A Tool for Capturing and Tracing the Software Architecture Design Process

María Luciana Roldán, Silvio Gonnet, and Horacio Leone

CIDISI – INGAR / UTN – CONICET, Avellaneda 3657, 3000, Santa Fe, Argentina
{lroldan, sgonnet, hleone}@santafe-conicet.gov.ar

Abstract. The design of a software architecture involves creative and unstructured activities with an opportunistic control flow. During this process, several models are generated, which can evolve in several versions and represent the intended artefact in different levels of abstraction. Given the difficulties in dealing with this complexity by using an improvised way, there is an urgent need for tools that support the capture and administration of this process. In this proposal, TracED, a computational environment and its underlying model to support the capture and tracing of software architectures design processes is presented.

Keywords: Design Process Support, Software Architecture.

1 Introduction

Software architectures design, as the design in other engineering disciplines, is a challenging task. Software architectures design process (SADP) involves several activities such as exploration, evaluation, and composition of design alternatives [1]. To address these activities, the research community has proposed modelling languages [2], design methods [3] and computer environments for architect assistance [1]. Those tools are basically focused on assisting designers in generating a software architecture that satisfies a set of requirements. However, documentation of associated rationale and design decisions are often left out, and are lost over time. Consequently, capturing design decisions is essential to capitalize previous designs [4]. In this paper, TracED, a computational environment to support the capture and tracing of SADP is proposed. TracED can be adapted to each new design problem. This adjustment is subjected to the concepts of a given design method, the employed architectural description language and the considered software architecture sub-domain (such as avionics, embedded systems, business). The possible operations that can be applied while SADP is being carried out also extend TracED. In Section 2, a conceptual and generic model for capturing and tracing a design process is presented. Then, suitable extensions are introduced for making it applicable to SADP. Afterwards, the main features of TracED are explained in Section 3. Finally, we conclude in Section 4.

XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), pp. 380-389, 2008.

2 A Model to Capture and Trace SADP

The proposed scheme considers SADP as a sequence of activities that operate on the products of the design process, called *design objects*. Typical *design objects* are models of the artefact being designed (i.e. the components and connectors that comprise the software architecture), and specifications to be met (i.e. quality requirements such as modifiability or performance). Naturally, these objects evolve as the SADP takes place, giving rise to several versions that must be kept. They are represented in two levels, the *repository* and the *versions' level*. The *repository* level keeps a unique entity for each *design object* that has been created and/or modified due to model evolution during a design project. This object is called *versionable object* (Fig. 1). Furthermore, relationships among the different *versionable objects* are maintained in the repository (*Association*, Fig. 1). On the other hand, the *versions' level* keeps the different versions of each *design object*. These are called *object versions* (Fig. 1). The relationship between a *versionable object* and its *object versions* is represented by the *version* association. Therefore, a given *design object* keeps a unique instance in the *repository* and all versions it assumes in different model versions belong to the *versions' layer*. At a given stage on the execution of a design project, the states assumed by the set of relevant *design objects* supply a snapshot of the state of SADP (called *model version*). According to the proposed model, a new model version is generated by applying a *sequence of operations* (ϕ_q) on a *predecessor model version*. Therefore, the representation scheme of model versions has a tree structure, where each *model version* is a node and the root is the *initial model version*. As this model evolution is posed as a history made up of discrete situations, Gonnet et al. [5] adopt the situation calculus for modelling such version generation process. They define a *belong*(*v*,*m*) predicate using the successor state axiom, which enables knowing the object versions (*v*) that belong to a model version (*m*). This makes possible to reconstruct a *model version* m_{i+1} by applying all operation sequences from the initial *model version* m_0. Lack of space prevents us from showing its formal representation.

The primitive operations that were proposed in [5] to represent the transformation of *model versions* are *add*, *delete*, and *modify*.

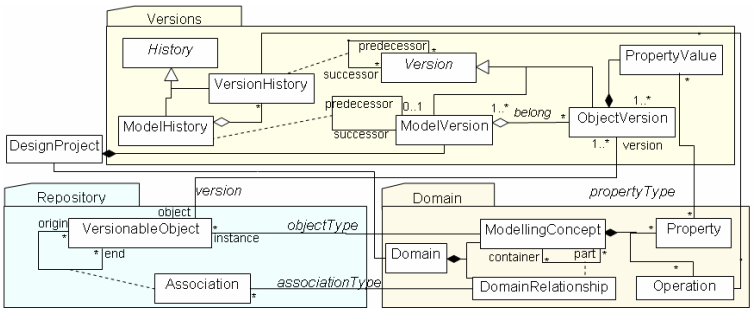

Fig. 1. Process version administration model.

By using the *add*(*v*) operation, an *object version v* that did not exist in a previous *model version* can be incorporated into a successor *model version*. Conversely, the *delete*(*v*) operation eliminates an *object version v* that existed in the previous *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 it.

Each operation applied to a *model version* is captured by means of *VersionHistory* relationships (Fig. 1). They allow tracing the model evolution keeping references among the *object versions* on which the *operation* was applied and the ones arising as result of its execution. Additionally, *VersionHistory* instances are aggregated in a *ModelHistory* instance, to represent the *sequence of operations* (activity) that caused a model evolution.

Fig. 2 illustrates the described schema to represent model versions, regarding a fragment of a SADP. The example presents two model versions 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* represent instances of concepts of architectural description languages such as ACME [2]; and the primitive operations have been extended by operations as *applyMVC* (explained in Section 2.2). This operation applies MVC style and refines the *WebApplication* component on *View*, *Model*, and *Controller* components, with their ports and connections. In Fig. 2, the *inferred models level* is obtained from views produced by the *version level* on the *repository*. A *Struts* system belongs to both inferred model versions. At repository level, it is represented by *S*, an instance of *versionable object*. *S* is linked with the versionable objects that represent *WebApplication*, *View*, *Model*, and *Controller* components (*W*, *V*, *M*, and *C* versionable objects, respectively). As Fig. 2 shows, *S* has an object version *V1S* that belongs to *model version* m_k and m_q. 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 m_q. As a consequence of *applyMVC* execution, *V1W* was deleted from the successor model version (m_q) and the new object versions representing the model, view and controller components were added to m_q.

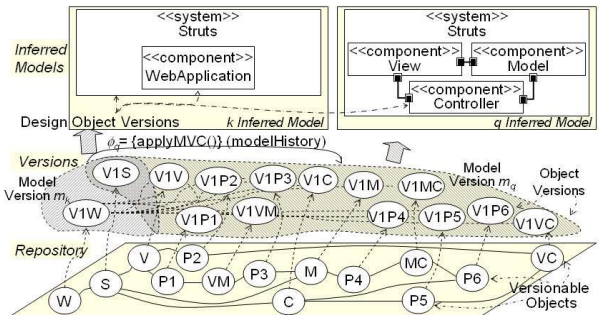

Fig. 2. Representation scheme of model versions.

2.1 The Domain Model

Domain Package (Fig. 1) enables the definition of concepts of software architectures domain, whose instances are going to be captured (such as system, component, connector, and port in Fig. 2). Therefore, for each design object type, an instance of *ModellingConcept* must be generated. Additionally, its properties are specified by a set of instances of *Property* class. Furthermore, the relationships among those concepts are instantiated from *DomainRelationship*. In Fig. 3, we illustrate a software architecture domain model, where the modelling concepts have been taken from ADD [3]. It comprises concepts such as quality and functional *requirement*, *scenario*, and *assessment*. Given that ADD proposes various types of views for representing the software architecture, ACME [2] has been chosen as the architectural description language that provides the notation for representing the structural view of the software architecture. Therefore, concepts such as component, connector, ports, roles and responsibilities have been added to the domain model.

2.2 The Operations Model

The primitive operations *add*, *delete*, and *modify* have to be extended with suitable operations for a design domain such as *applyMVC* operation employed in Fig. 2. To provide the foundations for computational tools, an object-oriented operations model is proposed, which is flexible enough for specifying the domain's operations.

Therefore, a *command* abstract class is introduced in the *Operations* package illustrated in Fig. 4. An *operation* is defined as a *macro command*, a subclass of *command* that simply executes a sequence of commands. For it, it must be defined its *arguments* and *body*. The commands of the body can be primitives (such as *add*, *delete*, or *modify*), iteration commands, variable assignment, or other existent operations. *Iteration* represents a command with a repetitive behaviour. It is specialized in *Loop*, which executes a body (commands sequence) for each element in a collection, and in *Next*, which iterates through a collection of elements accessing each element. Another command is *VariableAssignment* that represents the assignment of a value to a variable of a given type. Note that the *modelling concept* over which an operation is applied must be explicitly indicated.

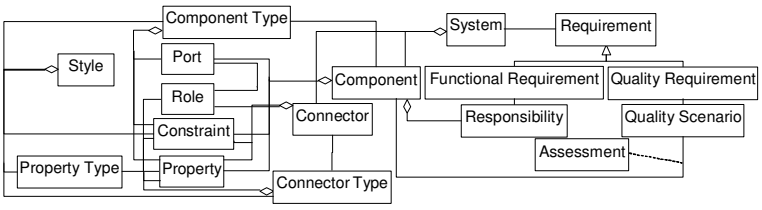

Fig. 3. A Domain Model.

Furthermore, every *command* has one or more data typed *arguments*, which are a kind of *variable*. Also, a *variable* has a type and can be declared and used in the body of an *operation*. *DataType* class generalizes the available types: *PrimitiveDataType*, *CollectionType*, and *ModellingConcept*. Moreover, *VariableAssignment* denotes the mapping between a *Variable* and a *RunTimeValue*. This interface represents the run-time values during the execution of an operation, which can be realized by different values like *literal*, *object version*, *modelling concept*, or *PropertyValue*.

Many possible operations can be instantiated from the operations model to be used during a SADP. In [6] we specify operations that range from basic ones like *addComponent*, *deleteComponent*, and *addScenario* to operations that apply a style or tactic, like *applyClientServer*, and *applyMVC*. Here, space limits us to just discussing the *addComponent* and *applyMVC* operations. Fig. 5 presents functional specifications of these operations. They give an outline of how the operations could be specified using a computational tool. The rest of the operations can be defined in a similar way, as they are defined in terms of 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*, which is carried out by a series of operations. First, a version of component *c* is added. After that, a set of responsibilities (l_{Resp}) and ports (l_{Ports}) are inserted. Some of these design objects can have associations at repository level, which are incorporated by means of a pseudooperation *addRelationship*.

Fig. 5 presents the specification of the *applyMVC* operation. The *Model*, *View*, and *Controller* components are generated by *applyMVC* execution, along with their responsibilities and ports, as well as the connectors between them. Additionally, the responsibilities and the scenarios to be satisfied by the original component *c* are delegated to new components (*delegateResponsibilites* and *delegateScenarios* operations), by interacting with the actor who executed the operation.

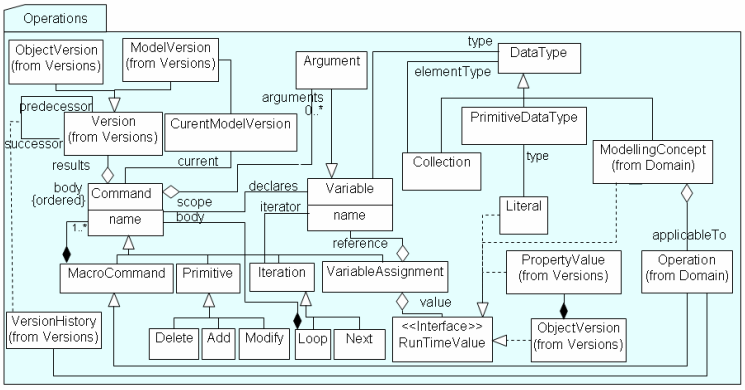

Fig. 4. Operations Package.

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

```
applyMVC(s, 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)
  deleteComponent(s, c) //Predecessor object version is removed
```

Fig. 5. Specifications of operations.

3 TraCEd

TracED is a research prototype for validating the proposed model to capture and trace engineering designs. It has been developed using Java language, MySQL database, and Hibernate framework. The major tools are *Domain Editor* and *Versions Manager*.

Domain Editor is the tool that enables the definition of a SADP domain. A partial view of the SADP domain specified in TracED is visualized in Fig. 6. Modelling concepts are organized hierarchically in a tree structure (upper-left corner of Fig. 6). Each concept is able to have zero or more descendents and a unique parent. This structure is obtained by specializing *modelling concept* (Fig. 1) in *abstract* and *concrete modelling concepts*. Abstract concepts generalize common properties and relationships used by a set of design objects. For example, *Requirement* generalizes *Quality* and *Functional Requirement* modelling concepts (Fig. 6).

Fig. 6 shows the properties tab of the *component* specification window, where a concept description can be assigned, and properties can be created or modified. In the example, there are two properties, *name* and *type*, defined as *String*. Additionally, Domain Editor enables to set binary relationships between modelling concepts, which are instances of *DomainRelationship* (Fig. 1).

For each modelling concept, a set of applicable operations must be specified. As can be observed in Fig. 6, three operations have been defined for *Component*: *addComponent*, *deleteComponent*, and *delegateResponsibility*. Domain Editor allows the specification of these *macrocommands* and their input *arguments*, by implementing the *Operations Model* (Fig. 4).

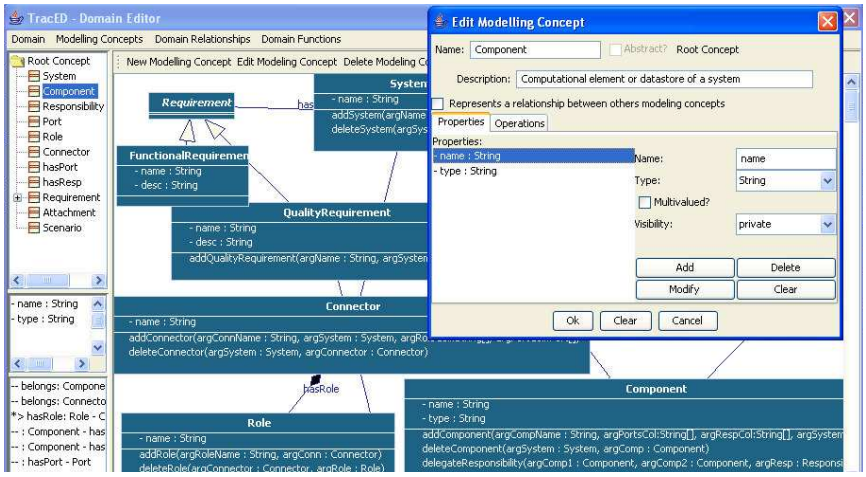

Fig. 6. Partial view of SADP Domain Model specified in TracED.

The window in Fig. 7 shows the definition of *addComponent* that is similar to the functional specification that was presented in Fig. 5. For defining it, the actor can select some existent operations. Additionally, Fig. 7 explains how the *VariableAssignment* is carried out using TracED, 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 *macrocommand*. Also, *addComponent* specification employs a *loop* command, which is defined over a responsibilities collection. It allows assigning one by one the responsibilities to the new component by using the *addResponsibility* operation.

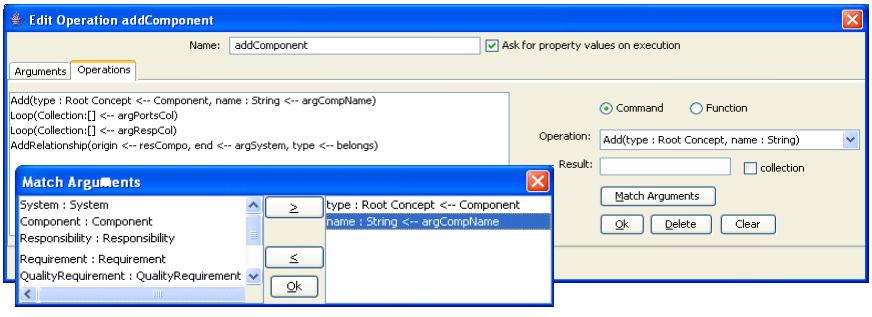

Fig. 7. Specification of the addComponent MacroCommand.

On the other hand, *Versions Manager* enables the execution of a design project. When a new design project is created, an existent 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. *Version Manager* is introduced by means of a conducting case study that resembles the SADP of Apache Struts, a free open-source framework for creating Java web applications [7], which utilizes the MVC architecture style. Struts will be designed by following the ADD method [3]; thus, will be presented as a recursive decomposition process, guided by architectural drivers.

Therefore, we start a new project called *StrutsProject*. An instance of *DesignProject* is created and associated with *SoftwareArchitecturesDomain* (which was partially defined in sections 2.1 and 2.2, and exemplified in Fig. 6 and 7). By doing that, the initial model version (*Root Model Version*) is created, forming the root of the tree structure of the version management scheme. It has no object versions and cannot be edited. Fig. 8 shows the Version Manager's window, with the first model version of Struts project. On the upper-left of this window appears the "model versions tree" navigator. To create a new model version, the predecessor model version must be selected.

The next step is to create *Model Version 1* (Fig. 8). At the model version's navigator panel, the user places the focus on this new model version, and executes one or more domain operations. Design Objects menu offers the available operations of the current domain grouped in submenus that correspond to each domain's modelling concept. In Fig. 8, it is visualized the execution of *addSystem*(*Struts*) operation, which belongs to the sequence of operations ϕ_1, which adds the first object version named *Struts*. At this point, the actor defines the architectural drivers for the intended architecture, which are: *Modifiability* and *Testability*. *Modifiability* requirement aims to achieve *separation of concerns*, *loose coupling*, and *modularity*, whereas *Testability* aims to extending, enhancing, or correcting system features easier. Therefore, the architect incorporates to the architectural design the desired quality requirements by executing a new sequence of operations ϕ_2 = {*addQualityRequirement*(*Modifiability*), *addQualityRequirement*(*Testability*)} on the previous model version (*Model Version 1*). The resulting model version (*Model Version 2*) is not shown due to lack of space. In order to make concrete quality requirements, *Quality Requirements* have to be translated into *Quality Scenarios*. Some quality scenarios considered by the architect are: i) It is easy to add different view types such as HTML and XML (*ScModifiability1*); ii) It requires little effort to incorporate new designers or programmers to a project (*ScModifiability2*). Regarding this scenarios (and other ones not depicted), a new *Model Version 3* is attained by the execution of ϕ_3 compound by a sequence of *addScenario* operations. Next, the actor goes on by incorporating structural elements. Therefore, the first component and its responsibilities are added by executing an *addComponent* operation. Such resulting model version is observed at the bottom window of Fig. 8 (Model Version 4). Then, at this point of the design process, the designer considers convenient the application of Model-View-Controller (*MVC*) style, given that this architectural style achieves the intended quality requirements.

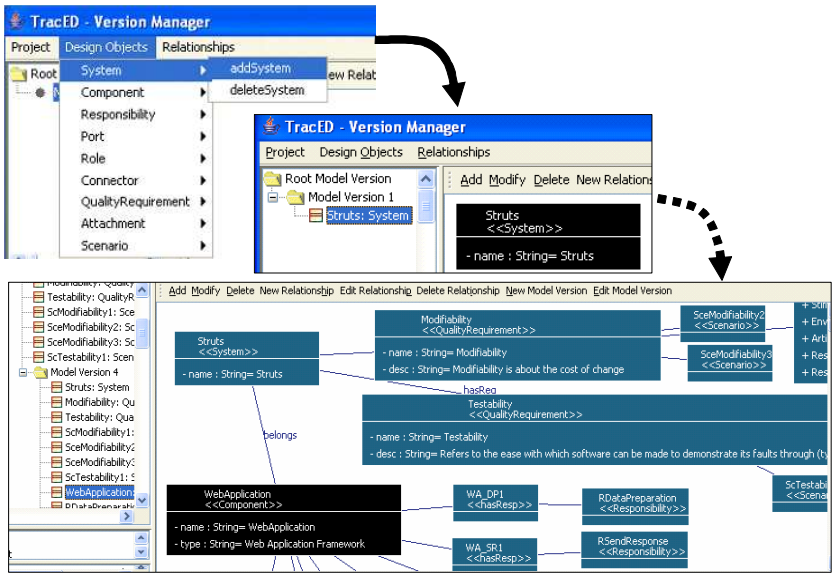

Fig. 8. Versions Manager – Carrying out the Struts project.

Thus, it is useful to apply *applyMVC* (Fig. 5) operation (which belongs to sequence of operation ϕ_5) on the current model version, by refining *WebApplication* component. As a result, a new *Model Version 5* is obtained where three new components *Model*, *View*, and *Controller* are present on *Struts* System. The new components are communicated by means of *ConnViewCtrlr*, *ConnModCtrlr* and *ConnModView connectors* that represent the interactions among them (Fig. 9).

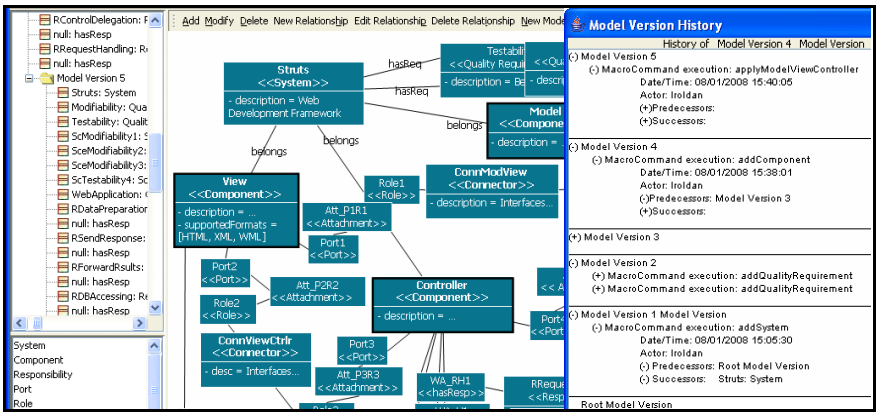

Fig. 9. Partial view of Model Version 5, after applyMVC execution, and its History window.

Struts' design process continues by relaxing MVC style through deleting the existent connector between *Model* and *View* component. Additionally, the architect provides a lower level abstraction by refining the *Controller* component.

As was pointed out previously, 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. 9 shows the History Window, which informs all applied operations from the root to *Model Version 5*.

4 Conclusions

In this paper, SADP is conceived as a composition of activities that operate on design products. TracED, a tool for capturing and tracing SADP has been presented. It allows defining a particular domain and the suitable operations for such a domain. This approach lays the foundations for future efforts related to synthesizing of activities patterns and reuse of previous designs. It must be evaluated the possibility of integrating TracED in a CASE environment as a plug-in. By working in background mode, it would assist to the designer in the capture of all operations carried out. A future extension of the present approach is to provide support for collaborative design process, by developing features for conflict detection and resolution tasks. A proposal for collaborative environments has been presented in [5].

Acknowledgments. The authors wish to acknowledge the financial support received from CONICET, Universidad Tecnológica Nacional and Agencia Nacional de Promoción Científica y Tecnológica (PICT 12628 and PICT 22118).

References

1. Jansen, A., van der Ven, J., Avgeriou, P., Hammer, D.: Tool support for architectural decisions. IEEE/IFIP WICSA 2007 (2007)
2. Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.): Foundations of Component-Based Systems. Cambridge University Press (2000) 47-68
3. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2nd Edition. Addison-Wesley (2003)
4. ISO/IEC JTC1/SC7, Recommended Systems and software engineering. Architectural description, ISO/IEC WD2 42010 (2008)
5. Gonnet, S., Leone, H., Henning, G.: A model for capturing and representing the engineering process. Expert Systems with Applications, Vol. 33. (2007) 881-902
6. Roldán, M.L., Gonnet, S., Leone, H.: A model for capturing and tracing architectural designs. IFIP IWASE 2006. Springer. Vol. 219 (2006) 16-31
7. Holmes, J.: Struts. The Complete Reference. McGraw-Hill/Osborne (2007)