2

An Architecture-Centric Approach for Information System Architecture Modeling, Enactement and Evolution

Hervé Verjus, Sorana Cîmpan and Ilham Alloui
University of Savoie – LISTIC Lab
France

1. Introduction

Information Systems are more and more complex and distributed. As market is continuously changing, information systems have also to change in order to support new business opportunities, customers' satisfaction, partners' interoperability as well as new exchanges, technological mutations and organisational transformations. Enterprise agility and adaptability leads to a new challenge: flexibility and adaptability of its information system. Most information systems are nowadays software-intensive systems: they integrate heterogeneous, distributed software components, large-scale software applications, legacy systems and COTS. In this context, designing, building, maintaining evolvable and adaptable information systems is an important issue for which few rigorous approaches exist. In particular information system architecture (Zachman, 1997) is an important topic as it considers information system as interacting components, assembled for reaching enterprise business goals according to defined strategies and rules. Thus, information system architecture supports business processes, collaboration among actors and among organizational units, promotes inter-enterprise interoperability (Vernadat, 2006) and has to evolve as business and enterprise strategy evolve too (Kardasis & Loucopoulos, 1998; Nurcan & Schmidt, 2009).

During the past twenty years, several works around system architecture have been proposed: they mainly focus on software system architecture (Bass *et al.*, 2003), enterprise and business architecture (Barrios & Nurcan, 2004; Touzi *et al.*, 2009; Nurcan & Schmidt, 2009). All of them mainly propose abstractions and models to describe system architecture. Research on software architecture (Perry & Wolf, 1992; Bass *et al.*, 2003) proposes engineering methods, formalisms and tools focusing on software architecture description, analysis and enactment. In that perspective, Architecture Description Languages (ADLs) are means for describing software architecture (Medvidovic & Taylor, 2000) and may also be used to describe software-intensive information system architecture. Such ADLs cope with software system static aspects at a high level of abstraction. Some of them deal with behavioral features and properties (Medvidovic & Taylor, 2000). Very few of the proposed approaches are satisfactory enough to deal with software-intensive system architecture dynamic evolution; *i.e.*, a software-intensive system architecture being able to evolve during enactment.

As an illustrative example of such a dynamically evolving software-intensive information system, consider the following supply chain information system that entails a manufacturing enterprise, its customers and suppliers. The supply chain information system is a software-intensive system comprising several software components. It is governed by an EAI (Enterprise Application Integration) software solution that itself comprises an ERP system. The ERP system includes components dedicated to handling respectively stocks, invoices, orders and quotations. These software elements form the information system architecture. In a classical scenario, a customer may ask for a quotation and then make an order. The order may or may not be satisfied depending on the stock of the ordered product. We may imagine several alternatives. The first one assumes that the information system is rigid (*i.e.*, it cannot dynamically evolve or adapt): if the current product stock is not big enough to satisfy the client's order, a restocking procedure consists in contacting a supplier in order to satisfy the order. We assume that the supplier is always able to satisfy a restocking demand. Let us now imagine that the restocking phase is quite undefined (has not been defined in advance – *i.e.*, at design time) and that it can be dynamically adapted according to business considerations, market prices, suppliers' availability and business relationships. Then, the supporting supply chain information system architecture would have to be dynamically and on-the-fly adapted according to the dynamic business context. Such dynamicity during system enactment is an important issue for which an architecture-centric development approach is suitable.

This represents an important step forward in software-intensive information system engineering domain, as software intensive information systems often lack support for dynamic evolution. When existing, such support doesn't ensure the consistency between design decisions and the running system. Thus, generally first the system model is evolved, and then the implementation, without necessarily mantaining the consistency between the two system representations. This leads to undesired situations where the actual system is not the one intended, or thought by the decision makers.

This chapter presents an architecture-centric development approach that addresses the above mentioned issues, namely dynamic evolution while preserving the consistency between the system design and implementation. Our approach entails architectural description formalisms and corresponding engineering tools to describe, analyze and enact dynamically evolvable software-intensive information systems.

It presents the overall development approach, briefly introducing the different models and meta-models involved as well as the different processes that can be derived from the approach (see section 2). Although the approach supports the entire development cycle, the chapter focuses on the way dynamic evolution is handled. More precisely it shows how information systems, described using suitable architecture-related languages (see section 3), can be architectured so that their dynamic evolution can be handled. Thus section 5 and 6 present the proposed mechanismes for handling respectively dynamic planned and unplanned evolutions of the information system architecture. These mechanisms are presented using evolution scenarios related to a case study which is briefly introduced in section 4. Section 7 presents related work. We end the chapter with concluding remarks in section 8.

2. On architecture-centric development

Considerable efforts have been made in the software architecture field (Medvidovic & Taylor, 2000; Bass *et al.*, 2003) (mainly software architecture modeling, architectural property

expression and checking) that place the architecture in the heart of a software intensive system life cycle. "Software architecture is being viewed as a key concept in realizing an organization's technical and business goals" (Carrière *et al.*, 1999). Software architectures shift the focus of developers from implementation to coarser-grained architectural elements and their overall interconnection structure (Medvidovic & Taylor, 2000). *In architecture-centric development approaches, the architecture of the system under construction is considered at different abstraction levels. Starting with a rather coarse grain representation of the system, the process stepwise refines this representation producing more detailed representations. At each phase, architectural properties can be defined and analyzed.* Architecture Description Languages (ADLs) have been proposed as well as architecture-centric development environments, toolkits (graphical modelers, compilers, analysis/verification tools, *etc.*) (Schmerl *et al.*, 2004; ArchStudio) which support software architects' and engineers' activities.

We consider the architecture-centric information system development as a model-driven engineering process (Favre *et al.*, 2006). Every process is centered on design models of systems to develop. Models are used for several purposes: to understand specific aspects of a system, to predict the qualities of a system, to reason on the impact of change on a system and to communicate with different system stakeholders (developers, commercials, clients, end-users, *etc.*). Among the objectives of such approaches is their ability to provide (at least partially) enough details to generate an implementation of the information system software components and their interconnections. Thus, the generated code is, itself, the expression of a model. In architecture-centric development approaches (Kyaruzi & van Katwijk, 2000) models represent mainly software architectures, but can also represent some expected properties or transformations that can be made on such architectures.

The architecture may be defined at several levels of abstraction. The transition from one level to another is done through a refinement process along which further details are added to the architecture description until reaching a desired concrete (implementation) level. The resulting concrete architecture can either be directly executed if the employed ADL has its own virtual machine or it can be used to generate an executable description for another target execution environment (*e.g.*, Java, C++, *etc.*).

As the system software architecture captures early design decisions that have a significant impact on the quality of the resulting system, it is important if not essential to check those decisions as early as possible. Software architecture analysis is an ineluctable activity within the development process. It focuses on structural and/or behavioral properties one can expect from both system functional and non-functional behaviors (*e.g.* are architectural elements always connected? Is the system behavior robust? *etc.*). Moreover, evolving a system must be accompanied by checking whether its correctness is still ensured or not after the changes. In software engineering processes, checking the correctness of a system relies on analyzing expected properties at either/both design time or/and runtime. This requires the availability of software tools/support for both checking if the desired properties are satisfied and detecting those that have been violated with the possibility of reconciling them. Ideally an approach that aims at considering the evolution along the whole lifecycle should provide mechanisms for analysis, detection of property violation and its reparation.

The introduction of architecture-centric approaches had as prior intent an improvement of the software development process, allowing people to gain intellectual control over systems

ever more complex and thus providing solutions for a major software engineering concern. Software-intensive system evolution is another major concern in software engineering (Andrade & Fiadeiro, 2003, Mens *et al.*, 2003), as human-centric activities are more and more supported by software applications that have to evolve according to changing requirements, technologies, business, *etc.* Software-intensive systems should be able to adapt according to those changes (Belady & Lehman, 1985). As changes may impact the information system architecture, the way of evolving the architecture is part of the information system evolution problem. Moreover, the problem of handling the evolution of a software-intensive information system taking into account its architecture is closely related to the problem of keeping the consistency between two layers: the software system *concrete* (source code, implementation) architecture, and, the information system *conceptual* (abstract, design) architecture as well as continuous switching between these layers (Perry & Wolf, 1992).

We distinguish four types of evolution (Cîmpan & Verjus, 2005) according to two criteria: (i) the architecture evolution is carried out *statically* (*i.e.*, while some of the information system executing software components are stopped) or *dynamically* (*i.e.*, while the system is being continuously executing), (ii) has the evolution been planned (*i.e.*, at design time) or not (*i.e.*, unplanned, may occur at any time during the information system enactment). A static evolution, be it planned or not, is de facto supported by all architecture-centric approaches. It is more or less supported by analysis tools to check the system correctness after the change implementation. A dynamic evolution is more difficult to handle, in particular if it has not been planned at the design time. Indeed this requires: (i) mechanisms to provide change specifications without stopping information system executing software components, (ii) performing the changes while preserving the information system correctness and (iii) preserving the consistency between the system implementation and its conceptual architecture.

As depicted by Figure 1, our architecture-centric approach supports information system development processes based on software architecture models. Different models and meta-models are proposed, as well as relations among them. Part of them are platform independent (PIM, represented in the upper part of the figure), while others are platform specific (PSM, represented in the lower part of the figure). The approach is suitable to description languages which have a layered construction. They entail a core, generic (and in our case enactable) description language as well as extension mechanisms enabling the description of domain specific languages. The figure gives a complete view of the different models and meta-models, yet not all of them are mandatorily used. Thus, different processes can be drawn from this picture. A very simple one would for instance consist in representing architecture in the core language, and use the associated virtual machine to enact it. A possible enhancement of this process would consist in defining properties the architecture should obey and check if it indeed does. This implies the additional use of an architecture analysis language to define such properties as well as the use of associated tools to verify whether the properties hold for the given architecture. If the enterprise environment imposes the use of particular platform, it is also possible that rather then using the virtual machine (VM), code is generated in a target language, using specific transformation rules. In this chapter, we do not address exhaustively how such processes are defined and carried out. Rather we focus on how evolution is supported at system enactment time.

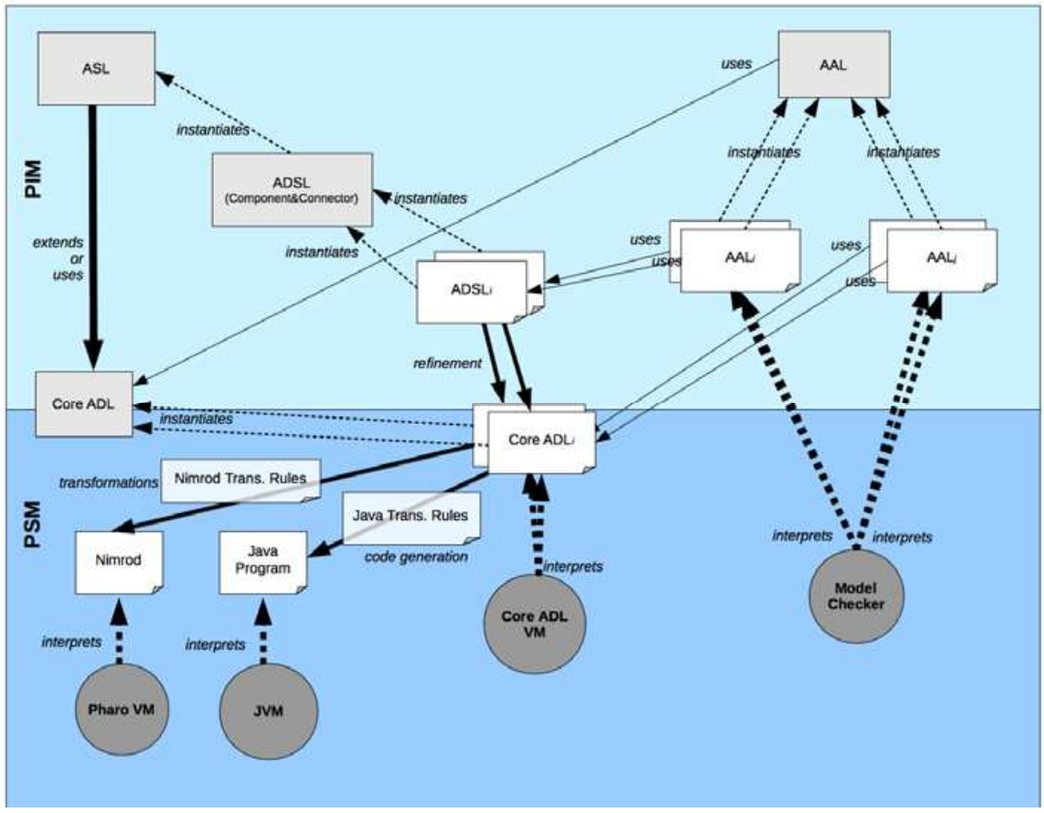

Fig. 1. Architecture centric development approach and processes

To illustrate how our architecture-centric information system development approach supports dynamic evolution of software–intensive information system architecture, we use as modeling language, an ADL allowing us to cope with unpredictable situations and dynamic changes: ArchWare ADL (Oquendo *et al.*, 2002; Oquendo 2004). This language is part of the ArchWare (ArchWare 2001) language family and can be used either as a specification language only or both as a specification and implementation language. In the first case, a target source code can be generated from specifications using mappings and adding implementation details related to the target environment. In the second case, an implementation is a specification, detailed enough, to be interpreted by the ArchWare ADL virtual machine. In both cases, user-defined expected architectural properties can be analyzed both at design time and runtime.

3. ArchWare architecture description languages, foundations and design

The ArchWare project (ArchWare, 2001) proposes an architecture-centric software engineering environment for the development of evolving systems (Oquendo *et al.*, 2004). The environment provides languages and tools to describe architectures and their properties, refine them as well as enact them using a virtual machine.

This section introduces part of the ArchWare language family – related to the description of architectures and their properties. The ArchWare language familly perfectly fits the above presented approach (cf. Figure 1). The description language family has a layered structure,

with a minimal core formal language and an extension mechanism that allows the users to construct more specific description languages.

The core formal language – ArchWare π-ADL. The ArchWare project proposes a meta-model, defined by an abstract syntax and formal semantic (Oquendo *et al.*, 2002). Several concrete syntaxes are proposed (Verjus & Oquendo, 2003; Alloui & Oquendo, 2003), ArchWare π-ADL (Cîmpan *et al.*, 2002; Morrison *et al.*, 2004) being the textual one. The core language is a well-formed extension of the high-order typed π-calculus (Milner, 1999) that defines a calculus of communicating and mobile architectural elements. These architectural elements are defined in terms of behaviors. A behavior expresses in a scheduled way both an architectural element internal computation and its interactions (sending and receiving messages via connections that link it to other architectural elements). These actions (concerning communication as well as internal computing) are scheduled using π-calculus based operators to express sequence, choice, composition, replication and matching. Composite architectural elements are defined by composing behaviors, communicating through connections. An architecture is itself an architectural element. Moreover, π-ADL provides a mechanism to reuse parameterised behavior definitions which can be embedded in abstractions. Such abstractions are instantiated as behaviors by application. As the core language is Turing complete, a virtual machine (Morissson *et al.* 2004) enables enactment of architectures that are defined using this language.

The extension mechanism – is represented in Figure 1 by ASL. The extension mechanism is based on architectural styles, representing a family of architectures sharing common characteristics and obeying a given set of constraints. ArchWare ASL (Architectural Sytle Language) is a meta-model allowing the definition of styles, and hence of domain specific languages (Leymonerie, 2004). More precisely, architectural element types can be introduced by a style, forming the style vocabulary. When a style is defined using ASL, it is possible to associate a new syntax; thus the style provides a domain-specific architecture description language. Architectural styles and associated languages can then be constructed using a meta-level tower. If using the n^{th} layer of the language family a style is defined, its associated syntax constitutes a *n+1* layer. By construction, an architecture defined using the n^{th} layer of the language family, has its corresponding description in the *n-1* layer.

The component-connector layer – corresponds to a particular domain language, dedicated to the definition of component-connector models of software architectures. In Figure 1, ADSL is the generic term for such domain specific language. Using the extension mechanism (ASL) a level *1* language has been constructed starting from the core language (level *0*). This language, named ArchWare C&C-ADL is associated to an architectural style in which architectural elements are either components or connectors (Cîmpan *et al.*, 2005; Leymonerie, 2004). Components and connectors are first class citizens and can be atomic or composed by other components and connectors. An architectural element interface, represented by its connections, is structured in ports. Each port is thus composed by a set of connections, and has an associated protocol (corresponding to a behavior projection of the element to which it pertains). Atomic as well as composite architectural elements may entail attributes used in their parameterisation. A composite element behavior results from the parallel composition of its composing element behaviors. The composite element has its own ports, which ports of composing elements are attached to.

The architecture analysis language – corresponds to AAL in Figure 1. Architectural properties can be expressed in the ArchWare framework by using a dedicated language: ArchWare Architecture Analysis Language (AAL) (Alloui *et al.*, 2003; Mateescu & Oquendo, 2006). AAL is a formal language based on first order predicate logic and μ-calculus (Bradfield and Stirling, 2001). Predicate logic allows users to express structural aspects while μ-calculus provides the expressive power needed for the representation of dynamic aspects of an evolving system. A property is expressed in AAL using a predicate formula (concerns the architecture structure, *e.g.*, the existence of a connection among two elements), an action formula (concerns the architectural element behavior, *e.g.*, a component must have a recursive behavior), a regular formula (regular expression over actions, *e.g.*, after a certain number of actions of a given type, an architectural element will perform a given action; the goal of such regular expressions is not to increase the language expressive power, but rather to enhance the property readability) or a state formula (state pattern, *e.g.*, a given behavior leads to an expected state, such as true or false). AAL toolset entails theorem provers (Azaiez & Oquendo, 2005) and model checkers (Bergamini *et al.*, 2004). User-defined properties are linked to the description of architectural elements they are about. Their evaluation/analysis may be carried out at both design time and runtime.

The architecture execution languages – correspond to some concrete runtime architecture-centric languages. Specific defined transformation rules are applied to architectural models to generate more concrete and/or detailed architectural models. In the proposed approach (see Figure 1) either Core ArchWare detailed architectural models are generated for being executed by the ArchWare Virtual Machine (Morrison *et al.*, 2004), or Java code is produced to be executed by a Java Virtual Machine (Alloui *et al.*, 2003b), or a Nimrod architectural model (Verjus 2007) is produced to be interpreted by Nimrod (implemented in Pharo, www.pharo-project.org).

4. Case study introduction and evolution scenarios

Given the four identified kinds of evolution (cf. section 2) in this chapter we focus on the dynamic evolution, be it planned or not. To illustrate the mechanisms allowing such evolutions, we consider a supply chain architecture that entails a manufacturing enterprise, its customers (clients) and suppliers. The supply chain architecture is governed by an EAI (Enterprise Application Integration) software solution that itself includes an ERP system. The ERP system includes components dedicated to handling respectively stocks, invoices, orders and quotations.

Static evolutions are not considered in this chapter. Such evolutions require the running system to be stopped before any modification. Then, it is up to the architect to modifiy statically the system architecture and to launch the system again. Research approaches dealing with static evolution are manifold and the reader may look closer at works presented in section 7.

Initial scenario. Whenever a client places an order to the EAI, s/he first asks for a quotation. In order to simplify the scenario, the decision about order commitment by evaluating the quotation is not covered here. The ordering system (one may frequently meet the term *component* in most ADLs) takes the order and updates the stock according to the demanded product and quantity). The restocking system may ask for restocking if the current product

stock is not big enough to satisfy the client's order. A restocking procedure consists in contacting a supplier in order to satisfy the order. We first assume that the supplier is always able to satisfy a restocking demand.

Dynamic planned evolution. The architecture that supports planned dynamic evolution is a self-contained architecture that is able to evolve in response to external and anticipated events. The architecture is able to dynamically and automatically evolve (*i.e.*, its structure and behavior may evolve – for example in our scenario by adding clients or modifying the invoicing system) without stopping the system and with no user's interaction. This kind of evolution requires anticipation: the evolution strategy is defined and embedded in the architecture description, before its execution. In our scenarios, the architecture evolves dynamically in order to support new clients or to change the ERP invoicing system (see section 5).

Dynamic unplanned evolution. In some situations (most real life systems), the evolution cannot be anticipated and the architecture is not able to self-adapt. We emphasize scenarios (section 6) for which the architecture has to evolve dynamically (*i.e.*, on-the-fly evolution), without stopping the system execution to support unpredictable situations. We show how the architect improves the restocking system by adding dynamically new suppliers and modifying the restocking process. This evolution scenario shows thus how our proposition addresses challenging topics such the dynamic and unplanned modification of the architecture structure (introducing new suppliers) and the dynamic and unplanned modification of the architecture behavior (changing the restocking process).

These evolution scenarios help to demonstrate how our approach supports controlled and consistent aware architecture dynamic evolution. For both planned and unplanned situations, the architecture consistency is ensured using architectural formal property verification.

5. Dynamic planned evolution: mechanisms and illustration using the supply chain architecture

The layer ArchWare C&C allows to handle dynamic planned evolution. As already mentioned, the language allows the definition of software architectures in terms of compositions of interacting components and connectors. The language (Cîmpan *et al.*, 2005) improves previous propositions, such as Dynamic Wright (Allen *et al.*, 1998), Piccola (Nierstrasz & Achermann, 2000) and π-Space (Chaudet *et al.*, 2000).

Being based on a process algebra, the language enables a system behavior representation. To represent architectural dynamic changes the C&C language introduces specific actions, such as a dynamic element creation and reconfiguration. Moreover, every architectural entity is potentially dynamic, its definition is used at the dynamic creation of several instances. Thus such a definition corresponds to a meta entity, a matrix containing an entity definition as well as information allowing the creation, suppression (dynamic or not) and management of several occurrences.

Components can be either atomic, either composite, *i.e.*, a composition of components and connectors. Independently of their atomic or composite nature, architectural elements can dynamically evolve. Their evolution has nevertheless particularities.

The evolution of atomic (cf. section 5.1) and composite elements (cf. section 5.2) is illustrated using the supply chain case study, for which the architecture has been defined in terms of components and connectors using the C&C language.

The Supply Chain architecture is presented in Figure 2. Defined as a composite component, the supply chain architecture entails two atomic components, a supplier and a client, and a composite component representing an ERP. The connector between the client and the ERP is equally represented, the other ones are basic connectors, and not represented in the figure. The ERP composite component entails four components, to handle respectively quotations, orders, stock availability and invoices. The quotation system and the order system are directly connected to one of the composite ports, allowing direct interaction with components from outside the composite. The stock control and the invoice system are intern to the composite, and are connected to the order system.

One of the supply chain architecture ports is dedicated to its evolution. Clients communicate with the ERP essentially for exchanging information related to quotes (quote demands and propositions) and orders (orders and invoices). Ports are dedicated to this purpose on both communicating parts.

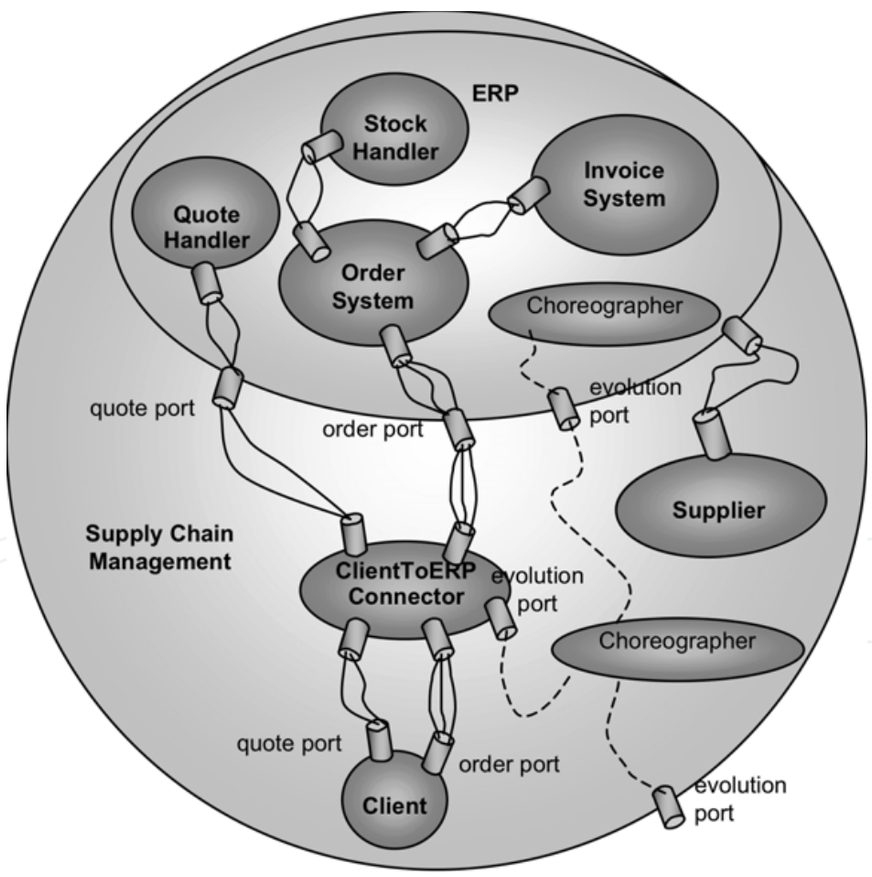

Fig. 2. The supply chain global architecture

The composite initialization and evolution are handled by its *choreographer*, as explained in section 5.2.

5.1 Evolution of atomic components and connectors

Atomic component and connectors definitions are structured in three parts, one for declaring attributes and meta ports, one to define the initial configuration (where instances of meta ports are created) and one representing the behavior. Component's behavior is named *computing*, while for connectors we use the term *routing*. The evolution of atomic components and connectors implies mainly changes in their interface, *i.e.*, addition or suppresion of ports. This has two implications on the behavior, who's representation does not change. The first implication is that part of it will be dedicated to handling the evolution while the rest of it, which we call nominal behavior, represents the main purpose of the element. The second implication is that the nominal behavior is generic, so that it can cope with the dynamic set of ports.

We will illustrate how dynamically evolving atomic architectural elements can be modeled by the example of the `ClientToERP` connector. The later has ports dedicated to the communication with clients and the ERP as well as an evolution port. As with all architectural elements described using ArchWare C&C-ADL, the declarations correspond to meta element declarations, meaning that several instances of the same meta element may co-exist at runtime. Thus, `clientQuotationP`, `erpQuotationP`, `clientOrderP`, `erpOrderP` as well as `newClientP` are meta ports. An instance of each is created in the configuration part. Additional instances may be created at runtime, as we will see. Meta elements provide an additional management level between types and instances, allowing to handle the dynamic evolution of architectures. In the initial configuration, an instance of each meta port is created (cf. Figure 3). Recursively, the connector has 3 choices: to transmit a demand/response for a product quotation, transmit a command, or handle an evolution request. The first two choices represent the nominal behavior. In the case of an evolution request, the connector creates two new instances of the `clientOrderP` and `clientQuotationP` ports, so that a new client can be connected to the ERP.

The nominal part of the behavior, which handles the quotation and the command transmissions, is generic, as it takes into account the fact that the number of clients, and hence the number of instances for `clientOrderP` and `clientQuotationP`, is unknown. Each meta entity (be it connection, port, component or connector) has a list containing its instances. The i^{th} instance of the meta entity is accessed using its name follwed by **`#i`**, while a random instance is accessed using the name followed by **`#any`**. Thus, in the connector behavior, `clientQuotationP`**`#any=i`**`~quotationReq` is a reference towards the connection `quotationReq` of a random instance of the meta port `clientQuotationP`, while keeping the reference in the `i` variable. Saving the reference towards the connection concerned by the request allows the connector to identify the request demander, and thus to return the response to the correct client.

This represention allows the connector between the clients and the ERP to evolve dynamically to enable the connection of new clients. In the next section we will show how composite dynamically evolving architectural elements can be described.

5.2 Evolution of composite architectural elements

In this section we have a look at how the arrival of new clients is represented at the supply chain architectural level. The supply chain is represented as a composite component. Each

composite element evolution is handled by a dedicated sub-component – the *choreographer*. The latter can change the topology whenever needed by: changing the attachments between architectural elements, dynamically creating new instances of architectural elements, excluding elements from the architecture, including elements which arrive into the architecture (coupling them with the rest of the architecture).

```
ClientToErpConnector is connector with {
   ports {  clientOrderP is OrderPort;
        clientQuotationP is QuotationPort;
        erpOrderP is OrderDemandPort;
        erpQuotationP is QuotationDemandPort;
        newClientP is port with {
             connections {
                  createI is connection(Any),
                  createO is connection(Any) }
             configuration { new createI; new createO }
             protocol {
                  via createI receive;
                  via createO send ;
                  recurse
             }
        }}
   configuration {   new clientOrderP; new clientQuotationP ;
        new erpOrderP; new erpQuotationP ; new newClientP}
   routing {
    choose{
        via clientQuotationP#any=i~quotationReq
             receive product:String, quantity:Integer;
        via erpQuotationP~quotationReq send product, quantity;
        via erpQuotationP~quotationRep receive price:Float;
        via clientQuotationP#i~quotationRep send price; }
   or   {
        via clientOrderP#any=i~orderReq
             receive product:String, quantity:Integer;
        via erpOrderP~orderReq send product, quantity;
        via erpOrderP~orderRep receive ack:String;
        via clientOrderP#i~orderRep send ack;
        if (ack=="OK") then {
             via erpOrderP~invoice receive  invoice: String;
             via clientOrderP#i~invoice send invoice;} }
   or {
        via newClientP~createI receive ;
        new clientOrderP; new clientQuotationP;
        via newClient~createO send }
    then recurse }
}
```

quotation communication

command communication

evolution management

Fig. 3. Connector between clients and the ERP

The `SupplyChain` choreographer (cf. Figure 4) handles the two evolution scenarios: the arrival of a new client and the reception of a new invoice system, which is transmitted to the `ERP` system. In the first case, the client is inserted into the `Client` meta component and an evolution message is sent to the `ClientToERP` connector, triggering the connector's evolution (cf. section 5.1). The `SupplyChain` choreographer attaches then the connector last created ports to the last client instance, *i.e.*, to the client that dynamically joined the supply chain.

```
SupplyChain is component with{
   ports {
        erpEvolveP is ERPEvolutionPort;
        newClientP is ClientPort;}
   constituents {
    clientComponent is Client;
    erpComponent is ERP;
    clientToErp is ClientToErpConnector;}
   configuration {
    new clientComponent; new erpComponent; new clientToErp;
    attach clientComponent~orderP to clientToErp~clientOrderP;
    attach clientToErp~erpOrderP to erpComponent~erpOrderP;
    attach clientComponent~quotationP
       to clientToErp~clientQuotationP;
    attach clientToErp~erpquotationP
       to erpComponent~erpQuotationP;}
   choreographer {
        choose {
             via erpEvolveP~newInvoice
                  receive newInvoiceComponent:InvoiceSystem;
             via erpComponent~erpEvolveP~newInvoice
                  send newInvoiceComponent;
             via erpComponent~erpEvolveP~ack receive ack:String;}
        or
        {    via newClientP~createOut receive c : Client;
             insert component c in Client ;
             via clientToErp~newClientP~createIn send ;
             via clientToErp~newClient~createOut receive ;
             attach clientComponent#last~orderP
                  to clientToErp~clientOrderP#last;
             attach clientComponent#last~quotationP
                  to clientToErp~clientQuotationP#last;}
     then recurse
   } }
```

Invoice system evolution

New client:

Demand the connector to evolve

Attach the new client to the connector

Fig. 4. The SupplyChain composite component

The ERP invoice system is replaced dynamically by a new one, and integrated in the ERP architecture. It is thus possible to change dynamically a system component. This is due, on the one hand, to the language formal foundations, the higher order π-calculus, which allows architectural elements to transit connections. On the other hand, the choreographer handles the connectivity among different architectural elements. It is thus possible to detach a component or to integrate a new one in a composite. The system topology changes in response to particular events.

New clients join the architecture dynamically; the connector evolves by creating communication ports for the new clients. This evolution scenario highlights the choreographer role in the evolution, its capacity to change dynamically the system topology. Other language mechanisms are used here, such as the management of meta elements' multiple instances and therefore the description of generic behaviors. The Client number of instances is unknown, and varies during execution. The Client meta entity handles the different instances, which can be on-the-fly created or come from outside the architecture. In this last case, typing constraints are imposed. The connector to which the Client component is attached has to evolve dynamically its interface (by adding new specific ports) and its behavior (the behavior definition does not change but is generic, so as to handle whatever number of clients).

The two evolution scenarios illustrate how ArchWare C&C-ADL allows the users to define the architecture of evolving systems. The evolutions presented contain some forms of mobility, as the new invoice system as well as new clients join the architecture at runtime. This is possible due to the use of the higher order π-calculus in the language foundations. Nevertheless we do not address other aspects related to mobility, only a rough management of the architecture state is made.

5.3 Property checking during evolution

Different kinds of properties are checked during the evolution. Some of them concern the typing and are intrinsically related to the language, and others are specified explicitly in order to represent domain-related properties to be checked. Concerning typing, for instance, the `newClient` connection of the `newClientP` port is typed so that only elements of type `Client` (or one of its sub-types) can transit via the connection. More precisely, in the previous example a component of type `Client` has to have two ports of type `QuotationDemandPort` and `OrderDemandPort`, *i.e.*, ports that have the same kind of connections and the same protocol. This ensures that a new component, to be integrated in the architecture as a client, is able to correctly interact with the rest of the system.

The explicit properties are more or less specific to the problem (in our case the supply chain). The main goal is to ensure the system integrity during the evolution, from structural as well as from behavioral points of view. An example of generic structural property is the *connectivity*: each element is connected to at least another element. While the system evolves the architectural elements have to remain correctly connected. Concerning the behavior, one can impose that each response to a command corresponds to an order made by a client. Properties can also combine both *structural* and *behavioral* aspects.

The following properties (structural and/or behavioral) are expressed using the ArchWare AAL analysis language (Alloui *et al.*, 2003a). Let us remind the reader that this language allows the users to define properties and comes with tools for property checking.

The property `connectivityOfERPArchitecture` expresses that each component has to be connected to at least another component. In our case study the architect has to verify that once the invoice system is changed, each component is connected to another component in the `ERP` composite component.

```
connectivityOfERPArchitecture is property {
-- each component is attached to at least another component
on self.components.ports apply
            forall { port1 | on self.components.ports apply
                  exists { port2 | attached(port1, port2) }}
}
```

The property `requestBeforeReplyOfOrderSystem` expresses the fact that the order system can send a response only after receiving a request. This property has to be verified also after the system evolution, *i.e.* when the invoice system is changed in the architecture.

```
requestBeforeReplyOfOrderSystem is property {
-- no reply without a request
  on OrderSystem.instances apply
   forall {os | (on os.actions apply isNotEmpty) implies
    (on os.orderP~orderReq.actionsIn apply
        exists {request | on os.orderP~orderRep.actionsOut apply
          forall {reply | every sequence {(not request)*. reply}
                          leads to state {false} }   } ) } }
```

This is expressed in AAL by a state formula that leads to `false` for all replies (belonging to `actionsOut`) sent before receiving a request (belonging to `actionsIn`).

The changes presented here were planned during the architecture design. The property checking takes place at design time too as the system evolves only in an anticipated way. That means each time that an architectural element is changed, related properties are checked on its new architecture description.

In the following section we will show how the unplanned architecture evolution can take place at runtime and how property checking is enabled before integrating the changes.

6. Dynamic unplanned evolution of the supply chain architecture

Let us go back to the original supply chain architecture. The case we are interested in is the one where no evolution was planned when the architecture was designed. So the architecture definition does not entail architectural changes to be triggered when given events occur (such as it was the case in the previous section) nor it is known what elements might evolve. Actually, the industrial reality shows that the maintenance and evolution of complex systems (client-server, distributed, *etc.*) is handled pragmatically, each case individually, without a methodical approach (Demeyer *et al.*, 2002).

The architecture proposed up to now (see section 4) responds to the problems the case study raises, without covering all possible evolution scenarios. What happens when the stocks for a product are not big enough to answer the demand, and the situation was not anticipated? What if the invoice system has to be outsourced, or if the supplier–client relation changes at runtime? The system initial architecture is unable to react to such unexpected events. So it has to evolve.

The scenario used for the unplanned dynamic evolution is different from the one presented in the previous section, although both are based on the initial scenario (section 3). More precisely a new restock system is to be added to the ERP.

6.1 Initial architecture: The supply chain system architecture before its evolution

We illustrate the dynamic unplanned evolution using an example described in the core language (rather than the C&C-ADL used for illustrating the dynamic planned evolution). Using the core language (also named π-ADL - see sections 2 and 3) enables to look at the evolution issue in its essence (independently from specific language layers) and to take advantage of the closeness with the virtual machine[1]. This induces a change in the architecture

[1] The virtual machine can only interpret the core ArchWare ADL language (Morisson *et al.*, 2004)

structure, as a description in the core language only uses the generic term of architectural abstraction (components and connectors used in the previous section are defined in terms of architectural abstractions (Cîmpan *et al.* 2005)). As the only terms are architectural abstractions, connected by connections (no ports, components nor connectors) we use a slightly different graphical notation as it is shown in Figure 5. There the architectural abstractions and their hierarchical composition for the initial architecture are presented.

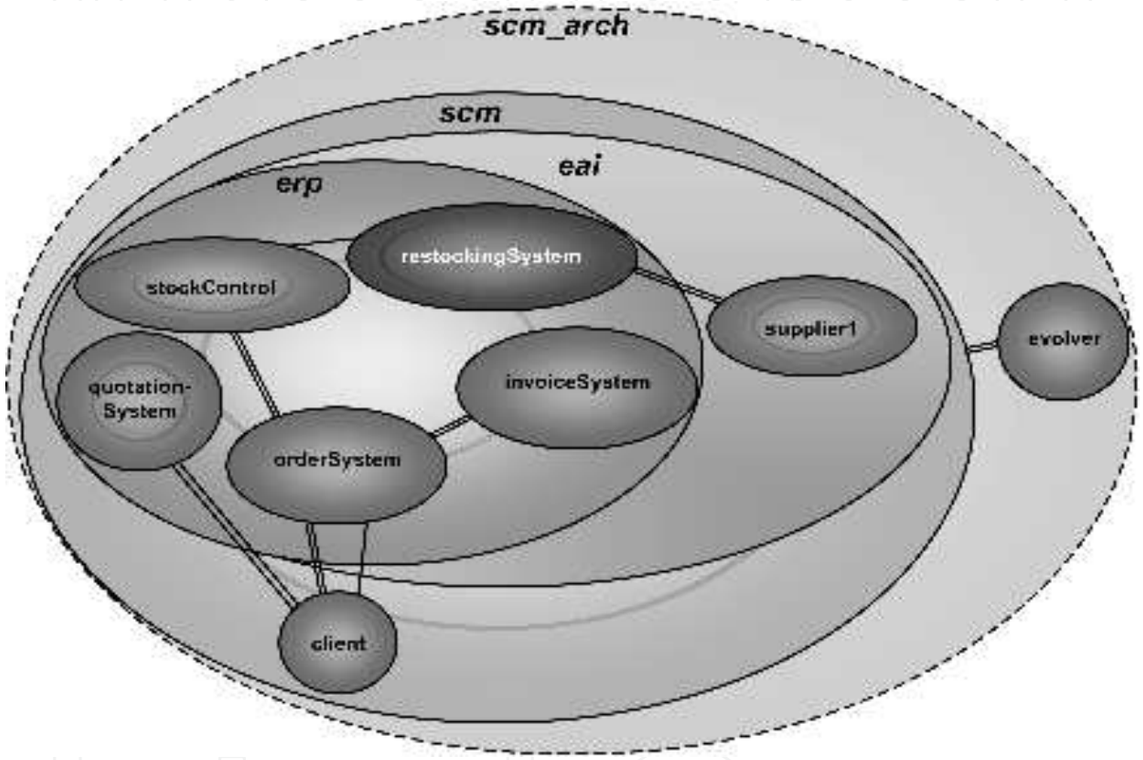

Fig. 5. The Supply Chain before its Evolution

The π-ADL descriptions for the client and supplier abstractions (cf. Figure 6) are rather simple. The client's behavior consists in sequencing the following actions: send a request for quotation, wait for the response; then, either make a request for quotation again (*i.e.*, when the previous one was not satisfactory), or place an order and wait for the invoice. The supplier receives a restocking request and satisfies it. In the initial scenario we chose a basic client-supplier relationship, in which any restock request is supposed to be satisfied (contractually this is of the suppliers' responsibility). The supplier acknowledges the request when it is ready to restock. We will see later how this relationship evolves.

```
value client is abstraction(String: quotationRequest, Integer: qty);{
   value quotationReq is free connection(String);
   value quotationRep is free connection(Float);
   value orderReq is free connection(String,Integer);
   value orderRep is free connection(String);
   value invoiceToClient is free connection(String);
   value quotationBeh is behaviour {
      via quotationReq send quotationRequest;
      via quotationRep receive amount:Float;
      unobservable; }
   quotationBeh();
   replicate {
       choose {
          quotationBeh();
        or
          behaviour {
             via orderReq send quotationRequest, qty;
             unobservable;
             via orderRep receive ack:String;
               if (ack == "OK) then {
                  via invoiceToClient receive invoice:String; } } } };
   done };
value supplier1 is abstraction(); {
   value restockingOrder1Req is free connection(String, Integer);
   value restockingOrder1Rep is free connection(String);
   replicate {
      via restockingOrder1Req receive wares:String, quantity:Integer;
      unobservable;
      via restockingOrder1Rep send "OK" };
   done };
```

Fig. 6. Descriptions for Client and Supplier

Building architectures in the core language is done by hierarchically composing abstractions. The ERP abstraction (cf. Figure 7) is composed by other abstractions. Let us remind the user that a behavior in the core language is closely related to a π-calculus process (Milner, 1999). Here the ERP abstraction is composed of abstractions representing systems for handling quotations, orders, invoices and restocks. The ERP abstraction is itself part of another abstraction, the EAI, itself part of another, and so on. The overall architecture (scm_arch) represents the supply chain management (cf. Figure 8) and its evolution capabilities. This abstraction composes the scm and evolver abstractions. In the π-calculus sense the two abstractions are autonomous processes, which are unified using their connection names and types (Oquendo *et al.*, 2002). Further in the chapter we will see the role played by each one of these abstractions.

```
value quotationSystem is abstraction(Float: price); {...}
value orderSystem is abstraction();{...}
value stockControl is abstraction(Integer: stock); {...}
value restockingSystem is abstraction();{...}
value invoiceSystem is abstraction();{...}
value erp is abstraction(Float: price, Integer: stock); {
       compose {      quotationSystem(price)
              and     orderSystem()
              and     invoiceSystem()
              and     stockControl(stock)
              and     restockingSystem() } };
value eai is abstraction(Float: price, Integer: stock); {
       compose {      supplier1(20)
              and
                      erp(price, stock) } } };
```

Fig. 7. The ERP abstraction

```
value scm_arch is abstraction(); {
        compose { scm()
                and
                evolver() } };
```

Fig. 8. The Supply Chain Abstraction (named scm_arch)

6.2 Language mechanisms for supporting dynamic unplanned evolution

The π-ADL evolution mechanisms are based on the π-calculus mobility (Milner, 1999). In π-ADL, an abstraction C (a behavior /process) can be sent from an abstraction A to another abstraction B. The latter can then dynamically apply it and may behave as the received abstraction C. As a consequence, the abstraction B has dynamically evolved, its current behavior might be radically different from the previous one (Verjus *et al.*, 2006). Such evolution is (1) dynamic because the new behavior (abstraction C) is dynamically received and (2) unplanned as the abstraction definition is unknown in advance. An architect can provide the abstraction definition at runtime, and thus represent the unplanned evolution (as opposed to the planned evolution illustrated in section 5).

To illustrate the evolution mechanisms let us consider a simple abstraction my_abst (cf. Figure 9). It receives a boolean (evolution) and an abstraction (evol_arch_part) on its connection evolRep. If the boolean value is true, my_abst behaves as the evol_arch_part abstraction definition received and applied. Otherwise (the boolean value is false), my_abst behaves in accordance to its initial description (// some code in Figure 9).

Thus, my_abst abstraction can be dynamically modified; such modification is unplanned as the evol_arch_part abstraction definition is unknown at design time and is provided at runtime by the evolver abstraction. The latter plays an important role in the evolution: it is always connected to the abstraction that is expected to evolve (the my_abst in this example).

```
value my_abst is abstraction(); {
   value evolReq is free connection();
   value evolRep is free connection(Boolean, abstraction() );
   . . . // some code
   via evolReq send;
   via evolRep receive evolution:Boolean,
                 evol_arch_part:abstraction(...);
   if (evolution) then {
      evol_arch_part(...) }
   else {
      . . .//some code } };
```

The new current abstraction's behaviour: the received abstraction is applied at runtime

Fig. 9. Abstraction Dynamic Evolution

```
value evolver is abstraction(); {
        value evolReq is free connection();
        value evolRep is free connection(Boolean, abstraction);
        value checkReq is free connection(abstraction);
        value checkRep is free connection(Boolean);
        evol_arch_part is ARCH-EVOLUTION;
        iterate{
        via evolReq receive;
        via checkReq send evol_arch_part;
        via checkRep receive propertyVerification : Boolean;
        if (propertyVerification)
                via evolRep send true, evol_arch_part ;
        }
};
```

Fig. 10. The Evolver Abstraction

The evolver abstraction is the communication means between the virtual machine and the external world *i.e.*, the architect who decides to evolve the architecture. As unplanned changes are to be implemented dynamically in the system, an important issue is the property preservation. Property verifications are made on the evol_arch_part abstraction, which represents (all) the evolved architecture (see section 6.4 for some insights on the evolved architecture according to the evolver abstractions' location(s)). This abstraction is sent using the mechanism we introduced to enable the on-the-fly exchange of abstractions: the use of a special abstraction type, ARCH-EVOLUTION. At design time, the evol_arch_part abstraction is declared of type ARCH-EVOLUTION (inside the evolver). This special type entails the evolution strategy, which can consist in using files, user's interfaces, *etc.* An example of such a strategy consists in using a file containing the new abstraction definition, at a place known by the virtual machine. The architect can place the evol_arch_part abstraction definition in this file. When the evolver sends the evol_arch_part abstraction, its definition is dynamically loaded by the virtual machine from the file.

However this is done only if property verification results are true, *i.e.*, the properties remain satisfied if changes contained in evol_arch_part are applied to the executing architecture. This verification is even more crucial than in the case of anticipated evolution since the content of evol_arch_part is not known in advance. The earlier property violations are detected, the lower the maintenance cost is. Thus the decision to evolve or not is taken in the evolver, depending on whether the properties are still verified after the evolution. Property

verifications are made on the `evol_arch_part` abstraction, which represents the evolved architecture and which is sent by the evolver to the property checker, also represented by an abstraction. The checker verifies the properties attached to the abstraction to be evolved tacking into account the changes contained in the `evol_arch_part` abstraction. It sends then a boolean value (`propertyVerification`) to the evolver: `true` if the properties still hold, `false` otherwise. If the properties hold, then the evolver sends the `evol_arch_part` abstraction to the abstraction to be evolved. Other strategies can be implemented, such as prompting the architect with the analysis results. The architect's choice to proceed or not is then reflected on the boolean value. Finally the changes are implemented on the running system taking into account the architecture execution state (Balasubramaniam et al., 2004).

6.3 Illustration of the dynamic unplanned evolution

This evolution scenario is interesting: on the one hand it implies evolving the system architecture structure by adding a second and new supplier; on the other hand, it enforces to change dynamically the restocking process to take into account that a restocking request may not be satisfied; in this case, a new supplier is appearing and the initial restocking request has to be split (with some quantity computations) among the two suppliers. The system behavior has to be changed dynamically according to the new configuration and process (Figure 11 and Figure 12).

The new architecture description is presented in Figure 12.

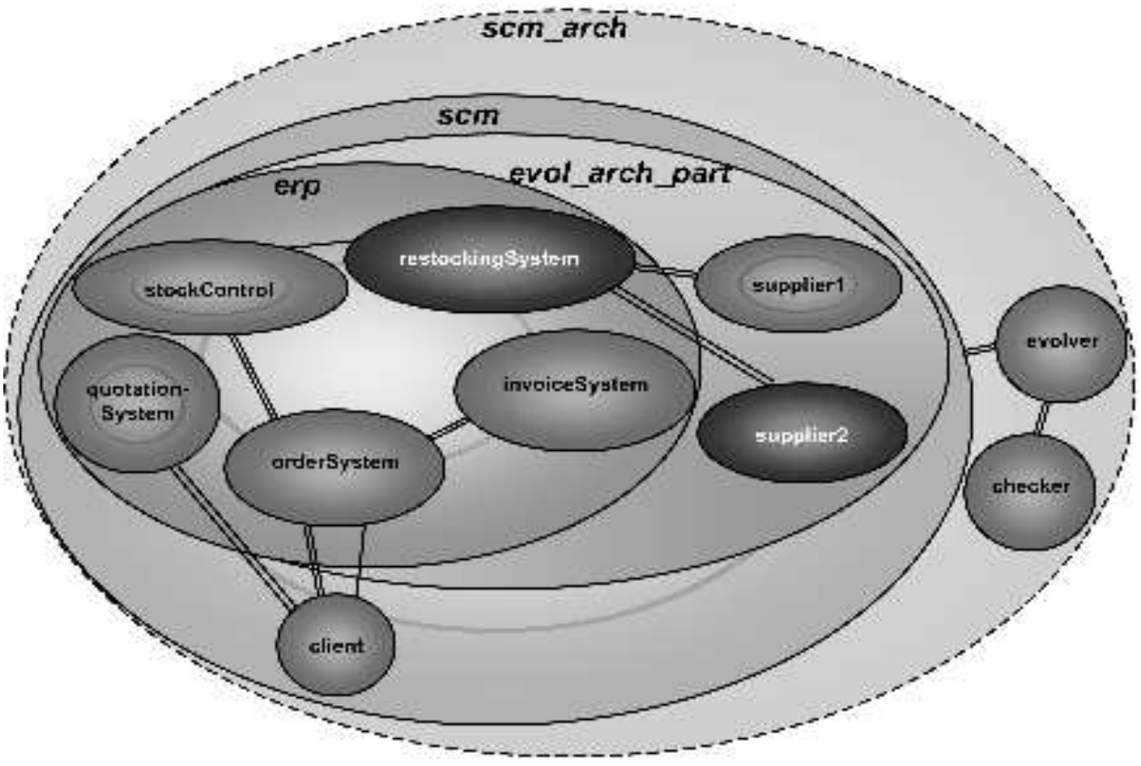

Fig. 11. The Evolved Architecture

```
value scm is abstraction(); {
   value evolReq is free connection();
   value evolRep is free connection(Boolean, abstraction()
);
   compose {
       behaviour {
          via evolReq send;
          via evolRep receive evolution:Boolean,
                 evol_arch_part:abstraction(Float,Integer);
          if (evolution) then {
                evol_arch_part(100.00, 32) }
          else { eai(100.00, 32) }}
     and
        client("rollings", 12) } };

value scm_arch is abstraction(); {
   compose {   scm()
       and     evolver()
       and     cheker()}
 };
```

The received abstraction is dynamically applied

Fig. 12. The evolved architecture (scm abstraction description)

Changes between the initial architecture (before evolution – Figure 5) and the modified architecture (after evolution – Figure 11) take place in the scm abstraction. The scm abstraction definition contains π-ADL code that manages the evolution: the scm abstraction behavior includes and applies both the eai and client abstractions (before evolution), and both the evol_arch_part and client abstractions (when the evolution occurs). Thus, when the evolution occurs, the evol_arch_part abstraction substitutes the eai abstraction. The evolver abstraction is unified with the abstraction that is supposed to evolve (scm). As explained in section 6.2, the evol_arch_part abstraction is dynamically received and then applied after property verification by the property checker.

As in the case of planned evolution, architectural properties may hold on both the structure and behavior of the system. Note that in the present case, the properties are expressed in AAL using the core ADL concepts; this is because the virtual machine can only interpret the core language (Morisson *et al.*, 2004). Examples of properties are:

Structural:

- connectivity: all architecture elements must be connected (no isolated element) in scm_arch;
- cardinality: there must be at least one supplier in scm_arch;

Behavioral :

- no regression of services provided within the architecture: client's request must always be satisfied;
- no regression of behavior safety: a supplier must first receive a request before any other action.

These properties are formalised using ArchWare-AAL as follows.

Architecture connectivity

In scm_arch, each architectural element (abstraction) abst1 is connected to at least another architectural element abst2 (non empty intersection of their connection sets).

```
connectivityOfscm_arch is property {
-- each abstraction must be connected to at least another one
    on self.abstractions apply
        forall { abst1 | on self.abstractions apply
        exists { abst2 | (abst2 <> abst1) and ((abst2.connections
          apply intersection(abst1.connections)) apply isNotEmpty) }  } }
```

Supplier cardinality

In scm_arch, there must be at least one abstraction supp of type Supplier.

```
atLeastOneSupplierInscm_arch is property {
-- there must be at least one supplier within the system
       on self.abstractions apply
         exists { sup | (sup.type="Supplier") }  }
```

Client's request satisfaction

Every client's request must be satisfied (through receiving an OK reply on orderReq connection). This property is expressed using a state formula on client's actions: every request followed by zero or more non OK reply followed by an OK reply is the expected behavior (leads to state True).

```
clientRequestSatisfaction is property {
-- the client must always have his(her) request satisfied
  on Client.instances apply
   forall {c | (on c.orderReq.actionsOut apply
        forall {request | on c.orderRep.actionsIn apply
          exists {reply | (reply = "OK") AND
                (every sequence { request.(not reply)*.reply }
                           leads to state {true}) }   } ) } }
```

Supplier's safe behavior

Each supplier must receive a request before any other action. This is expressed by a state formula on supplier's actions: every sequence starting with zero or more actions that are not of the restocking order type (*i.e.,* couple (wares, quantity)) and ending by a reply, is not an expected behavior (leads to state false).

```
requestBeforeReplyForSupplier is property {
-- no reply action before receiving a request
  on Supplier.instances apply
   forall {s |
    (on s.restockingOrderReq.actionsIn apply
       exists {request | (request.type='(String, Integer)')
               AND (on s.restockingOrderRep.actionsOut apply
                    forall {reply |
                       every sequence {(not request)*. reply}
                              leads to state {false} } )  }) } }
```

Before implementing changes contained in ARCH-EVOLUTION, user-defined properties are analyzed taking into account those changes. In our scenario, the four properties remain satisfied when introducing a second supplier: (a) the connectivity is still ensured, (b) there is still at least one supplier, (c) the client's request can be satisfied by supplier1 and if needed with the help of supplier2, (d) the supplier1's behavior and now the supplier2's behavior must remain safe. Consequently evolving the scm_arch this way does not a priori threaten the stability of the system architecture as unexpected behaviors are detected during property analysis.

Once the property verification successfully performed by the property checker and the evol_arch_part abstraction applied, the scm abstraction adopts a new behavior, defined dynamically by the architect, according to the adopted evolution strategy. This behavior adds a new supplier (supplier2) and the restocking process in the ERP is changed taking into account this new supplier. The Figure 13 does not show non-modified abstractions (for conciseness and clarity purposes). One can note that the new supplier (supplier2) does not behave as the existing supplier (supplier1) (*i.e.*, both suppliers are different): the evolution we introduced (see section 4) requires two different suppliers; we assume that a restocking request to the new supplier (supplier2) will only be satisfied if the requested quantity is less or equal to the supplier2's stock quantity (for a given product). The restocking process takes now into account the existence of a new supplier, and the initial demand may be split (according to the quantity requested) and handled respectively by the two suppliers.

We have shown in this section that: (i) the system is able to dynamically evolve with architectural elements that are dynamically and on-the-fly provided (not known in advance), (ii) required changes are transmitted to the executing architecture through a particular abstraction (evolver), (iii) the architect can check the architecture before and/or after the evolution using user-defined properties that are expressed in a dedicated architecture property definition language, (iv) changes are applied according to the results of the property verification.

```
value supplier2 is abstraction(Integer capacity); {
  value restockingOrder2Req is free connection(String, Integer);
  value restockingOrder2Rep is free connection(String, Integer);
  via restockingOrder2Req receive wares:String, quantity:Integer;
  unobservable;
  if (quantity > capacity) then {
       via restockingOrder2Rep send "NOK",capacity; }
  else { via restockingOrder2Rep send "OK",capacity;       }
  done };
value restockingSystem is abstraction(); {
  value restockingReq is free connection(String, Integer);
  value restockingOrder2Req is free connection(String, Integer);
  value restockingOrder2Rep is free connection(String, Integer);
  value restockingOrder1Req is free connection(String, Integer);
  value restockingOrder1Rep is free connection(String);
  via restockingReq receive wares:String, quantity:Integer;
  via restockingOrder2Req send wares, quantity;
  via restockingOrder2Rep receive ack:String, qtyReceived:Integer;
  if (ack == "NOK") then {
       via restockingOrder1Req send wares, (quantity-qtyReceived);
       unobservable;
       via restockingOrder1Rep receives ack2:String; }
  unobservable;
  done };
value erp is abstraction(Float: price, Integer: stock); {
       compose {      quotationSystem(price)
                and    orderSystem()
                and    invoiceSystem()
                and    stockControl(stock)
                and    restockingSystem() } };
value ARCH-EVOLUTION is abstraction(Float:price, Integer: stock); {
       compose { erp(price, stock)
                and    supplier1()
                and    supplier2(20)  } };
```

The abstraction that will be sent to the `scm` abstraction and applied by this latter

Fig. 13. Definition of the ARCH-EVOLUTION abstraction

6.4 Discussion

Let us now focus on evolution mechanisms illustrated in this section. When unpredictable situations occur, the architect has to dynamically (at runtime) provide an abstraction definition entailing the desired architectural changes. This abstraction, typed ARCH-EVOLUTION is (1) checked against architectural properties, (2) sent to the abstraction that is supposed to evolve and (3) dynamically applied by this latter (see section 6.2). The scope of an architectural modification is related to the dedicated abstraction (evolver) that manages such modification. More precisely it is related to the exact place the evolver takes in the architecture, *i.e.*, which abstraction it is bound to. A given modification that is defined within an ARCH-EVOLUTION abstraction may only impact the abstraction (and sub-abstractions) that receives this ARCH-EVOLUTION abstraction from the evolver. As a consequence, the evolvers (abstractions) are architectural evolution elements. They may be considered as non-functional architectural elements. Thus, it is up to the architect to decide, at design time, where to place evolvers. The architect has to decide which abstractions may

evolve without knowing how these abstractions will evolve. The architect adopts a strategy that can vary from using a unique evolver attached to the abstraction that composes the entire system to using as many evolvers as abstractions in the architecture. Both strategies have advantages and drawbacks. The first alternative forces the architect to dynamically provide the code that corresponds to the new (modified) entire architecture even if only a small change is occurring; it implies that that property checking is also performed on the entire architecture. The second alternative is quite heavy as it imposes that an evolver should be unified with every existing architectural abstraction (but when a change is occurring, only the code that corresponds to the evolved abstraction is redefined). This decision is related to the number of architectural abstractions and the underlying complexity of the full specification code (expressed in ArchWare ADL): it is an architectural design issue that can be solved.

Furthermore as the ADL proposes abstraction composition and evolution-dedicated abstractions, a given architecture may considerably evolve with cascading evolution situations. An open issue is, then, when and how an architecture is deviating so far from its initial configuration that we may consider it as another architecture (not as an evolved one).

During planned evolution or unplanned evolution, user-defined properties are evaluated taking into account the new architecture. It is worth noting in the scenarios of section 6.3, that while some property expressions like `connectivityOfScm_arch` are not affected by the change, other properties like `requestBeforeReplyForSupplier` should evolve to express `supplier2's` expected safe behavior as well.

7. Related work

This section presents the work related to the dynamic evolution of software-intensive information systems using a software architecture-centric approach.

(Bradbury *et al.*, 2004) presents a survey of self-management in dynamic software architecture specifications. The authors compare well known architecture description languages and approaches in a self management perspective; dynamic architectural changes have four steps : (1) initiation of change, (2) selection of architectural transformation, (3) implementation of reconfiguration and (4) assessment of architecture after reconfiguration. This chapter focuses on the three first steps. In the Bradbury *et al.* survey, most of the studied ADLs support all of the basic change operations (adding or removing components and connectors) but other more complex operations are not fully satisfied. Particularly, dynamically modifying internal component behavior remains an issue that is successfully addressed only by few ADLs.

The representation of evolvable software-intensive information system architectures is related to architecture description languages and their capabilities, *i.e.*, ADLs that allow the architect to express dynamic evolvable architectures, including adaptive architectures. Few ADLs support dynamic architecture representation: Darwin (Magee *et al.*, 1995), Dynamic Wright (Allen *et al.*, 1998), π-Space (Chaudet & Oquendo, 2000), C2SADEL (Medvidovic *et al.*, 1999; Egyed & Medvidovic, 2001; Egyed *et al.*, 2001), Piccola (Nierstrasz & Achermann, 2000), Pilar (Cuesta *et al.*, 2005), ArchWare π-ADL (Oquendo *et al.*, 2002; Oquendo 2004), ArchWare C&C-ADL (Cîmpan *et al.*, 2005). Most of them are not suitable to support

unplanned dynamic architecture evolution as they consider different representations for the concrete and abstract levels, and use reflection mechanisms to switch among these representations: a dynamic architecture is first defined at abstract level and is then reflected (1) into a dynamic evolvable concrete software-intensive system (Cazzola *et al.*, 1999; Tisato *et al.*, 2000) or (2) into another, evolved abstract representation (Cuesta *et al.*, 2001; Cuesta *et al.*, 2005). The link between the abstract level and the concrete one is not maintained, leading to a situation in which only anticipated modifications can be supported dynamically. ArchWare π-ADL uses a unique representation for both levels (Verjus *et al.*, 2006).

Thus, handling the software evolution is closely related to the problem of keeping the consistency between the abstract and the implementation levels and continuously switching between these levels. This issue is particularly important in the case of runtime evolution. The consistency among the abstract and the concrete levels can be seen in two directions: top-down from the abstract level to the concrete one (such as considered in model-driven and architecture refinement approaches) and bottom-up from the concrete level to the abstract one (such as considered by architecture extraction approaches). Our approach adresses the top-down consistency.

Going from abstract architectural representations to more concrete ones is inherent in architecture-centric development, and to the model-driven development in general. The architecture-centric development highly depends on maintaining the consistency among levels. Traditionally, when changes on the abstract architecture occur, it is up to the developer to modify the code accordingly (sometimes assisted by semi-automated code generation tools). Some architecture-based development approaches maintain mappings between single versions of the architecture and their corresponding implementations (Carriere *et al.*, 1999; Medvidovi *et al.*, 1999; Erdogmus, 1998; Egyed 2000; Van der Hoeck *et al.*, 2001; Egyed *et al.*, 2001; Dashofy *et al.*, 2002; Aldrich *et al.*, 2002).

(Egyed & Medvidovic, 2001) approach introduces an intermediate "design" level between architectural (abstract) level and implementation (concrete) level. The consistency between these levels is managed using mapping rules between UML diagrams with OCL constraints (at design level) and C2 concepts (at architectural level). The transformation-based consistency checking is ensured by IVita (Egyed & Medvidovic, 2001). This approach assumes that changes are applied off-line.

ArchEvol (Nistor *et al.*, 2005) proposes to accurately determine which versions of the component implementations belong to the initial version of the architecture and which belong to the branched version of the architecture. ArchEvol defines mappings between architectural descriptions and component implementations using a versioning infrastructure (by using conjointly Subversion, Eclipse and ArchStudio) and addresses the evolution of the relationship between versions of the architecture and versions of the implementation. ArchJava (Aldrich *et al.*, 2002) is an extension to Java that unifies the software architecture with implementation, ensuring that the implementation conforms to the architectural constraints. The latter mainly concern the communication integrity, *i.e.*, implementation components only communicate directly with the components they are connected to in the architecture. The limitations of ArchJava are inherent to Java systems, that are difficult to dynamically evolve without stopping the executing system. In (Garlan *et al.*, 2004), the code is monitored, changes are made on abstract architectures using a change script language and

then mapped into the code. Each change operator has its transformation into lower level changes. If the change script execution fails at the code level, the change is aborted. As this work is made in the context of self-adaptation, issues such as how the change is triggered are taken into account. The evolution takes place at runtime, and concerns both the design and the code.

(Huang *et al.*, 2006) focus on dynamic software architecture extraction and evolution by catching the executing component-based system state and system behavior: an architectural representation is deduced and can be modified at runtime. The approach can be seen as a unifying one. However, the deduced architectural representation is incomplete and the approach is limited to existing component-based systems.

Thus, the issue of dynamic unplanned changes is not satisfactory addressed. Aside ArchWare, none of the existing proposals unifies the abstract level and the implementation level in a complete and consistent manner. This is related to the fact that these proposals consider software-intensive information system architecture at abstract levels only. Most of the ADLs provide high level architectural means by focusing on abstract architectures. As we can see, there is an unbalance between the two identified issues, namely the description of dynamic architectures and the handling of unexpected changes. The former is rather well addressed by existing proposals, but the latter is practically uncovered. For an ADL, to consider both issues is important, the ADL should not only be a design language but also an implementation one, *i.e.*, architectures become executable representations. We have investigated in this chapter dynamic evolvable software-intensive information system architectures in a model-based development perspective. Model-based development is centered around abstract, domain-specific models and transformations of abstract models into more specific underlying platforms. Our approach addresses both abstract and concrete architectural models in a model-based development framework and is quite related to the Monarch approach (Bagheri & Sullivan, 2010). Nevertheless Monarch does not deal with dynamic architecture evolution support.

8. Conclusion

In this chapter we have presented an architecture model-based development approach guiding the development of dynamic evolvable software-intensive information system architectures. Our approach supports dynamic evolvable architecture development process covering modeling, analysis and execution. As the evolution is an important facet of a software-intensive system (Lehman, 1996), our proposal aims at integrating evolution modeling and evolution mechanisms into an executable and formal ADLs that can serve at both abstract and concrete levels (bridging the gap between both levels). In this chapter, we use ArchWare languages and technologies. Our proposal deals with the dynamic evolution of architecture using specific architectural elements and ADL built-in mechanisms. We claim that software-intensive system architectures have to incorporate, at the design time, evolution mechanisms making those architectures evolution-aware at runtime.

Architectural changes can occur at different levels of abstraction and may concern architecture structure and behavior, internal architectural elements' structure and behavior as well as their related properties. The dynamic support elements and mechanisms we have introduced and formally defined serve not only at classical architectural (abstract) level but

also as implementation means. In other words, the software architecture, if completely detailed and defined, can be the entire executing system itself. In our proposal, even in the case of information systems incorporating heterogeneous and existing components or legacy systems, information system components' "glue" is described as behavior that can be modified at runtime. Then, components can be dynamically removed, modified or added and the way they interoperate can also be dynamically modified.

This issue positively impacts software-intensive information system maintenance activities and underlying costs, budgets, efforts and skills.

As for proposed mechanisms for unplanned evolution, through the concept of evolver and the ADL virtual machine, we consider them as a significant research advance as at the best of our knowledge. Moreover as the framework relies on core ADL concepts of abstraction composition and evolution-dedicated abstractions, a given architecture may considerably evolve with cascading evolution situations. We did not discuss when and how an architecture is deviating so far from its initial configuration so that we may consider it as another architecture/system (and not as an evolved one). This issue can be further addressed and related to architectural evolution patterns and paths introduced in (Garlan *et al.*, 2009). We also think that the ADL-based evolution support we propose is a good candidate for self-adaptation system design and analysis but further investigations and case studies are mandatory.

The scenarios used in this chapter, illustrate changes that are related to the composition of the system (by adding for example a supplier) as well as the behaviour of the system (in other words the business process) by modifying the restocking process. Other case studies have been realized using the ArchWare approach and technologies, *i.e.*, for a Manufacturing Execution System for a Grid-based application in a health-related project (Manset *et al.*, 2006). Other reserarch activities are directly inspired from these results (Pourraz *et al.*, 2006).

9. References

Abrial, J.R. (1996). *The B Book*, Assigning Programs to Meanings, Cambridge University Press, Cambridge, 1996.

Aldrich, J.; Chambers, C. & Notkin, D. (2002). ArchJava: Connecting Software Architecture to Implementation, *Proceedings of the 24th International Conference on Software Architecture (ICSE 2002)*, Orlando, Florida, May 2002.

Allen, R. ; Douence, R. & Garlan D. (1998). Specifying and Analyzing Dynamic Software Architectures, *Proceedings on Fundamental Approaches to Software Engineering*, Lisbon, Portugal, March 1998.

Alloui, I. & Oquendo, F. (2003). UML Arch-Ware/Style-based ADL, Deliverable D1.4b, ArchWare European RTD Project, IST-2001-32360, 2003.

Alloui, I. ; Garavel, H. ; Mateescu, R. & Oquendo F.(2003a). The ArchWare Architecture Analysis Language, Deliverable D3.1b, ArchWare European RTD Project, IST-2001-32360, 2003.

Alloui, I. ; Megzari, K. & Oquendo F. (2003b). Modelling and Generating Business-To-Business Applications Using an Architecture Description Language - Based

Approach, *Proceedings of International Conference on Enterprise Information Systems (ICEIS)*, Anger, France, April 2003.

ArchStudio http://www.isr.uci.edu/projects/archstudio.

Andrade, L.F. & Fiadeiro, J.L. (2003). Architecture Based Evolution of Software Systems, In *Formal Methods for Software Architectures*, M.Bernardo & P.Inverardi, pp. 148-181, LNCS 2804, 2003.

ArchWare Consortium (2001). *The EU funded ArchWare – Architecting Evolvable Software – project*, http://www.arch-ware.org, 2001.

Azaiez, S. & Oquendo, F. (2005). Final ArchWare Architecture Analysis Tool by Theorem-Proving: The ArchWare Analyser, Delivrable D3.5b, ArchWare European RTD Project IST-2001-32360, 2005.

Bagheri, H. & Sullivan, K. (2010). Monarch: Model-based Development of Software Architectures, *Proceedings of the 13th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Oslo, Norway, October 2010.

Balasubramaniam, D.; Morrison, R.; Mickan, K.; Kirby, GNC.; Warboys, B.C.; Robertson, I.; Snowdon, B; Greenwood, R.M. & Seet, W. (2004). Support for Feedback and Change in Self-adaptive Systems, In *Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04)*, Newport Beach, CA, USA, ACM, October /November 2004.

Barrios, J. & Nurcan, S. (2004). Model Driven Architectures for Enterprise Information Systems, In *Proceedings of 16th Conference on Advanced Information Systems Engineering, (CAISE'04),* Springer Verlag (pub), Riga, Latvia, June 2004.

Bass, L.; Clements, P. & Kazman, R.(2003). *Software architecture in practice*, Second Edition, Addison-Wesley, 2003.

Belady, L. & Lehman, M.(1995). *Program Evolution Processes of Software Change*, Academic Press, London, UK, 1995.

Bergamini, D.; Champelovier, D.; Descoubes, N.; Garavel, H.; Mateescu, R. & Serwe, W. (2004). Final ArchWare Architecture Analysis Tool by Model-Checking, *ArchWare European RTD Project IST-2001-32360*, Delivrable D3.6c, December 2004.

Bradbury, J.S.; Cordy, J.R.; Dingel, J.& Wermelinger, M.(2004). A survey of self-management in dynamic software architecture specifications. In *Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS '04)*. Newport Beach, CA, USA, ACM, October /November 2004.

Bradfield, J. C.& Stirling, C. (2001). Modal logics and mu-calculi: an introduction , In *Handbook of Process Algebra*, Elsevier, pp. 293–330, 2001.

Carriere, S.; Woods, S. & Kazman, R. (1999). Software Architectural Transformation, In *Proceedings of 6th Working Conference on Reverse Engineering*, IEEE Computer Society, Atlanta, Georgia, USA, October 1999.

Cazzola, W.; Savigni, A.; Sosio, A. & Tisato, F. (1999). Architectural Reflection : Concepts, Design and Evaluation, Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano. Retrived from http://www.disi.unige.it/CazzolaW/references.html, 1999.

Chaudet, C. & Oquendo, F. (2000). π-SPACE: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems, In *Proceedings of 15th*

IEEE International Conference on Automated Software Engineering, Grenoble, France, September 2000.

Cîmpan, S.; Oquendo, F.; Balasubramaniam, D.; Kirby, G. & Morrison, R. (2002). The ArchWare ADL: Definition of the Textual Concrete Syntax, *ArchWare Europeean RTD Project IST-2001-32360*, Delivrable D1.2b, December 2002.

Cîmpan, S. & Verjus, H. (2005). Challenges in Architecture Centred Software Evolution, In *CHASE: Challenges in Software Evolution*, Bern, Switzerland, 2005.

Cîmpan, S.; Leymonerie, F. & Oquendo, F. (2005). Handeling Dynamic Behaviour in Software Architectures, In *Proceedings of European Workshop on Software Architectures*, Pisa, Italy, 2005.

Cuesta, C.; de la Fuente, P.& Barrio-Solorzano, M. (2001). Dynamic Coordination Architecture through the use of Reflection, In *Proceedings of the 2001 ACM symposium on Applied Computing*, Las Vegas, Nevada, United States, pp. 134 – 140, March 2001.

Cuesta, C.; de la Fuente, P.; Barrio-Solorzano, M. & Beato, M.E. (2005). An abstract process approach to algebraic dynamic architecture description, In *Journal of Logic and Algebraic Programming*, Elsevier, Vol. 63, pp. 177-214, ISSN 1567-8326, 2005.

Dashofy, E. M.; van der Hoek, A. & Taylor, R. N. (2002). Towards architecture-based self-healing systems, In *Proceedings of the First Workshop on Self-Healing Systems (WOSS '02)*,. D. Garlan, J. Kramer, and A. Wolf, Eds., Charleston, South Carolina, November 2002.

Egyed, A. & Medvidovic, N. (2001). Consistent Architectural Refinement and Evolution using the Unified Modeling Language, In *Proceedings of the 1st Workshop on Describing Software Architecture with UML*, co-located with ICSE 2001, Toronto, Canada, pp. 83-87, May 2001.

Egyed, A.; Grünbacher, P. & Medvidovic, N.(2001). Refinement and Evolution Issues in Bridging Requirements and Architectures, In *Proceedings of the 1st International Workshops From Requirements to Architecture (STRAW)*, co-located with ICSE, Toronto, Canada, pp. 42-47, May 2001.

Egyed, A. (2000). Validating Consistency between Architecture and Design Descriptions, In *Proceedings of 1st Workshop on Evaluating Software Architecture Solutions (WESAS)*, Irvine, CA, USA, May 2000.

Erdogmus H. (1998). Representing Architectural Evolution, In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, pp. 159-177, November 1998.

Favre, J.-M. ; Estublier, J. & Blay, M. (2006). *L'Ingénierie Dirigée par les Modèles : au-délà du MDA*, Edition Hermes-Lavoisier, 240 pages, ISBN 2-7462-1213-7, 2006.

Ghezzi, C.; Jazayeri, M. & Mandrioli D. (1991). *Fundamentals of Software Engineering*, Prentice Hall, 1991.

Garlan, D.; Cheng, S.-W.; Huang, A.-C.; Schmerl, B. & Steenkiste, P. (2004). Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure, *IEEE Computer*, Vol. 37, No. 10, October 2004.

Garlan, D.; Barnes, J.M.; Schmerl, B. & Celiku, O. (2009). Evolution styles: Foundations and tool support for software architecture evolution, In *Proceedings of the 7th Working*

IEEE/IFIP Conference on Software Architecture (WICSA'09), pp. 131–140, Cambridge, UK, September 2009.

Huang, G.; Mei, H. & Yang, F.-Q. (2006). Runtime recovery and manipulation of software architecture of component-based systems, In *Journal of Automated Software Engineering*., Vol. 13, No. 2, pp 257-281, 2006.

Kardasis, P. & Loucopoulos, P. (1998). Aligning Legacy Information Systems to Business Processes , In *Proceedings of International Conference on Advanced Information Systems Engineering (CAISE'98)*, Pisa, Italy, June 1998.

Kyaruzi, J. J. & van Katwijk, J. (2000). Concerns On Architecture-Centered Software Development: A Survey, In *Journal of Integrated Design and Process Science*, Volume 4, No. 3, pp. 13-35, August 2000.

Lehman M. M. (1996). Laws of Software Evolution Revisited, In *Proceedings of European Workshop on Software Process Technology (EWSPT 1996)*, p. 108-124, Nancy, France, October 1996.

Leymonerie F. (2004). ASL language and tools for architectural styles. Contribution to dynamic architectures description, *PhD thesis, University of Savoie*, December 2004.

Magee, J.; Dulay, N.; Eisenbach, S. & Kramer J. (1995). Specifying Distributed Software Architectures, In *Proceedings of 5th European Software Engineering Conference (ESEC '95)*, LNCS 989, pp. 137-153, Sitges, September 1995.

Manset, D.; Verjus, H.; McClatchey, R. & Oquendo, F. (2006). A Formal Architecture-Centric Model-Driven Approach For The Automatic Generation Of Grid Applications. In *8th International Conference on Enterprise Information Systems (ICEIS'06)*, Paphos, Chyprus, 2006.

Mateescu, R. & Oquendo, F. (2006). π-AAL: an architecture analysis language for formally specifying and verifying structural and behavioural properties of software architectures, In *ACM SIGSOFT Software Engineering Notes*, Vol. 31, No. 2, pp. 1-19, 2006.

Medvidovic, N. & Taylor, R.N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages, In *IEEE Transactions on Software Engineering*, Vol. 26, No. 1, pp. 70-93, 2000.

Medvidovic, N.; Egyed, A. & Rosenblum, D. (1999). Round-Trip Software Engineering Using UML: From Architecture to Design and Back, In *Proceedings of the 2nd Workshop on Object-Oriented Reengineering*, pp. 1-8, Toulouse, France, September 1999.

Mens, T.; Buckley, J.; Rashid, A. & Zenger, M. (2003). Towards a taxonomy of software evolution, In *Workshop on Unanticipated Software Evolution*, (in conjunction with ETAPS 2003) Varsaw, Poland, April 2003.

Milner, R. (1999). *Communicating and Mobile Systems: the π -calculus*, Cambridge University Press, 1999.

Morrison, R.; Kirby, GNC.; Balasubramaniam, D.; Mickan, K.; Oquendo, F.; Cîmpan, S.; Warboys, BC.; Snowdon, B. & Greenwood, RM. (2004). Support for Evolving Software Architectures in the ArchWare ADL, In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 4)*, Oslo, Norway 2004.

Nierstrasz, O. & Achermann, F. (2000). Supporting Compositional Styles for Software Evolution, In *Proceedings of International Symposium on Principles of Software Evolution*, IEEE, Kanazawa, Japan, pp. 11-19, November 2000.

Nistor, E.; Erenkrantz, J.; Hendrickson, S. & van der Hoek, A. (2005). ArchEvol: Versioning Architectural-Implementation Relationships, In *Proceedings of the 12th International Workshop on Software Configuration Management*, Lisbon, Portugal, September 2005.

Nurcan, S. & Schmidt, R. (2009). Service Oriented Enterprise-Architecture for enterprise engineering introduction, In *Proceedings of 13th IEEE International Enterprise Distributed Object Computing Conference*, pp. 247-253, Auckland, New Zeeland, September 2009.

Oquendo, F. (2004). π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures, In *ACM SIGSOFT Software Engineering Notes*, Volume 29, No. 3, pp. 1-14, 2004a.

Oquendo, F.; Alloui, I.; Cîmpan, S. & Verjus, H. (2002). The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantic, *ArchWare European RTD Project IST-2001-32360*, Deliverable D1.1b, 2002.

Oquendo, F.; Warboys, B.; Morrison, R.; Dindeleux, R.; Gallo, F.; Garavel, H. & Occhipinti, C. (2004). ArchWare: Architecting Evolvable Software, In *Proceedings of the 1st European Workshop on Software Architecture (EWSA 2004)*, St Andrews, UK, pp. 257-271, 2004.

Perry, D.E. & Wolf, A.L. (1992). Foundations for the study of software architecture, In *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, 1992.

Pourraz,, F. ; Verjus, H. & Oquendo, F. (2006). An Architecture-Centric Approach For Managing The Evolution Of EAI Service-Oriented Architecture, In *8th International Conference on Enterprise Information Systems (ICEIS'06)*, Paphos, Chyprus, 2006.

Tisato, F.; Savigni, A.; Cazzola, W. & Sosio, A. (2000). Architectural Reflection - Realising Software Architectures via Reflective Activities, In Proceedings of the 2nd Engineering Distributed Objects Workshop (EDO 2000), University of Callifornia, Davis, USA, November 2000.

Touzi, J.; Benaben, F.; Pingaud, H. & Lorre, J. (2009). A model-driven approach for collaborative service- oriented architecture design, In *International Journal of Production Economics*, Vol. 121, Issue 1, p. 5-20, 2009.

Van der Hoek, A.; Mikic-Rakic, M.; Roshandel, R. & Medvidovic, N. (2001). Taming architectural evolution, In *Proceedings of the 8th European Software Engineering Conference*, ACM Press, pp.1-10, Viena, Austria, September 2001.

Verjus, H. & Oquendo, F. (2003). Final XML ArchWare style-based ADL (ArchWare AXL), *ArchWare European RTD Project IST-2001-32360*, Deliverable D1.3b, June 2003.

Verjus, H. ; Cîmpan, S. ; Alloui, I. & Oquendo, F. (2006). Gestion des architectures évolutives dans ArchWare, In *Proceedings of the First Conférence francophone sur les Architectures Logicielles (CAL 2006)*, Nantes, France, September 2006, pp. 41-57.

Verjus, H. (2007). Nimrod: A Software Architecture-Centric Engineering Environment - Revision 2, *Nimrod Release 1.4.3, University of Savoie - LISTIC*, Number LISTIC No 07/03, June 2007.

Vernadat, F. (2006). Interoperable enterprise systems: architecture and methods, Plenary Lecture at *12th IFAC Symposium on Information Control Problems in Manufacturing*, Saint-Etienne, France, May 2006.

Zachman, J. (1997). Enterprise Architecture : The Issue of the Century, In *Database Programming and Design*, Vol. 10, p. 44-53, 1997.

IntechOpen