SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 9, no. 1, pp. 67-97 (2010)

A model for capturing the software architecture design process of mobile systems

M. Luciana Roldán[1,2] M. Celeste Carignano[1,2] Silvio Gonnet[1,2] Horacio Leone[1,2]

[1] Instituto de Desarrollo y Diseño (INGAR)
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)
Universidad Tecnológica Nacional (UTN)
Avellaneda 3657, Santa Fe, 3000, Argentina

[2] Centro de Investigación y Desarrollo de Ingeniería en Sistemas de Información (CIDISI)
Universidad Tecnológica Nacional – Facultad Regional Santa Fe (UTN - FRSF)
Lavaise 610, Santa Fe, 3000, Argentina

e-mail: {lroldan, celestec, sgonnet, hleone}@santafe-conicet.gov.ar

Abstract

Numerous efforts have addressed the problem of defining the fundamental architectural building blocks and methods for modelling software architectures in dynamic mobile environments. However, there is a lack of tools for documenting the evolution of the products generated during the design of software architectures for mobile systems. Based on a generic versioning administration scheme, a model to capture and manage the products of a software architecture design process is proposed placing the focus on mobility concerns. This model follows an operational approach, where design decisions are represented as architectural operations that are captured when they are applied during a design project. The capture of this information enables the tracing of such a design process and its resulting products.

Keywords: Design Process Support, Mobility, Software Architecture

1 Introduction

During the last few years, we have witnessed an exceptional technological revolution. This technological explosion has made it possible the development of complex mobile applications, which have evolved faster and faster (Roman et al., 2000). These new applications must be adaptable to technological changes, so they need to be flexible and extensible enough to support new features or to change the existing ones. As a means of controlling such a complexity in systems construction and evolution, software architecture has emerged (Medvidovic et al., 2003; Mikic-Rakic et al., 2008). Mainly, mobility constitutes an additional factor of complexity because in a mobile computing system, components may move across a network of locations, changing the environment in which computations need to be performed. Numerous efforts have addressed the problem of defining the fundamental architectural building blocks and methods for modelling software architectures in dynamic mobile environments (Ali et al., 2008; Lopes et al., 2002; Medvidovic and Mikic-Rakic, 2001; Schäfer, 2006). However, there is a lack of systematic methods and techniques to assist the designer in developing, documenting and evolving mobile software architectures. Important contributions recognize that software architecture is the result of architectural design decisions made over time (Tyree and Akerman, 2005; Kruchten et al., 2006; Jansen et al., 2007) and its documentation should not only describe the architecture of a system, but also "why" that architecture looks the way it does (Jansen et al., 2008). Therefore, the software architecture design decisions underlying the architecture provide that "why" (Jansen et al., 2008). If the knowledge concerning the domain analysis, the patterns used, the design options evaluated, and the decisions made, is not captured, it is lost and thus unavailable to support subsequent decisions (Ali Babar and Gorton, 2007). The design decisions captured and traced can be used as a memory aid for those who participate in making decisions and as a source of information for stakeholders when they need it (Burge et al., 2008). All these concepts about software architecture design rationale are strictly applicable and necessary when the architectural design is focused on mobile systems.

Regarding the administration of the products of a design process and their evolution, software configuration management systems (SCM) may provide such assistance (Westfechtel and Conradi, 2003). As Westfechtel (1999) has pointed out, SCM systems have proved to be an indispensable aid in organizing the products generated along big development efforts. However, their underlying data models to represent versions are very simple. But, more importantly, SCM systems have been created just to focus on the products of development processes, neglecting the representation of the activities that have generated them, the decisions that have been taken, the people and computerized tools that have performed such activities and the rationale underlying the adopted decisions, etc. Thus, they do not satisfy the need to capture the design knowledge. Once a design stage is complete, what remains is mainly the mobile software architecture but there is no explicit representation of how this product was obtained. Consequently, to overcome such troubles, it is necessary to recognize the design activities that are carried out to evolve from the initial design specifications to the final architecture; at the same time, it is crucial to identify the design decisions associated with each activity, along with their corresponding assumptions, simplifications, and underlying rationale. In fact, it is this design experts' knowledge that has to become explicit. Particularly, given that software architectures are centred on non-functional requirements of a complex system, new tools are needed, which allow the designers to set the correspondences between such requirements and the proposed architecture of a mobile system. Therefore, the key issue this contribution addresses is a model to capture and trace a whole software architecture design process of a mobile application, from the requirements stage to the last version of the designed architecture. The representation of evolution in architectural configurations during system runtime, as it is considered by Georgas et al. (2005), is not within the scope of this paper.

The proposed approach follows an operational perspective, where the design decisions are represented as design operations applied to the several versions of the products generated during the design process. The proposal constitutes a means of documenting the design process, by capturing each executed operation when the design is carried out, and maintaining the design history. This feature establishes a distinction from the proposal of Jansen et al. (2008), which considers capturing the design decisions after completing (part of) the design process. Additionally, the model is flexible enough to be fitted into mobile applications domain. It can easily incorporate mobility related concepts, such as stationary or mobile, and logical or physical components.

This article is organized as follows: Section 2 outlines the generic versioning scheme on which the model is based. It is not intended for a specific domain; on the contrary, it could be applied to different domains such as software (Roldan et al., 2006) and chemical engineering (Gonnet et al., 2007). Then, that section defines suitable extensions to make it applicable to the software architecture design process of mobile systems. Therefore, the model is outlined in an object-oriented approach to provide the foundations for the development of a computational tool that enables the capture and tracing of a design process; particularly, the definition of the specific concepts and operations for mobility domain are included. The concepts and operations applicable to this kind of systems are derived from ADLs for mobility and a software architecture method. Afterwards, Section 3 presents a case study of a mobile sales system and Section 4 introduces a prototype to validate our approach. Finally, this paper ends in Section 5 emphasizing the main contributions of the proposed model.

2 A model to capture and trace design processes

The underlying versioning scheme of our proposal considers a design process as a sequence of activities that operate on the products of the design process, called *design objects*. In particular, this contribution focused on the software architecture design process (SADP). Therefore, *design objects* can be the building blocks of the artefact being designed (i.e. the physical components and connectors on which the logical components and connectors are deployed), and the specifications to be met (i.e. quality requirements such as availability or performance). Consequently, 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 the 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.

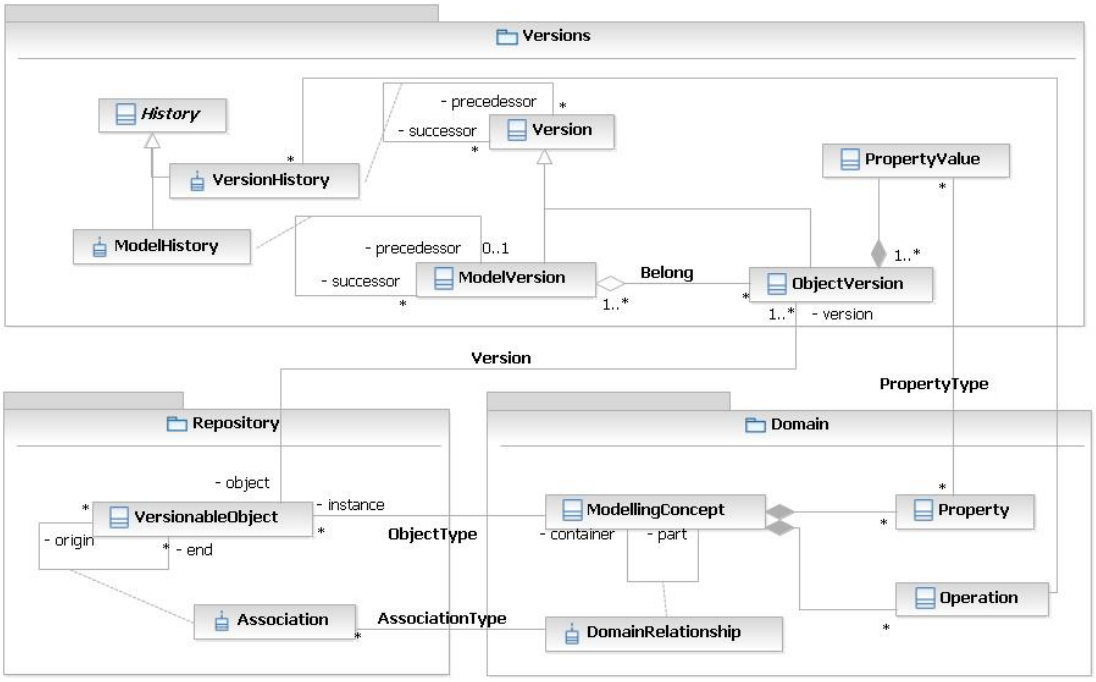

Fig. 1. Process version administration model (PVAM).

Therefore, a given *design object* keeps a unique instance in the *repository* and all the versions it assumes in the different model versions belong to the *versions level*. 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, which are denominated *model version*. According to the proposed model, a new model version is generated by applying a *sequence of operations* (ϕ) 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*. Given the fact that the model evolution is posed as a history made up of discrete situations, Gonnet et al. (2007) adopt the situation calculus for modelling such version generation process. They define a *belong*(*v*,*m*) predicate using the format of successor state axioms, which allows to know the object versions (*v*) that belong to a model version (*m*). This makes the reconstruction of a *model version* m_{i+1} possible by applying all operation sequences from the initial *model version* m_0. The whole formal representation of this scheme is not within the scope of this paper and is available in Gonnet et al. (2007). The primitive operations that were proposed in such a versioning scheme to represent the transformation of *model versions* are *add*, *delete*, and *modify*. 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 keep 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* that caused a model evolution.

Fig. 2 illustrates the described schema for representing the evolution of model versions, regarding a fragment of a software architecture design process for a mobile application. Further details of the design process of a similar mobile system will be provided in Section 3. 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* ϕ_q. In this case, *design objects* represent instances of concepts obtained from architectural description languages such as Con Moto (Schäfer, 2006); and the primitive operations have been extended by operations like *applyThreeLayers* (explained in Section 2.2). This operation applies a particular case of *Layers* pattern (Buschmann et al., 1996), which refines a logical component into three logical components arranged in layers. In this case, an *applyThreeLayers* operation belongs to the sequence of operations ϕ_q. The intention of the architect is to split the original component responsibilities into three groups, following a configuration where the lower layers offer services to the upper ones. Therefore, the *ServerSideApp* component is refined in *ServerApplicationLayer*, *ServerMiddlewareLayer*, and *ServerDataAccessLayer* components, with their ports and connections. Fig. 2 presents a partial view of *repository*, *versions*, and *inferred model* levels. The inferred models level is obtained from views produced by the versions level on the repository. A *SalesSytem* system and an *ApplicationServer* physical stationary component belong to both inferred model versions. At repository level, *ApplicationServer* is represented by $ApplicationServer_{vo}$, an instance of *versionable object* (the instances of *SalesSystem* are not shown for simplicity). $ApplicationServer_{vo}$ is linked with the versionable objects that represent *ServerSideApp*, *ServerApplicationLayer*, *ServerMiddlewareLayer*, and *ServerDataAccessLayer* components ($ServerSideApp_{vo}$, $ServerApplicationLayer_{vo}$, $ServerMiddlewareLayer_{vo}$, and $ServerDataAccessLayer_{vo}$ versionable objects, respectively). As Fig. 2 shows, $ApplicationServer_{vo}$ has an object version $ApplicationServer_{v1}$ that belongs to the model version m_k and m_q. In addition, *ServerSideApp* component only belongs to the *k* inferred model version. At versions level, this logical stationary component has an object version $ServerSideApp_{v1}$, which belongs to the model version m_k but it does not belong to m_q. As a consequence of *applyThreeLayers* execution, $ServerSideApp_{v1}$ was deleted from the successor model version (m_q) and the new object versions representing the three layers were added to m_q ($ServerApplicationLayer_{v1}$, $ServerMiddlewareLayer_{v1}$, and $ServerDataAccessLayer_{v1}$).

The versioning scheme captures the applied sequence of operations, by means of an instance of *ModelHistory*. This instance links the previous model version (m_k in Fig. 2) with the successor model version (m_q in Fig. 2). Furthermore, the *VersionHistory* instances, that represent each applied individual operation, are part of the *ModelHistory* instance. Each *VersionHistory* instance keeps not only what the executed operation was, but

also what the predecessor object versions (*ServerSideApp*$_{v1}$ in Fig. 2) and the successor object versions (results) were.

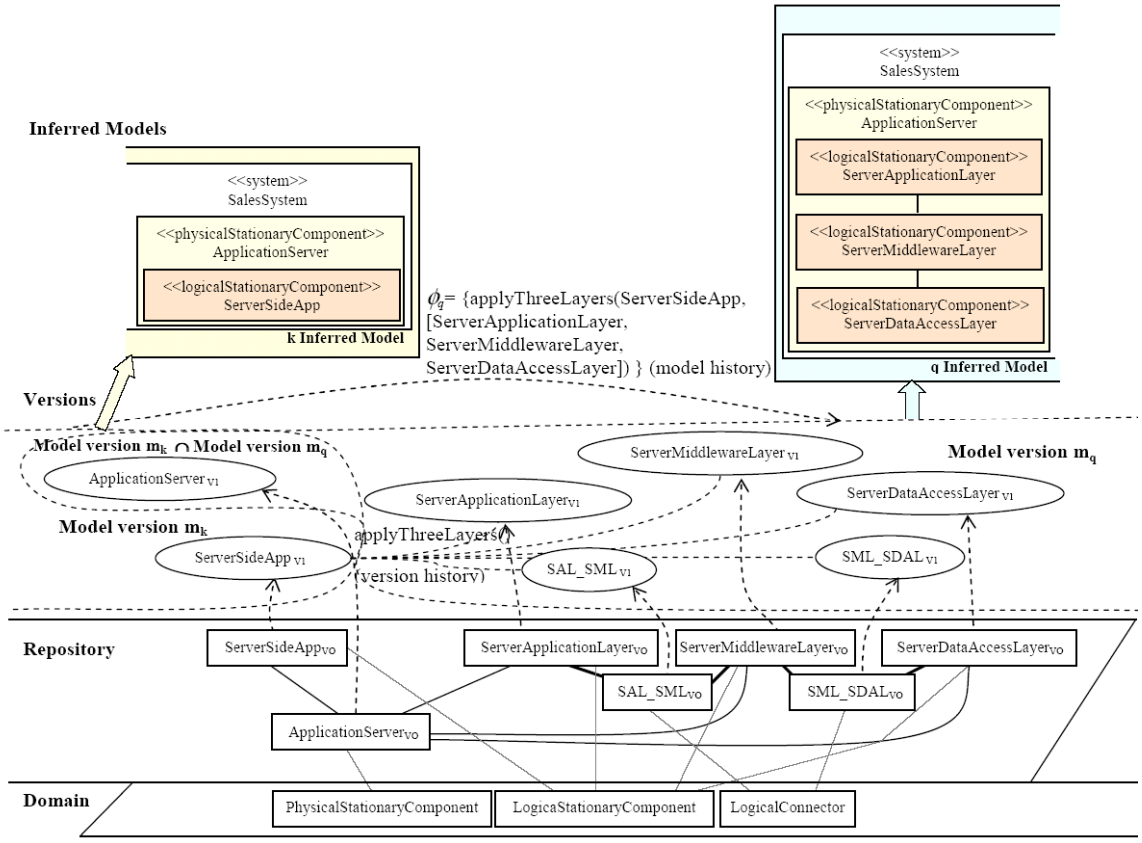

Fig. 2. A fragment of a design process captured using PVAM.

2.1 The Domain Model

The *Domain Package* (Fig. 1) enables the definition of concepts of software architectures domain of mobile systems, whose instances are going to be captured (such as logical stationary component and logical connector 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, which includes all the concepts the architect manages. A first set of modelling concepts that defines the domain model proposed in this paper arises from the software architecture design method that the architect adopts. In this case, it is assumed that the architect prefers the ADD method (Bass et al., 2003). 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 (responsibilities) is allocated to them. ADD's input is a set of requirements (functional and quality requirements). *Quality requirements* are expressed as a set of system specific *quality scenarios*, whereas *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. When a method iteration is finished, the designer verifies how well the architectural design achieves the scenarios and sets an *assessment*. Given that ADD proposes various types of views to represent the software architecture under design, the architect chooses Con Moto (Schäfer, 2006) as the architectural description language that provides the notation to represent the software architecture of an

application with mobility features using a behavioural view, together with a structural and deployment view. In this way, regarding such an ADL, a new set of concepts are identified and included in the domain model.

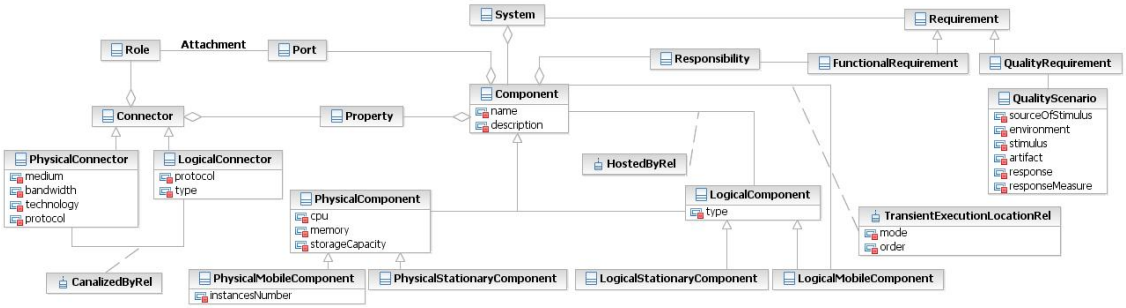

Fig. 3. A Domain Model.

Con Moto provides the building blocks to create an architectural model that reflex the physical structure of a mobile system. They are *physical components* (such as devices, servers and workstations) and *physical connectors* (meaning the communication links that form the network topology). *Physical components* act as the execution environment for logical components. Their computational resources such as *cpu* and, *memory* can be expressed as *properties*. Also, regarding *physical connectors*, properties like *bandwidth* and *protocol* can be indicated. In addition, Con Moto provides the elements to model in detail the logical structure of a system, which comprises information about software components (represented by *logical components*), and their dependencies or interactions (represented by *logical connectors*). The deployment of a *logical component* on a *physical component* is expressed in Fig. 3 by the relationship *hostedByRel*. Physical or logical components which are able to change their location need to be represented in the domain model; therefore, the domain model defines a second hierarchy level of components type. Regarding physical components, the domain includes *physicalStationaryComponent* and *physicalMobileComponent* modelling concepts. The first one comprises physical units such as servers and workstations, while the second one comprises mobile devices (such as handhelds and cellular telephones). Considering logical components, new modelling concepts are proposed: *logicalStationaryComponent* and *logicalMobileComponent*. The first one represents classical software components that run in a given physical host, and the second one comprises components like mobile agents and components replicated in more than one host; these ones are capable of passing the control of an execution to a replica situated in another physical location, and resuming the execution in the last point. To represent this potential behaviour in an architectural design, the domain model includes a *transientExecutionLocationRel* modelling concept, which sets a link between a given logical mobile component and another component. This could express two situations: i) the first component is able to migrate to the second one in order to run there, or ii) a replica of the first component might exist on the second one, which could take the control of an execution. *Mode* and *order* properties indicate which kind of mobility is represented and the order in a list of possible locations to which a component could move (Field et al., 2006; Lima et al., 2004).

In order to allow communication, physical as well as logical components have *ports*, which permit the communication of processes through connectors. Con Moto makes possible to describe an architecture where a *logical connector* is embedded in a given *physical connector*. In other words, it makes possible to represent that two logical components deployed on different physical components can communicate between themselves using a logical connector. This logical connector is canalized by a physical connector set between the two physical components. This architectural representation is achieved by *canalizedByRel* concept, which is included in the domain model (Fig. 3). In spite of not being considered by Con Moto, we have included the *role* modelling concept in order to make consistent the domain model with other ADLs that are not specific for mobility, like ACME (Garlan et al., 2000). A role is simply an interface of a connector. Thus, the way of linking components and connectors is by attaching a *port* with a *role* (*attachment* association in Fig. 3).

It should be noted that *hostedByRel*, *transientExecutionLocationRel*, and *canalizedByRel* could be modelled as domain relationships, as it was done in Fig. 2, where *hostedByRel*-type links between components were represented as associations (instances of domain relationships). However, in Fig. 3, they have been reified as modelling concepts to enable the capture of their versions.

The defined domain model provides enough concepts to model the artefacts of a software architecture design process of a mobile system, since it covers the main techniques or paradigms of code mobility (also called as code migration). It is straightforward to represent a potential case of "remote evaluation" (Schäfer, 2006; Bieszczad and White, 2007), where a program (a *logicalMobileComponent*-type object, *lmc1*) could be transferred from a node (*physicalStationaryComponent*-type object, *psc1*) to another (*psc2*) to be executed there. In this case of mobility, the first node is in control of the operation and the results are returned to it. This fact can be represented in an architectural design by means of a *hostedByRel*-type object between the component *psc1* and *lmc1*, which indicates that *lmc1* has been originally deployed to be executed in *psc1*. Depending on several factors (such as processing constraints), *lmc1* is able to move to another location. This possible situation could be represented in a model version by adding a *transientExecutionLocation*-type object, *tel1*, to link *lmc1* with a different *physicalStationaryComponent*-type object, *psc2*. In this particular case, the *mode* property value of *tel1* object must indicate that it is a "remote evaluation" case of mobility, and the order is 1. If the mobile component is intended to migrate to more than one component, additional *transientExecutionLocation*-type objects must be added to link each possible destination component, indicating the order (*order* property) to follow during the migration process (with consecutive numbers).

Additionally, the proposed design domain (Fig. 3) supports the representation of "code-on-demand". In this case, a program and its necessary data (a logicalMobileComponent-type object, *lmc1*) is transferred to a node (a *physicalStationaryComponent*-type object, *psc1*), which is the component that controls the operation. Similar to the previous case, an explicit *transientExecutionLocation*-type relationship object, *tel1*, links *psc1* with *lmc1*. In this particular case, the value of *mode* property of *tel1* version must indicate that it is a "code-on-demand" case of mobility.

Another kind of mobility that can be represented using the concepts included in the design domain (Fig. 3) is "mobile agents". By means of this technique, a mobile agent or program (*logicalMobileComponent*-type object) is transferred to a new node (*physicalStationaryComponent*-type object) to be executed there, being the results returned to the former location of the agent. In this case, given the agent autonomy, the agent itself controls the operation. In this situation, the value of *mode* property of the *transientExecutionLocation* object version indicates that "mobile agent" is the case of mobility employed. Additionally, some responsibility-type object versions could be linked to the *logicalMobileComponent*-type object that represents the mobile agent. Examples of responsibilities might be learning, cooperating, moving to a new location, and modifying its behaviour.

The proposed domain model (domain package, Fig. 1) is flexible enough to define a design domain suitable for the architect's needs and preferences. The domain package is understood as a language that allows us to define the various design object types with the properties and relationships between them. In this section, the domain model is instantiated with concepts of ADD and Con Moto. However, it could be instantiated with the concepts defined by any method and/or ADL. For example, if Ambient-PRISMA (Ali et al., 2006; Ali et al., 2005) is considered, the *ModellingConcept* class must be instantiated to represent the Ambient-PRISMA concepts defined in Ali et al. (2005), such as "*Ambient*" and "*Aspect*". Also, specific relationships defined by Ambient-PRISMA can be instantiated from *DomainRelationship* class. An example is "is-located-in" relationship, which allows linking concepts like *Component* and *Ambient*.

2.2 The Operations Model

The primitive operations *add*, *delete*, and *modify* provided by the versioning scheme to represent the transformation of *model versions* are not enough for representing the complex activities of designing the architecture of mobile systems. To make possible the extension of the available set of operations with operations suitable for such design domain, an operations model is proposed. To provide the foundations for computational tools, an object-oriented operations model is proposed, which is flexible enough to specify the domain operations to be used by the architects.

Therefore, a *Command* abstract class is introduced in the *Operations* package illustrated in Fig. 4. An *operation* is defined as a *macro command* that simply executes a sequence of commands (see in Fig. 4 *Operation* class represented as subclass of *MacroCommand* abstract class). The *arguments* and *body* of an *operation* must be defined to specify it. The commands of the body can be primitives (such as *add*, *delete*, or *modify*), iteration commands, variable assignment commands, or other existent operations. *Iteration* is a predefined command with a repetitive behaviour. It is specialized in *Loop* command, which executes a body (a sequence of ordered commands) for each element in a collection, and in *Next* command, which accesses sequentially to each element of a collection. Another predefined command is *VariableAssignment*, which represents the assignment of certain value to a variable with a given type. It should be noted that the *modelling concept* over which an operation is applied must be explicitly indicated.

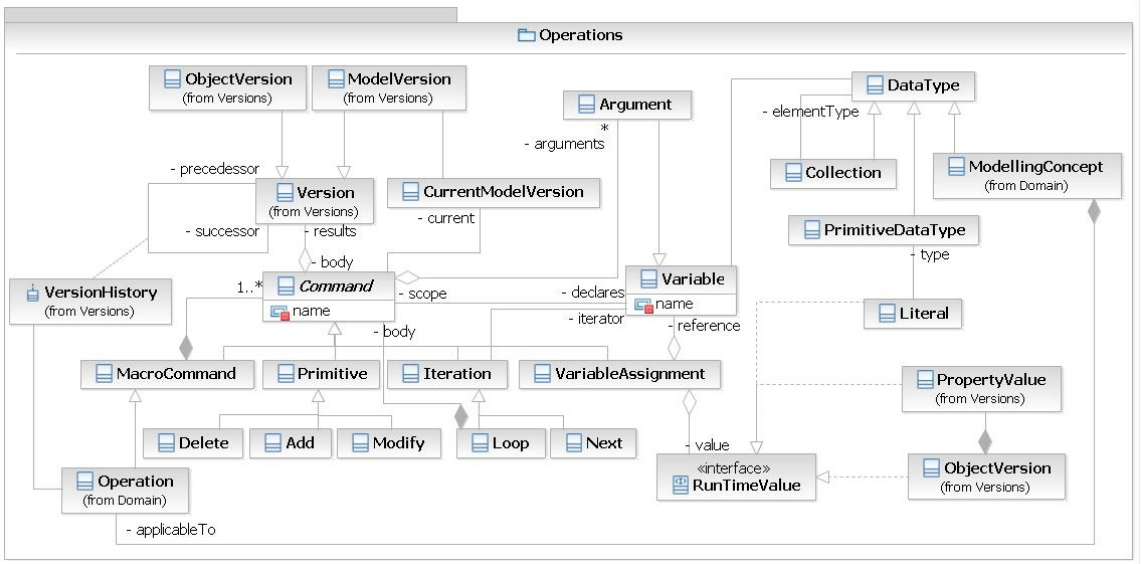

Fig. 4. Operations Package.

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*. The interface *RunTimeValue* represents the run-time values assumed by a *variable* or *argument* during the execution of an operation, which can be realized by *literals*, *object versions*, *modelling concepts*, or *property values* (*Literal*, *ObjectVersion*, *ModellingConcept*, *PropertyValue* in Fig. 4).

Many possible operations can be instantiated from the operations model. In a previous work (Roldán et al., 2006) several operations have been specified. They range from basic ones, like *addComponent*, *deleteComponent*, and *addScenario,* to operations that apply a style or tactic, like *applyClientServer* and *applyMVC*. Although these operations are valid for any architectural domain, this contribution focuses on

specific operations to design the software architecture of mobile applications. Table 1 classifies a set of operations related to structural concepts in three different levels of complexity:

i) *Basic Operations*: operations that allow to create and delete basic design objects (like physical or logical *components* and *connectors*);

ii) *Special Operations*: more complex operations that involve object refinement or delegation;

iii) *ApplyPattern Operations*: high level operations that generate a new set of design objects, which have a configuration based on an architectural pattern; in some cases, they do not modify the structure of the architectural model, but affect some properties of design objects.

Table 1. Basic, Special, and ApplyPatterns Operations

Basic Operations	
addPhysicalStationaryComponent	deletePhysicalStationaryComponent
addPhysicalMobileComponent	deletePhysicalMobileComponent
addLogicalStationaryComponent	deleteLogicalStationaryComponent
addLogicalMobileComponent	deleteLogicalMobileComponent
addPhysicalConnector	deletePhysicalConnector
addLogicalConnector	deleteLogicalConnector
addHostedByRel	deleteHostedByRel
addTransientExecutionLocationRel	deleteTransientExecutionLocationRel
addCanalisedRel	deleteCanalisedRel
addQualityRequirement	deleteQualityRequirement
addFunctionalRequirement	deleteFunctionalRequirement
addPort	deletePort
addRole	deleteRole
addProperty	deleteProperty
addResponsibility	deleteResponsibility
addScenario	deleteScenario
Special Operations	
setPhysicalConnector	delegateResponsibility
setLogicalConnector	delegateScenario
refineComponent	verifyScenario
refineResponsibility	setAttachment
ApplyPatterns Operations	
applyClientServer	applySynchronization
applyMVC	applyInformationBroker
applyThreeLayers	applyLogicalMobility

Fig. 5 presents functional specifications for some of the basic operations defined in Table 1. The rest of the operations are 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 *addPhysicalMobileComponent*(*s*, *pc*, *lports*, *lresps*, *attr*) operation allows adding a physical mobile component *pc* to a system *s*. As it can be seen in Fig. 5, this operation is carried out by a series of commands. First, a version of a physical mobile component *c* is added (*add(c)*). After that, a set of responsibilities (specified by the *lresp* list argument) and ports (detailed by the *lports* list argument) are inserted. The last argument of the *addPhysicalMobileComponent* operation is an ordered list of attribute values, such as processor, available memory, maximum available storage and a number of instances in the system. It should be taken into account that the addition of a new element into a model version means the instantiation of a given modelling concept (which is part of the domain model in

Fig. 3). Thus, this fact implies the creation of a versionable object at repository level and an object version at version level. Also, these design objects are associated to others at repository level, by including an instance of association (Fig. 1) between them. In functional specifications of operations *addRelationship* command express how these associations have to be accomplished.

As it can be observed, functional specifications give an outline of how the operations may be defined using a computational tool. The employed syntax should be recognized by a computational tool based on the proposed operations model (Fig. 4). Section 4 introduces an example of how TracED, a prototype that implements the proposed operations model, allows the definition of these operations likeway.

```
addQualityRequirement(s, qr)
add(qr)
addRelationship(s, qr)
```

```
addScenario(qr, sce)
add(sce)
addRelationship(qr, sce)
```

```
addPhysicalMobileComponent(s, pc, lports,
lresps, {cpu-val, mem-val, storage-val,
instantes-val})
add(pc, {cpu-val, mem-val, storage-val,
instances-val})
for each p in lports
   addPort(pc, p)
end for
for each r in lresps
   addResponsibility(c,r)
end for
addRelationship(s, pc)
```

```
addPhysicalStationaryComponent(s, pc, lports,
lresps, {cpu-val, mem-val, storage-val})
add(pc, {cpu-val, mem-val, storage-val})
for each p in lports
   addPort(pc, p)
end for
for each r in lresps
   addResponsibility(c,r)
end for
addRelationship(s, pc)
```

```
addLogicalStationaryComponent(cont, c, lports,
lresps)
add(c)
for each p in lports
   addPort(pc, p)
end for
for each r in lresps
   addResponsibility(c,r)
end for
addHostedByRel(cont,  c)
```

```
addLogicalMobileComponent(cont, c, lports,
lresps, type)
add(c)
for each p in lports
   addPort(pc, p)
end for
for each r in lresps
   addResponsibility(c,r)
end for
addTransientExecutionLocationRel(cont-c, cont,
c , {type, 0})
```

```
addPhysicalConnector(pc, role1, role2)
add(pc)
addRole(role1)
addRole(role2)
addRelationship(pc, role1)
addRelationship(pc, role2)
```

```
addLogicalConnector(lc, role1, role2, pcList)
add(lc)
addRole(role1)
addRole(role2)
addRelationship(lc, role1)
addRelationship(lc, role2)
for each pc in pcList
   addChannelRel(rel, lc, pc)
end for
```

```
addHostedByRel(c1, c2)
add(c1c2)
addRelationship(c1c2, c1)
addRelationship(c1c2, c2)
```

```
addTransientExecutionLocationRel(tel, cont,  c,
attributes)
add(tel, attributes) // attributes order: type,
locationOrder
addRelationship(tel, cont)
addRelationship(tel, c)
```

```
addCanalisedByRel(r, lcnn, pcnn)
add(r)
addRelationship(r, lcnn)
addRelationship(r, pcnn)
```

Fig. 5. Specification of some basic operations.

Similarly, it is possible the definition of special operations (some examples are shown in Fig. 6). These operations increase their abstraction level, so they need auxiliary functions to be specified. Functions do not constitute architectural operations. They are mainly interactive functions that require the intervention of the architect. For example the operation *selectPort*(*c*) asks the user to choose one port of a ports list, to do some action with the selected one. Another sort of auxiliary functions are "*get*" functions. An example is *getResponsibilities*(*c*), which collects all the *responsibility*-type object versions associated with a component *c* in a specific model version. Such a function is part of an special operation called *delegateResponsibility*(c_1, c_2). It enables to delegate a responsibility of component c_1 to component c_2. In a similar way, the operation *delegateScenario* proceeds. As part of this group, the architect has also defined *setLogicalConnector* and

setPhysicalConnector, which permit not only to create a (physical or logical) connector object version, but also to set the attachment with the components communicated by it. In addition, *setLogicalConnector* includes a list of physical connectors as the last argument (*pcnnList*), which will act as the physical channels for the logical connections.

```
setPhysicalConnector(pc, comp1, comp2)
addPhysicalConnector(pc, pc-role1, pc-role2)
p1 := selectPort(comp1)
p2 := selectPort(comp2)
setAttachment(att1, pc-role1, p1)
setAttachment(att2, pc-role2, p2)
```

```
setLogicalConnector(cnn, c1, c2, pcnnList)
addLogicalConnector(cnn, [cnn-r1, cnn-r2],
pcnnList)
c1-p := selectPort(c1)
c2-p := selectPort(c2)
setAttachment(cnn-r1, c1-p)
setAttachment(cnn-r2, c2-p)
```

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

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

```
delegateScenario(c1, c2)
lScens = getScenario(c1)
for each s in lScens
  if (delegate?(c2, s))
    addRelationship(c2, s)
  end if
end for
```

Fig. 6. Specification of some special operations.

ApplyPattern operations encapsulate well known solutions for recurring design problems that arises in specific design situations. Given that patterns are a means of documenting software architectures, the execution of an *applyPattern* operation describes the vision the architect has in mind when designing a software system (Buschmann et al., 1996). These operations apply an architectural pattern either refining a component object version (and therefore, deleting this original component) into a new set of components and connectors that are instantiated from a pre-existing style, or just adding a set of elements (i.e.: component and connectors with their ports, responsibilities and roles) in order to create a new architectural configuration. Due to the very high level of abstraction of *applyPattern* operations, they need to interact with the designer asking about responsibilities and scenarios delegation, as well as how to attach connectors between external and new components.

In Fig. 7, some operations that involve the application of architectural pattern are specified. The first one, named *applyThreeLayers*, contains the semantic of the Layers pattern (Buschmann et al., 1996). It decomposes an application into groups of subtasks in which each group of subtasks is at a particular level of abstraction. In this case the operation is specified for three layers. This operation can be executed by an architect during a software architecture design process, when his / her design decision is to create different layers to deal with a specific aspect of the application and uses the services of the next lower layer. This operation specified in Fig. 7 requires four arguments: the first argument is the component to be split in the proposed layers; the following three arguments are the name of the layers to be added, which should be ordered from the highest level to the lowest level layer. Each component layer object version is added by means of an *addLogicalStationaryComponent* operation (which also creates and assigns a pair of ports) and indicates in one argument which the host component is. In addition, the responsibilities of the original component are delegated to new components using *delegateResponsibilites* operations, which interact with the actor who executed the operation. Finally, *applyThreeLayers* asks the designer how to attach connectors between external and refined components by means of a predefined interactive function (*PortMap?*).

ApplySynchronization operation (Fig. 7) encloses the semantic of a pattern suitable for mobile applications (Roth, 2002), which deals with the problem of keeping synchronized two databases located on different devices or computers, which are weakly connected. In this context, several users change some of the data on different devices, often simultaneously; thus, data tends to be out of sync. This architectural pattern provides a solution by including in the architecture a logical component that acts as a *synch engine*, in each device or computer. As it is observed in Fig.7, arguments must be provided to indicate the logical stationary components where both *synch engines* components are going to be located. As this kind of software components do not concern directly to business logic, they can be included as part of a middleware layer.

Therefore, the operation adds two *logicalStationaryComponent* objects and assigns to them a pair of predefined ports and responsibilities. The responsibilities include exchanging the modifications or differences occurred since the last synchronization (*rDifferenceExchange*), detection of potential conflicts (*rConflictDetection)*, and resolution of conflicts (*rConflictResolution*).

ApplyConnectionBroker (which is also specified in Fig. 7) encloses the semantic of the Information Broker architectural pattern identified by Risi and Rossi (2004), who proposed a catalogue of architectural patterns for mobility. The aim of the *Connection Broker* pattern is to manage and select the more suitable of two possible connections from a component application to a database. A possibility is enabling the connection to the main database that resides in a server, and the other is establishing a connection with a replica of such a database. The replicated data base is a reduced version of the main data base and possibly is out of date. To provide this decision capability to an architecture, an intermediary component (the *Connection Broker*) is included by the *applyConnectionBroker* operation. This component is in charge of setting a suitable connection to any of the databases for inserting/updating/recovering data, based on a set of rules that indicates when a connection is convenient and when not.

```
applyThreeLayers(c1, l1, l2, l3)
host := getHostComponent(c1)
addLogicalStationaryComponent(host, l1, [p1,
p2])
addLogicalStationaryComponent(host, l2, [p3,
p4])
addLogicalStationaryComponent(host, l3, [p5,
p6])
addLogicalConnector(l1l2, [r1, r2])
addLogicalConnector(l2l3, [r3, r4])
setAttachment(p2,r1)
setAttachment(p3,r2)
setAttachment(p4,r3)
setAttachment(p5,r4)
delegateResponsibility(c1, l1)
delegateResponsibility(c1, l2)
delegateResponsibility(c1, l3)
lp = getPorts(c1)
for each p in lp
  np = PortMap?(p) // Ask the user the port to
map
  r = getRol(p)
  addRelationship(np, r)
end for
delete(c1)
```

```
applyConnectionBroker(c, appComp, [dest1,
dest2])
addLogicalStationaryComponent(c, broker, [b-p1,
b-p2, b-p3])
setLogicalConnector(b-lc1, broker, appComp)
setLogicalConnector(b-lc2, broker, dest1)
setLogicalConnector(b-lc3, broker, dest2)
```

```
applySynchronization(c1, c2,  db1, db2)
addLogicalStationaryComponent(c1, syncEngine1,
[se1-p1, se1-p2])
addResponsibility(syncEngine1,
rDifferenceExchange)
addResponsibility(syncEngine1,
rConflictDetection)
addResponsibility(syncEngine1,
rConflictResolution)
addLogicalStationaryComponent(c2, syncEngine2,
[se2-p1, se2-p2])
addResponsibility(syncEngine2,
rDifferenceExchange)
addResponsibility(syncEngine2,
rConflictDetection)
addResponsibility(syncEngine2,
rConflictResolution)
setLogicalConnection(se1-db1, syncEngine1, db1)
setLogicalConnection(se2-db2, syncEngine2, db2)
setLogicalConnection(se1-se2, syncEngine1,
syncEngine2)
```

```
applyLogicalMobility(mComp, cont, {order})
addTransientExecutionLocationRel(mComp-cont,
cont,  mComp,{"logical-mobility", order})
```

```
applyPhysicalMobility(mComp, cont, {type,
order})
addTransientExecutionLocationRel(mComp-cont,
cont,  mComp, {type, order})
```

Fig. 7. Specification of operations that apply architectural patterns.

Another architectural pattern is materialized in *applyLogicalMobility* operation (Fig. 7). The mechanism of "logical code mobility" (Bieszczad and White, 2007; Fuggetta et al., 1998) should be applied in a situation where replicas of certain computational unit are available in the local code repositories of two different execution environments (physical components or hosts). This pattern incorporates in an architectural model the behaviour that makes possible moving the execution of a code unit from one machine to other. In this way, a replica of the executing unit running on the first machine is loaded from a local repository and initialized with the transferred state to resume its execution. The arguments of *applyLogicalMobility* operation are the unit code (*logical mobile component*) that will change its execution location, the next physical component where this component will run, and the properties values that indicate the execution location order assigned to this new location.

The specification of *applyLogicalMobility* operation comprises an *addTransientExecutionLocationRel* operation. As a consequence, an object whose type is *transientExecutionLocationRel* is added. This object relates the mobile component with the physical component that will temporally host it. Particularly, the *type* property of the added object will receive the value of "Logical Mobility". In this way, *applyLogicalMobility* operation allows the designer to express in the architectural model the kind of code mobility that the future system will support, as well as the locations in which a logical mobile component will be executed.

Similar operations for *physical code mobility* (Fuggetta et al., 1998) such as Remote Evaluation, Code on demand, and Mobile Agent could be specified by assigning suitable property values. It should be noted that the operation and domain packages (Fig. 4 and 1, respectively) allow us to define a particular design domain suitable for the architect's necessities and preferences. During the domain definition, the operations that the architect will be able to use in the mobile architecture design process should be defined. The proposed model is a first step towards the development of computational tools to support the design process.

3 Case Study

The case study describes the design process of the software architecture of a Sales Force Management System. The system is intended to improve the efficiency of the sales representatives of an enterprise in the field. They meet clients to promote and sell the company products making use of a local database suited in their mobile devices and entering new orders. However, when connectivity with the company database server is possible, the information must be gathered (sent) from (to) it. Obviously, the company database must periodically be synchronized with the databases in each mobile device.

Functional requirements for such a system are the following:

i) Salespeople can place orders from different locations using their mobile devices, preferably in real time (*FunctionalRequirement_1*).

ii) An order need to be validated. The application running in the salesperson mobile device has features for checking the inventory levels of the ordered products and the client credit situation (that controls if the clients have no debts with the company, their credit limits, and the possible means of payment). To accomplish the order validation process, the mobile device needs to connect to the company server, to get the client credit information and up-to-date inventory information (*FunctionalRequirement_2*).

iii) Once an order has been validated and confirmed, it is incorporated to the company server (*FunctionalRequirement_3*), in order to update the inventory levels and start its processing. A

new order in the system may imply the generation of a manufacturing order or the acquisition of raw materials.

The software architecture of this mobile system must satisfy the aforementioned functional requirements and some non-functional or quality requirements (Bass et al., 2003), which are described as follows:

i) It is possible to place an order even if there is no connectivity with the company server (*Availability_QR1*). If the application that runs in a salesperson's mobile device is not able of reaching the company database, the order cannot be validated. Therefore, it is saved on the database located at the mobile device, in "awaiting confirmation" state. This quality requirement is known as *Availability* (Bass et al., 2003) or *Reliability* (ISO, 2001).

ii) When connectivity is re-established, the company database is synchronized with the mobile databases to keep data updated, and, if it is possible, to change the status of orders from "awaiting confirmation" to "confirmed" (*DataFreshness_QR2*).

iii) The system supports different sorts of mobile devices, which means that the processing capability of the devices is not homogenous. Consequently, when the processing power and the required memory of a device is not enough for executing some application features (such as the ones related to order validation) it is possible run them on the company application server (*Performance_QR3*). This quality requirement is called *Performance* (Bass et al. 2003) or *Efficiency* (ISO, 2001).

The design process of the architecture for the introduced mobile system will be conducted by following the ADD method; thus, it will be presented as a decomposition process, guided by architectural drivers. The design decisions made by the architect (or designer) at each point of the design process are captured by a sequence of operations, which is applied to a previous model version and generates a new software architecture model version. In the present case study, the architect employs the design domain for mobile architectures defined in the previous section.

The designer (an architect) begins with an empty root model version (*Root Model Version*), where he/she adds the first object version that represents the system whose architecture is going to be built. Thus, the first model version *ModelVersion1* is obtained after applying the sequence of operations $\phi_1 = \{addSystem(SalesSystem)\}$.

Sequence of Operations ϕ_1 for achieving ModelVersion1
addSystem(SalesSystem)

Then, the architect continues by identifying the requirements for the intended system. This means the execution of a sequence of operations ϕ_2, which comprises a series of *addQualiyRequirement* and *addFunctionalRequirement* operations. As a consequence, *functionalRequirement*-type and *qualityRequirement*-type objects are included giving rise to a new model version (*ModelVersion2*).

Sequence of Operations ϕ_2 for achieving ModelVersion2
addQualityRequirement(SalesSystem, Availability_QR1)
addQualityRequirement(SalesSystem, DataFreshness_QR2)
addQualityRequirement(SalesSystem, Performance_QR3)
addFunctionalRequirement(SalesSystem, FunctionalRequirement_1)
addFunctionalRequirement(SalesSystem, FunctionalRequirement_2)
addFunctionalRequirement(SalesSystem, FunctionalRequirement_3)

During the design process the architecture is represented combining structural, behavioural and deployment elements from different architectural views (Bass et al., 2003; ISO, 2008). The architect begins by adding the object versions that represent the physical (stationary or mobile) components involved in the architecture,

which are going to act as the hosts of logical components. On the company main office, the system is deployed on an application server and a database server. So, the architect adds two physical stationary components: *ApplicationServer* and *DataBaseServer*. The software components responsible of the main business logic will run on the first component (*ApplicationServer*), and the second will host the database and data access services. As salespeople will carry their own mobile devices (*SalesForceMobileClient*) to query the system and to generate orders, the architect includes instances of such devices in the new model version. The ϕ_3 sequence of operations is given in the next listing, which generates *ModelVersion3*.

Sequence of Operations ϕ3 for achieving ModelVersion3

```
addPhysicalStationaryComponent(SalesSystem, ApplicationServer, {'Core2 T5200 1.60GHz','2Gb','160Gb'},
                                                                [p1, p2,p3], [])
addPhysicalStationaryComponent(SalesSystem, DataBaseServer, {'Xeon 3.20GHz', '2Gb', '120Gb'}, [p4], [])
addPhysicalMobileComponent(SalesSystem, SalesForceMobileClient, {'Intel 312MHz 300Mhz','128Mb','2GB', 20},
                                                                [p5, p6], [])
```

Above sequence of operations includes *addPhysicalStationaryComponent* and *addPhysicalMobileComponent* operations. Some argument values have to be provided to set the property values for created object versions. These physical components are not isolated; they have to communicate with each other by means of different physical connections. Thus, the architect decides that an Ethernet connection is suitable between the *ApplicationServer* and *DataBaseServer* components. This decision is materialized by a *setPhysicalConnector* operation execution. Another decision is to set two alternative and redundant physical connections between the *SalesForceMobileClient* and the *ApplicationServer*. Both connections are wireless, but differ in the communication protocol they use, *3G* and *WiFi*. As a result of applying the ϕ_4 sequence of operations, *ModelVersion4* is obtained.

Sequence of Operations ϕ4 for achieving ModelVersion4

```
setPhysicalConnector(WiFi, ApplicationServer, SalesForceMobileClient)
setPhysicalConnector(3G, ApplicationServer, SalesForceMobileClient)
setPhysicalConnector(Ethernet, ApplicationServer, DataBaseServer)
```

Fig. 8 illustrates the model evolution that takes place from applying the sequence of operations ϕ_4 on *ModelVersion3*, which adds the physical connectors between the mobile devices and the stationary components. There, a series of *setPhysicalConnector* operations forms the sequence of operations ϕ_4. The intention of the architect is setting up the physical connections between the physical components in *ModelVersion3*. As it was specified in Fig. 6, *setPhysicalConnector* implies both the addition of a new *connector*-type object and its *role*-type objects, and the attachments to link the respective *port*-type objects.

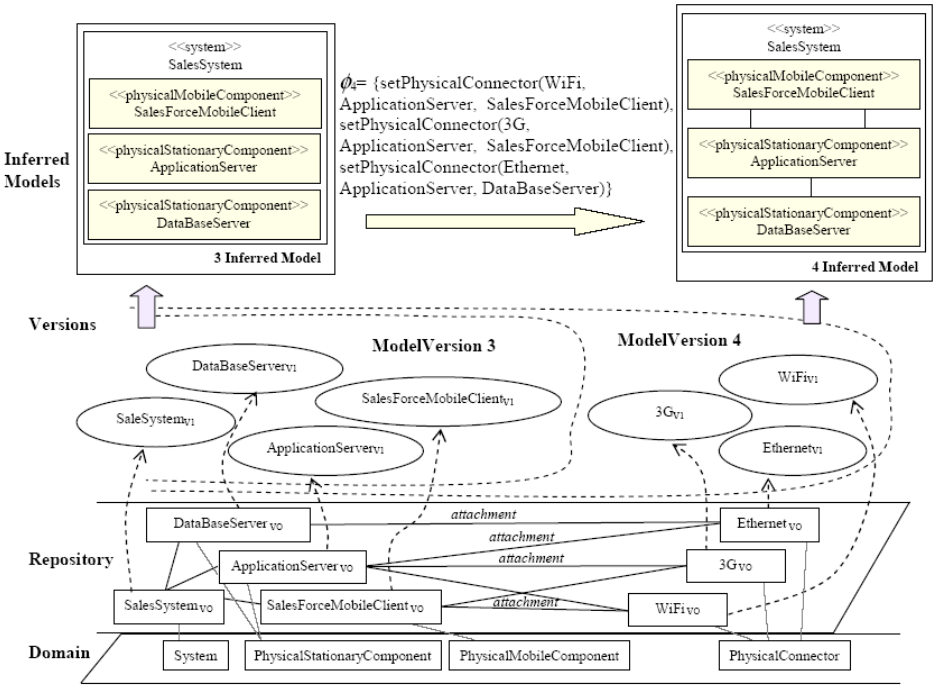

Fig. 8. Obtaining *ModelVersion4* from *ModelVersion3*

At the top of Fig. 8, the *inferred model* is visualized, which is obtained from views of the *versions level* on the *repository*. At versions level (Fig. 8), $SalesSystem_{v1}$ (*system*-type object version) $SalesForceMobileClient_{v1}$, $ApplicationServer_{v1}$ and $DataBaseServer_{v1}$ (*component*-type object versions) belong both to *ModelVersion3* and *ModelVersion4* (ports are not shown for simplicity). At repository level, these design objects are represented by $SalesSystem_{vo}$, $SalesForceMobileClient_{vo}$, $ApplicationServer_{vo}$, $DataBaseServer_{vo}$, respectively. These objects are instances of *versionable object* (Fig. 1). As a consequence of the execution of *setPhysicalConnector* operation, object versions for representing three physical connectors are added to *ModelVersion4* ($3G_{v1}$, $WiFi_{v1}$, and $Ethernet_{v1}$). Such object versions maintain a versionable object instance at repository level. Moreover, associations between the included objects are maintained at repository level, like the *attachment* associations between components (Fig. 8).

Afterwards, the software architecture continues evolving in a series of model versions, which arise as a result of adding all the object versions necessary to represent the logical components involved in the software architecture, their responsibilities and properties, and the *connector*-type objects that provide the communication between them. Having laid out the physical configuration of the system in terms of the physical components and connectors, logical components with a high level of abstraction that represent the application to be developed are created and deployed in each physical component. On the one hand, a *ServerSideApp logicalStationaryComponent*-type is added to *ApplicationServer*. On the other hand, a *MobileClientSideApp* is created and deployed on the *SalesForceMobileClient* physical component. Then, a logical component responsible of providing data services (*DataServicesApp*) is incorporated in *DataBaseServer* physical component. These three components are added to the architecture providing information about responsibilities, and leaving information about connection between them for a next step in the design process. Therefore, the sequence of operation for achieving *ModelVersion5* is detailed as follows.

Sequence of Operations ϕ_5 for achieving ModelVersion5

```
addLogicalStationaryComponent(ApplicationServer, ServerSideApp,
      [SSA_UIResp, SSA_OrderAcceptResp, SSA_OrderValidationResp, SSA_DataQueryResp,
       SSA_SynchResp, SSA_CreditVerificationResp, SSA_InventoryManagementResp])
addLogicalStationaryComponent(SalesForceMobileClient, MobileClientSideApp,
                              [MCSA_UIResp, MCSA_OrderAcceptResp, MCSA_OrderValidationResp,
```

MCSA_DataQueryResp, MCSA_SynchResp])
addLogicalStationaryComponent(DataBaseServer, DataServicesApp,
[DSA_ConnectionResp, DSA_UserValidationResp, DSA_DataBaseManagementResp])

At this point in the design process, the architect wants to add further decomposition to the system. The logical components that will compose the architecture belong to different levels, where the high level elements rely on the lower-level ones. Therefore, since a layered architecture is suitable to the intended architecture model, the designer identifies three levels of abstraction. The higher layer contains logical components related to the application itself. The next layer contains the middleware, which provides services transparently like database synchronization and data routing through the most convenient connection. The lowest layer deals with data access features. These decisions are materialized by the applying of a Layers pattern on two logical components of the *ModelVersion5* (*ServerSideApp* and *MobileClientSideApp*). Thus, the sequence of operations ϕ_6 reachs *ModelVersion6*, where logical components (layers) such as *ServerApplicationLayer*, *ServerMiddlewareLayer*, *ServerDataAccessLayer*, *MobileApplicationLayer*, *MobileMiddlewareLayer*, and *MobileDataAccessLayer* are created (Fig.9). As it can be observed from specification in Fig. 7, *ApplyThreeLayers* is a refining operation as the original component is deleted from the resulting model version. The current model version (*ModelVersion6*) is shown in Fig. 9 by using a deployment view.

Sequence of Operations ϕ_6 for achieving ModelVersion6
applyThreeLayers(ServerSideApp, [ServerApplicationLayer, ServerMiddlewareLayer, ServerDataAccessLayer])
applyThreeLayers(MobileClientSideApp, [MobileApplicationLayer, MobileMiddlewareLayer, MobileDataAccessLayer])

Having organized the logical arrangement of logical components, the architect includes a set of *component*-type objects. In the application layer of the mobile side of the architecture (Fig. 11), the designer includes: *UIManager*, *OrderReceiver*, and *OrderValidator* logical components (and their ports). *UIManager* is in charge of the user interface and accepting the user requests. *OrderReceiver* has responsibilities of processing orders and queries of the user, and *OrderValidator* has responsibilities of checking order correctness, credit situation and product inventory levels. Particularly, the responsibilities of the last component are CPU and memory intensive. Therefore, given the constraints on these mobile devices resources, the architect foresees the necessity of adding the component object version as an instance of *LogicalMobileComponent* design object type. In this way, the applied sequence of operations ϕ_7 on *ModelVersion6*, which generates *ModelVersion7*, includes some *addLogicalStationaryComponent* and *addLogicalMobileComponent* operations. Afterwards, to allow the interaction between these components logical connections are set to achieve the functional requirements of the system.

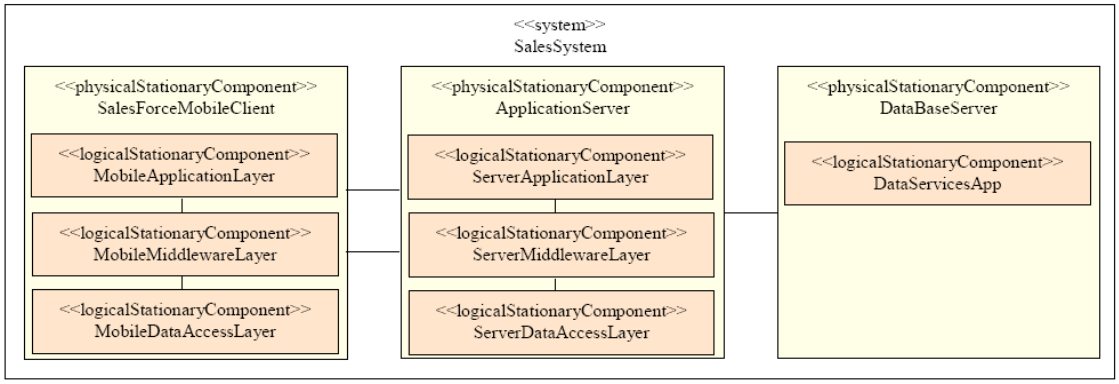

Fig. 9. Deployment view of *ModelVersion6*

Sequence of Operations ϕ_7 for achieving ModelVersion7
addLogicalStationaryComponent(MobileApplicationLayer, UIManager, [UI-p1],[])
addLogicalStationaryComponent(MobileApplicationLayer, OrderReceiver, [OR-p1, OR-p2, OR-p3, OR-p4, OR-p5],[])
addLogicalMobileComponent(MobileApplicationLayer, OrderValidator,[OV-p1, OV-p2, OV-p3],[])

Sequence of Operations ϕ_8 for achieving ModelVersion8
setLogicalConnector(LC1, UIManager, OrderReceiver,[])
setLogicalConnector(LC2, OrderReceiver, OrderValidator,[])

Then, in a similar way, the architect adds analogous components in the application layer of the *ApplicationServer*. This is made by using a series of *addLogicalStationaryComponent*, which forms the sequence of operations ϕ_9. As a result *ModelVersion9* arises including three new *logicalStationaryComponent*-type objects named *OrderBusinessLogic*, *InventoryBusinessLogic*, and *CreditBusinessLogic* (Fig. 10). The first one is in charge of saving all coming orders and recovering existent orders requested from devices and other applications. The second one has the responsibility of managing products and stock (inventory) information. Some of the mentioned responsibilities could be delegated from the container layer, whereas others could be passed as an argument. To simplify the case study, *lresps* argument is left empty and it is assumed that the architect will carry out responsibility delegations in later model versions. The next step is setting the logical connection between them (given a sequence of operations ϕ_{10}, which includes a series of *setLogicalConnector* operations). Fig. 11 shows the architecture layout at application layer (client and server) by means of a C&C view.

Sequence of Operations ϕ_9 for achieving ModelVersion9
addLogicalStationaryComponent(ServerApplicationLayer, OrderBusinessLogic, [OBL-p1, OBL-p2], [])
addLogicalStationaryComponent(ServerApplicationLayer, InventoryBusinessLogic, IBL-p1, [])
addLogicalStationaryComponent(ServerApplicationLayer, CreditBusinessLogic, CBL-p1, [])

Sequence of Operations ϕ_{10} for achieving ModelVersion10
setLogicalConnector(LC3, OrderReceiver, OrderBusinessLogic, [WiFi, 3G])
setLogicalConnector(LC4, OrderValidator, InventoryBusinessLogic, [WiFi, 3G])
setLogicalConnector(LC5, OrderValidator, CreditBusinessLogic, [WiFi, 3G])

Afterwards, *ServerDataBaseManager* is added to provide data access from components of the *ApplicationServer* to the *DataBaseServer*, in a transparent way. Therefore, the architect executes an *addLogicalStationaryComponent* indicating the component that hosts the new component (see sequence of operations ϕ_{11}). The resulting model version is *ModelVersion11*.

Sequence of Operations ϕ_{11} for achieving ModelVersion11
addLogicalStationaryComponent(ServerDataAccessLayer, ServerDataBaseManager, SDB-p1,[])

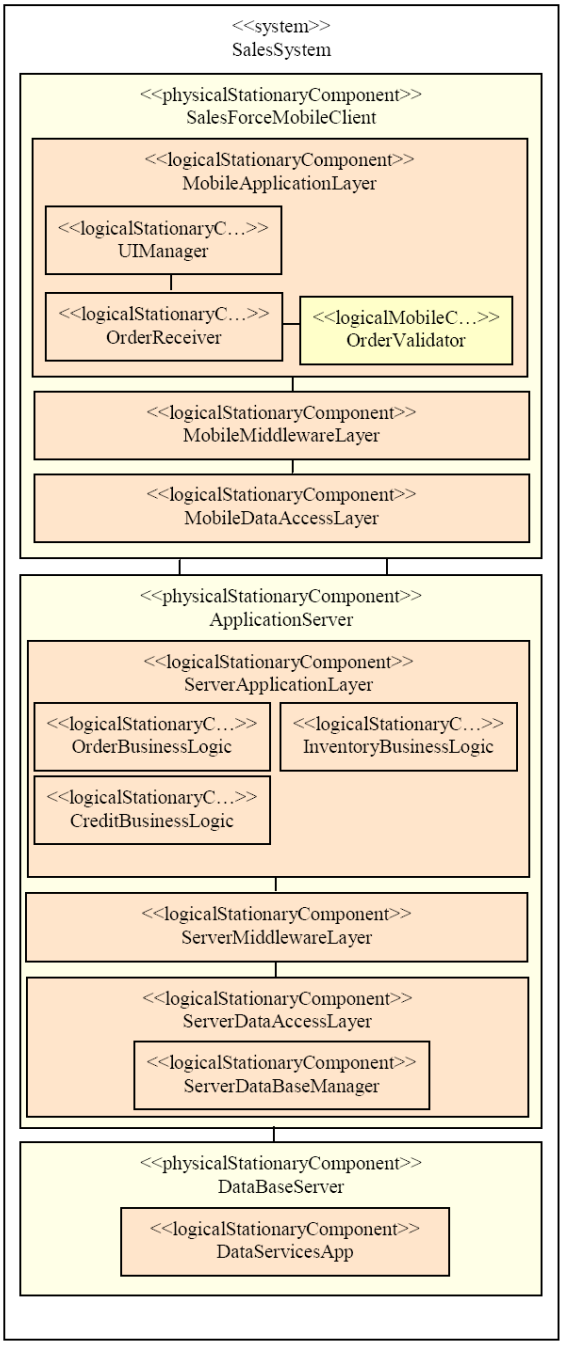

Fig. 10. Deployment view of *ModelVersion11*

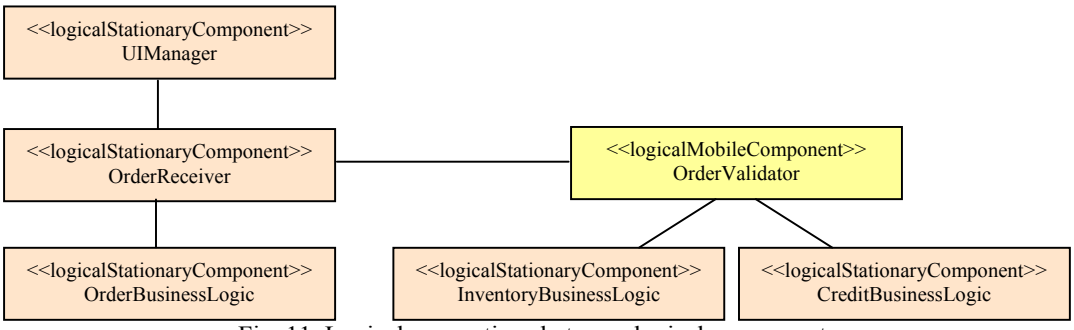

Fig. 11. Logical connections between logical components

At this point in the design process, the architecture achieves part of the intended functional requirements (*FunctionalRequirement_1*, *FunctionalRequirement_2*, *FunctionalRequirement_3*). Now, the architect focuses on the needed quality requirements. Firstly, the *Availability_QR1* requirement is regarded. This requirement demands to the application to keep on working despite no connectivity with the main database. Therefore, the solution is to include a temporal o minimal database located at the mobile device, to save application data even whether connection with the remote server application is not possible. Additionally, the mobile application should be able to decide how to route the generated orders (controlling the flow of the application business logic). On the other hand, the database redundancy makes it necessary some synchronization mechanism to maintain updated the company database.

Therefore, the architect's design decision is to provide further granularity to mobile-side application by adding two new component object versions. At application layer, a component responsible for a limited business logic (named *MinimalOrderBussinessLogic*), and at data access layer, a component to manage the access to a reduced local database (named *LocalDataBaseManager*). This is materialized in the sequence of operations ϕ_{12}, which applied on *ModelVersion11* generates *ModelVersion12*.

Sequence of Operations ϕ_{12} for achieving ModelVersion12
addLogicalStationaryComponent(MobileApplicationLayer, MinimalOrderBusinessLogic, [MOB-p1, MOB-p2, MOB-p3], [])
addLogicalStationaryComponent(MobileDataAccessLayer, LocalDataBaseManager, [LDB-p1], [])

Fig. 12 shows a partial view of *ModelVersion12* and its inferred model. This figure illustrates a fragment of the design process. At the top of it, partial views of the inferred model versions are represented, making use of a mix of structural and deployment view. There, three consecutive model versions are presented (*ModelVersion12*, *ModelVersion13* and *ModelVersion14*). In *ModelVersion12*, two object versions of components can be observed (*MinimalOrderBusinesLogic* $_{v1}$ and *LocalDataBaseManager* $_{v1}$) as well as the *hostedByRel*-type objects that indicate where they are located ($MDBA\_LDDBM_{v1}$, *MAL_MOBL* $_{v1}$). *ModelVersion13* is generated from *ModelVersion12* by applying the *sequence of operations* ϕ_{13}. The applied operations set the logical connections among logical components situated at *SalesForceMobileClient. LC6* (between *OrderReceiver* and *MinimalOrderBusinesLogic*) is an internal communication between components belonging to the same layer. *LC7* logical connector (between *MinimalOrderBusinesLogic* and *LocalDataBaseManager*) represents the communication between components belonging to adjacent layers. These new connectors also have a double representation at *repository* level (given by $LC6_{vo}$ and $LC7_{vo}$ versionable objects) and at *versions* level (given by $LC6_{v1}$ and $LC7_{v1}$ object versions).

Sequence of Operations ϕ_{13} for achieving ModelVersion13
setLogicalConnector(LC6, OrderReceiver, MinimalOrderBusinessLogic, [])
setLogicalConnector(LC7, MinimalOrderBusinessLogic, LocalDataBaseManager, [])

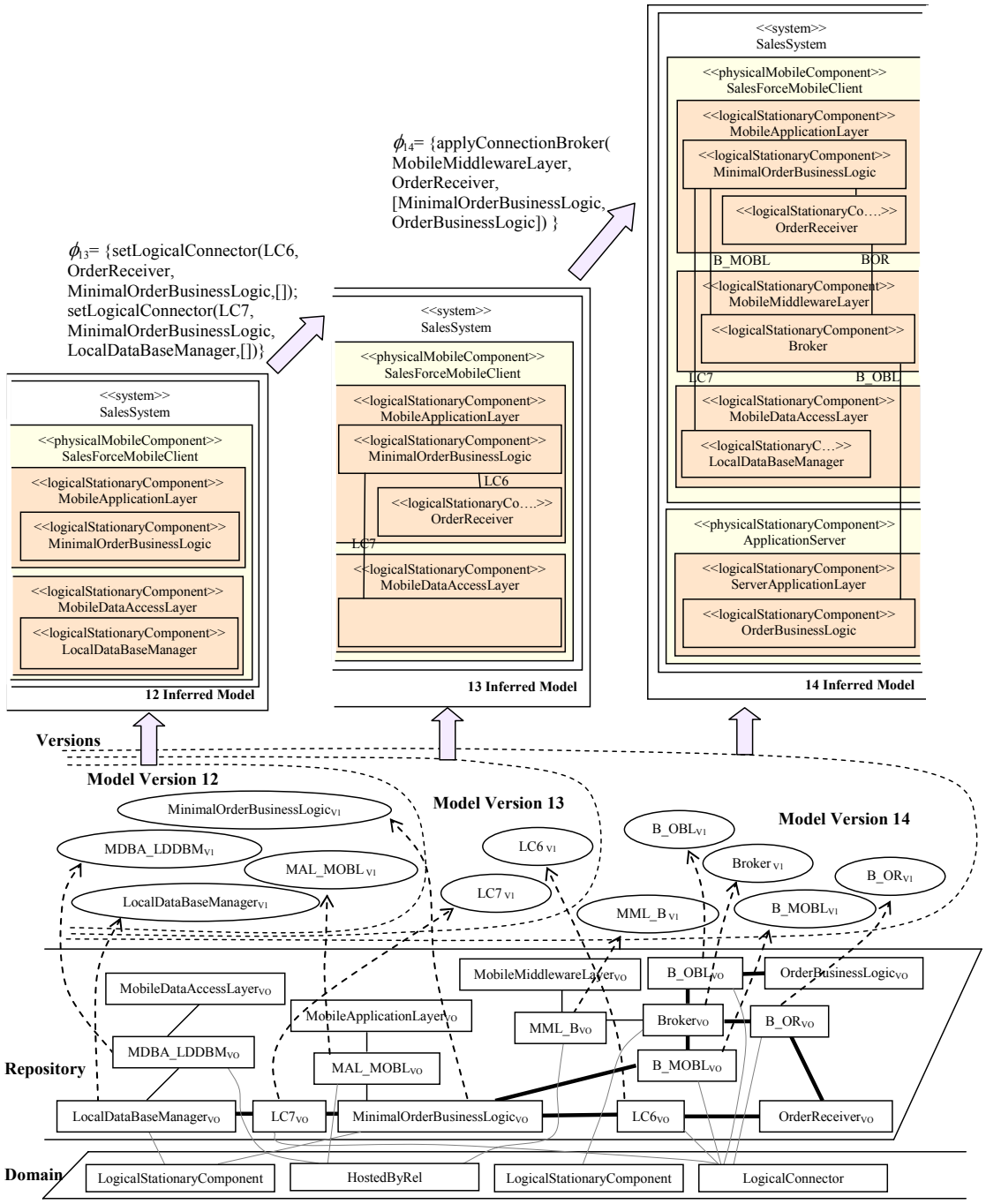

Fig. 12. Model evolution from *ModelVersion12* to *ModelVersion14*

The next step in the design process is to apply the *ConnectionBroker* pattern to include elements of the logic of the application. *ApplyConnectionBroker* operation (Fig. 7) pursues *Availability_QR1* requirement. The operation arguments are *OrderReceiver* component (Fig. 12) and the two alternative components with it could communicate: *MinimalOrderBusinessLogic* (a local component) or *OrderBusinessLogic* (a remote component). The first argument is the component that hosts the broker component. Particularly, in this software architecture model, the broker is located in the *MobileMiddlewareLayer* component object. The

Broker component and a set of responsibilities that enable it to determine the logical path to follow, based on a set of policies or rules, are added.

Sequence of Operations ϕ_{14} for achieving ModelVersion14
applyConnectionBroker(MobileMiddlewareLayer, OrderReceiver, [MinimalOrderBusinessLogic, OrderBusinessLogic])

The *applyConnectionBroker* operation specifies that the *Broker* object is going to be located at the *MobileMiddlewareLayer* by including a *hostedByRel*-type object that relates them (MML_B_{v1} at versions level and MML_B_{vo} at repository level, Fig. 12). Additionally, a set of *logicalConnector*-type object is present in *ModelVersion14* in order to communicate the broker component with the application layer components that require its services (located both at the mobile client and at the application server). They are *B-MOBL*, *B-OR*, *B-OBL*. As it can be observed, the versioning scheme represents all incorporated objects at *repository* level (given by $Broker_{vo}$, MML_B_{vo}, $B\text{-}MOBL_{vo}$, $B\text{-}OR_{vo}$, $B\text{-}OBL_{vo}$ objects) and at *versions* level (given by $Broker_{v1}$, MML_B_{v1}, $B\text{-}MOBL_{v1}$, $B\text{-}OR_{v1}$, $B\text{-}OBL_{v1}$ object versions).

As it was mentioned previously, some mechanisms are necessary to synchronize the databases, in order to satisfy the *DataFreshness_QR2* quality requirement. Therefore, the architect considers applying the *Synchronization* pattern (Fig. 7). The first two arguments of this operation are the component-type object versions where both synchronizing engines (logical components) are going to be located. Also, the two last argument values are the component-type objects that represent the databases to synchronize (*ServerDataBaseManager* and *LocalDataBaseManager*). The sequence of operations ϕ_{15} comprises an *applySynchronization* operation as follows.

Sequence of Operations ϕ_{15} for achieving ModelVersion15
applySynchronization(ServerMiddlewareLayer, MobileMiddlewareLayer, ServerDataBaseManager, LocalDataBaseManager)

An unachieved quality requirement is *Performance_QR3*. The designer decides that the way of achieving it is allowing the mobile devices to delegate the execution of CPU and memory consuming tasks on the company server. *OrderValidator* (*logical mobile component*) is identified as a component that develops processing intensive activities, which it is hosted in *SalesForceMobileClient*, more specifically in the *ServerApplicationLayer*. Even though *OrderValidator* runs on *SalesForceMobileClient* another possible physical location where the *OrderValidator* may run should be provided. By executing the *applyLogicalMobility* operation a new possible executing location is assigned, thus obtaining *ModelVersion16* Fig. 13 shows the *ModelVersion16*, wich employs a deployment view (Fig. 13 - a) and a C&C view (Fig. 13 - b) to illustrate the architectural model. It explains the logical connections among logical components situated at adjacent layers.

Sequence of Operations ϕ_{16} for achieving ModelVersion16
applyLogicalMobility(OrderValidator, ServerApplicationLayer, 1)

At this point in the design process, the architect should analyse and evaluate if all the main requirements of the architecture have been considered.

4 TracED

TracED is a research prototype that implements the proposed model to capture and trace software architectural designs. It has been developed using Java language, MySQL database, and Hibernate framework. The major tools are *Domain Editor* and *Versions Manager* (Roldán et al., 2008). Both tools were developed based on the object-oriented models proposed in previous sections.

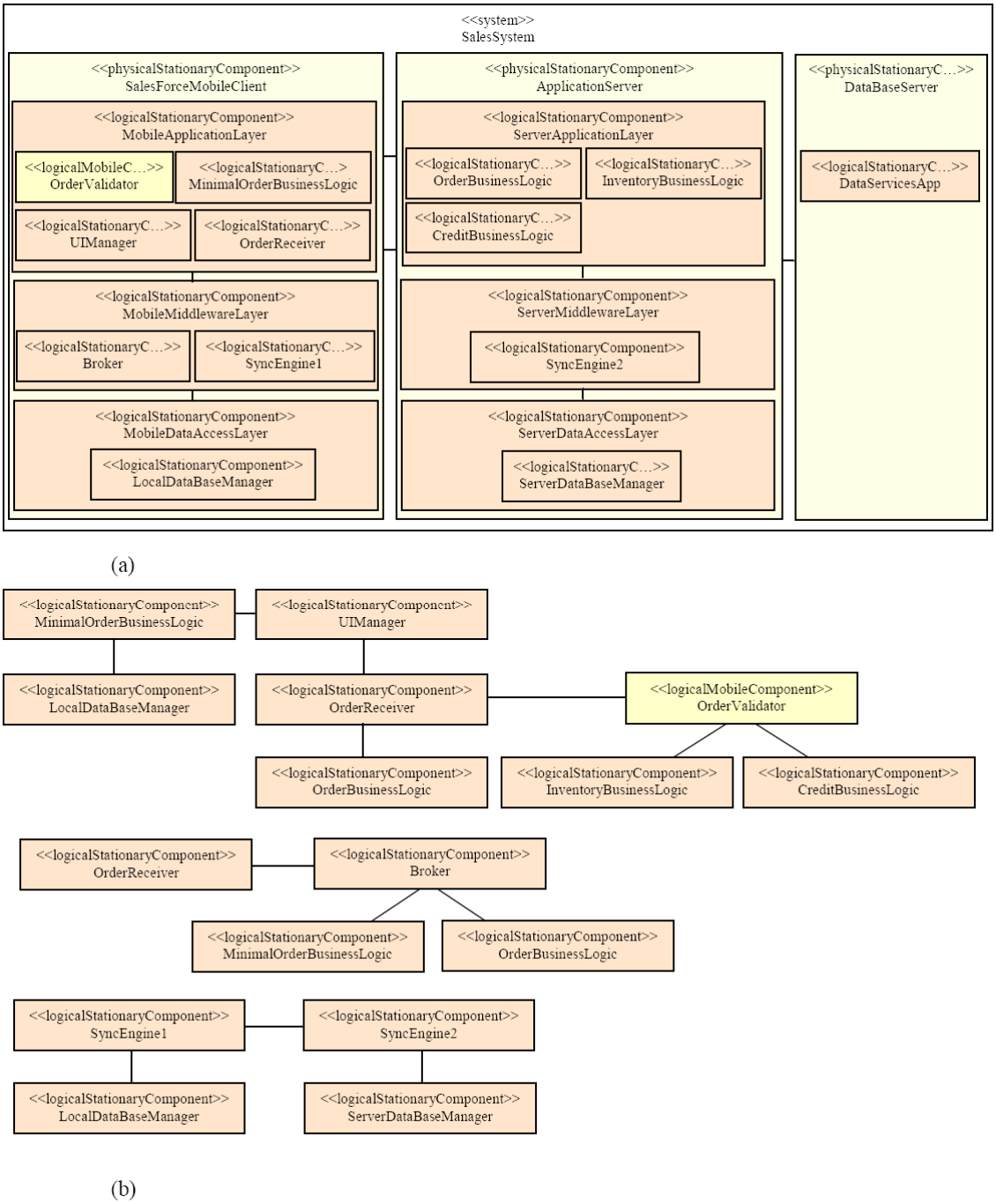

Fig. 13. a) Deployment view of *ModelVersion16*. b) C&C view of *ModelVersion16*.

The *Domain Editor* enables the definition of a design domain. A partial view of a software architecture domain for mobile systems defined by using TracED, is visualized in Fig. 14. Modelling concepts are organized hierarchically in a tree structure (upper-left corner of Fig. 15). Each concept can 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, and *Connector* generalizes *PhysicalConnector* and *LogicalConnector* (Fig. 14). Also,

Component is specialized in a hierarchy of special components, defining four concrete modelling concepts: *LogicalMobileComponent*, *LogicalStationaryComponent*, *PhysicalMobileComponent* and *PhysicalStationaryComponent*. Domain Editor allows the user to set binary relationships between modelling concepts, which are instances of *DomainRelationship* (Fig. 1).

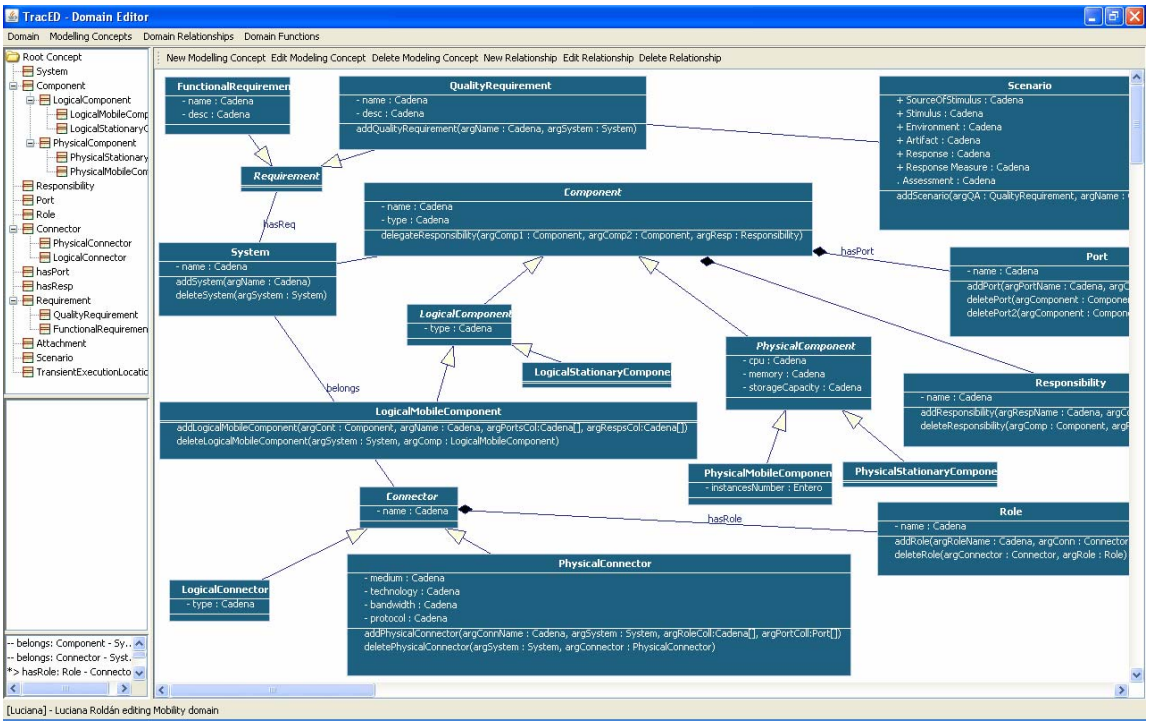

Fig. 14. Partial view of Domain Model for Mobile Software architectures in TracED

Fig. 15 shows the specification window of the *LogicalMobileComponent.* In *Properties* tab, a concept description can be assigned, and properties (like *name* and *type*) can be created and modified. In Fig 15, *Operations* tab is active, where a set of operations applicable to the current modelling concept can be specified. In the case of *LogicalMobileComponent*, the architect has defined *addLogicalMobileComponent* and *deleteLogicalMobileComponent* operations. The definition of a new operation means the instantiation of the *Operations Model* (Fig. 4). As *LogicalMobileComponent* has been defined as a subconcept of *Component*, it inherits the operations defined by this abstract concept, such as *delegateResponsibility.*

The window in Fig. 16 shows the definition of *addLogicalMobileComponent* that is similar to the functional specification presented in Fig. 5. To define it, the architect selects some of the available operations from the combo box situated on the right. The *VariableAssignment* between an input argument value or a previous result and the argument of another operation sub-command is carried out by matching them one to one (*Match Arguments* button). In this way, *VariableAssignment* instances are created, which bind input argument values or previous results and arguments of other sub-commands. The binding of arguments (variables) and their values (which are unknown at the moment of the specification) are set beforehand for the execution of the *macrocommand*. Also, *addLogicalMobileComponent* has a *loop* command as part of its body, which is defined over a ports collection. It allows assigning the ports to the new component one by one by using the *addPort* operation (see the window at the bottom of Fig. 16).

The other main component of TracED, *Versions Manager*, enables the execution of design projects. When a new design project is created, an existent domain is selected for it. A project is carried out working with the

Version Manager window, by executing the available domain operations, which include new design objects as instances of the modelling concepts defined in the selected design domain.

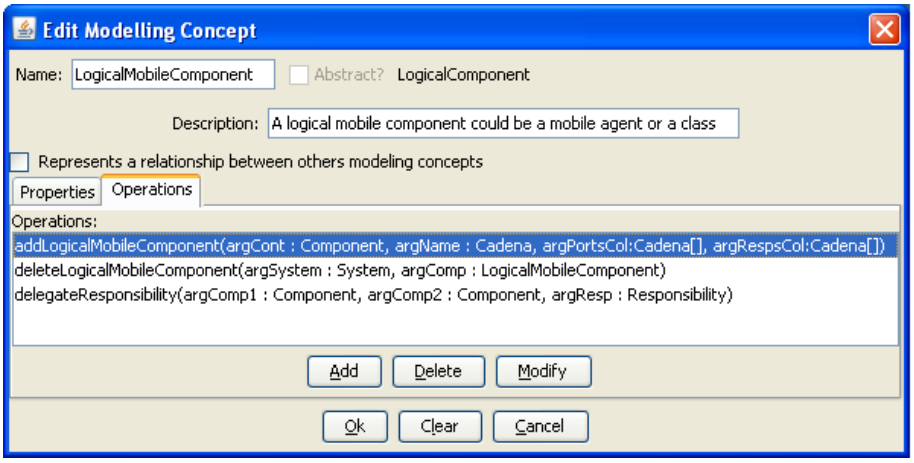

Fig. 15. Specification of Logical Mobile Component Modelling Concept

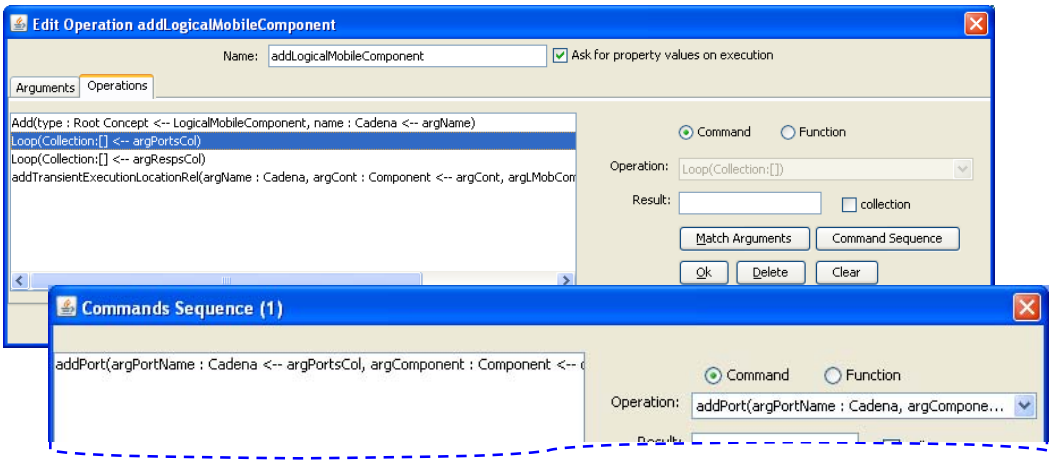

Fig. 16. Specification of the *addPhysicalMobileComponent* MacroCommand

To show how TracED is used to develop and capture a design process, part of the presented case study is carried out. The new project is called *SalesForceManagementProject*. An instance of *DesignProject* (Fig. 1) is created and associated with the domain for mobile software architecture, which it was described in sections 2.1 and 2.2 and was specified in TracED by using Domain Editor. By doing that, the initial model version (*Root Model Version*) is generated. *Root Model Version* is the root of the tree structure of the version management scheme and it does not have object versions and cannot be edited. Fig. 17 shows the Version Manager window, with the first model version of *SalesForceManagementProject* project, named *SalesSystem*, which was added by an *addSystem* execution (sequence of operations ϕ_1, as it was described in the case study of the previous section). 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 model version that arises from applying a sequence of operations ϕ_2 is shown as in the snapshot in Fig. 17, which includes the functional and quality requirements for *SalesSystem*.

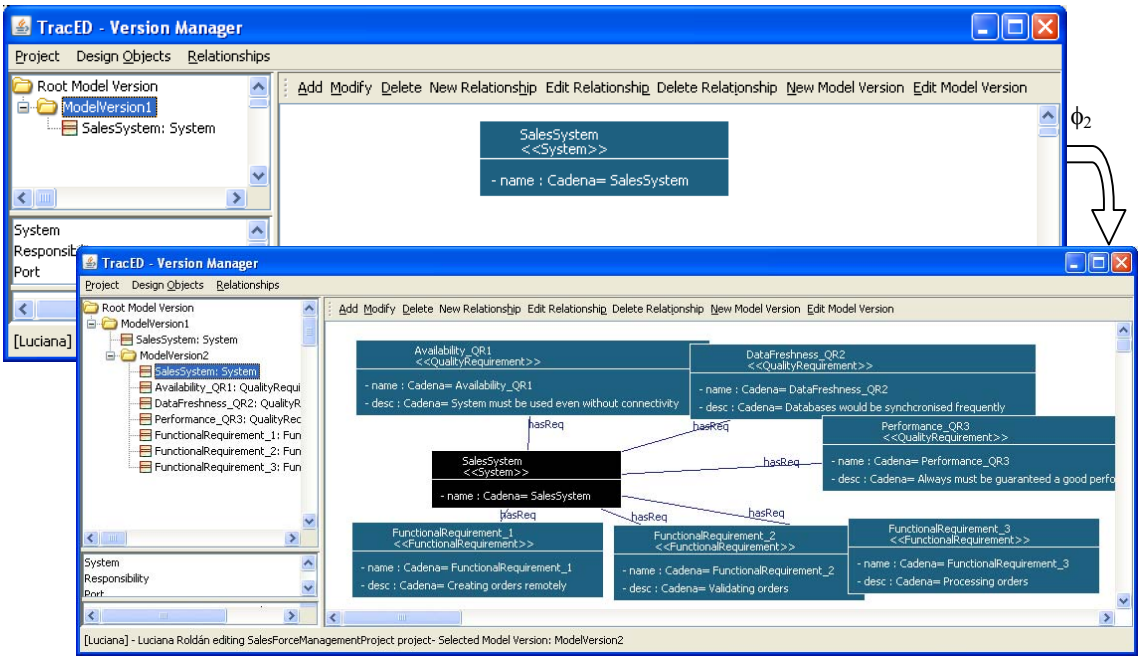

Fig. 17. First two model versions of *SalesForceManagementProject*

Following in this way, the rest of the case study is developed with the support provided by TracED. This prototype allows users to systematize the capture of each new model version, the object versions that belong to them, and the architectural operations that originated them.

Moreover, TracED allows the architect to recover the history of a given model version. He/she can query, for example, which is the predecessor model version of a given model version, and all its object versions. By selecting a model version from the model version navigator panel, it is possible to see what happened over time. Fig. 18 shows fragments of the History Window, which informs all operations that have been applied from the root (initial model version) to a selected model version. In this window, it is also possible to see detailed information about each applied operation. For instance, the moment when an operation was applied, who the actor involved was and the names assigned to new object versions (successor object versions). In this case, the architect queries about how *ModelVersion15* was obtained. At the bottom of Fig. 18, it is informed that an *addSystem* operation has been applied on *Root Model Version* to generate *ModelVersion1*. At the top of *ModelVersion1*, the sequence of operations that generates *ModelVersion2* is depicted. In a similar way, the window informs all the model versions generated to reach the requested one. Thus, moving the focus to the top of Fig. 18, it is observed that the operation executed on *ModelVersion14* was *applySynchronization*, which gave rise to *ModelVersion15*.

Due to its prototypical status, TracED currently has some limitations. On the one hand, TracED just provides some predefined queries for asking the model about the history of a design project, like the one presented in Fig. 18. However, the incorporation of new features for creating and executing queries is easy, as the information on which the queries find the answers has already been captured by the model and no additional extension is needed in that direction. On the other hand, TracED lacks of appropriated views to present an architectural model through different view points of view types (IEEE, 2000; Clements et al., 2002). Further work is necessary to achieve this goal. Despite its limitations, TracED has allowed us to verify the viability of the proposed models.

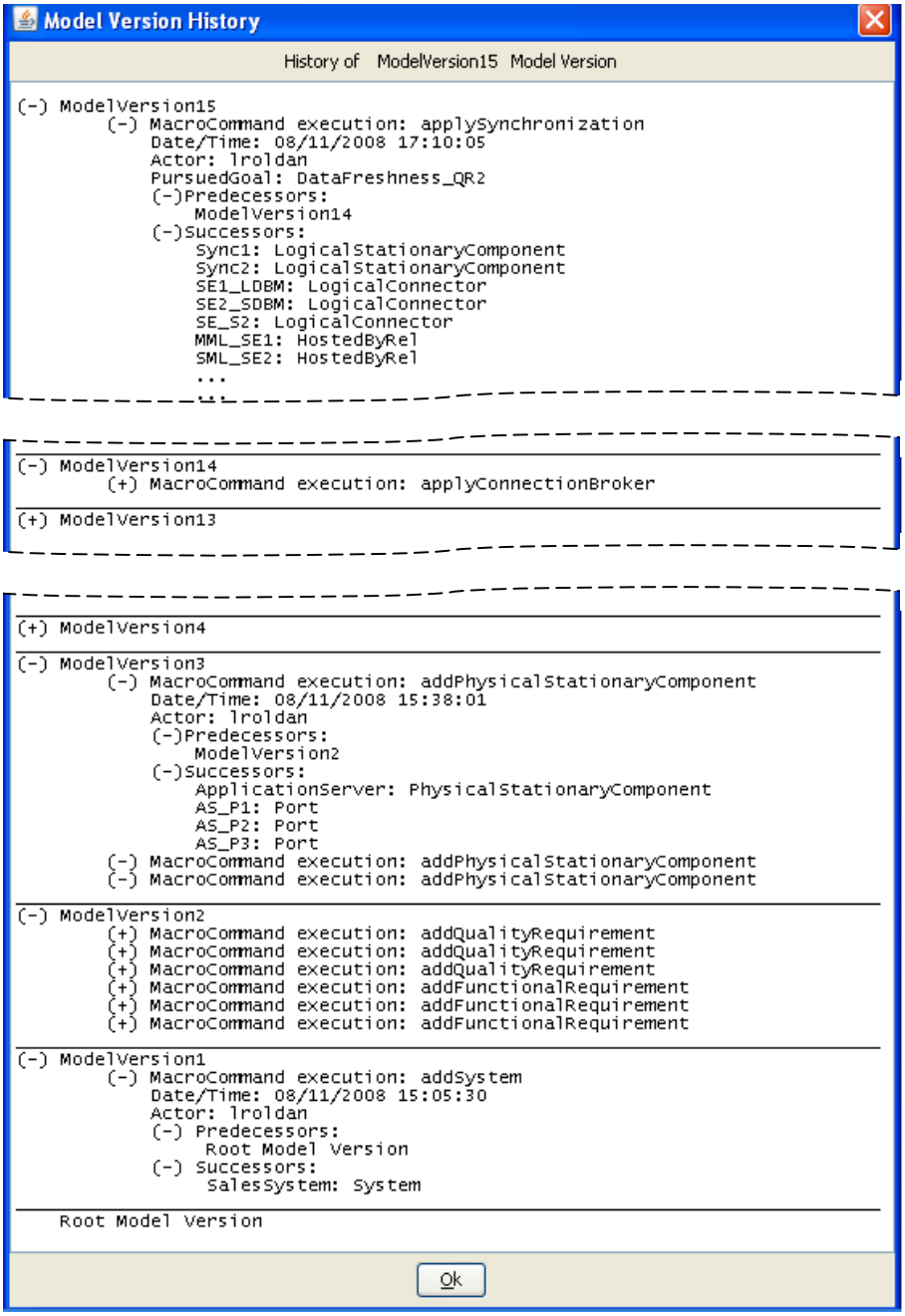

Fig. 18. A history window of *ModelVersion15*.

To improve the usability of TracED, it should work in an integrated way with CASE tools that support other design activities. In this way, TracED would perform the capture of all applied operations, by working in background mode, without designer noticing it. By using the features of the Operations Model (Fig. 4), new and higher level of abstraction operations could be specified for the CASE tool, which will be available for being included in the CASE tool menus.

5 Related Works

Recently, there has been proposed several approaches for representing architectural design decisions. They aim to assist software architects in their decision-making activities by capturing and characterizing architectural knowledge. Most of these tools are based on conceptual o semi-formal models, which provides a characterization and interpretation of SADP, and what their authors considers that is important to capture.

Tyree and Akerman (2005) have proposed a template of attributes to represent architectural design decisions, which extends the documentation of design decisions described in Clements et al. (2002). Such an approach allows the designers documenting some critical evolutions of SADP. The approach discussed in Capilla et al. (2007) is similar, but instead of providing a complete list of attributes to describe a design decision, they propose the use of mandatory and optional attributes that can be tailored according to different needs for making more agile the efforts of capturing a design decision. In addition, they include specific attributes and relationships aimed to support the evolution of design decisions. Archium is a tool that models design decisions and their relationships with resulting components (Jansen et al., 2007). It is based on a conceptual model for representing architectural design decisions and their context, which allows keeping the evolution of an architecture design by keeping architectural deltas (changes). The perspective employed in this approach is different from ours since the design decisions are not explicitly captured; they remain as tacit knowledge. Hence, the tacit knowledge embedded in the captured design is used to trace back from the changes to the decisions they originated from (Jansen et al. 2008). Contrary to this perspective, our approach captures the design decisions by materialising them in a sequence of operations that is applied on the current model version. In this way, the design decisions are captured "during" the SADP and not "after". The Architecture Rationale and Elements Linkage (AREL) approach, models architecture design as causal relationships between design concerns, decisions and outcomes (Tang et al., 2007). Ali Babar and Gorton (2007) propose another framework for the capture and recover of architecture knowledge called Process-centric Architecture Knowledge Management Environment (PAKME). PAKME uses a data model for characterising architectural constructs (such as design decisions, alternatives, rationale, and quality attributes), their attributes and relationships. Each design decision is captured as a case along with rationale and contextual information using a template.

In spite of the fact that most of them support the notion of design decisions, none of the aforementioned approaches represent design decisions as concrete executions of design operations as our approach does. In addition the proposed versioning administration model 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 among architectural elements and architectural decisions. Moreover, none of the existent proposals for representing architectural design decisions address the particularities of software architecture design of mobile systems. In general, they are intended for generic domains and are hard to extend to more specific design domains. Although some contributions (Ali et al., 2008; Lopes et al., 2002; Medvidovic and Mikic-Rakic, 2001; Schäfer, 2006) propose the fundamental architectural building blocks and methods for modelling software architectures for dynamic mobile environments, they hardly support the designers in developing, documenting and evolving mobile software architectures.

6 Conclusions

Software architectures for mobile systems demand tools for managing the different versions generated during their design process. Many ADLs support the static description of a system, but most of them provide no facilities for specifying architectural changes. Mobility-specific ADLs are not the exception. These factors lead to problems like knowledge vaporization, given that just the final artefact is kept and the design decisions made by the involved architects are lost. Consequently, there is an urgent need of tools able to capture and manage this process. In this contribution, we propose a model that employs an operational approach, which allows expressing and capturing each architectural decision as a design operation. Therefore, designs are captured by introducing a minimal impact on the design activities performed by architects. This feature

establishes a distinction from contributions that propose documenting the architectural design process after completing (part of) it.

The proposed model supports mobility concerns and it is flexible enough to make it possible the definition of the fundamental architectural building blocks and particular operations suitable for the architect's necessities and preferences. This definition allows to represent the several methods for modelling software architectures in dynamic mobile environments and to capture processes that follow the defined methods. Furthermore, our proposed model allows to specify operations in a goal-oriented way, indicating which is the pursued goal by a given operation execution. In such a way, explicit associations between the applied operations, their results, and the achieved architectural requirements could be captured and recovered.

Acknowledgements

The authors wish to acknowledge the financial support received from CONICET, Universidad Tecnológica Nacional and Agencia Nacional de Promoción Científica y Tecnológica (25/O118, PAE – PICT 2007 - 02315, IP-PRH 2007).

References

Ali Babar, M., Gorton, I. (2007). A tool for managing software architecture knowledge. In: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent.

Ali, N., Millán, C., Ramos, I. (2006). Developing mobile ambients using an aspect-oriented software architectural model. Lecture Notes in Computer Science, Springer, Vol. 4276, pp. 1633-1649.

Ali, N., Ramos, I., Carsi, J. (2005). A conceptual model for distributed aspect-oriented software architectures. In: Proceedings of the International Conference on Information Technology: Coding and Computing, Vol. 2, pp. 422-427.

Ali, N., Solís, C., Ramos, I. (2008). Comparing architecture description languages for mobile software systems. In: Proceedings of the 1st international Workshop on Software Architectures and Mobility (Leipzig, Germany). SAM '08. ACM, New York, NY, pp. 33-38.

Bass, L., Clements, P., Kazman, R. (2003). Software architecture in practice, 2nd Edition. Addison-Wesley, Boston.

Bieszczad, A. , White, T. (2007). Code Mobility and Mobile Agents. In: Bellavista, P., Corradi, A. (Eds.), The Handbook of Mobile Middleware, Auerbach Publications, New York.

Burge, J., Carroll, J., McCall, R., Mistrík, I. (2008). Rationale-Based Software Engineering. Springer-Verlag.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996). Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons.

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

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R. (2002). Documenting Software Architectures: Views and Beyond. Pearson Education.

Field, Z., Dewar, R., Trinder, P., Du Bois, A. R. (2006). Two executable mobility design patterns: mfold and mmap. In: Proceedings of the 2006 Conference on Pattern Languages of Programs (Portland, Oregon, October 21 - 23, 2006). PLoP '06. ACM, New York, NY, pp. 1-11.

Fuggetta, A., Picco, G. P., Vigna, G. (1998). Understanding Code Mobility. IEEE Transactions on Software Engineering 24(5), 342-361.

Garlan, D., Monroe, R.T., Wile, D. (2000). Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (Eds.), Foundations of Component-Based Systems. Cambridge University Press, pp. 47-68.

Georgas, J., van der Hoek, A., Taylor, R. (2005). Architectural runtime configuration management in support of dependable self-adaptive software. In: Proceedings of the 2005 workshop on Architecting dependable systems, pp. 1-6.

Gonnet, S., Leone, H., Henning, G. (2007). A model for capturing and representing the engineering process. Expert Systems with Applications, 33(1), 881-902.

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

ISO (2001). ISO/IEC 9126-1, Software Engineering – Product Quality, Part1: Quality Model.

ISO (2008). ISO/IEC JTC1/SC7, Recommended Systems and software engineering. Architectural description, ISO/IEC WD2 42010.

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

Jansen, A., van der Ven, J., Avgeriou, P. Hammer D. (2007). Tool support for Architectural Decisions. In: Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), pp. 44-53

Kruchten, P., Lago, P., van Vliet, H. (2006). Building up and reasoning about architectural knowledge. In: Proceedings of the Second International Conference on the Quality of Software Architectures.

Lima, E. F., Machado, P. D., Sampaio, F. R., Figueiredo, J. C. (2004). An approach to modelling and applying mobile agent design patterns. SIGSOFT Software Engineering Notes 29 (3), 1-8.

Lopes, A., Fiadeiro, J.L., Wermelinger, M. (2002). Architectural Primitives for Distribution and Mobility. In: Proceedings of 10th Symposium on Foundations of Software Engineering. ACM Press, pp. 41-50.

Medvidovic, N., Mikic-Rakic, M. (2001). Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility. In: Proceedings of the Software Engineering and Mobility Workshop (Toronto, Canada, May).

Medvidovic, N., Mikic-Rakic, M., Mehta, N. R., Malek, S. (2003). Software Architectural Support for Handheld Computing. IEEE Computer 36 (9), 66-73.

Mikic-Rakic, M., Malek, S., and Medvidovic, N. (2008). Architecture-driven software mobility in support of QoS requirements. In: Proceedings of the 1st international Workshop on Software Architectures and Mobility (Leipzig, Germany), SAM '08. ACM, New York, NY, pp. 3-8.

Risi, W. A., Rossi, G. (2004). An architectural pattern catalogue for mobile web information systems. International Journal of Mobile Communications 2(3), 235-247.

Roldán, M.L., Gonnet, S., Leone, H. (2006). A model for capturing and tracing architectural designs. In: IFIP IWASE 2006. Springer. Vol. 219, pp. 16-31.

Roldán, M.L., Gonnet, S., Leone, H. (2008). A Tool for Capturing and Tracing the Software Architecture Design Process. In: Proceedings of the XXXIV Conferencia Latinoamericana de Informática (CLEI 2008), Santa Fe, Argentina, ISBN 978-950-9770-02-7, pp. 380-389.

Roman, G., Picco, G., Murphy, A. (2000). Software Engineering for Mobility: A Roadmap. In: Proceedings of the International Conference on Software Engineering, Future of SE Track (Limerick, Ireland), pp. 241-258.

Roth, J. (2002). Patterns of Mobile Interaction. Personal Ubiquitous Computing 6 (4), 282-289.

Schäfer, C. (2006). Modeling and Analyzing Mobile Software Architectures. Lecture Notes in Computer Science, Springer, Vol. 4344, pp. 175 – 188.

Tang, A., Jin, Y., Han, J. (2007). A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software, 80, 918–934.

Tyree, J., Akerman, A. (2005). Architecture decisions: demystifying architecture. IEEE Software 22 (2), 19-27.

Westfechtel, B., Conradi, R. (2003). Software Architecture and Software Configuration Management. Lecture Notes in Computer Science, Springer, Vol. 2649, pp. 24-39.

Westfechtel, B. (1999). Models and tools for managing development processes. Lecture Notes in Computer Science, Springer, Vol. 1646.