USING AN ARCHITECTURE-CENTRIC APPROACH FOR FORMALIZING AND DEPLOYING SERVICES ORIENTED ARCHITECTURE

Hervé VERJUS *, Kamel MANSOURI **, Mohamed Salah KHIREDDINE **

* : LISTIC - ESIA Lab., University of Savoie, France.

herve.verjus@esia.univ-savoie.fr

**: L.R.P. Lab., University of Batna, Algeria.

Mansouri_kl@yahoo.fr

mskhireddine@caramail.com

Keywords : COTS-based system, software architecture, styles, services oriented architecture.

ABSTRACT

The development of big software applications is oriented toward the integration or interoperation of existing software components (like COTS and legacy systems) [11]. This tendency is accompanied by a certain number of drawbacks for which classical approaches in software composition cannot be applied and fail. COTS-based systems are built in ad-hoc manner and it is not possible to reason on them no more it is possible to demonstrate if such systems satisfy important properties like Quality Of Service and Quality Attributes.

The recent works issued in web field allow the definition and the use of a complex web service architecture [12]. Languages such as WSFL [13], XLANG [14] and BPEL4WS [15] support these architectures called Services Oriented Architectures. The definition of software systems using these languages benefits some existing technical solutions such as SOAP [16], UDDI [17], etc., that permit the distribution, the discovery and the interoperability of web services.

However, these languages do not have any formal foundation. One cannot reason on such architectures expressed using such languages: properties cannot be expressed and the system dynamic evolution is not supported. On the other hand, software architecture domain aims at providing formal languages for the description of software systems allowing to check properties (formal analyses) and to reason about software architecture models.

The paper proposes a formalisation of COTS-based system (their structure, their behaviours) using architectural styles. The ADL used is π-ADL (based on the π-calculus, supporting style description). The paper will also present our approach consisting in refining an abstract architecture to an executable and services-oriented one.

1. INTRODUCTION

Information systems are now based on aggregation of existing components that have to cooperate in a precise manner in order to satisfy user needs and software functionalities.

Information systems are more and more complex, need more and more functionality provided by several software applications that already exist (COTS or legacy systems). Reusing and assembling existing components (COTS or/and legacy systems) are questions that cope with some difficulties that are not covered by classical component-based programming solutions like EJB, COM+, CCM, etc. As these are specifications for components development, they do not address the case of COTS-based systems, where source code is not available or/and has been previously developed with other specifications and programming languages. The EAI (Enterprise Application Integration) domain provides integration models and techniques for assembling heterogeneous software applications in a pragmatic way. EAI emerging solutions encompass (1) a distributed architecture using web services and (2) a description of the web services centric architecture, expressed using a web services orchestration/choreography language (i.e. XLANG, WSFL, BPEL4WS, etc.). Information systems based on such technology integrate heterogeneous software components, COTS, using a process-based integration approach, where the process description has to insure the execution correctness of the system. Such information systems, building from COTS, will be called COTS-based systems in the following.

In such context, an issue is still open: the adequation between the information system provided (i.e.

its composition) and the functionalities it would able to provide (i.e. to the end user). Because EAI solutions fail in insuring that the information systems provided, succeeds in end-user needs satisfaction.

This paper presents a first attempt in formally describe an information systems building from COTS (or legacy systems). The approach used is based on a architecture-centric development process where the system description is the heart of the process. Using such approach, the (abstract) description can be checked, refined in order to obtain more concrete descriptions that will be executed. In our case, the concrete description (the one that has to be executed) is expressed in a commonly used language dedicated to web services systems description. We assume in this paper that COTS can interoperate as web services.

The paper will first present our approach. In section 3, we introduced our reference architecture for building COTS-based systems. This architecture is based on web services. Then, we will introduce part of the formal description for describing such systems in section 4. We will briefly present the code generation (section 5) and we will then conclude in section 6.

2. FROM A SOFTWARE ARCHITECTURE SPECIFICATION TO A SERVICES ORIENTED ARCHITECTURE

The architecture of a software system defines the elements that compose the system, and how they interact. The software architecture definition can be made informally, or by using a dedicated language. Different abstraction levels are considered for describing the software architecture. The use of formal architecture refinement guarantees the preservation of properties specified at abstract levels all the way towards architecture implementation.

The architecture centric development process (see figure 1) aims at providing means for defining software systems at a very abstract level. Such descriptions can be then validated in order to check systems properties and are refined in a more concrete description (that allows to deploy the system in a concrete environment).

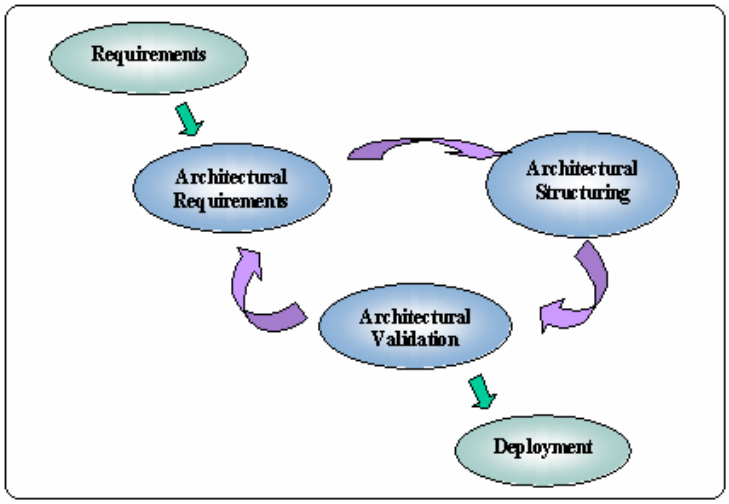

Figure 1 - Architecture centric development process

We decided to describe information systems using a formal language (a textual language). The formal description can then be refined in order to obtain a concrete representation. We have to manage the concrete representation generation starting from a formal one. During the generation step, all of the system properties have to be preserved. The targeted representation use web services as integration technology among our components: the COTS.

In our concrete architecture, web services are used as COTS facets, which allow them to interoperate (figure 2). In such concrete context, all well-known languages (WSFL, XLANG, BPEL4WS, etc.) and technologies (WSDL, SOAP, etc.) may be candidates for supporting the deployment and the execution of our systems using web services.

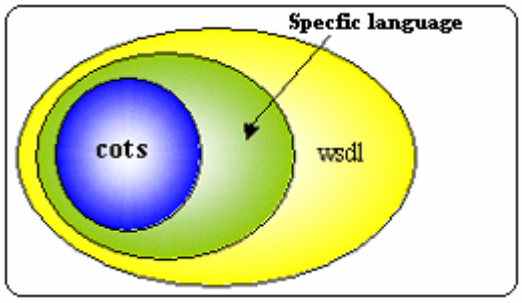

Figure 2 - COTS wrappers

In order to (1) support COTS-based information systems (particularly required control when building a system from COTS [24]) and (2) to take into account web services as implementation technology, we defined a particular services oriented architecture (SOA) that will be presented in the next section.

3. A SERVICE ORIENTED ARCHITECTURE FOR BUILDING COTS-BASED SYSTEMS

The architecture of a software system defines the elements that compose the system (i.e. often called "components"), and how they interact in order to satisfy the system requirements [18].

Research in this domain does not offer architecture that can be taken as a reference for a designed COTS based system, especially when dealing with several constraints such as [9]:

- **heterogeneity**: COTS are heterogeneous;
- **autonomy**: COTS must participate in the system and still remaining in their environments (local autonomy) if needed. Even, information managed by COTS can be locally confidential;
- **evolution**: COTS can be solicited in different ways. They can leave the system, or participate in at any moment according to the process requirements;
- **cooperation**: COTS are brought to cooperate in order to reach the system's goals;
- **control**: operational mode of the system (the manner for managing cooperation) has to be controlled and flexible;
- **distribution**: COTS can be distributed on a large scale.

For satisfying the above-stated clauses consistency, we propose the following assumptions:

- COTS are considered as black boxes that do not share any external resource.
- COTS are considered as services suppliers [4].
- COTS are autonomous and have no knowledge of other COTS availability, nor they have information on their environment.
- COTS have no idea about the objective for which they are going to participate.

Dealing with our objectives as well as with the COTS features, we define a reference architecture. This architecture is a services oriented architecture that is deducted from a reference architecture for building COTS-based federations (in which the control among COTS can be tuned) [10, 24].

From the implementation point of view (i.e. the concrete representation), the architecture is basically a set of web services interacting together. From the abstract point of view, the previously shown architecture is inherited from the one presented in [24] for which COTS orchestration (i.e. choreography in SOA) is an important topic.

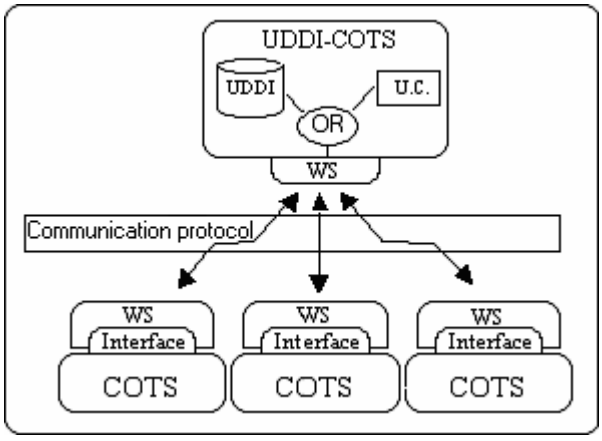

Figure 3 – The reference architecture

We restricted the architecture presented in [4][9][23] in order to simply our study (i.e. the part called *Control Foundation* is the one that we reuse in the architecture shown in figure 3, with some extensions taking into account web services control needs).

In this architecture we propose a COTS-UDDI, that is composed of:

- **Publish policy service (UDDI)**: stores the information of all services offered by the COTS, in a neat manner, and permits, by calling functions, to get their positions. It may have services that achieve the same goal, in this case we assume that services are similar and we note S1=S2, these services are stocked in even level in UDDI.
- **Orientation service (OR)** : the unit of orientation is a process that manages interactions between the supplier and the requester of services; on the request, orientation service gets the adequate service, in cooperation with the UDDI. It sends then the request to the supplier and waits for its response. On error, the request is redirected.
- **Controls process service (UC)**: it schedules the main activities of the system (other activities are controlled at the level of every COTS). It is the services orchestration unit.

When a client (in terms of classical client-server architecture) needs a service, it sends its request to the O.R. which gets information about the service from UDDI (information that consists on supplier and the supplier's capabilities). On the response, the O.R. unit can either suspend or send its request to the supplier. At the end of the process, it even sends the answer to the client.

4. SYSTEM ARCHITECTURAL DESCRIPTION USING STYLES

In the case of complex COTS-based systems (like information systems) classical approaches fail:

- Industrial need systems that are adapted to their requirements: the design (including properties) of such systems is a crucial step but systems designs/models have to be validated before implemented.
- COTS are specific software components with which components classical integration patterns or idioms are not relevant: COTS have to be characterized as well as their integration (the "glue" has also to be formalized).

COTS-based system models (when existing) cannot be checked nor validated. That is, one cannot reason on models nor analysis can be made on such models. This lack of formalization has following consequences:

- design (of complex systems) expertise cannot be caught nor maintained;
- there is a gap and discrepancies between the design and the execution. It is impossible to guarantee that the execution will be conformant to the design;
- the COTS-based systems evolution (replacement, deletion, addition of COTS, changing system behaviour, etc.) is not well supported nor it can be validated;

- crucial properties (safety, completeness, consistency, etc.) of the systems are not taken into account.

The work on the software architectures formalisation proposes solutions that might meet the needs issued by the identified limitations.

The architecture-centric development process we propose (see figure 4) is quite different to the classical software development process [20]: if the system behaviour does not fit the requirements, the architecture description can be modified without restarting entirely the development process. Representations (architectural descriptions) are also checked at every stage of the process before generating the code.

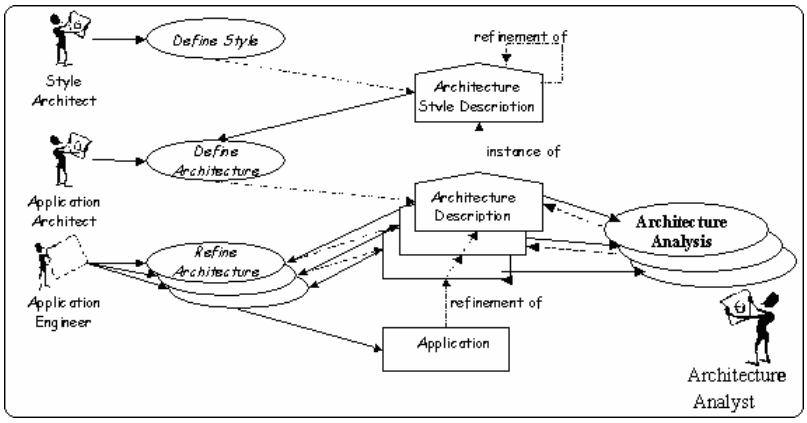

Figure 4 - Architecture centric development process

The work on architecture centric approaches for software development has been very fruitful during the past years, leading, among other results, to the proposition of a variety of Architecture Description Languages (ADLs) [28], usually accompanied by analysis tools. The enthusiasm around the development of formal languages for architecture description comes from the fact that such formalisms are suitable for automated handling. These languages are used to formalize the architecture description as well as its refinement. The benefits of using such an approach are manifold. They rank from the increment of architecture comprehension among the persons involved in a project (due to the use of an unambiguous language), to the reuse at the design phase (design elements are reused) and to the property description and analysis (properties of the future system can be specified and the architecture analyzed to check their verification).

The different ADLs proposed share some common concepts, especially on the way the structural aspects are treated. Thus components entail the functionality of the future systems and interact and communicate via connectors. Interfaces of components and connectors are structured as sets of ports. An architecture description is a configuration of such interacting components and connectors. Of course there are variations from one language to another, due to their historic evolution and to their purpose. Thus some languages are general purpose (ACME [7], π-Space [24]) while others are dedicated to a specific domain (like META-H [27], which is dedicated to the real-time multiprocessor avionics system architecture).

If the structural aspect are covered by all the ADLs, the behavior is handled by only some of them (like Wright [1], π-Space [24], CHAM [2]). Wright and π-Space are based on process algebra (CSP for Wright and π-calculus for π-Space), which allows the behavior description. The use of π-calculus in the case of π-Space leads to the possibility of describing dynamic architectures. A CHAM allows to specify behavior as a succession of chemical reactions in a chemical solution.

A central aspect of architectural design is the use of recurring organizational patterns and idioms – or architectural styles [5]. A number of benefits in using architectural styles are identified [5], such as design and code reuse, ease on system understanding, interoperability improvement, style-specific analyses an visualization.

Architectural styles are means for intensive design reuse. They provide a design framework to software architects. They are also means for guaranteeing that the architecture will exhibit certain properties. Thus knowing that a component or an architecture follows a particular architectural styles, induces that the considered component or architecture has all the properties ensured by the style.

Styles can range from very generic ones to very specific ones: the expression of a specific style is an architecture. The range of generality is achieved by building hierarchies of styles. Thus, a style can have sub-styles that enforce the constraints imposed by it, making the definition more specific.

A style represents a family of architectures that are compliant to the style. The more specific a style definition, the smaller the family of architectures it represents. In the case of completely specific styles this set has cardinality 1, i.e. the style represents a family of architectures with only one element.

Some of the existing ADLs propose mechanisms for style definition. It is the case for Aesop [6], Armani [8], Acme [7], Unicon-2 [3], σπ-Space [25] and π-ADL [20].

Among these languages, π-ADL is the only one that :

- allows the architecture structural modeling as well as the behavioral description (as an extension of π-calculus);
- supports properties/constraints definition;
- supports dynamic evolution of the systems;
- proposes styles mechanism.

Basically, a style is instantiated by architecture descriptions; at instantiation all the constraints have to be verified. Sub-styles can be defined, by adding constraints to an existing one. For formalizing the particular SOA (our reference architecture) we propose, we defined dedicated styles inherited from the component/connector style already defined [21]. The following figure (figure 5) presents an abstract of our styles library definition. It concerns the orientation unit introduced in the previous section.

OR_component **is style extending** (component) **where**
Cammen --! In this code segment, all declared ports are necessarily used.
Ports
Requested_port_service : In_P_Requested_ service
Response_port_service : Out_P_Response_service
Requested_port_service_from_supply:
Out_P_Requested_service
Response_port_service_from_supply: In_P_Response_service
Requested_port_service_info: Out_P_Requested_service_info
Response_port_service_info:In_P_Response_service_info
Free_port_service : Out_P_OR_free_service
notification_port : In_out_P_OR_notification
attributes
constituents
constraints
forall(p |p **in type** In_P_Requested_ service; q |q **in type** Out_P_Response_service
implies
every sequence {true*.< via p send any>.(not< via q resive any >)*.< via p send any>.true*} !– assure that it would not have answer without services
leads to state{false}) !- demand beforehand.
!– assure that we cannot free a non allocated service.
forall(p |p **in type** In_P_ Response_service_info; q |
q **in type** Out_P_OR_free_service
implies every sequence {true*.(not<via p send any>).< via q send any >.true*} leads to state{false})
attachments

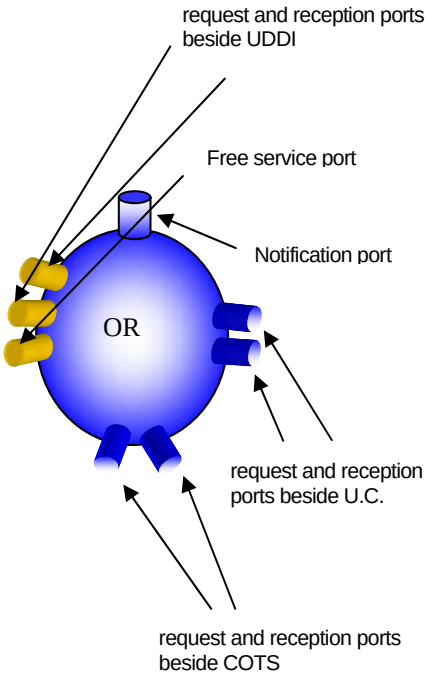

Figure 5 - Orientation unit description

In the π-ADL code fragment above,

- in *cammen* section, static ports are defined for OR unit (whereas, port types are declared somewhere in the code);
- in *constraints* section: a set of properties for the OR unit (such properties will then be checked). Such property expresses that OR provides responses to a client only if it has previously sent requests. This property is expressed in AAL [22] (based on temporal logic) that is encapsulated in π-ADL styles mechanisms for properties expression.

The figure 5 is an extract of the complete π-ADL styles[1] library for describing COTS-based systems according to the architecture presented in section 3.

Starting from the described styles, software designer is now able to define architectures following these styles. It means that an architecture referencing our architectural styles inherits all properties defined in the styles; such architecture can be checked according the styles definition and then may be instantiated. Styles may be also specialized in sub-styles. In such case, sub-styles inherit again all of the properties of their super-styles.

The next section will show the refinement step consisting in generating implementation code (in order to execute instantiated architectures) from a π-ADL specification (a formal description of a COTS-based architecture using our architectural styles).

5. FROM π-ADL DESCRIPTION TO XLANG CODE GENERATION

A π-ADL description (a specification) cannot be directly executed. As we presented in sections 2 and 4, such specification has to be refined to an implementation architecture (figure 6). The targeted and executable architecture is a specific services oriented architecture for building COTS-bases systems. Such web services architecture can also be described using classical web-centric languages such as WSFL, XLANG, BPEL4WS, etc.

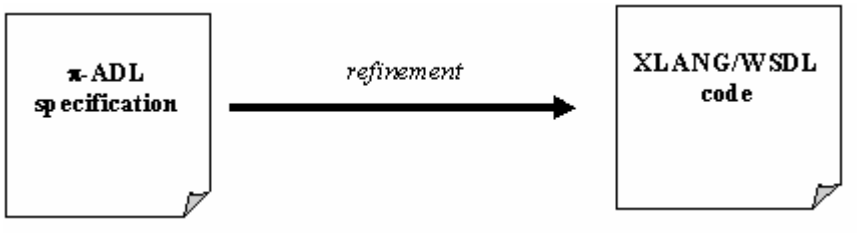

Figure 6 – Refinement step

We aim at generating XLANG and WSDL code using rewriting rules. Such rules support the transformation from a π-ADL specification to XLANG and WSDL code. As example, we focus on the Unit Control. The rewriting rules are presented:

5.1. XLANG generation code rules:

Inaction behaviour: the empty process does not contain any actions. It plays a role rather like the null statement of ordinary programming languages. The rule of transformation from π-ADL to XLANG is:

{Done} → <empty/>

Ex:{done} **Ex** :<empty/>

[1] The complete π-ADL styles definition can be found in [26].

Sequence action: A sequence contains zero or more actions or processes which are executed sequentially. The sequence concludes when its final action or process is terminated. The rule of transformation from π-ADL to XLANG is:

{(action.)*action} → sequence ::= sequence [action | process]*

Ex :

```
{
dec_Speedometer .
Sleep
}
```

Ex :

```
<xlang:sequence>
 <xlang:action operation=" dec_Speedometer " activation="true"/>
 <xlang:action operation="Sleep"/>
</xlang:sequence>
```

Switch action : The switch process supports conditional behaviour. The branches of the switch are considered in the order in which they appear. The first branch whose condition holds true provides the process for the switch. If no branch with a rule is taken, then the default branch will be proceeded. If the default branch does not exist, a default branch with an empty process is deemed to be present. The switch is completed when the process form of the selected branch completes.

Case { project_list } →

project_list ::= variant_project_list

| union_project_list

| any_project_list

variant_project_list ::= identifier do clause [or identifier do clause]* or default do clause

union_project_list ::= type do clause [or type do clause]* or default do clause

switch ::= switch branch* default?

default ::= default process

branch ::= branch case process

case ::= case Qname

Ex :

```
behaviour {via x receive y : String.
case { y= "A" do {….}
or y=B do {…}
}
```

Ex :

```
<switch xmlns:y="http://…/y">
 <branch> <case> y:A </case>
  <sequence> … </sequence>
 </branch>
 <branch> <case> y:B </case>
  <sequence> … </sequence>
 </branch>
</switch>
```

Composition process : The Composition process executes each of the specified sub-processes concurrently.

compose { [composition_list]* } → all ::= all process*

Ex :

```
compose {
behaviour X1{} and
behaviour X2{}
 }
```

Ex :

```
<all>
 <sequence> <!-- behaviour X1 --> </sequence>
 <sequence> <!-- behaviour X2 --> </sequence>
</all>
```

As XLANG code mainly expresses services orchestration (i.e. basically control flow in our previous

transformation rules), WDSL code states for service interfaces definition. Some rules for generating WSDL code are shown in the following.

5.2. WSDL generation code rules:

In order to simply the transformation, we assume that behaviour express no more than one service.		
Value ID behaviour { [clause] }	→	<definition> [Clause]<service name = ''ID''>[Clause]</service></definition>
Ex:		**Ex**:
Type x **is** y		<Types> <xsd : element name =x type=''xsd : y''name = ''x'' </Types>

When XLANG and WSDL pieces of code have been generated, the concrete architecture can now be executed by deploying it, using a specific web-services application server (like BizTalk, etc.). From the COTS side point of view, each COTS has to be integrated with their corresponding web services.

6. CONCLUSION AND ONGOING WORK

Building COTS-based system generally fails due to non formal approaches (often ad-hoc solutions like EAI) used. In [4], [23]and [10] we claim that designing and building COTS-based systems addresses lots of issues: the gap between the design level and the implementation one is one of them. Because COTS (as well as legacy systems) already exist, it is not possible to generate all of the software system code but we have to deal with the "glue" between such software components (COTS, etc.). Instead of managing an ad-hoc approach (i.e. rewriting approach, etc.) consisting in refinement steps from specification to implementation code generation, we have to focus on the "glue" that have to guarantee the properties of the COTS-based system the designer is interested in. Our approach is divided in two parts:

the definition of an architecture that is convenient for the design of COTS-based systems as well as it is also closed to a concrete architecture (in our case, a Services-Oriented Architecture);

an architecture-centric development process using a formal ADL as a specification language. Such language allows us to define our reference architecture in architectural styles. COTS-based systems architectures are expressed using these styles and can be validated against properties (both structural and behavioural). The (abstract) architecture of a COTS-based system is then refined in order to produce an implementable SOA that is entirely compliant and checked with the abstract architecture that is expressed in XLANG and WSDL piece of code. But the behavior of COTS-based system largely depends of each COTS involving in the system; that is, it is not possible to reason on each of the architecture components and one cannot demonstrate all of the properties a designer is interested in (completeness, safety, security, vivacity, etc.) but only a sub set of them.

We will focus on services composition at a high level of abstraction (composition properties will be formally studied) that would be permit to formally and entirely specify SOA and to generate an executable architecture. In such case, we will be able to reason on each of web service as well as on their respecting properties (QoS) and on their composition. We also will to apply our approach in an industrial context like an information system deployed with an EAI solution.

REFERENCES

[1]. Allen, R. and Garlan, G. "A formal basis for architectural connection", *ACM Transaction on Software Engineering and Methodology*, 1997.

[2]. Wermelinger M., "Towards a Chemical Model for Software Architecture Reconfiguration", *Proceedings of the 4th International Conference on Configurable Distributed Systems*, 1998.

[3]. DeLine, R.: "Toward User-Defined Element Types and Architectural Styles", *Second International Software Architecture Workshop*, San Francisco, 1996.

[4]. Estublier, J., Verjus, H., and Cunin, P.-Y., "Building Software Federation", *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, Las-Vegas, USA, June 2001.

[5]. Garlan, D. "What is Style?", *Proceedings of Dagshtul Workshop on Software Architecture*, February 1995.

[6]. Garlan, D., Monroe, R. and Wile, D., “Exploiting style in architectural design environments”, *Proceedings of SIGSOFT'94 Symposium on the Foundations of Software Engineering*, ACM Press, December 1994.

[7]. Garlan, D., Monroe, R. and Wile, D., “Acme: an Architecture Description Interchange Language”, *Proceedings of CASCON'97*, November 1997.

[8]. Monroe, R., “Capturing Software architecture Design Expertise with Armani”, *School of Computer Science Carnegie Mellon University*, Pittsburgh, January 2001.

[9]. Verjus, H., “Conception et construction de fédérations de progiciels“, *PhD thesis* (in french), LLP-ESIA Lab., University of Savoie, Annecy, France, September 2001.

[10]. Verjus, H., Cîmpan, S., Telisson, D., "Formalising COTS-based federations using software architectural styles", in *Proceedings of the 15th International Conference Software & Systems Engineering and their Applications*, December 2-5, 2002, Paris, France.

[11]. Abts C.,"COTS Software, Integration and Usage Issues", Center for Software Engineering, USC 27 September 2000.

[12]. IBM Web Services Architecture Team, 'Web Services Architecture Overview', IBM, Septembre 2000.

[13]. Leyman F., 'Web Services Flow Language', IBM Software Group specification, Mai 2001.

[14]. Satish T., 'XLANG, Web Services for business process design', Microsoft, Mai 2001.

[15]. Curbera F. and All, “Business Process Execution Language for Web Services”, Ver1.0, IBM 31 July 2002.

[16]. Ehnebuske, D. and All, Simple Object Access Protocol (SOAP), Ver1.1, 08/05/2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[17]. Bellwood, T. and all, UDDI, Ver 3.0, 19/07/2002, http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[18]. Luckman, C., Vera, J., and Meldal, S., « Three Concept of system Architecture », 07/1995.

[19]. B.W. Boehm « A Spiral Model for Software Development and Enhancement ». ACM SIGSOFT Software Engineering Notes, vol. 11, No. 4/ 8/ 1986.

[20]. Oquendo F., Alloui I., Cîmpan S., Verjus H., The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics, ArchWare European RTD Project IST-2001-32360, Deliverable D1.1b, 12/ 2002.

[21]. Leymonerie F., Cîmpan S., Oquendo F., «ADL Foundation Style», 5/ 2003.

[22]. Oquendo F., Alloui I., Mateescu R.; Architecture Analysis Language, Project Deliverable D3.1, Ver 1.0, 31/1/2003

[23]. J. Estublier, H. Verjus, P.Y. Cunin, « Designing and Building Software Federations », *Proceedings of 1st Conference on Component Based Software Engineering, (CBSE)*, Warsaw, Poland, September 2001.

[24]. Chaudet, C. and Oquendo, F., “π-SPACE: A Formal Architecture Description Language Based on Process Algebra For Evolving Software Systems”, *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, Grenoble, France, September 2000.

[25]. Leymonerie, F., Cîmpan, S. and Oquendo, F., “Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux : Application à J2EE”, *14emes Journées Internationales Génie Logiciel & Ingénierie de Systèmes et leurs Applications*, Paris, December 2001.

[26]. Mansouri, K, « Définition de styles architecturaux pour le description de systèmes logiciels à base de composants de type COTS, selon une approche services web », master thesis (in french), University of Savoie, France, September 2003.

[27]. Binns, P., Engelhart, M., Jackson, M. and Vestal, S., “Domain Specific Software Architectures for Guidance, Navigation, and Control”, International *Journal of Software Engineering and Knowledge Engineering*, 1996.

[28]. Medvidovic, N., and Taylor, R.N., “A Classification and Comparison Framework for Software Architecture Description Languages”, *Technical Report UCI-ICS-97-02*, Department of Information and Computer Science, University of California, Irvine, February 1997.