DYNAMIC ARCHITECTURE BASED EVOLUTION OF ENTERPRISE INFORMATION SYSTEMS

Sorana Cîmpan, Herve Verjus and Ilham Alloui
University of Savoie – Polytech' Savoie - LISTIC Lab
B.P. 806 - 74016 Annecy Cedex - France
Phone: +33 (0)4 50 09 65 80
sorana.Cîmpan@univ-savoie.fr, herve.verjus@univ-savoie.fr, ilham.alloui@univ-savoie.fr

Keywords: Software architecture, formal language, evolution scenarios, enterprise information system.

Abstract: Enterprise Information Systems have to co-evolve with the enterprise they support. Their evolution is the one of an important software system. Software evolution should be addressed at all developpement phases in order to notably reduce costs (Lehman, 1996). The issue of software systems evolution has been addressed mainly at the code level. In this paper we present how evolution of enterprise information systems can take place at higher abstraction levels, when using an architecture-centred development process. The evolutions addressed are dynamic, i.e. they take place at runtime and concern both planned and unplanned evolutions of the enterprise information system.

1 INTRODUCTION

Human-centric activities are more and more supported by software applications, most enterprises relying on an enterprise information system, which has to evolve according to requirements, new technologies, etc. Thus, evolution and quality of software systems is a major issue (Andrade *et. al.*, 2004, Mens *et. al.*, 2003), related to changes that may occur at different level (market, functionalities, needs, etc.).

The evolution is often considered at the latter stages of software system development process, *i.e.* implementation and execution, mostly by adopting pragmatic approaches (Demeyer *et. al.*, 2002), but it is rarely studied in the earlier stages (design, modelling, specification). We agree with (Lehman, 1996), which indicates that evolution should be studied at each software development process stage in order to notably reduce costs. We claim (Verjus *et. al.*, 2006) that some evolutions could be taken into account during the design and would not have to be postponed to latter phases, namely the implementation or runtime.

In this paper, we focus on system evolution using a software architecture centric approach experimented and validated in the ArchWare European project (ArchWare, 2001). The project address evolutions that may have impact on the system software architecture; maintenance tasks that have no impact on the architecture are managed using classical approaches.

We classified the system evolution according to moment when it takes place, *i.e.* static or at runtime, and to their predictability during the design, *i.e.* planned and unplanned (Cîmpan and Verjus, 2005). This paper does not address static evolutions, which are taken into account by all the (architectrure) modelling approaches (Ding *et. al.*, 2001, Barais *et. al.*, 2005, Tibermacine *et. al.*, 2005). A simplist description of such changes (be them planned or not) is: stop the system, do the change, check consistency, run again the system. We focus here on cases where the system cannot be stopped, and thus the change has to take place at runtime. We illustrate how our architecture-centric approch allows taking into account, during runtime, planned and unplanned dynamic evolutions.

Planned dynamic evolutions are managed at the architectural level and enacted automatically (self-contained architecture) without external help. Unplanned dynamic evolution management implies that the considered architecture provides an evolution entry point and that an evolution mechanism is available for the external environment. Thus, human or other external means could dynamically evolve the system (architecture) by using such entry points. Unplanned dynamic

evolution appears to be the most common situation (Mens *et. al.*, 2005, Demeyer *et. al.*, 2002, Mens *et. al.*, 2003), particularly when considering the maintenance of existing software systems. We show how our approach allows to manage software architecture evolution using a formal architecture description language, ArchWare ADL (Oquendo *et. al.,* 2002). The language covers both structural and behavioural aspects of architecture descriptions as well as the expression of architectural constraints and properties.

This paper is organized as follows: section 2 presents the ArchWare related foundations and technologies; section 3 introduces scenarios that will illustrate evolution (both planned and unplanned). Then, we focus on planned dynamic evolution in section 4 and on unplanned dynamic evolution in section 5. Conclusions and perspectives will close the paper.

2 THE ARCHWARE PROJECT AND LANGUAGES

The work presented here has been partially funded by the European Commission in the framework of the IST ArchWare Project (Archware Consortium, 2001) and by the French ANR Cook project (Cook 2006). The ArchWare project proposes an innovative architecture-centric software engineering framework, *i.e.* architecture description and analysis languages, architectural styles, refinement models, architecture-centric tools, and a customisable software environment. The main concern is to guarantee required quality attributes throughout evolutionary software development (initial development and evolution), taking into account domain-specific architectural styles, reuse of existing components, support for variability on software products and product-lines, and run-time system evolution. The Cook project studies the role of software architectures in the reengineering of object-oriented applications (Pollet *et. al.*, 2007).

In ArchWare, a software architecture is considered as a set of typed nodes connected by relations. When describing architectures, the nodes are termed components and the relations termed connectors. These components and connectors and their compositions have specified behaviours based on π-calculus (Milner 1999), and are annotated with quality attributes. ArchWare proposes a set of languages for: (1) describing the architecture (ArchWare ADL), (2) architecture properties (ArchWare AAL), (3) architecture refinement (ArchWare ARL). ArchWare ADL offers different language layers for describing architecture, from the more generic one (the core language), to language that are more and more specific. Such layers can be defined by the user, using the style mechanism. (Cîmpan *et al.,* 2005) presents the layered construction of the language.

The core description language ArchWare π-ADL is based on the concept of formal composable components and on a set of operations for manipulating these components (Oquendo *et. al.*, 2002). The ADL supports the concepts of behaviours and abstractions of behaviours, to represent respectively running components and parametric component types. Behaviour is described using all the basic π-calculus operations as well as composition and decomposition. Communication between components is via channels represented by connections (representing component interfaces as well). The ArchWare ADL allows the definition of evolvable architectures, *i.e.*, where new components and connectors can be incorporated and existing ones can be removed, governed by explicit policies.

A language based on well-known component connector, Archware C&C-ADL, is proposed as a layer built on top of the core language. Given its formal foundation on π-calculus, its ability to represent both structural and behavioural aspects of architectures, as well as architectural constraints ArchWare ADL has an expressive power higher than most existing ADLs (Medvidovic *et. al.*, 2000, Verjus *et. al.*, 2006).

In this paper we use the Archware C&C-ADL for illustrating the planned dynamic evolution and the core language for illustrating the unplanned dynamic evolution. In both cases, properties are represented using ArchWare AAL.

3 EVOLUTION SCENARIOS

The chosen evolution scenarios are related to a Supply Chain Management System (SCMS), entailing an enterprise information system, with its clients and its suppliers. The enterprise information system and its suppliers constitute what we will call hereafter an EAI solution. The SCMS architecture will be modified according to particular requirements and/or constraints. Other case studies using the ArchWare approach in have been published (Blanc dit Jolicoeur *et. al.*, 2002, Pourraz *et. al.*, 2006, Ratcliffe *et. al.*, 2005, Revillard *et. al.*, 2005).

Initial scenario. Whenever a client passes an order to the EAI, it first asks for a quotation. If the quotation satisfies the client, it makes an order. The ordering system (or component), takes the order, updates the stock and may ask a supplier for restocking if the current stock is not enough to satisfy the order.

Planned dynamic evolution scenario. Two dynamic evolutions are planned. One concerns the dynamic change of the invoice system, and another the arrival of new clients in the global architecture. The architecture is self-contained and evolves in response to external stimulus .

Unplanned dynamic evolution scenario. The initial restocking process no loger fits the requirements. We show how the architect can improve the restocking process by adding a new supplier and by modifying the restocking request process. This evolution will both modify the structure as well as the intern behaviour of components.

4 PLANNED DYNAMIC EVOLUTION

We consider in this section dynamic evolutions that were planned by the system architect, while architecting the system. ArchWare C&C-ADL offers mechanisms for representing such evolving architecures (Cîmpan *et. al.*, 2005). The language continues and improves previous language propositions for the description of dynamic architectures, such as Dynamic Wright (Allen *et. al.*, 1998) or π-Space (Chaudet *et. al.*, 2000).

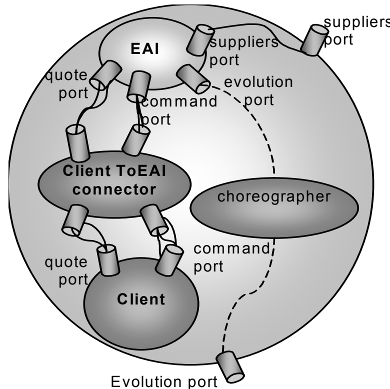

Figure 1: The SCMS global architecture.

In our scenario the clients comunicate with the EAI component either for demanding a quote for a product (using their respective quote port), either for passing an order for a product (using their respective order port).

As it is represeted by a composite component, the supply chain has a special architectural element, the choreographer, which handles its evolution. The choreographer is implicitly connected to all components and connectors in the composite. Its ArchWare C&C-ADL description is given later.

Two planned dynamic evolutions are considered here: (1) the integration of a new invoice system, intern to the EAI and (2) new clients join the supply chain (transparently for EAI and the existing clients).

First planned dynamic evolution. The EAI component entails four components, dedicated to the management of respectively quotes, orders, stocks and invoices, connected as shown in Figure 2. The invoice handler is intern to the composite, the other components being attached to composite ports. The connectors among components are basic, and not represented here.

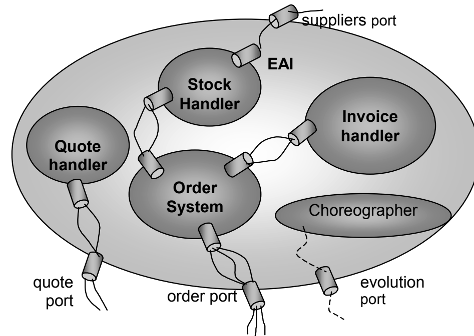

Figure 2: The EAI composite component.

Let us have a look at the ArchWare C&C-ADL definition of the order component, which is an atomic component with three ports. (cf. Figure 3).

```
OrderHandler is component with{
    ports {
        stockP is InvoiceDemandPort;
        invoiceP is InvoiceDemandPort;
        orderP is OrderPort;}
    configuration { new stockP ; new stockP; new
orderP }
    computation {
      via orderP~orderReq receive product:String,
quantity:Integer;
      via stockP~invoiceReq send product,
quantity ;
      via stockP~invoiceRep receive  ack: String;
      via orderP~orderRep send  ack;
      if (ack=="OK") then {
        via invoiceP~invoiceReq send product,
quantity ;
        via invoiceP~invoiceRep receive  invoice:
String;
        via orderP~invoice send  invoice
      }
      recurse
    } }
```

Figure 3: Order system atomic component.

The orderP port is connected to one of the composite ports, and allows the reception of orders from clients, containing both a product identification and a desired quantity. The stockP port allows the order system to ask the stock handler to check if the product is available in the required quantity. The stock handler confirms the resource availability or indicates it's unavailability. The message is transmitted to the client. If the product is available, the order system asks the invoice handler (to which it is connected via the the invoiceP port) an invoice, which once received is sent to the client. This behaviour is repeted recursively.

The ArchWare C&C-ADL EAI composite component definition is presented in Figure 4. Every component in a composite is potentially dynamic, several instances can be created at runtime. The **constituents** part entails the declaration of different component types, an instance of each being created in the **configuration** part. The later entails equally the expression of different attachements among components.

```
EAI is component with{
   ports {
       erpOrderP is OrderPort;
       eaiQuotationP is QuotationPort;
       eaiEvolveP is EAIEvolutionPort;    }
   constituents {
    orderComponent is OrderHandler;
    invoiceComponent is InvoiceHandler;
    stockComponent is StockHandler;
    quotationeComponent is QuoteHandler}
   configuration {
    new orderComponent; new invoiceComponent;
new stockComponent; new quotationeComponent;
    attach orderComponent~orderP to eaiOrderP;
    attach quotationComponent~quotationP to
eaiQuotationP;
    attach orderComponent~stockP to
stockComponent~stockP;
    attach orderComponent~invoiceP to
invoiceComponent~invoiceP;  }
   choreographer {
    via eaiEvolveP~newInvoice receive
newInvoiceComponent:InvoiceSystem;
   detach orderComponent from invoiceComponent ;
   insert component newInvoiceComponent in
invoiceComponent;
   attach orderComponent~invoiceP to
invoiceComponent#last~invoiceP;
   via eaiEvolveP~ackP send "ok";
   recurse;
   }} – end of the meta-component EAI
```

Figure 4: EAI composite component.

As already mentioned, a composite evolution is handled by its choreographer. For the EAI composite, a port is dedicated to the reception of the evolution message (the evolution decision is taken elsewhere). The EAI composite is ready to evolve its invoice system, a new component version can be received via the evolution port. Once the new invoice component is received, the choreographer detaches the current orderComponent and the invoiceComponent. The newly received invoice handler is inserted as instance of invoiceComponent type. We recall that all components are dynamic, meaning that several instances can co-exist at runtime. The last instance can be addressed using the component type name followed by **#last**. Thus using invoiceComponent**#last** the choreographer attaches the new invoice component version to the order system.

Second planned dynamic evolution. We will see how new clients can dynamically join the supply chain, this evolution being handled by the SupplyChain choreographer. The arrival of a new client is completely transparent to the EAI component. The connector that links the clients to the EAI component also evolves, but this evolution will not be shown here.

```
SupplyChain is component with{
   ports {
       eaiEvolveP is EAIEvolutionPort;
       newClientP is ClientPort; }
   constituents {
    clientComponet is Client;
    eaiComponent is EAI;
    clientToEai is ClientToEAIConnector;}
   configuration {
    new clientComponet; new eaiComponent; new
clientToEai;
    attach clientComponent~orderP to
clientToEai~clientOrderP;
    attach clientToEai~eaiOrderP to
eaiComponent~eaiOrderP;
    attach clientComponent~quotationP to
clientToEai~clientQuotationP;
    attach clientToEai~eaiquotationP to
eaiComponent~eaiQuotationP;     }
choreographer {
choose {
  via eaiEvolveP~newInvoice
     receive newInvoiceComponent:InvoiceSystem;
  via eaiComponent~eaiEvolveP~newInvoice
     send newInvoiceComponent;
  via eaiComponent~eaiEvolveP~ack
     receive ack:String;}
or {
  via newClientP~createOut receive c : Client;
  insert component c in Client ;
  via clientToEai~newClientP~createIn send ;
  via clientToEai~newClient~createOut receive ;
  attach clientComponent#last~orderP to
    clientToEai~clientOrderP#last;
  attach clientComponent#last~quotationP to
    clientToEai~clientQuotationP#last;}
then recurse
}  } – end meta-component SupplyChain
```

Figure 5: Supply chain composite component.

The supply chain choreographer handles both the arrival of a new client, and the reception of a new version of the invoice handler (which is passed to the EAI component). Any new client is inserted in the composite as instance of the Client component type. A request for evolution is sent to the ClientToEAI connector, which will create in response two ports in order to allow the connection of the new client. Once the connector indicates that the new ports have

been created, the choreographer makes the attachement between the client and the newly created connector ports. The access to the last instance (for both the client and the connector ports) are made as previously using the **#last** suffix.

Using these two evolution examples we shown how ArchWare C&C-ADL allows the definition of dyanmicaly evolving architectures.

Besides the verification of component types, as in the reception of a new invoice component or a new client, other properties verification can be performed in order to ensure the global coherence for the system. Such properties can concern both structural and behavioural aspects. Examples of structural properties include components connectivity (no unconnected component), the existence of particular components (imposing the existence of at least one instance of invoice component is the EAI composite), etc. Behavioural properties concern the components or ports behaviour. In Figure 6, the property requestBeforeReplyOfOrderSystem states that an order system cannot send a reply befor receiving a request. This property has to be preserved after the system evolves (reception of the new invoice system, or new client).

```
requestBeforeReplyOfOrderSystem is property {  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} }   } ) }     }
```

Figure 6: Property ensuring the order between send and request.

The evolutions presented in this section have to be planned during the system architecture definition. In the following we will show how it is possible to handle unplanned dynamic evolutions.

5 UNPLANNED DYNAMIC EVOLUTION

Research activities leaded in the ArchWare project encompass a virtual machine able to interpret architectural descriptions coded using the core language ArchWare π-ADL (cf. section 1 above). In the following, the illustrating example is specified using the core language (and not in the C&C layer, as in the previous section). The ArchWare environment proposes specific components that allow managing such evolutions (Oquendo *et. al.*, 2004).

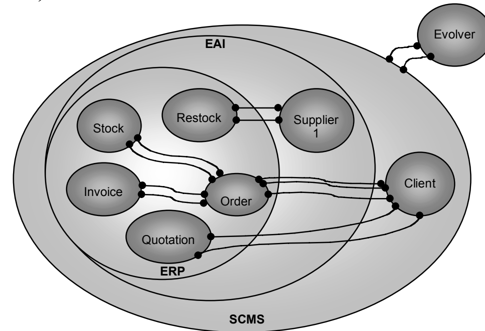

Figure 7: The initial architecture.

```
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();
    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);
   via restockingOrder1Req receive wares:String,
quantity:Integer;
   unobservable;
   via restockingOrder1Rep send "OK";
   done };
value quotationSystem is abstraction(Float:
price); { … }
value orderSystem is abstraction();{ … }
value stockingControl 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 stockingControl(stock)
        and restockingSystem() } };
value eai is abstraction(Float: price, Integer:
stock); {
    compose {       supplier1(20)
        and         erp(price, stock)
        } }
```

```
};
```

Figure 8: Initial architecture description (extract).

We do not present here all the tools and how they interact and cooperate together but we focuse on how the description language and its associated virtual machine allow such evolution to take place.

Before focusing on the architecture evolution process, let us recall that whenever a customer is asking for an order, if the desired product quantity is less that the corresponding stock quantity, a restocking request is emited to a supplier in order to satisfy the customer order request.

At the beginning, (cf. Figure 8), we assume that a sole supplier is involved and is always able to satisfy restocking requested by the restocking system. (cf. Figure 7). Let us imagine now that this supplier is no longer able to satisfy restocking requests, or the restocking manager has decided to change the restocking process by involving more than one supplier. If the desired restocking quantity exceeds the initial supplier (named `supplier1`) restocking ability, a second request is then addressed to another supplier (named `supplier2`); the second restocking request quantity is computed by subtracting the quantity the `supplier1` is willing to provide to the customer's initial request. This evolution scenario is interesting: on one hand it implies changes in the system architecture structure by adding a new supplier (cf. Figure 9); on another hand, it enforces the dynamic change of the restocking process for taking into account that a restocking request may not be satisfied; in this case, a new supplier joins the architecture and the initial restocking request has to be split among the two suppliers. The system behavior has to be dynamically changed according to the new configuration and process (cf. Figure 9).

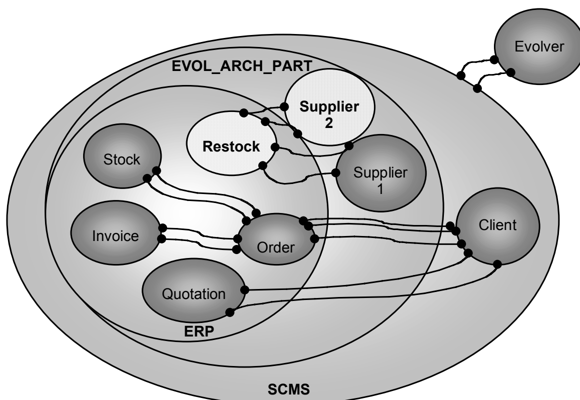

Figure 9: The architecture after evolution.

Let us see now how the evolution is managed at the architectural code level and enacted by the virtual machine. The architectural element named `evolver` (cf. Figure 10) is notified (via the `evolReq` connection) by the SCMS abstraction as soon as an evolution (an architectural change) is mandatory.

```
value evolver is abstraction(); {
    value evolReq is free connection();
    value evolRep is free connection(Boolean,
abstraction);
    via evolReq receive;
    choose {{via evolRep send false, any();}
    or  {value evol_arch_part is ARCH-EVOLUTION;
        via evolRep send true, evol_arch_part ;}}
};
value scms 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) }
      }
     andclient("rollings", 12)}
};
-- The SCMS abstraction that is the entire system
architecture
value scms_arch is abstraction(); {
   compose {    scms()
        and evolver()}
};
```

Figure 10: The evolver and scms abstractions containing evolution mechanisms and managing evolution process.

The way the architectural elements are organized is quite important regarding the evolution: because the `evolver` is attached to a specific abstraction, only this abstraction (and all of its sub-abstractions) may evolve. If it is unknown (or difficult to anticipate) on which abstraction the evolution will occur, one should attach the evolver to the abstraction that contains the entire system (the root abstraction). The side effect of attaching the `evolver` to the root abstraction is that for a small evolution (implying a very small part of the system), all of the architecture has to be expressed again in the evolution abstraction (called `ARCH-EVOLUTION` in the following and in the code examples) as shown later.

Notice that we do not make any assumptions about the nature of the evolution and what really the changes and the evolution policy will be. However, we have to specify where the evolution may occur. In the previous architectural description (cf. Figure 10), when the evolver is requested, either (`choose` in the piece of code) the architect decides that no evolution/change is required (*i.e.*, the `evolution` parameter received on the connection is equal to `false`), either the architect asks for an evolution (*i.e.*, the `evolution` parameter received on the connection

is equal to true). In the latter case, the virtual machine looks for a specific architectural abstraction (called ARCH-EVOLUTION), by dynamically loading an ARCH-EVOLUTION.adl file. Thus it is up to the architect to define the evolution and to produce the ARCH-EVOLUTION.adl file when it is required. The ARCH-EVOLUTION abstraction is first sent on the evolRep connection, and then dynamically applied: the evolved architecture currently behaves as the one integrating the changes expressed in the ARCH-EVOLUTION abstraction (cf. architectural abstraction 7). The evolved architecture now contains a new supplier (called supllier2) and the restocking process has changed accordingly. Notice that the new supplier (supplier2) does not behave as the existing supplier (supplier1) (*i.e.*, it is not the same) because the evolution we introduced (cf. section 3) 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 has now to take into account the existence of a new supplier, and the initial demand may be split (according to the quantity requested) between two suppliers (cf. Figure 11). The eai abstraction has been replaced by the evol_arch_part abstraction (corresponding to the ARCH-EVOLUTION described in the ARCH-EVOLUTION.adl file), integrating all architectural changes (cf. Figure 11). The transformation from the initial architecture (without evolution) and the evolved one (integrating the changes) takes place in the scms abstraction (cf. Figure 10), by using the evolReq and evolRep connections as we saw. These mechanisms allow several evolution strategies, each coded in a specific abstraction expressed in the evolver architectural element.

```
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 stockingControl(stock)
        and restockingSystem()
}
-- The evolved abstraction implying two suppliers
value ARCH-EVOLUTION is abstraction(Float:price,
Integer: stock); {  compose {   erp(price, stock)
        and supplier1()
        and supplier2(20)   }
};
```

Figure 11: ARCH-EVOLUTION.adl architectural description (extract).

Dynamically changing an architecture may cause drawbacks (inconsistency, lost properties). In order to limit their importance, architectural constraints can be defined and the architecture can be analyzed whenever the architect wishes. Such constraints and properties are expressed using the ArchWare AAL language (Alloui *et. al.*, 2003). Details on the properties checking and architecture validation and verification can be found in (Alloui *et. al.*, 2003, Alloui *et. al.*, 2005). Remember that the modified architecture can be checked against the initial architecture identified properties. If such verification succeeds, changes can be applied by dynamically evolving the architecture as presented.

6 CONCLUSION

Enterprise information systems require evolution in order to support enterprise activities, to adapt to market changes and enterprise evolutions. In our scenario, we illustrated 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).

Architecture evolution support is an important issue (Andrade and Fiadeiro, 2003). Current researches in this area are concentrated either on low abstraction levels (implementation), either on a very high levels (traditionnaly called conceptual level).

Firstly, we claim that the ADL has to support evolution: architecture evolution has to be expressed using the language itself. Few ADLs have such features (Darwin (Magee *et al.,* 1995), π-Space (Chaudet and Oquendo, 2000), Piccola (Nierstrasz and Achermann, 2000)), while most of them are based on process algebras. Secondly, the architecture descriptions have to be enactable and on the fly change mechanisms have to be provided. This latter point is very important. Most research activities focusing on architecture evolution can only address static evolution because they are not based on an adequate ADL: changes on architectures are made either on abstract architectures (Egyed and Medvidovic, 2001), either directly in the code (cf. (Pollet *et al.,* 2007) for a good survey on research leaded in this topic). In this context, the consistency between the two abstractions levels becames an important issue (Oreizy *et al.,* 1998, Garlan *et. al.*, 2003, Carriere *et. al.*, 1999, Erdogmus, 2000, Van der Hoeck *et. al.*, 2001, Aldrich *et. al.*, 2002, Pinzger *et al.*, 2004, Rank, 2005, Roshandel *et. al.*, 2004, Nistor *et. al.*, 2005).

The approach presented in this paper proposes the following interesting features:

- ArchWare ADL is a formal high level Architectural Description Language that covers structural and behavioural architecture description; it is also an interpretable language (accompanied by a virtual machine) with specific evolution support mechanisms;
- The ArchWare development process is shorter than most of the software-intensive system development processes: conceptual (or abstract) level and implementation are unified. Thus, there is no need to manage consistency between them avoiding gaps and discrepancies: the executing system architecture is the specified one;
- The ArchWare ADL formal foundations allow the architect to formally describe enterprise information systems, to check the system architecture;
- The evolution is managed directly and dynamically at the architectural code, each change consequence(s) on the architecture being verifyied before being applied.

In our illustrating scenario, we showed that the initial system is able to evolve dynamically by applying architectural abstractions that are unknown at the beginning (before system execution) but that can be formalized during system (architecture) execution. Evolution could be a cascading process for which changes may be applied on a previously modified architecture (with properties checking at each evolution step).

REFERENCES

Allen R., Douence R. , Garlan D., 1998. Specifying and Analyzing Dynamic Software Architectures, *In Proceedings on Fundamental Approaches to Software Engineering*, Lisbon, Portugal.

Alloui I., Garavel H., Mateescu R., Oquendo F., 2003. The ArchWare Architecture Analysis Language. *ArchWare Deliverable D3.1b*.

Alloui I., 2005. Property verification and change impact analysis for model evolution, *1ères journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, Paris.

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

Andrade L.F., Fiadeiro J.L., 2003. *Architecture Based Evolution of Software Systems*. LNCS 2804: 148-181.

ArchWare Consortium, 2001. The EU funded IST–2001–32360 ArchWare – Architecting Evolvable Software - project : http://www.arch-ware.org

Barais O., Lawall J., Le Meur A-F., Duchien L., 2005. Providing Support for Safe Software Architecture Transformations. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA),* Pittsburgh, USA.

Blanc dit Jolicoeur, L., Braesch, C., Dindeleux, R., Gaspard, S., Le Berre, D., Leymonerie, F., Montaud, A., Chaudet, C., Haurat, A., Théroude, F., 2002. Final Specification of Business Case 1, Scenario and Initial Requirements. *Deliverable D7.1b, ArchWare project.*

Carriere S., Woods S., Kazman R, 1999. Software Architectural Transformation. In *6th Working Conference on Reverse Engineering*, IEEE Computer Society.

Chaudet C., Oquendo F., 2000. π-SPACE: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems**,** *15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble - France.

Cîmpan S., Verjus H., 2005. Challenges in Architecture Centred Software Evolution, *CHASE: Challenges in Software Evolution*, Bern, Swizerland.

Cîmpan S., Leymonerie F., Oquendo F., 2005. Handeling Dynamic Behaviour in Software Architectures. In *the European Workshop on Software Architectures,* Pisa, Italy.

Cook 2006 http://www.iam.unibe.ch/~ducasse/Research/Cook/index.html

Demeyer S., Ducasse S., Nierstrasz O., 2002. *Object-Oriented Reengineering Patterns*, Morgan Kaufmann, 1-55860-639-4.

Ding L., Medvidovic N., 2001. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution, In *2001 Working IEEE/IFIP Conference on Software Architectures (WICSA-2)*, Amsterdam, Netherlands.

Egyed, A., Medvidovic, N., 2001. Consistent Architectural Refinement and Evolution using the Unified Modeling Language, In *1st Workshop on Describing Software Architecture with UML, co-located with ICSE 2001*, Toronto, Canada.

Erdogmus, H., 1998. Representing Architectural Evolution, In *Proceedings of CASCON '98*. Toronto, Ontario, Canada.

Garlan D., Cheng S.-W., Schmerl B., 2003. Increasing System Dependability through Architecture-based Self-repair. In *Architecting Dependable Systems*, R. de Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag.

Lehman M. M., 1996. Laws of Software Evolution Revisited, In *European Workshop on Software Process Technology*.

Magee J., Dulay N., Eisenbach S. Kramer J., 1995. Specifying Distributed Software Architectures. In *5th European Software Engineering Conference (ESEC '95)*, Sitges, LNCS 989.

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.

Medvidovic N., Taylor R.N., 2000. A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70-93*.

Mens T., Buckley J., Rashid A., Zenger M., 2003. Towards a taxonomy of software evolution, In *Workshop on Unanticipated Software Evolution*, Varsovie, Poland.

Mens T., Wermelinger M., Ducasse S., Demeyer S., Hirschfeld R., 2005. Challenges in software evolution, In *8th International Workshop on Principles of Software Evolution*. IEEE Press.

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

Nistor E., J. Erenkrantz, S. Hendrickson, and A. v. d. Hoek, 2005. ArchEvol: Versioning Architectural-Implementation Relationships, In *12th International Workshop on Software Configuration Management (SCM05),* Lisbon, Portugal.

Nierstrasz O., Achermann F., 2000. Supporting Compositional Styles for Software Evolution, *International Symposium on Principles of Software Evolution*, IEEE, Kanazawa, Japan.

Oquendo F., Alloui I., Cîmpan S., Verjus H., 2002. The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics. *ArchWare Deliverable D1.1b*.

Oquendo F., Warboys B., Morrison R., Dindeleux R., Gallo F., Garavel H., Occhipinti C., 2004. ArchWare: Architecting Evolvable Software. *In First European Workshop on Software Architecture (EWSA 2004)*, St Andrews - UK.

Oreizy P., Medvidovic N., Taylor R., 1998. Architecture-based runtime software evolution, *In Inernational Conference on Software Engineering*, Kyoto, Japan.

Pinzger M., Fischer M., Gall H., 2004. Towards an Integrated View on Architecture and its Evolution, *Elsevier Electronic Notes in Theoretical Computer Science*, Rome, Italy.

Pollet D., Ducasse S., Poyet L., Alloui I., Cîmpan S., Verjus H., 2007. Towards A Process-Oriented Software Architecture Reconstruction Taxonomy, In *11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, Amsterdam, the Netherlands.

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.

Rank S., 2005. Architectural Reflection for Software Evolution, In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2005)*, held at ECOOP, Glasgow, UK

Ratcliffe O., Cîmpan S., Oquendo F., 2005. Case study on architecture-centered design for monitoring views at CERN, In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005),* Pittsburgh, Pennsylvania, USA.

Revillard J., Benoit E., Cîmpan S., Oquendo F., 2005. Architecture-centric development for Intelligent Instrument Design, *IEEE Int. Conf. on Computational Intelligence for Measurement Systems and Applications (CIMSA 2005),* Giardini Naxos, Italie.

Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., 2004. Mae: A System Model and Environment for Managing Architectural Evolution, *ACM Transactions on Software Engineering and Methodology*, Vol. 13, Issue 2, pages 240-276.

Tibermacine C., Fleurquin R., Sadou S., 2005. Preserving Architectural Choices throughout the Component-Based Software Development Process. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05),* Pittsburgh, Pennsylvania, USA.

Verjus H., Cîmpan S., Alloui I., Oquendo F., 2006. Gestion des architectures évolutives dans ArchWare, *1ère Conférence francophone sur les Architectures Logicielles (CAL 2006)*, Nantes, France.