Nimrod: A Software Architecture-Centric Engineering Environment
Revision 2 - Nimrod Release 1.4.3

Hervé Verjus
University of Savoie Polytech'Savoie - LISTIC Lab
B.P. 80439 - 74944 Annecy-le-Vieux Cedex - France
Phone: +33 (0)4 50 09 65 80
herve.verjus@univ-savoie.fr

Keywords: ADL, evolution, Nimrod, software architecture, formal language, object oriented programming language.

Abstract: The development of large software applications is complex. Object-oriented paradigm does not propose a suitable paradigm. ADLs offer new artefacts at different level of abstraction. Among other, components and connectors are commonly used artefacts for specifying large software-intensive systems. Few ADLs propose execution mechanisms and very few address software evolution. Nimrod is a new software architecture-centric approach with an ADL (named Nimrod ADL). Nimrod ADL aims at proposing appropriate level of abstraction for formalizing, executing and evolving large and complex software architectures.

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. But as market frequently changes, information system has also to evolve accordingly (new needs, new functionalities, new processes). Then, information system evolution is an important topic and issue.
During the last decade, software engineers, practitioners and researchers have been focused on software architecture (Shaw and Garlan, 1996). Software architecture encompasses software elements and their relationships at different level of abstraction (very abstract level, also called conceptual, and concrete level that is very closed to the code).
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 for validation purpose). Once the information system has been identified and formalized, the architect may reason on it (Alloui, 2005).
Several ADLs were proposed (Medvidovic and Taylor, 2000) that mainly focus on architecture design, at a high level of abstraction. In such context, managing the gap between abstract level and implementation level is an important issue. Our approach does not propose to distinguish both level but proposes to unify design and implementation by considering relevant ADL abstractions and by providing behaviour expression and execution mechanisms. Thus, the architecture designed is also the one that will be enacted.

As the evolution is often considered at the latter stages of software system development process (i.e. implementation, execution), mostly by adopting pragmatic approaches (Demeyer et al., 2002), it is not often studied in the earlier stages (design, modelling, specification). Software evolution (Lehman, 1996) would be studied at each software development process stage in order to notably reduce costs. The Nimrod approach allows the architect to dynamically evolve enterprise system architecture. As there is no abstract and implementation level separation, the de-

signed and evolved architecture will be the executed one.

The paper will first present in section 2 the ArchWare project and relative technologies that corresponds to the formal Nimrod's foundations. In section 3, we will present the Nimrod approach for formalizing, enacting and evolving software-intensive system architectures. The, section 4 will illustrate Nimrod environment through an example, while section 5 will conclude.

2 ARCHWARE ADL AS A BASIS

2.1 ArchWare architecture-centric approach

The architecture centric development process (see figure 1) aims at providing means for defining software intensive 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).

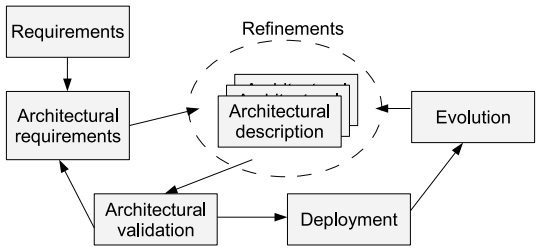

Figure 1: Architecture centric development process.

The ArchWare project (European IST-5 project, number 32360 - (Consortium,)) proposes an innovative architecture-centric software engineering framework, i.e. architecture description and analysis language, 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. 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, 1989), 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. (Cimpan 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) and is thus more complete and more expressive power than the most ADLs (Medvidovic and Taylor, 2000; Verjus et al., 2006). 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 (Cimpan et al., 2005), is proposed as a layer built on top of the core language.

2.2 The ArchWare environment

The ArchWare runtime framework (Morrison et al., 2004) includes an execution engine of architectures based on evolutionary processes of development, a refinement process of architecture description and mechanisms supporting the interoperability of the environment tools and components (that can be also COTS). Details of the ArchWare environment can be found in (ArchWare, 2001 and Oquendo et al., 2004). The ArchWare architecture centred tools provides supports for:

- the definition of the architecture,
- the validation of such architectures (using analysis tools and software graphical animation tool),
- the checking of the functional and extra functional properties of architectures,
- the refinement of architecture descriptions from an abstract level to a concrete level,

- the code generation of the systems in various programming languages (using explicit rules).

2.3 Architecture evolution support

One of the ArchWare environment key features is the evolution support ability (Cimpan and Verjus, 2005). On one hand, ArchWare ADL is the language allowing to describe evolvable architectures (i.e. architectures that can dynamically evolve); on the other hand, the ArchWare environment contains an ADL virtual machine (Morrison et al., 2004) that supports dynamic evolution (the architecture description code can be modified while being interpreted). Then, an architecture description can be dynamically changed and the runtime architecture change accordingly. When an architecture evolves dynamically, one may check the new architecture against properties or not (it is up to the architect).

3 NIMROD ADL AND ENVIRONMENT

3.1 Nimrod engineering approach and meta-model

Our purpose is to allow software practitioners to express software architecture at different abstraction levels, from a very abstract one to a more concrete. Nimrod[1] is a software architecture-centric engineering environment that targets software architecture formalization, execution, evolution and validation. At a first glance, Nimrod provides an ADL (Architecture Description Language) that is built with high level software architectural concepts. Architectures expressed in Nimrod ADL are executable. Nimrod ADL meta-model is built on top of the Nimrod root meta-model. The architectural Nimrod root meta-model we propose is simple and can be extended as needed. It has two main concepts that are *ArchitecturalElement* and *Relationship* (see figure 2). As the basic idea, a relationship links two architectural elements whatever the relationship is. We make no assumption about the kind of relationship the architect will need: it is an abstract concept and could be extended as needed. Starting from the Nimrod root meta-model, software architect can define models (containing architectural elements and relationships) or/and can define other meta-models from the Nimrod root meta-model (as consequences, other ADLs could be defined, all based on *ArchitecturalElement* and *Relationship* concepts - such Nimrod ADLs constitute what we call Nimrod ADLs family). The conceptual and layered definition of the Nimrod ADL and ADLs family follows a model-driven engineering approach.

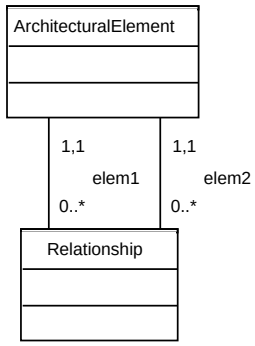

Figure 2: The Nimrod meta-model.

Arround this, specific architectural models can be defined from one of an already defined meta-models, built on top of the Nimrod root meta-model. Architectural meta-models can be expressed as needed (defining a vocabulary for each abstraction level the architect needs) and architectural models can be expressed at each level of abstraction using a particular meta-model. Any architectural model can be then validated against a specific meta-model (and, by extension, against a hierarchy of meta-models). The Nimrod meta-model definition and validation mechanisms are note detailed deeper in this paper.

3.2 Nimrod ADL

ADLs are often based on the component and connector artefacts (Shaw and Garlan, 1996; Allen et al., 1997; Medvidovic and Taylor, 2000). As it has been already wrote in several surveys ADLs consider and cover very differently software architecture: some ADLs only focus on structural aspects of architecture, some other only focus on architectural properties while some other covers structural aspects, with architectural behaviour and properties (Oquendo et al., 2002; Oquendo et al., 2004; Pourraz et al., 2006; Verjus et al., 2006). But ADLs fail in their usability: when an ADL is simple it lacks in many points (and is often used as an academia notation for expressing very simple architectures) and when an ADL is taking into account several architecture aspects (structure, behaviour, properties) it is too complex and is no more used. We then will focus on Nimrod ADL even if as we previously wrote, Nimrod allows the architect to express his own architectural meta-models and models according his needs and the

[1]This works is partially funded by the French ANR Cook project

abstraction level he wants to consider.

First idea: Nimrod has to be simple.
Nimrod ADL is not complex, neither it is too simple: Nimrod ADL is a software architecture meta-model defined using the Nimrod root meta-model. Its concepts are expressed in terms of ArchitecturalElement and Relationship.
Nimrod ADL contains few concepts:

- Component
- Connection
- CompositionRel(ationship)
- CommunicationRel(ationship)

Such concepts are sufficient in order to support:

- Architectural structure: what are the architectural components and how they are composed together.
- Architectural behaviour: how components communicate by using sending and reception of messages.

Components can be composites (when they are composing other sub-components) and have connections in order to communicate with other components. Sub-components are linked to their parent by the way of composition relationships (CompositionRel) that also define the scope of the components. In such approach, components are the compositional units of a Nimrod architecture Connections are linked together by the way of communication relationships (CommunicationRel). Their unification is done according (data/structure) types they support.

Second idea: Nimrod does not reinvent the wheel !
ADLs are interesting in order to describe abstractions (Medvidovic and Taylor, 2000; Oquendo et al., 2002) and architectural elements composition. They provide vocabulary that ease to understand system structure and communication among elements. But they also lack in some points: for instance, only structural architectures cannot be executed and ADLs that support architectural behaviour expression (Leymonerie et al., 2002) are not very suitable neither for supporting all of the most programming control structures (loops, conditional expressions, iterations, recursion) nor for defining variables, data types, collections. Some programming languages are more well adapted in such situation.

Nimrod has to be used for expressing, executing and evolving software architectures and combines strengths of ADLs with strengths of Object-Oriented Programming Languages (OOPL).

ADL strengths:

- Expressiveness
- Levels of abstraction
- Communication between components
- Composition (components as compositional unit)

OOPL strengths:

- Control structure
- Data types/Collections definition
- Dynamic languages (some of them, by the way of virtual machines)

Nimrod component is the entity that combines strengths of both fields as it is the compositional unit and is also the computational unit (where behaviour is expressed). Software architects define components that interact together, each having a behaviour (in the sense of π-calculus process (Milner, 1989)).

Nimrod has been implemented using Smalltalk, with respect to the π-calculus semantics (Nimrod is in somewhere closed to the ArchWare ADL (Oquendo et al., 2002; Oquendo, 2004). Smalltalk (Sharp, 1997) is a object-oriented programming language that is extensible and dynamic. Smalltalk code is interpreted using a virtual machine. Nimrod ADL syntax is very simple and intuitive. One of the core features of Nimrod is the ability to define and (re)use behaviour. Behaviours are implemented as specific Smalltalk processes (i.e. runnable BlockClosure instances). Each component has a behaviour definition that can be executed: each execution is a light-weight thread (in Smalltalk sense). According to the π-calculus semantic, a communication among two processes can be done when the first process is ready for sending a message while the second process is ready for receiving this message. Nimrod deals with connection unifications as ArchWare ADL did (Oquendo et al., 2002; Oquendo, 2004; Verjus et al., 2006).

Third idea: Nimrod (not only) targets (mobile) software architecture evolution.
Software architecture evolution is still an important issue (Mens et al., 2003; Mens et al., 2005; Cimpan and Verjus, 2005; Andrade and Fiadeiro, 2003; Verjus et al., 2006). Nimrod ADL and Nimrod environment support architecture evolution: the Nimrod ADL provides means for handling evolution while Nimrod environment integrates evolution mechanisms (that are

basically built on top of the Smalltalk VisualWorks environment). Nimrod ADL proposes evolution primitives (only manually evolution is actually supported) on components. For example:

```
myClient := Component name: 'client'.
"some code"
myClient manualEvolution
```

Listing 1: architectural evolution primitive

The previous piece of code (see listing 1) requests an evolution in the context of *myClient* component. At runtime, the architecture execution will be suspended and will request the user (the architect) for an evolution within the context (in other words the scope) of the *myClient* component. Either the user is deciding to resume the execution without any change, either the user is modifying the architecture description (the code expressed in Nimrod ADL) and then is resuming the architecture execution. No checking mechanism is implemented for validating changes: that means, user may change the architecture with some side effects that may corrupt the architecture integrity (such checking mechanisms will be implemented in a near future). It is up to the user to change the architecture consistently. Nimrod ADL supports behaviour (process) mobility. For example, a component *A* interacts with a component *B* by the way of connections (both components have connections that are unified). The component *A* is a composite and it composes a sub-component (lets say *subC*). Imagine that the *subC* component has to be moved from component *A* to component *B*. In order to do that, the component *A* has first to be decomposed in order to make free (to separate) the component *subC* from the component *A*. Then, the *subC* is sent to the component *B* by the way of message passing (the message is the *subC* component itself). When *subC* has been received by the component *B*, the component *B* composes *subC* component and connection unifications can be done (if required and if connections are types compatible). During the message passing, the *subC*'s behaviour is suspended and, then can be resumed at any time within the component *B*'s scope.

3.3 Architecture definition mechanisms

Nimrod provides mechanisms in order to express architecture definitions. A Nimrod architecture definition is basically a architectural configuration (Shaw and Garlan, 1994; Shaw and Garlan, 1996) also called a style (Garlan, 1995; Shaw and Garlan, 1996; Leymonerie, 2004). A Nimrod architectural definition can be then instantiated that gives an architecture ready to be executed. One of the main advantage of using architecture definition is that a type can be used for creating components conform to that architectural definition. A Nimrod architectural definition is an architectural and a conceptual building block that can be reused and extended as needed. As consequence, an architecture definition can be a part in another architecture definition. All of the complexity of a particular definition is hidden to its external world as an architectural element may compose sub architectural elements that are encapsulated in. We will detail architectural definition in a future document.

3.4 Nimrod environment

The Nimrod environment is developed using Smalltalk VisualWorks. The smalltalk virtual machine is the Nimrod ADL virtual machine as the Nimrod is implemented in Smalltalk. Once an information system architecture has been expressed in Nimrod ADL, the Nimrod ADL is then interpreted as a VisualWorks Smalltalk program. Interesting VisualWorks features (Inspector, Debugger, etc.) are suitable when interpreting, inspecting debugging and evolving Nimrod architecture code.

4 A CLIENT-SERVER ARCHITECTURE EXAMPLE EXPRESSED AND ENACTED USING NIMROD

4.1 Formalizing an architecture

The following piece of code is a quite simple client-server architecture. As you will see, component is the main artefact: the entire architecture (*compArchi* in the following listing 2) is also a component that composes a *client* component and a *server* component. The *compArchi* component :

- can be reused "as is";
- can be reused in other architectures with some adaptations: by adding *compArchi*'s connections in order to communicate with other components and by modifying its behaviour (for sending and/or receiving messages with other components i.e. its "external world").

```
" It is a typical client-server architecture
    example - During the architecture execution,
    the user is asked for an evolution"
" He may decide to continue the execution as is or
     to change the architecture by modifying, for
    example, the server behaviour"
```

```
| client server con1 con2 con3 con4 con5
    comp2 compArchi|

"An anonymous sub-component"
comp2 := Component new.
comp2 name: #subComponent1.
comp2 behaviour: [
Transcript show: 'comp2 behaviour
    execution'; cr].

"Client component"
con1 := Connection name: 'con1' type:
    Integer type: String.
con2 := Connection name: 'con2'.
con2 types add: Boolean; add: Component;
    add: Integer.
client := Component name: 'client'.
client addConnection: con1; addConnection:
     con2.
client composes: comp2.
client behaviour: [
        client decomposes: comp2.
        con1 send: comp2.
        con2 receive].

"Server component"
con4 := Connection name: 'con4' type:
    Integer type: String.
con5 := Connection name: 'con5' type:
    Boolean type: Component type: Integer.
con3 := Connection name: 'con3'.
con3 types add: Integer; add: String.
server := Component name: 'server'.
server addConnection: con4; addConnection:
     con5.
server behaviour: [
         | myComponent |
        myComponent :=  con4 receive.
        server composes: myComponent.
        myComponent exec.
        server unobservable.
        con5 send: 'OK'].

"compArchi component: the overall
    architecture"
compArchi := Component name: 'client-
    server'.
compArchi composes: client.
compArchi composes: server.
compArchi unifies: con1 with: con4.
compArchi unifies: con5 with: con2.

compArchi behaviour: [
        client exec.
        compArchi unobservable.
        compArchi manualEvolution.  "ask
             for an evolution that will be
             manually managed"
        server exec].

compArchi exec.
```

Listing 2: the client-server architecture expressed using Nimrod ADL.

Each component (*client* and *server*) of the architecture is described separately. Then components composition and unification are expressed within a scope of a another component (compArchi) that contains the other. The compArchi's behaviour defines when client and server communicate and when they are executed. The last line of code starts the execution (*exec*) of the overall architecture.

4.2 Executing an architecture

As Nimrod component behaviours are implemented as Smalltalk's light-weight processes, they run concurrently according to the VisualWorks light-weight process management semantic. Note that Nimrod ADL separates component definition and component execution. The latter corresponds to the execution of the component's behaviour that is part of the component definition. As consequence, one may change the component definition (and for example, its behaviour) without disturbing the execution (as a Smalltalk process).

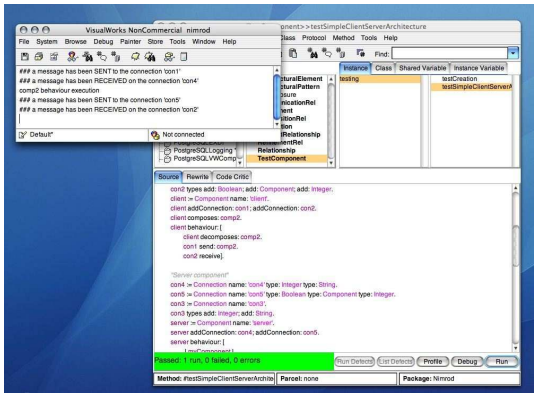

Figure 3: a client-server architecture using Nimrod.

The figure 3 shows the outputs results of the client-server architecture execution.

```
server behaviour: [
                | myComponent |
                myComponent :=  con4 receive.
                server composes: myComponent.
                myComponent exec.
                server unobservable.
                con5 send: 'OK'].
```

Listing 3: reception of a mobile component.

The *comp2* component (see listing 2) has been sent from the *client* to the *server*. When received on the *server*'s *con4* connection (see listing 3), the *server*

composes the received component and executes its internal behaviour. Such example illustrates a mobility simple scenario. Checking mechanisms have to be implemented in order to support consistent component mobility according to component internal state management, persistency, etc.

4.3 Evolving the architecture

(Cimpan and Verjus, 2005; Verjus et al., 2006) present a taxonomy of software architecture evolution: static or dynamic evolution, planned or unplanned evolution. Architectural evolution can be managed automatically or manually. Nimrod currently supports static and dynamic evolution, that is either planned or unplanned.

```
" some code     see the listing describing the
    client-server architecture "
        compArchi behaviour: [
                client exec.
                compArchi unobservable.
                compArchi manualEvolution.  "ask
                    for an evolution that will be
                    manually managed"
                server exec].

        compArchi exec.
```

Listing 4: reception of a mobile component.

During the (*compArchi*) architecture execution, and according to its behaviour definition (see listing 4), a manual evolution is requested.

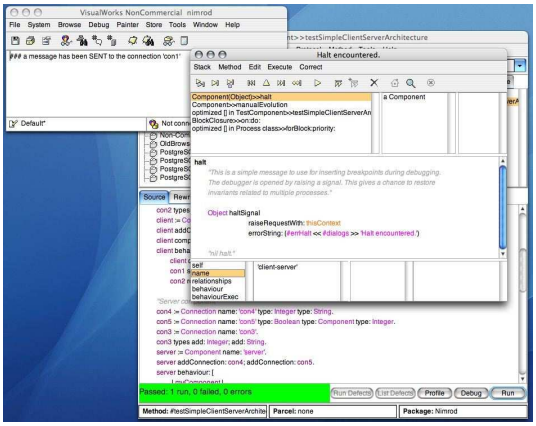

Figure 4: a manual architectural evolution is requested.

It is up to the architect to decide when the evolution has to be occurred (in the code). In the previous piece of code (see listing 2), an evolution is requested just before the *server* component execution, while the *client* is currently being executed. The figure 4 shows that the architecture execution is suspended until the architect will resume it.

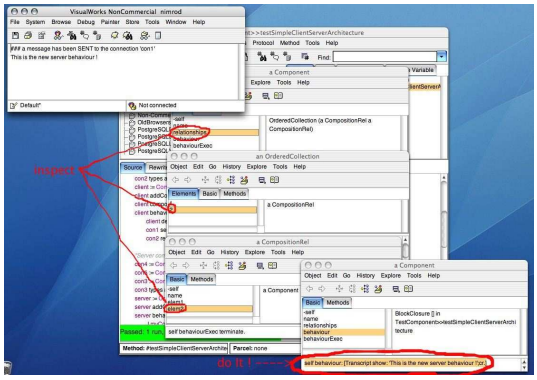

Figure 5: once the changes have been expressed using Nimrod ADL, they can be applied on the current runnable architecture.

The architect may do some changes. In the presented example, he is changing the *server*'s behaviour (by using the VisualkWorks's Inspector tool and by directly modifying the behaviour code). Then, the architect resumes the architecture execution that is executing the *server*'s behaviour that just has been changed. As you will see in the figure 5, the execution outputs are not the same as the previous execution (without any change see the figure 3), and take into account the recent changes. The architecture has been dynamically (on the fly) changed.

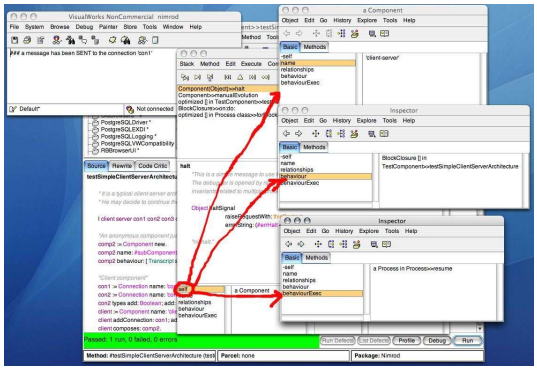

Figure 6: architectural elements inspection.

When the architecture is suspended, the architect may inspect architectural elements (components, connections, etc.). The figure 6 shows that the current component whose name is "client-server" (that is declared as *compArchi* in the listing 2), has its executed behaviour implemented as a Smalltalk suspended process.

5 CONCLUSION AND ONGOING WORK

Architecture evolution support is an important issue (Andrade and Fiadeiro, 2003). Current researches on this area are concentrated either on low level of abstraction (implementation level), either on a very high level (that is traditionnaly called conceptual level). Firstly, we claim that the ADL has to support evolution: architecture evolution would have to be expressed using the language itself. Few ADLs have such feature (Darwin (Magee et al., 1995), π-Space (Chaudet and Oquendo, 2000), Piccola (Nierstrasz and Achermann, 2000), ArchWare (Verjus et al., 2006; Cîmpan et al., 2007; Oquendo et al., 2004)), while most of them are based on π-calculus. Secondly, the ADL has to be runnable and has to support on the fly changes mechanisms. This latter point is very important. Most research activities focusing on architecture evolution are statically onces because they are not based on an adequate ADL: changes on architectures are made either on abstract architecture (Egyed and Medvidovic, 2001), either directly in the code (see (Pollet et al., 2007) for a good survey on research leaded in this topic). As consequences, research works address consistency issue between abstract and implementation levels (Oreizy et al., 1998; Garlan et al., 2003; Carriere et al., 1999; Erdogmus, 1998; Aldrich et al., 2002; Pinzger et al., 2004; Pinzger et al., 2005; Rank, 2005; Roshandel et al., 2004; Nistor et al., 2005). Nimrod is a first step towards a simple, runnable ADL that supports evolvable architectures. Nimrod combines ADLs and OOPL strengths in order to be usable as a new ”way of programming” software-intensive system architectures. Using Nimrod architects express runnable architectures by composing components. Evolving such architectures is no more a big deal while the Nimrod integrates evolution mechanisms and tools. The Nimrod ADL artefacts (components, connections) are high level abstractions, commonly used in ADLs while the component's behaviour is expressed as sequence of actions (as in the π-calculus).

We will aim at implementing Nimrod checking mechanisms:

- according to π-calculus semantics;
- when evolving an architecture, the changes would not have to be applied (”do it !” in the figure 5) if they violate architectural constraints and properties.

Another issue is to integrate architectural changes impacts analysis. Nimrod only supports manual evolution management: we will investigate automated evolution management that would consist in automatically searching for an appropriate evolution strategy and corresponding changes among several. Similar mechanisms (i.e. components that would manage evolution) such as those described in (Verjus et al., 2006) could be good candidates.

REFERENCES

Aldrich, J., Chambers, C., and Notkin, D. (2002). Architectural reasoning in ArchJava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367, Malaga, Spain. Springer Verlag.

Allen, R., Douence, R., and Garlan, D. (1997). Specifying dynamism in software architectures. In Leavens, G. T. and Sitaraman, M., editors, *Proceedings of ESEC '97 Workshop on Foundations of Component-Based Systems*, pages 11–22, Zurich.

Alloui, I. (2005). Property verification and change impact analysis for model evolution. In *1ères journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, pages 169–174.

Andrade, L. and Fiadeiro, J. (2003). Architecture based evolution of software systems. In Springer-Verlag, editor, *Formal Methods for Software Architecture*, volume 2804 of LNCS, pages 148–181.

Carriere, S., Woods, S., and Kazman, R. (1999). Software architectural transformation. In Society, I. C., editor, *Proc. 6th Working Conference on Reverse Engineering*.

Chaudet, C. and 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 (ASE'00)*, Grenoble, France.

Cimpan, S., Leymonerie, F., and Oquendo, F. (2005). Handling dynamic behaviour in software architectures. In *European Workshop on Software Architectures*, Pisa, Italy.

Cimpan, S. and Verjus, H. (2005). Challenges in architecture centred software evolution. In *CHASE: Challenges in Software Evolution*, pages 1–4, Bern, Switzerland.

Cîmpan, S., Verjus, H., and Alloui, I. (2007). Dynamic architecture based evolution of enterprise information systems. In *International Conference on Enterprise Information Systems (ICEIS)*.

Consortium, A. Archware - architecting evolvable software - ist european project 2001-32360.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Egyed, A. and 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, pages 83–87, Toronto, Canada.

Erdogmus, H. (1998). Representing architectural evolution. In *Proceedings of CASCON '98*, pages 159–177, Ontario, Canada.

Garlan, D. (1995). What is style? In *Proc. First International Workshop Software Architecture*.

Garlan, D., Cheng, S., and Schmerl, B. (2003). Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*. Springer-Verlag.

Lehman, M. (1996). Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin. Springer.

Leymonerie, F. (2004). *ASL language et outils pour les styles architecturaux. Contribution à la description d'architectures dynamiques*. PhD thesis, University of Savoie, Annecy.

Leymonerie, F., Cimpan, S., and Oquendo, F. (2002). Etat de l'art sur les styles architecturaux: classification et comparaison des langages de description d'architectures logicielles. *Génie Logiciel*, 62.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings ESEC '95*, volume 989 of *LNCS*, pages 137–153. Springer-Verlag.

Medvidovic and Taylor (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 123–131. IEEE Computer Society.

Mens, T., Wuyts, R., Volder, K. D., and Mens, K. (2003). Workshop proceedings — declarative meta programming to support software development. *ACM SIGSOFT Software Engineering Notes*, 28(2).

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall.

Morrison, R., Balasubramaniam, D., Kirby, N., Mickan, K., Oquendo, F., Cimpan, C., Warboys, B., Snowdon, R., and Greenwood, M. (2004). Support for evolving software architectures in the archware adl. In *4th Working IEEE/IFIP Int. Conf. on Software Architecture*, pages 69–78, Oslo, Norway.

Nierstrasz, O. and Achermann, F. (2000). Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan. IEEE.

Nistor, E., Erenkrantz, J., Hendrickson, S., and d. Hoek, A. V. D. H. (2005). Archevol: Versioning architectural-implementation relationships. In *12th International Workshop on Software Configuration Management (SCM05)*, Lisbon, Portugal.

Oquendo, F. (2004). π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14.

Oquendo, F., Alloui, I., Cimpan, S., and Verjus, H. (2002). The archware adl: Definition of the abstract syntax and formal semantics. Deliverable D1.1b, ArchWare Consortium, ArchWare European RTD Project IST-2001-32360.

Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., and Occhipinti, C. (2004). Archware: Architecting evolvable software. In *In proceedings of the first European Workshop on Software Architecture (EWSA 2004)*, pages 257–271, St Andrews, UK.

Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA. IEEE Computer Society.

Pinzger, M., Gall, H., and Fischer, M. (2005). Towards an integrated view on architecture and its evolution. *Electronic Notes in Theoretical Computer Science*, 127(3):183–196.

Pinzger, M., Gall, H., Girard, J.-F., Knodel, J., Riva, C., Pasman, W., Broerse, C., and Wijnstra, J. G. (2004). Architecture recovery for product families. In *Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5)*, LNCS 3014, pages 332–351. Springer-Verlag.

Pollet, D., Ducasse, S., Poyet, L., Alloui, I., Cîmpan, S., and Verjus, H. (2007). Towards a process-oriented software architecture reconstruction taxonomy. In *Conference on Software Maintenance and Reengineering (CSMR), Best Paper Award*.

Pourraz, F., Verjus, H., and Oquendo, F. (2006). An architecture-centric approach for managing the evolution of eai services-oriented architecture. In *Eighth International Conference on Enterprise Information Systems (ICEIS 2006)*, pages 234–241, Paphos, Cyprus.

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*, pages 51–58, Glasgow, UK.

Roshandel, R., Hoek, A. V. D., Mikic-Rakic, M., and Medvidovic, N. (2004). Mae: a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276.

Sharp, A. (1997). *Smalltalk by Example*. McGraw-Hill.

Shaw, M. and Garlan, D. (1994). Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University, School of Computer Science.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

Verjus, H., Cimpan, S., Alloui, I., and Oquendo, F. (2006). Gestion des architectures évolutives dans archware. In *1ère Conférence francophone sur les Architectures Logicielles (CAL 2006)*, pages 41–57, Nantes.