A Model for Capturing and Tracing Architectural Designs

M. Luciana Roldán, Silvio Gonnet, Horacio Leone
CIDISI, Universidad Tecnológica Nacional
INGAR, Universidad Tecnológica Nacional, CONICET
Avellaneda 3657, 3000, Santa Fe, Argentina
{lroldan, sgonnet, hleone}@ceride.gov.ar

Abstract. Software architecture constitutes the primary design of a software system. Consequently, architectural design decisions involved in architecture design have a key impact on the system in such aspects as future maintenance costs, resulting quality, and timeliness. However, the applied knowledge employed and the design decisions taken by software architects are not explicitly represented in the design despite their important role; consequently, they remain in the mind of designers and are lost with time. In this work, a model for capturing and tracing the products and architectural design decisions involved in software architecture design processes is proposed. An operational perspective is considered in which design decisions can be modelled by means of design operations. The basic ontology of situation calculus is adopted to formally model the evolution of a software architecture.

1 Introduction

Software Architecture Design Process (SADP) involves several activities such as exploration, evaluation and composition of design alternatives which make it a difficult, complex process [1]. In order to address those activities, the research community has been working intensively in the achievement of modelling languages [2, 3], design methods [4] and computer environments for architect assistance [1, 5]. Those tools are basically focused on assisting designers in generating a software architecture design to satisfy a set of requirements. However, documentation of associated rationale, design decisions, and applied knowledge are often omitted. Such 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 support design processes. Thus, most

architectural design knowledge and architectural design decisions taken through SADP remain in the minds of experienced designers, and are lost with time. Consequently, capturing design decisions is of great importance to capitalize previous designs and to provide the foundations for learning and training activities. Precisely, this latter issue has been the goal of other contributions [6, 7] which recognise that the design rationale should be incorporated into the SADP.

Therefore, this work introduces a model for capturing and tracing the SADP and its products. Its goals are to make explicit the states of the SADP and the way in which they were generated. The model is based on a generic Process Version Administration Model (PVAM) [8], which provides mechanisms for capturing and managing versions generated during the course of an engineering design project.

In the next section a conceptual model is presented, introducing the extensions for making PVAM applicable to SADP. After that, the operation capturing system is described, where the products and operations of the SADP are represented. The proposed model is illustrated in Section 4 with a case study about the design of a monitoring system for an industrial process. Finally, conclusions and future research guidelines are discussed.

2 A Conceptual Model for Capturing Architectural Design Processes

The proposed scheme considers the SADP as a sequence of activities operating on the products of the design process, which are called *design objects*. Examples of *design objects* are components and connectors of the architecture being designed, or functional and quality requirements and scenarios to be met. Naturally, these objects evolve as the SADP takes place, giving rise to several versions. In order to maintain these versions, the previously proposed PVAM [8] is considered. The general scheme employed 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* represents the artifact 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*.

At the v*ersions 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* which represent the versions of the objects that compose a given model at a time point. The relationships between a *versionable object* and one of its *object versions* is represented by the *version*(v, o) predicate. Therefore, a given *versionable object* keeps a unique instance in the *repository* and the *versions* it assumes in different *model versions* belong to the *versions level*.

Based on that scheme, the model evolution is posed as a history made up of discrete situations. The situation calculus [9] is adopted for modelling such version generation process. A new model version m_n is generated when an activity a is

executed. An activity *a* is materialised by a sequence of operations ϕ and the new model version m_n is the result of applying such sequence ϕ to the components of a previous model version m_p. 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 precedent model version. Therefore, the new model version m_n is achieved by performing the following evaluation: $apply(\phi, m_p) = m_n$.

The primitive operations that were proposed to represent the transformation of *model versions* are *add*, *delete*, and *modify*. By using the *add(v)* operation, an *object version* that did not exist in a previous *model version* can be incorporated into a successor *model version*. Conversely, the *delete(v)* operation eliminates an *object version* that existed in the previous *model version*. Also, if a *design object* has a version v_p, the *modify*(v_p, v_s) operation creates a new *version* v_s of the existing *design object*, where v_s is a successor *version* of v_p. Thus, an *object version v* 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, the 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.

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

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

From these definitions, and using the format of successor state axioms proposed by [9], 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 ($added(v, apply(\phi, m))$); or (ii) *v* already belonged to the previous *model version m* ($belong(v, m)$) and it is not deleted when ϕ is applied to it ($\neg deleted(v, apply(\phi, m))$).

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

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

Once the versions belonging to a *model version* are defined, the relationships existing among *object versions* have to be specified. First, it should be noted that in this proposal, *object versions* belonging to a *model version* are not explicitly associated to other versions belonging to the same model version. 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 by them. This fact is represented in

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)).

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

The primitive operations *add*, *delete*, and *modify* introduced are not enough to capture and trace a SADP execution. Then, PVAM must be extended in terms of the suitable operations for this design domain, like the ones listed in Table 1. This operations range from the most basic to the most complex ones:

- *Basic*: operations that allow creating and deleting basic design objects (like *components* and *connectors*);
- *Special*: more complex operations that involve object refinement or delegation;
- *Styles/Mechanisms application*: these operations 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 certain design objects properties.

Table 1. Possible Operations for the Software Architecture Design Domain

Basic Operations		
addComponent	addScenario	deleteQualityRequirement
addConnector	addTypeComponent	deleteResponsibility
addFunctionalRequirement	addTypeConnector	deleteRole
addPort	deleteComponent	deleteScenario
addProperty	deleteConnector	deleteTypeComponent
addQualityRequirement	deleteFunctionalRequirement	deleteTypeConnector
addResponsibility	deletePort	
addRole	deleteProperty	
Special Operations		
refineComponent	delegateResponsibility	verifyScenario
refineResponsibility	delegateScenario	
Styles/Mechanisms application		
applyIntermediaryBlackboard	applyRuleEngine	applyPoolOfConnections
applyControlLoop	applyClientServer	

These operations are defined in terms of primitive operations as *add*(*c*), and non-primitive ones (see Table 1), as *addPort*(*c*, *p*). The execution of one of these operations implies that a sequence of primitive operations *add*, *delete,* and/or *modify* are applied to a previous 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 the *addComponent*(*s*, *c*, l_{Resps}, l_{Ports}) operation. It adds a component *c* to a system *s*. Therefore, if it is applied to a model version *m*, then a version of a

component *c* having a set of *responsibilities r* and *ports p*, will belong to the successor model version (*apply(ϕ, m)*), as it is defined in Expression 5.

$$\begin{gathered}(\forall\ \phi, s, c, l_{Resps}, l_{Ports}, m)\ addComponent(s, c, l_{Resps}, l_{Ports}) \in \phi \Rightarrow \\ added(c, apply(\phi, m)) \wedge added(rel(s,c), apply(\phi, m)) \wedge \\ ((\forall\ r \in l_{Resps})\ added(r, apply(\phi, m)) \wedge added(rel(c,r), apply(\phi, m))) \wedge \\ ((\forall\ p \in l_{Ports})\ added(p, apply(\phi, m)) \wedge added(rel(c,p), apply(\phi, m)))\end{gathered} \tag{5}$$

Similarly to Expression 5, the definition of new operations allows enlarging the set of operations. This can be done without modifying the successor state axiom (Expression 3).

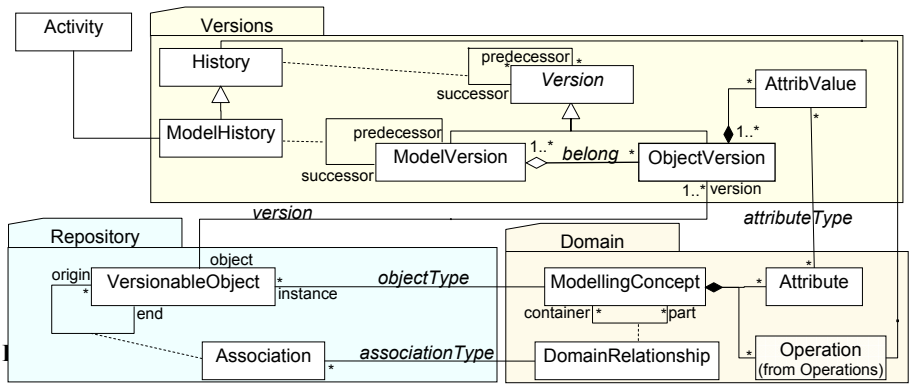
The precondition for applying the *addComponent* operation is specified in Expression 6, where the *poss*(*op*, *m*) predicate expresses that an operation *op* is applicable to a given model version *m*.

$$\begin{gathered}(\forall\ s, c, l_{Resps}, l_{Ports}, m)\ poss(addComponent(s, c, l_{Resp}, l_{Ports}), m) \Leftrightarrow \\ belong(s, m) \wedge \neg\ belong(c, m) \wedge \\ (\forall\ r \in l_{Resps}) \neg\ belong(r, m) \wedge (\forall\ p \in l_{Ports}) \neg\ belong(p, m)\end{gathered} \tag{6}$$

3 The Version Support System for Capturing Architectural Design Processes

3.1 Defining the Operations Model

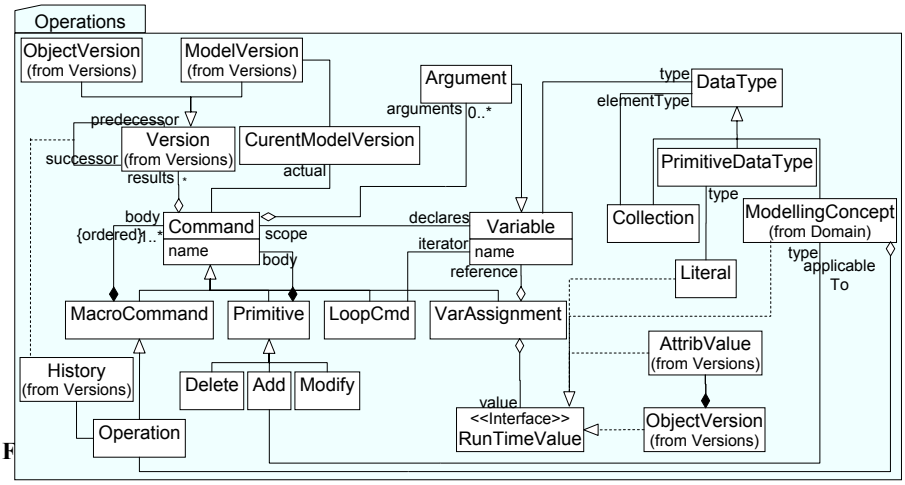
The class diagram illustrated in Fig. 1 shows the main concepts of PVAM introduced in the previous section. The relationship between a *versionable object* and one of its *object versions* is represented by the *version* relationship. Furthermore, it is assumed that design objects are identified and classified according to the different types (see Section 3.2). The design object type is represented by *ModellingConcept* class (Fig. 1).



As outlined before, each transformation operation applied to a *model version* incorporates the necessary information to trace a model evolution. This information is represented by *history* relationships between the *object versions* to which the *operation* is applied and the ones arising as the result of its execution (Fig. 1). In order to represent architecture evolution, a model version has zero or more successor model versions (noted by * cardinality at *successor* role of *History* association shown in Fig. 1).

PVAM must be capable of extending in terms of the suitable operations for SADP domain. Subsequently, in this section the operation model is presented, which allows specifying and instantiating specific domain operations.

Operations are associated with a *modelling concept* and are defined as ordered sets of commands (Fig. 2). Those commands can be *primitives* or *operations* that can be used to define other operations. *Primitives* encapsulate the semantics defined by Expressions 1, 2 and 3. The execution of an *operation* generates one or more *results*, which can be a set of *versions*. Furthermore, *history* class is instantiated, linking the *predecessor* with the *successor versions*.



F

The operation definition is represented using the basic structure of the Abstract Syntax Kernel Metamodel for Expressions defined by the UML 2.0 OCL Specification [10]. To implement operations, the well-known Command design pattern was used [11]. Therefore, a *command* abstract class is introduced into the *Operations* package illustrated in Fig. 2. 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 commands that constitute its body are some other already defined commands, which are available for use in the specification (primitives, loop, variable assignment, or other operations). Note that the *modelling concept* over which an operation is applied must be explicitly indicated. Furthermore, there are other concrete classes that specialise the *command* class, and that can be part of a *macro command*. One of them is the *LoopCmd*, which represents a loop construct over a collection variable and has a body that is executed for each element in the collection. Another valid command is *VariableAssignment* that represents the assignment of a value to a variable of a given type.

As shown in Fig. 2, 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 types described by the model are grouped by the abstract class *DataType*. *DataType* subclasses are *PrimitiveDataType*, *CollectionType*, and *ModellingConcept*. *PrimiteDataType* includes *Integer*, *Float*, *String* and *Boolean* types. *Collection* describes a list of elements of a particular given type that are ordered, have no duplicates and are parameterized with an element type. *ModellingConcept* is imported from *Domain Package* and enables specifying arguments that explicit the type of an expected object version to be added during the execution of an *add* primitive.

As regards *VariableAssignment*, it denotes the mapping between a *Variable* and a *RunTimeValue*. This interface is not defined to specify operations. It is included to

represent the run time values during the execution of an operation. *RunTimeValue* can be realized by different values like *literal*, *object version*, *modelling concept*, or *AttribValue* (value of an attribute of an object version, Fig. 2), depending on the variable type.

3.2 Products of SADP

In order to capture the versions generated during a SADP, the PVAM must be extended according to the particular design objects produced by that process. To this purpose, the *Domain Package* shown in Fig. 1 must be extended with concepts of the SADP domain. The products that constitute the design object types are taken from the Attribute-Driven Design Method (ADD) proposed in [4], and the architectural description language ACME [2]. The class diagram shown in Fig. 3 introduces these concepts and their relationships. This model is implemented by the instantiation of the classes of *Domain package* (Fig. 1). The classes presented in Fig. 3 are going to be instances of *ModellingConcept* and their properties are going to be instances of *Attribute*. Finally, the relationships of Fig. 3 will be instantiated from *DomainRelationship* in *Domain package*.

The ADD method is based on a recursive 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*). The *quality requirements* are expressed as a set of system specific *quality scenarios*, and the *functional requirements* are translated into a set of *responsibilities* [4]. Quality scenarios and responsibilities can be delegated to other components when the original component is refined. When the method iteration is finished, the designer verifies scenarios and sets an *assessment*.

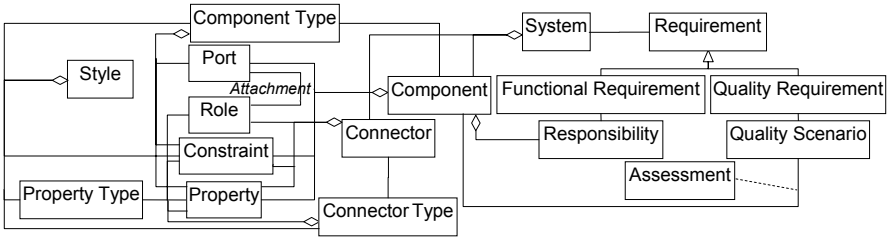


Fig. 3. Domain model for architecture based design

In ADD, the different model versions are represented using various types of views. Only the component view is considered within the scope of this work in order to describe the architecture. Accordingly, ACME [2] 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, each of which is termed *port*. The *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. 3, the *attachment* concept is not considered as a *modelling concept* but as a relationship. Moreover, ACME proposes elements to document extra-structural properties of a system's architecture, as *Properties*. Furthermore, it is possible to attach *constraints* to design elements. With the aim of providing a more powerful language, ACME defines *component*, *connector*, and *property type* building blocks. On the basis of these modelling concepts, it is possible to define *Families* or *Styles*. They are defined by a set of *property*, *component,* and *connector types* and a set of *constraints*.

3.3 Architectural Operations Specification

As it was outlined in Section 3.1, PVAM must be extended in terms of the suitable operations for the SADP domain, like the ones listed in Table 1.

Fig. 4 presents functional specifications for some of the basic operations defined in Table 1. The other operations are defined in a similar way, but they are not shown due to lack of space. As seen in Fig. 4, the operation *addComponent*(*s*, *c*, l_{Resps}, l_{Ports}) 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 (detailed by list l_{Ports}) are inserted. These operations are carried out by the *addResponsibility*(*c, r*) and *addPort*(*c, p*) operations. Finally, a relationship between the new component *c* and an existing system *s* is included. This last operation is performed by the *add* primitive operation (*add*(*rel*(*s*, *c*))). These operation specifications are implemented as instances of the *Operation* model introduced in Fig. 2.

In the same way as for basic operation, it is possible to define the special operations. Fig. 5 presents some examples. A function with a '?' symbol at the end indicates that it is interactive; thus, the user is asked about how to proceed. The interactive commands can be implemented as a special case of *VarAssignment* command (Fig. 2).

```
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
  add(rel(s, c))

addPort(c, p)
  add(p)
  add(rel(c, p))

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

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

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

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

Fig. 4. Specifications of basic operations

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 *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.

```
delegateResponsibility(c1, c2)
  lResps = getResponsibility(c1)
  for each r in lResps
     if (delegate?(c2, r))
        delete(rel(c1, r))
        add(rel(c2, r))
     end if
   end for

delegateScenario(c1, c2)
  lScens = getScenario(c1)
  for each s in lScens
     if (delegate?(c2, s))
        delete(rel(c1, s))
        add(rel(c2, s))
     end if
  end for

refineComponent(c, lComps, llPorts, llResps , lConns, llRoles, llAtts)
  i = 0
  for each cc in lComps
      lr = llResps(i) // cc responsibilities listi
      lp = llPorts(i) // cc ports listi
      addComponent(getSystem(c),cc,lr,lp)
      i++
  end for
  i = 0
  for each cn in lConns
      lr = llRoles(i) // cn roles listi
      la = llAtts(i) //  port list which should attach cn roles
      addConnector(getSystem(c), cn, lr, la)
      i++
  end for
  // delegate scenarios and responsibilities to new components
  // (interactive)
  for each cc in lComps
      delegateScenario(c, cc)
      delegateResponsibility(c, cc)
  end for
  // create new connections between internals and external components
  // (interactive)
  lp = getPorts(c)
  for each p in lp
      np = PortMap?()
      r = getRol(p)
      delete(rel(p, r))
      add(rel(np, r))
  end for
  deleteComponent(getSystem(c),c)
```

Fig. 5. Specifications of special operations

The *refineComponent*(*c*, l_{Comps}, l_{lPorts}, l_{lResps}, l_{Conns}, l_{lRoles}, l_{lAtts}) operation, another example of special operation (Fig. 5), decomposes a component *c* into one or more components given by the list l_{Comps}. The ports and responsibilities of the new components are given by the lists l_{lPorts} and l_{lResps}, respectively. Furthermore, a set of connectors among the new components is added. These connectors are specified by l_{Conns} whose roles are given by the list l_{lRoles} and the attachments by the list l_{lAtts}.

The operations that apply an architecture style [12], or an architectural pattern [13], refine a preexistent component with a new set of components and connectors that are instantiated from an architectural style/pattern. They interact with the designer asking for the responsibilities and scenarios delegation, as well as connectors mapping between external components and refined components. An example of *applyStyle* operation is defined in Fig. 6. In this case, the *applyControlLoop* operation is specified. This style proceeds from the process control paradigm and defines the architecture to activate various monitoring policies when different events coming from sensors are produced [14]. The monitoring policies may in turn produce other events or actions in response to predefined

situations. Note that this operation can be considered as a specialization of *refineComponent* operation. The knowledge on how to proceed in the refinement of component *c* is given by the control loop style. Therefore, a series of *addComponent* operations is performed. The *addComponent*(*s*, {*Diagnosis*, *TDiagnosis*}, [P_1,P_6]) operation indicates that a component and two ports must be created. The component is called *Diagnosis*, whose modelling concept is *TDiagnosis*, an instance of *ComponentType* (see Fig. 3), and the ports are denominated P_1 and P_6.

```
applyControlLoop(c)
  s = getSystem(c)
  addComponent(s,{Diagnosis, TDiagnosis}, [P1,P6])
  addComponent(s,{PolicyManager, TPolicyManager}, [P2,P3])
  addComponent(s,{Reactor, TReactor}, [P4, P5])
  addConnector(s,{CDgnPMgr,TCDgnPMgr},[R1,R2],[P1,P2])
  addConnector(s,{CPMgrRct,TCPMgrRct},[R3,R4],[P3,P4])
  delegateScenario(c,Diagnosis)
  delegateScenario(c,PolicyManager)
  delegateScenario(c,Reactor)
  delegateResponsibility(c,Diagnosis)
  delegateResponsibility(c,PolicyManager)
  delegateResponsibility(c,Reactor)
  // Set mappings between previous connector and new components
  lp = getPorts(c)
  for each p in lp
    np = PortMap?(p) // Ask the user the port to map
    r = getRol(p)
    delete(rel(p, r))
    add(rel(np, r))
  end for
  deleteComponent(s, c)
```

Fig. 6. Specification of applyControlLoop operation

4 Case Study: Monitoring System for an Industrial Process

The following case study describes the design of a monitoring system for an industrial process (see Fig. 7). It is based on classical case studies presented in other contributions [1, 4]. Monitoring activities are focused on the two core distillation columns: an extractive distillation column and a solvent stripping one, working together in a highly integrated manner. The system should monitor control loops and temperature sensors, by continued acquisition of real-time process data, tracking set-point values, alarm conditions and outputs of valves, and comparing them with normal pattern behaviour. The system should also monitor process state, using real-time process data previously processed in combination with expert knowledge in order to maintain process stability and performance. Further functionalities are control flowrate sensors and validate material balances. In order to meet all these functional requirements, the system should be connected to input and output devices. Input devices allow the system to get the real time data from the process equipment and output devices are used by the system to inform the plant operator about process anomalies, like: solvent inventory buildup, sensor fault, abnormal process pattern, etc. The main functions considered in designing the monitoring system include: administration of users (process operator, plant supervisor, etc.) and permissions,

configuration of input/output devices, priority-based event management, process diagnosis, specification of warning and process protective actions.

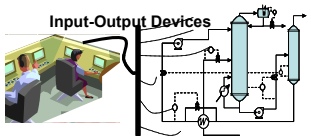


Fig. 7. Monitoring system for an industrial process

For reasons of space, only a sequence of operations of the model evolution is analyzed. Let us consider an intermediate *model version i* (see Fig. 8) where the main components are: *Control&Diagnosis* (with responsibilities in priority based event management, protective actions execution, warning launch, input/output devices configuration); *UserInterface* (with responsibilities related to user interaction issues: set parameters values, show information, rule administration); *SensorActuatorLayer* (with responsibilities like sending out commands to actuators, receiving information from sensors); and *Configuration*. From this model version, the designer chooses to refine the *Control&Diagnosis* component by applying the *applyControlLoop* operation. This operation creates three new *components*: *Diagnosis*, *PolicyManager*, and *Reactor*. The *applyControlLoop* operation (see Fig. 6) asks the necessary information for delegating responsibilities, and for reconnecting previous connections to the new configuration.

Fig. 8 shows a partial view of the *Version* and *Repository* levels from which model version views can be inferred. This figure is focused on the version of *Control&Diagnosis* evolution to a set of versions of components and connectors due to *applyControlLoop* operation. A *view* of a *model version* is obtained from the knowledge in the *Version* and *Repository* levels. The object versions belonging to a *model version* are inferred by the *belong*(*v*, *m*) predicate (Expression 3). Fig. 8 shows some object versions that belong to *model version i* (*Control&Diagnosis*$_{v,1}$, *P1C&D*$_{v,1}$, *P2C&D*$_{v,1}$, *P3C&D*$_{v,1}$). Given an object version (*Control&Diagnosis*$_{v,1}$), it is possible to know its versionable object (*Control&Diagnosis*$_o$), which is linked with its design object type (modelling concept *component*, defined in *Domain*). All this information makes possible to reconstruct the elements of a model version view, as it is the *Control&Diagnosis* component which is obtained from object version *Control&Diagnosis*$_{v,1}$ and versionable object *Control&Diagnosis*$_o$. On the other hand, the expression 5 enables to retrieve the relationships among the object versions that belong to a given model version. *Control&Diagnosis*$_o$ has three ports named *P1C&D*$_o$, *P2C&D*$_o$, and *P3C&D*$_o$ which have their respective object versions *P1C&D*$_{v,1}$, *P2C&D*$_{v,1}$, and *P3C&D*$_{v,1}$. Therefore, component *Control&Diagnosis*$_{v,1}$ has ports *P1C&D*$_{v,1}$, *P2C&D*$_{v,1}$, and *P3C&D*$_{v,1}$.

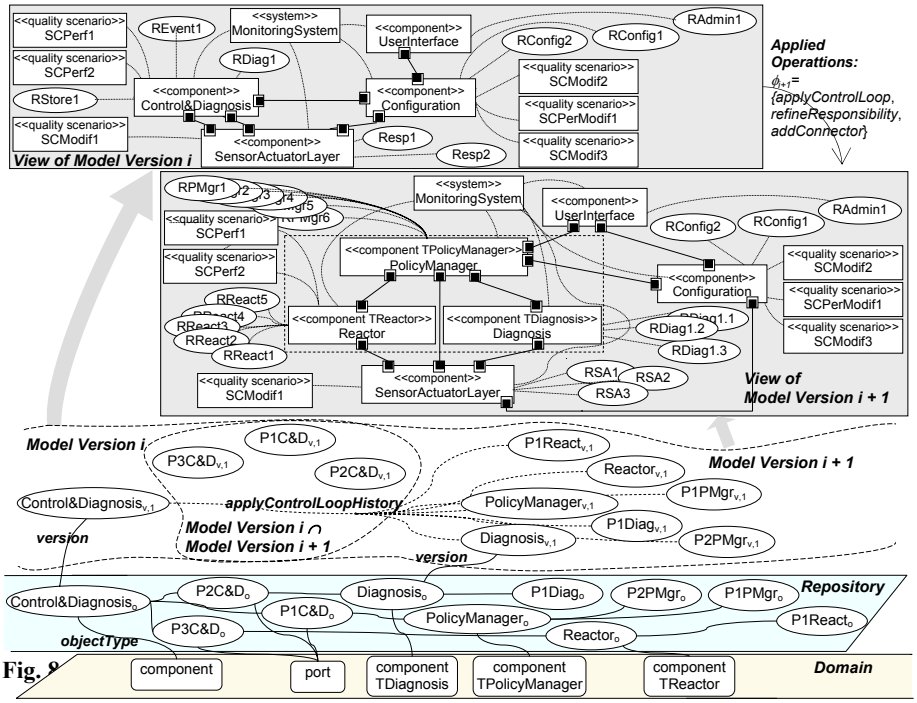


Fig. 8

The *applyControlLoop* operation is applied on *Control&Diagnosis* component (*Control&Diagnosis*$_{v,1}$ object version). This operation is traced by an instance of the *history* link (Fig. 1) which associates the previous object version (*Control&Diagnosis*$_{v,1}$) with the successor object versions (*Reactor*$_{v,1}$, *Diagnosis*$_{v,1}$, *PolicyManager*$_{v,1}$, *P1React*$_{v,1}$, *P1Diag*$_{v,1}$, *P1PMgr*$_{v,1}$, *P2PMgr*$_{v,1}$ in Fig. 8). *P1C&D*$_{v,1}$, *P2C&D*$_{v,1}$, and *P3C&D*$_{v,1}$ object versions belong to both *model version i* and *model version i+1* because they were delegated from the original component (*Control&Diagnosis*) to the newer ones by *applyControlLoop* operation.

Additionally, other operations were applied on model version *i* to obtain model version *i+1* that are not illustrated in Fig. 8 at version and repository levels. One of them arises due to the need of associating *PolicyManager* and *Configuration* components, so a new *connection* and their *roles* objects are added, applying *addConnector* operation. Using again operation *addConnector*, a new connection between *PolicyManager* and *SensorActuatorLayer* is added. It enables *PolicyManager* to receive information from, and send information to, *SensorActuator* (see Fig. 8, *View of model version i+1*).

It is important to note that the proposed extension of PVAM enables applied operations on SADP's products (Fig. 3) to be captured. For example, responsibilities are refined using *refineResponsibility* operation. The *RDiag1* responsibility (Fig. 8, *view of model version i*) was refined on the following responsibilities: i) listening notifications of situations coming from *SensorActuator* (*RDiag1.1*); ii) getting

devices information (*RDiag1.2*); iii) probing device (*RDiag1.3*) (Fig. 8, *view of model version i + 1*).

4.1 Retrieving the History of Architectural Design Processes

The model introduced allows tracing and recovering the history of the architectural design activities carried out by the designer during SADP. It is possible to ask about the history of model versions in terms of operation sequences that have generated a given model version, and also consult on the history of a particular object version, which allows to know how the evolution took place through the different versions. Fig. 9 shows an example of a history query to perform on the hypothetical monitoring system designed in current section. An actor would whish to know the sequence of operations that originated model version *i+1* from the precedent model version *i*. The applied operations were *applyControlLoop*, *refineResponsibility* and *addConnector*, which can be seen in Fig. 8 of the case study. The resulting information allows knowing who carried out the operations, at what time and date, their arguments, the new elements incorporated to the design, the set of elements eliminated and what kind of modelling concepts they were. As shown in Fig. 9, additional information can be obtained, like the suboperations implied at the execution of the current one. Knowing which were the operations that gave rise to model version *i+1* is useful for understanding the rationale associated with such a step because the architect knows the semantic of the operation and the intent.

5 Conclusions

The model proposed in this paper, an extension of PVAM, captures the operations that generate each design product during the SADP. Furthermore, it also offers an explicit mechanism to manage the different model versions generated during the SADP. Thus, it allows the tracing of the SADP and its resulting products, setting the grounds for learning and future reuse of the design process. This is a fundamental step towards the development of computational tools to support the SADP and to guide designers in the different activities of a design project. A related work [6] proposes a set of requirements which such tools should satisfy in order to adequately support the evolution of software architectures. The approach presented in this work meets a wide spectrum of those requirements: (i) *First class architectural concepts*, represented by the extensible domain model proposed; (ii) *First class architectural design decisions*, enabling specification of adequate operations for software architecture design representing design decisions made by the architect; (iii) *Under-specification and incompleteness*, allowed by the model evolution through discrete situations (model versions) increasing the level of abstraction; (iv) *Explicit architectural changes*, allowing capturing, managing and tracing of products of SADP, using explicit history links between different versions, which means that the operations applied through the design process are saved and, therefore, it is possible to reconstruct the history from an initial model version; (v) *Support for modification, subtraction, and addition type changes*, implemented by

the primitive operations add, delete and modify. Those operations are also used in the definition of higher level operations representing more complex design operations like refining or styles application.

Model Version: Model Version i+1
Precedent Model Version: Model Version i
Applied Operations:

Operation: applyControlLoop
Model Version: Model Version i **Actor:** Architect1
Arguments:

Argument Name	Value	Data Type
Source Version	Control&Diagnosis	Component

Results:

Object Version	Modelling Concept	Date Time
PolicyManager$_{v.1}$	Component TPolicyManager	01-06-2006 10:56
Reactor $_{v.1}$	Component TPolicyManager	01-06-2006 10:56
.....	.....	.....
RPMgr1 $_{v.1}$	Responsibility	01-06-2006 10:56

Deleted versions:

Object Version	Modelling Concept	Date Time
Control&Diagnosis $_{v.1}$	Component	01-06-2006 10:56
....	.....	.....
Rel_C&D_Diag $_{v.1}$	Relation	01-06-2006 10:56

SubOperations:

(+) delegateResponsibility
(+) delegateResponsibility
...

Operation: refineResponsibility
Model Version: Model Version i **Actor:** Architect1
(+)

Operation: addConnector
Model Version: Model Version i **Actor:** Architect1
(+)

Fig. 9. Partial view of the sequence of operations applied to model version i

Situation calculus, the formal background of the framework, allows us to represent the activities carried out during a SADP, and therefore, it enables the designer to get a better understanding of the information on how the various design objects (systems, components, connectors, functional requirements, quality requirements, quality scenarios, assessment, etc.) have been obtained. Thus, the history of operations performed on versions of design objects can be kept. Besides, this conceptual framework also provides the foundations for the proposal of formal means for detecting potential conflicts.

The framework could incorporate extensions to the Domain package, integrated to the version administration model, defining other characteristics not included by ADD or ACME. Furthermore, it uses an operational perspective where design decisions can be modelled by means of design operations. This approach is employed in other contributions [1, 4]. The structure of the conceptual framework allows the easy definition of specific design operations, like *applyControlLoop*, by instantiating the *Operation* model (Fig. 2). This extension is possible without modifying the successor state axiom (Expression 3).

References

1. A. Díaz Pace, *A Planning-Based approach for the exploration of Quality-Driven design alternatives in Software Architectures*, Tesis Doctoral (UNICEN, 2004).
2. D. Garlan, R. T. Monroe, D. Wile, Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, edited by G.T. Leavens and M. Sitaraman (Cambridge University Press, 2000), pp. 47-68.
3. N. Medvidovic, D. Rosenblum, D. Redmiles, J. Robbins, Modeling Software Architectures in the Unified Modeling Language, *ACM Transaction on Software Engineering and Methodology*, **11**(1), 2-57 (2002).
4. L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice: Second Edition* (Addison-Wesley, 2003).
5. F. Bachmann, L. Bass, M. Klein, Preliminary Design of ArchE: A Software Architecture Design Assistant, Carnegie Mellon University, Technical Report CMU/SEI-2003-TR-021, 2003.
6. A. Jansen, J. Bosch, Evaluation of Tool Support for Architectural Evolution, in: Proceedings of the 19th IEEE International Conference on Automated Software Engineering (2004), pp. 375-378.
7. A. Tang, J. Han, Architecture Rationalization: A Methodology for Architecture Verifiability, Traceability and Completeness, in: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (2005), pp. 135-144.
8. S. Gonnet, *Un modelo integrado para la captura y administración del proceso de diseño*, Tesis Doctoral (UNL, 2003).
9. R. Reiter, *Knowledge in Action: Logical Foundation for Describing and Implementing Dynamical Systems* (The MIT Press, 2001).
10. Object Management Group, OCL 2.0 Specification (2005), 2005-06-06.
11. E. Gamma, R. Helm, R. Johnson, K. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).
12. M. Shaw, D. Garlan, *Software Architecture, Perspectives on an Emerging Discipline* (Prentice-Hall, 1996).
13. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture. A System of Patterns* (John Wiley & Sons, 1996).
14. M. Shaw, Beyond Objects: A Software Design Paradigm Based on Process Control, Carnegie Mellon University, Technical Report CMU-CS-94-154, 1994.

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).