the case of software architectures design process, tools like ACMEStudio [42] and ArchStudio [43] have been proposed. They have been developed in Eclipse, so they can be extended by adding TracED as a plug-in. In this way, TracED would perform the capture of all applied operations, by working in background mode, without designer noticing it. Additionally, given the capabilities provided by the Operations Model, new and higher level of abstraction operations could be specified for the CASE tool, able of being included in the user menus.

Next research steps are concerned to how to exploit all captured design information (applied design operations and resulting version) in order to generate design knowledge valuable for carry out other design projects, facilitate learning activities for inexpert designers and make the understanding of past design decisions easier. Some ideas to explore theses issues are the use of case based reasoning and Bayesian nets for generating design assistance guidelines, and the proposal of specific query mechanisms to yield (semi) automatic documentation.

Acknowledgments

The authors wish to acknowledge the financial support received from CONICET (PIP 2754), Universidad Tecnológica Nacional (25/O118 – UTI1083) and Agencia Nacional de Promoción Científica y Tecnológica (PAE PICT 02315). Also, they thank the anonymous reviewers that contributed with their comments to improve the quality of this paper.

References

[1] Keane A, Nair P. Problem solving environments in aerospace design. Adv Eng Software 2001;32(6):477–87.
[2] Chapman C, Pinfold M. Design engineering – a need to rethink the solution using knowledge based engineering. Knowl-Based Syst 1999;12(5–6):257–67.
[3] Goel V. A comparison of design and nondesign problem space. Artif Intell Eng 1994;9(1):53–72.
[4] Boyle JM. Interactive engineering system design: a study for artificial intelligence applications. Artif Intell Eng 1989;4(2):58–69.
[5] Goldschmidt G. Capturing indeterminism: representation in the design problem space. Des Stud 1997;18(4):441–55.
[6] Lee J, Lai K. What's in design rationale? Human–Comput Interact 1991;6(3–4):251–80.
[7] Lee J. Design rationale systems: understanding the issues. IEEE Expert 1997;12(3):78–85.
[8] Bañares-Alcántara R, King J. Design support systems for process engineering III. Design rationale as a requirement for effective support. Comput Chem Eng 1997;21(3):263–76.
[9] Kunz W, Rittel HWJ. Issues as elements of information systems. Working paper 131, Institute of Urban and Regional Development, University of California, Berkeley; 1970.
[10] Carnduff T, Goonetillake J. Configuration management in evolutionary engineering design using versioning and integrity constraints. Adv Eng Software 2004;35(3–4):161–77.
[11] Westfechtel B. Models and tools for managing development process. Lecture notes in computer science, vol. 1646. Berlin: Springer; 1999.
[12] Gonnet S, Leone H, Henning G. A model for capturing and representing the engineering process. Expert Syst Appl 2007;33(4):881–902.
[13] Bass L, Clements P, Kazman R. Software architecture in practice. 2nd ed. Boston: Addison-Wesley; 2003.
[14] Jansen A, Bosch J. Software architecture as a set of architectural design decisions. In: Proceedings of the 5th IEEE/IFIP working conference on software architecture (WICSA 2005); 2005.
[15] ANSI/IEEE Std 1471. Recommended practice for architectural description of software-intensive systems, ISO/IEC 42010; 2007.
[16] Garlan D, Monroe RT, Wile D. Acme: architectural description of component-based systems. In: Leavens GT, Sitaraman M, editors. Foundations of component-based systems. Cambridge University Press; 2000. p. 47–68.
[17] Medvidovic N, Taylor RN. A classification and comparison framework for software architecture description languages. IEEE Trans Software Eng 2000;26(1):70–93.
[18] Jansen A, van der Ven J, Avgeriou P, Hammer D. Tool support for architectural decisions. In: Proceedings of the 6th working IEEE/IFIP conference on software architecture (WICSA 2007); 2007.
[19] Tang A, Nicholson A, Jin Y. Using bayesian belief networks for change impact analysis in architecture design. J Syst Software 2007;80(1):127–48.
[20] Bachmann F, Bass L, Klein M. Preliminary design of ArchE: a software architecture design assistant. Technical report CMU/SEI-2003-TR-021, Software Engineering Institute, Carnegie Mellon University; 2003.
[21] Roldán ML, Gonnet S, Leone H. A model for capturing and tracing architectural designs. In: IFIP international federation for information processing, vol. 219/2006. Advanced software engineering: expanding the frontiers of software technology. Boston: Springer; 2006. p. 16–31.
[22] Concheri G, Milanese V. MIRAGGIO: a system for the dynamic management of product data and design models. Adv Eng Software 2001;32(7):527–43.
[23] Reiter R. Knowledge in action: logical foundation for describing and implementing dynamical systems. Cambridge, MA: The MIT Press; 2001.
[24] Lin F. Situation calculus. In: van Harmelen F, Lifschitz V, Porter B, editors. Handbook of knowledge representation; 2008. p. 649–69.
[25] OMG. Unified modeling language (OMG UML). Superstructure, V2.1.2; 2007. <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF>.
[26] Eggersmann M, Gonnet S, Henning G, Krobb C, Leone H, Marquardt W. Modeling and understanding different types of process design activities. Latinoam Appl Res 2003;33(2):167–75.
[27] Wojcik R, Bachmann F, Bass L, Clements P, Merson P, Nord R, et al. Attribute-driven design (ADD), Version 2.0. Technical report CMU/SEI-2006-TR-023, Software Engineering Institute, Carnegie Mellon University; 2006.
[28] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns. Elements of reusable object-oriented software. Reading, MA: Addison Wesley; 1995.
[29] Roldán ML. Un modelo para la representación de conocimiento y razonamiento en el proceso de diseño basado en arquitecturas de software. Tesis doctoral, Facultad Regional Santa Fe, Universidad Tecnológica Nacional; 2009.
[30] OMG. Object constraint language, OMG available specification, Version 2.0; 2006. <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>.
[31] MySQL documentation team. MySQL reference manual; 2006.
[32] Bauer C, King G. Hibernate in action. Greenwich: Manning Publications; 2005.
[33] Holmes J. Struts: the complete reference. New York: McGraw-Hill/Osborne; 2007.
[34] Clements P, Kazman R, Klein M. Evaluating software architectures. Methods and case studies. Addison Wesley; 2002.
[35] Bañares-Alcántara R, Lababidi HMS. Design support systems for process engineering II. KBDS: an experimental prototype. Comput Chem Eng 1995;19:279–301.
[36] Pohl K, Weidenhaupt K, Dömges R, Haumer P, Jarke M, Klamma R. PRIME—toward process-integrated modeling environments. ACM Trans Software Eng Methodol 1999;8(4):343–410.
[37] Robbins J, Hilbert D, Redmiles D. Extending design environments to software architecture design. Automated Software Eng 1998;5(3):261–90.
[38] Díaz-Pace JA, Campo M. Exploring alternative software architecture designs: a planning perspective. IEEE Intell Syst 2008;23(5):66–77.
[39] Tang A, Avgeriou P, Jansen A, Capilla R, Ali Babar M. A comparative study of architecture knowledge management tools. J Syst Software 2010;83(3):352–70.
[40] Burge J, Brown D. Software engineering using rationale. J Syst Software 2008;81(3):395–413.
[41] Lee L, Kruchten P. A tool to visualize architectural design decisions. QoSA 2008:43–54.
[42] Institute for software research. ArchStudio 4. Software and systems architecture development environment. <http://www.isr.uci.edu/projects/archstudio/index.html>.
[43] Schmerl B, Garlan D. AcmeStudio: supporting style-centered architecture development (research demonstration). In: Proceedings of the 26th international conference on software engineering; 2004.