

# EVALUATION AND COMPARISON OF ADL BASED APPROACHES FOR THE DESCRIPTION OF DYNAMIC OF SOFTWARE ARCHITECTURES

Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem  
*University of Sfax, laboratory LARIS-FSEGS*  
*B.P. 1088 Sfax, Tunisia*  
*Email: mohamed.hadjkacem@fsegs.rnu.tn*

Khalil Drira  
*LAAS-CNRS*  
*7 Avenue du Colonel Roche 31077 Toulouse, France*

**Keywords:** ADL, dynamic configuration of architectures, behaviour of applications, component, configuration.

**Abstract:** This paper presents an evaluation study of Architecture Description Languages (ADL) which allows to compare the expressive power of these languages for specifying the dynamicity of software architectures. Our investigation enabled us to release two categories of ADLs: configuration languages and description languages. Here, we address both categories, and we focus on two aspects: the behaviour of software components and the evolution of the architecture during execution. In addition, we explain how each ADL handles these aspects and demonstrate that they are generally not or not enough dealt with by most of the ADLs. This motivates future extensions to be undertaken in this domain. Throughout this paper, we illustrate the comparison of these two aspects by describing an example of a distributed application for collaborative authoring support.

## 1 INTRODUCTION

Software applications are more and more distributed, large and complex. Such applications are made up of a generally significant number of software entities, scattered on the network, which cooperate in order to provide services to their users. This complexity is related to the great geographical and structural dispersion of the entities constituting the application, to the material and software heterogeneity, and the abundant interaction between them. Moreover, these applications must usually undergo various modifications during their life cycle in order to face new requirements of the users and / or the new technologies.

In our study, we mainly deal with distributed collaborative applications, such as the collaborative authoring of research papers. This kind of applications is characterized by a dynamic architecture which may evolve during system execution. The evolution of an architecture is generally expressed in terms of reconfiguration operations which correspond to the addition/removal of a component or a connection between components. In addition, a reconfiguration can be induced by an internal (component) event or by an external (user) event. Accordingly, many reconfigurations can not be known in advance and many others may depend on the behaviour of some components. Hence, the design of such systems becomes an in-

tricate task requiring rigorous approaches. Accordingly, it is necessary to have a specification language which support the description of dynamic software architectures allowing reconfigurations induced by internal and external events. In our approach, we made recourse to ADLs and we tried to select the most suitable ADL for the specification of our applications. The investigation of most existing ADLs enabled us to evaluate them with respect to the above mentioned aspects. This paper presents the main results of this comparative study.

In our study of ADLs, we focus on specifying the dynamic of architectures as well as the behaviour of components. Indeed, we measure for each language its expressive power according to these two aspects. It should be stressed that few works considered these aspects while evaluating ADLs. The majority of them were interested rather in classifying ADLs with respect to static aspects. The most outstanding one is the work of Medvidovic and Taylor (Medvidovic and Taylor, 2000) that tried to classify existing ADLs. This work identifies the main characteristics which have to be present in an ADL. Riveill and Senart (Riveill and Senart, 2002) and Barais (Barais, 2002) essentially sought to classify ADLs in two classes: configuration languages and description languages. They evaluate the ability of these languages to describe reconfiguration. Finally, other works like the

one of Allen and al. (Allen et al., 1998) consist in extending Wright so that it supports the run-time re-configuration.

Our study has enabled us to identify several weak points in ADLs. These weak points are in several orders. Indeed, the dynamic of architectures is not well supported by ADLs although there are some proposals which are interesting (Darwin (Dulay et al., 1993) and Olan (Bellissard et al., 1996)). Moreover, the dynamic of software architecture is not much expressed. ADLs are interested only in the systems having a fixed number of configurations which have to be known in advance (Wright (Allen et al., 1998)). Thus, it is not possible to perform arbitrary reconfiguration operations and particularly those which are external during the application execution. Finally, the behavioural aspect is almost absent in the majority of ADLs except for some such as Rapide (Rapide, 1997) which allows to describe the behaviour using process algebra. However, the behaviour description expresses only communicating events and does not consider re-configuration operations.

This article is organized as follows. Section 2 defines basic ADLs concepts. In Section 3 we introduce our case study: an application for collaborative authoring support. Sections 4 and 5 present the ADLs studied and evaluate their ability of describing the dynamic of an architecture and the behaviour of components. Finally, in Section 6, we present a synthesis of the obtained results.

## 2 WHAT IS AN ADL?

An ADL (Accord, 2002), (Medvidovic and Taylor, 2000) is defined as a textual or graphic notation, formal or semi-formal. It allows to specify software architectures and it is generally accompanied with specific tools. In our study, we noted a large variety of notations offered by ADLs. Nevertheless, the three concepts of component, connector and configuration are considered essential.

A *component* represents the basic unit of a system. We find among them clients, servers, data bases, objects, modules, etc.

A *connector* is an architectural entity which models the interactions between components. Components and connectors are generally accessible only through interfaces (which we often call ports for the component and roles for the connector).

A *configuration* or *architecture* is an arrangement, a topology, or a graph of components and connectors which describes how the components are connected to each others.

Based on the ability of an ADL to describe the evolution of an architecture at run-time, we distinguish

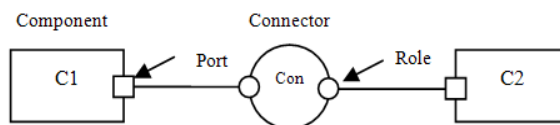


Figure 1: Configuration of an application

two classes of ADLs: configuration languages and description languages. Contrary to description languages, configuration languages permit to specify operations like adding a new component/connector or removing a component/connector.

## 3 CASE STUDY: COLLABORATIVE AUTHORING

We consider in this paper an applicative example: the collaborative authoring (Pinheiro et al., 2002). It is a cooperative application enabling a set of authors to cooperate in order to draft a common document.

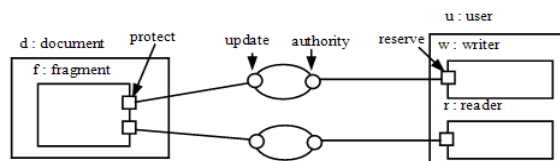


Figure 2: The collaborative authoring configuration in ADL

This illustration allows, to compare the similarities and the differences between these ADLs. This application illustrates an example of a system which owns a dynamic architecture. During execution a new component: reader or writer may be connected to the application without disturbing its functioning. This modification in the architecture is controlled by the rules stating that two writers must never access simultaneously to the same fragment. Adding a new writer component it is induced by an external (user) event. In addition, we may consider internal events which may also change the architecture at the run-time. For example, a writer component may be disconnected from the application when the time allocated for him is expired.

## 4 CONFIGURATION LANGUAGES

The configuration languages (Accord, 2002), (Riveill and Senart, 2002) provide a model of notation to spec-

ify the application configuration. In these languages, a component is considered as an instantiate entity. Several instances of the component can be created and can coexist during the execution. Configuration languages allow to describe an application with the aim of its deployment and of its execution. They belong to a family of languages which are accompanied with tools of modelling, parsers and code generators.

We present in the following sections the main properties of the languages Darwin, ArchJava and Olan and measure for each one its ability of describing the dynamic of architectures and the behavioural of components.

## 4.1 The Darwin language

Darwin (Dulay et al., 1993) uses the three basic concepts i.e. the component, the connector and the configuration but only the component is explicitly expressed. It allows a partial specification of the dynamic of architecture in terms of component creation during the execution of an application.

Darwin provides two mechanisms to specify the component instantiation: *lazy* and *dynamic instantiation*. Thanks to these two mechanisms, the architecture of an application is not considered any more as a set of components fixed at the design phase. It is, then, possible to specify when and where the dynamic creation of software components can take place.

*Lazy instantiation:* Lazy instantiation (Accord, 2002) enables to declare components which will not be immediately instantiated. These components will be indeed created when the service which they provide is requested. This type of mechanism enables to instantiate only one component by interconnection and allows describing structures whose number of components can be determined only dynamically.

---

```

component Edition {
  inst
  writer : Writer;
  fragment : dyn Fragment;
  bind
  writer.reserve -- fragment.protect;
}

```

---

Figure 3: Lazy instantiation in Darwin

Contrary to traditional components which will be created at the application installation, the component `fragment: dyn Fragment` is not instantiated as long as the user does not reach the service provides by the component. In this example, the component *fragment* will be instantiated and protected with the first call of the `fragment.protect`. Lazy instantiation enables to instantiate only one component by interconnection clause (Barais, 2002) and this cannot

always be used. So, Darwin proposes the dynamic instantiation concept.

*Dynamic instantiation:* Unlike lazy instantiation, multiple instances can be dynamically created from only one interconnection clause. Dynamic instantiation (Riveill and Senart, 2002) allows to specify the connection for a component type rather than for an instance of this type. So, it enables multiple instantiations through only one clause of interconnection. The declaration of this instantiation `dyn Fragment` is achieved in the interconnection clause `bind` (Figure 4).

---

```

component Edition {
  inst
  writer : Writer;
  bind
  writer.reserve -- dyn Fragment ;
  writer.update -- Fragment.protect;
}

```

---

Figure 4: Dynamic instantiation in Darwin

The difference between this example and the previous one is that the component *Fragment* is instantiated `writer.reserve -- dyn Fragment;` in the interconnection clause `bind`. So, the user can create several instances of the component *Fragment* in only one interconnection clause.

To conclude, we can deduce that the description of the dynamic of an architecture using Darwin is limited. Indeed, Darwin does not allow expressing different reconfiguration actions usually desired by the application administrator. In addition, it is not possible, for example, to remove nor to designate dynamically created components. Thus, a primitive component cannot communicate with dynamically created components (Barais, 2002). Furthermore, Darwin does not offer the basis for analysing application behaviour, because its model does not enable to describe the features of a component and of its services (Allen, 1997). Indeed, a component is considered as a black box.

## 4.2 The ArchJava language

ArchJava (Aldrich et al., 2001) aims to improve the programs comprehension, to guarantee and to allow a better evolution of the application architecture.

In order to create and connect dynamically some components, ArchJava offers functions to create, destroy and connect new components. The dynamic component creation is done using the operator `new`. A component can be connected using the operator `connect` with a `port` instance. The component destruction is not managed explicitly. When a component is neither referred nor connected to other components, its memory will be released by the garbage

collector. It may be necessary, in some applications, to connect several components to the same port without knowing their number in advance. ArchJava, then, enables to define a port interface. A new port instance will then be created with each connection (Barais, 2002).

We return to our example and we suppose that a writer can reserve several fragments at the same time.

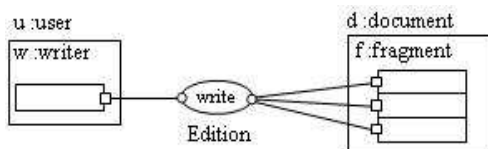


Figure 5: Fragment reservation in ArchJava

At the architectural level (Figure 5), when a request arrives (for example a request for fragment reservation), the component *writer* asks for a new connection. The component *Edition* creates, then, a new component *Fragment* using the operator `new` and connects it to the component *writer* thanks to the operator `connect`. Then, the *writer* communicates with the component *Fragment* via its port `reserve`.

According to our experimentation with ArchJava we can conclude that this language is an easy ADL thanks to its syntax similar to java. It allows to describe the dynamic of architectures in terms of creation, connection and destruction of components. However, this description is very limited since external (user) events are not taken into account. Moreover, ArchJava does not allow to specify the behaviour of components. Like in Darwin, in ArchJava a component is considered as a black box.

### 4.3 The Olan language

Olan (Bellissard et al., 1996) aims principally to provide a complete environment for construction, configuration and deployment of distributed applications built on the assembly of heterogeneous components.

Basically, Olan is a formalism which does not support the update of architectures at runtime. This language has been augmented with an OCL (Object Constraint Language) enabling then to describe operations changing the architecture. The obtained language is based on a set of rules of the form (*Event*, *Condition*, *Action*). Like Darwin, Olan offers lazy and dynamic instantiation. However, Olan provides a new schema of dynamic instantiation called *collection*.

*Collection*: The collection concept (Accord, 2002; Riveill and Senart, 2002) has been introduced to facilitate the use of multiple components with the same type. When it is created, a collection does not contain any component. A collection has two operators

`new` and `delete` allowing to create or destroy dynamically a component instance.

The clause `fragment = Collection [0..n]` of *Fragment*; defines a collection of fragments in a document. The component designation can take two forms. First, the collection members can be dynamically identified through some properties related to its attributes. In this case, components are managed implicitly. It is the designation mode by association. The second designation form is rather explicit. It considers a component as a member of the collection and it is designated with its index in the collection.

---

```
Writer.reserve(request,idf) => fragment.update(request)
where fragment.idf = idf using authority;
```

---

Figure 6: Component designation from a collection

This example considers designation by association. In the clause below, *Writer* indicates a *fragment* (identified by its `idf`) from a collection in order to modify it. It cannot modify the fragment only if it is authorized. Thus, the main advantage is to create sets of components having the same functions but with variable number of components.

Accordingly, we can say that Olan architectural model is similar to the Darwin's one extended with the concept of *collection*. Thanks to this concept, Olan allows to express the dynamic of architectures with more precision than Darwin. Indeed, Olan enables to describe a component by separating interface description from its implementation and its configuration. Thus, when one of the parts evolves, the others are not modified. In addition, Olan offers a kind of adaptation form, since it is possible to modify the system in reaction to some events. However, events taken into account in the model are not external. The modification types allowed are too limited. Like Darwin, Olan does not offer the basis for analysing component behaviours.

## 5 DESCRIPTION LANGUAGES

Description languages (Accord, 2002) allow to describe the application evolution and to specify modifications which are to be carried out on the initial architecture. They offer the possibility to validate architecture by checking specific constraints or by simulation. They belong to language families which are accompanied with tools for modelling, code generation, code execution or simulation.

We present in the following sections the mechanisms offered by the languages Rapide, Wright and Acme to describe the dynamic of architectures and behaviour of components.

## 5.1 The Rapide language

Rapide, (Rapide, 1997) aims to check validity of software architectures by simulation. Rapide provides tools for dynamically updating the software architecture of an application. We demonstrate this, on the collaborative authoring application:

We suppose that the interface types of *Writer*, *ControlCentre*, and *Msg* are already defined. The interface for *Writer*, for example, defines all functions allowing to create, reserve, and modify a *Fragment*. The interface for *EditionCenter* declares two actions called *in* and *out* to allow an *EditionCenter* to accept or reject *Writer*. The *ArchEdition* architecture (an architecture generator) defines the communications between the interfaces of *Writers* and *ControlCenter* using connection rules. For example, the first connection links a component *Writer* whenever an accepted event with that *Writer* is received.

---

```

Example Edition
type Writer is ... type Fragment is ... type Msg is ...
W : Writer; ?F : Fragment; ?d : Data;
type EditionCenter is interface
  action in Accept(X: Writer);
        out Hnadoff(X: Writer);
behaviour
Start => Send (d);
(?d in Data) Replay (?d) where Authority => update(?d);
end EditionCenter;
architecture ArchEdition return EditionCenter is
SFO: ControlCenter; ...
connections
Accept(?W) => link(?W);
?W.Update(?d) where ?W.Authority(SFO) || > SFO.Receive(?d);
Handoff(?W) => unlink(?W);
constraint
observe from W.Send, SFO.Receive
  match ((?d in data)
(W.Send(?d) -> (SFO.Receive(?d) or Empty)))†(“*");
End; end ArchEdition;

```

---

Figure 7: Dynamic architecture in Rapide

The rule states that if a *Writer* (a match for *?W*) sends a request containing a message *?d*, then *?W* is bound to the *Writer* and *?d* to the message. The connection is made only when the *Writer* is accepted. If the *ArchEdition* refuses a *Writer*, then it unlinks from it. From now, no communications from the *Writer* are received. This is a dynamic interface connection architecture. It defines communication between a variable numbers of *Writers* components. The second connection rule defines a conditional broadcast between all *Writers* and the *ControlCenter*.

Accordingly, Rapide allows to describe the dynamic of architectures in terms of components creation, dynamic nomination of participants and essentially a fan-in connection. It describes the behaviour of components in the behaviour clause. However, this description expresses only communicating events and does not consider reconfiguration operations.

## 5.2 The Wright language

Wright (Allen, 1997) is focused on the architecture specification. It is mainly interested in rigorous reasoning about protocols. For this purpose it uses the CSP language.

The first version of this language (Allen, 1997) allows the description of static architecture only. enabled to describe only static architectures. In 1998, Wright was extended to support the dynamic architectures description: *configuration program*. To illustrate this, we return to our example and we consider a configuration in which a server is composed of two components managing the access in reading and writing documents. The system is composed of a client and a server (*Primary: FlakyServer*) interconnected via a connector. This server can frequently run out. The idea consists in enabling the system to change configuration by replacing the primary server with (*Secondary: SlowServer*) until the latter is ready to function again.

---

```

Configurator Edition
new.U : User
-> new.Primary : FlakyServer
-> new.Secondary : SlowServer
-> new.L : FaultTolerantLink
-> attach.U.p.to.L.u
-> attach.Primary.p.to.L.s WaitForDown
where
WaitForDown = Primary.control.serverDown -> detach.Primary.p.from.L.s
              -> attach.Secondary.p.to.L.s -> WaitForUP
              □ §
WaitForUp = Primary.control.serverUp -> detach.Secondary.p.from.L.s
            -> attach.Primary.p.to.L.s -> WaitForDown
            □ §

```

---

Figure 8: Dynamic Wright specification

In this configuration program, the initial sequence of actions (*new* and *attach*) builds the initial configuration. Then *WaitForDown* describes two situations: the system can run and successfully terminate (§) or a fault can occur. If the primary server goes down, the secondary server is in state on and the link connector is reconfigurable, the primary server is detached and is replaced by the secondary server. The new configuration then resumes its execution until it terminates or the primary server comes up. *WaitForUp* specifies when the secondary server is off and the connector is reconfigured. In this case, the secondary server is detached and is replaced with the primary server.

In order to dynamically manage the changes of an architecture, Wright uses the configuration program. This makes the correspondence between the submitted events and the reconfigurations to be performed (Riveill and Senart, 2002). Hence, Wright allows to specify, in an abstract and formal way, the dynamic aspect: creation, connection, and remove of components. However, in Wright, the specification of recon-

figurations is limited. Indeed, Wright allows to specify only configurations which are known in advance.

### 5.3 The Acme language

Acme (Garlan et al., 1997) is a simple, generic software architecture description language, provides in part, a common interchange format between other ADLs. The concepts offered by Acme (component, connector, system, etc.) are sufficient to describe the architecture structure but are insufficient to specify precise information such as the dynamic of architecture or the behaviour of components.

We consider again the collaborative editing example. Here, we consider two components (Writer and Fragment) bound by *Rpc* connector. The *Writer* sends its requests via the *reserve* port and the *Fragment* receives its requests via the *protect* port. The *Rpc* connector has two roles *callee* and *caller*. For to describe the dynamic of architecture, Acme allows the translation of a completely static language toward a more dynamic language such as Darwin and Rapide. In this example (Figure 9), some properties were added. For example, the property `Aesop-style:style-id = client-Server;` shows that the writer in Acme is defined as a style in Aesop (Aesop is an ADL).

---

```

System Edition = {
Component writer = {
Port reserve;
Properties {
  Aesop-style : style-id = client-Server;
  source-code : external = "writer.c";
} ... }
Connector rpc = {
Roles {caller, callee}
Properties { ... }
Attachments {
writer.reserve to rpc.caller ;
fragment.protect to rpc.callee
}
}

```

---

Figure 9: Architecture with properties consideration

To summarize, we can say that Acme does not propose any new concept for describing the dynamic aspect. Acme allows to describe only the static aspect. In addition, the concepts offered by Acme are insufficient to specify the behaviour of components.

## 6 EVALUATION RESULTS

ADLs can be divided into two classes. The configuration languages and the description languages. Every ADL has a particularity. Darwin and Olan emphasize the dynamic aspect whereas Wright and Rapide emphasize the behavioural aspect.

With regard to the configuration languages, we can say that: Darwin's lazy instantiation does not respond to all of dynamic architecture needs. Whereas the dynamic instantiation has a major drawback: it is impossible to call dynamically created instances. The collection concept offered by Olan allows to add and delete components during execution. ArchJava enables component creation, connection and destruction but it does not allow to specify the behaviour of components.

With regard to the description languages, we can mention that: Rapide allows to describe the behaviour of components. However, this description expresses only communicating events and does not consider reconfiguration operations. In Wright, the architectural changes are managed inside the configuration program. But these changes must be known in advance.

Finally, we can say that the dynamic aspect is not well supported by ADLs. In spite of the great number of interesting proposals, these solutions are not enough thorough. The majority of ADLs, are able to manage only systems having a finite number of configurations and known in advance. Then, it is impossible to arbitrarily execute reconfiguration operations during run-time. Concerning the behavioural aspect, we noted that even if the behaviour is described, this emphasizes only the communication between components and does not consider the events causing reconfigurations.

## 7 CONCLUSION

In this paper, we have presented an overview of different architecture description languages and their properties. We classified them according to configuration and description languages, and we focused in particular on their support for specifying the dynamic of architectures and the behaviour of components. We have noted that the majority of ADLs are concentrated on the static description of architectures but the dynamic of architecture and the behaviour of component are not enough expressed. Generally, the choice of the appropriate ADL is difficult and crucial. Such a language must at the same time be intuitive and formal. Many approaches based on the formal specifications have also been used to describe the distributed system behaviours. However, although they facilitate the description of communications, they do not allow to describe the components and their attributes.

UML2.0 is a promising notation to describe the architecture of application. Moreover, it is a standard which includes various diagrams and notations mastered through the majority of software architects. Rather than building a new ADL, an interesting re-

search project consists to develop a generic environment around UML2.0 allowing to specify and design a complete system. It allows to describe the static and the dynamic aspect of architecture and the behaviour of components.

## REFERENCES

- Accord (2002). Assemblage de composants par contrats en environnement ouvert et réparti, état de l'art sur les langages de description d'architecture (adls). Projet ACCORD, Technical Report Livrable 1.1-2, RNTL, France.
- Aldrich, J., Chambers, C., and Notkin, D. (2001). Architectural reasoning in archjava. In *Proceedings of the OOPSLA '01 Workshop on Specification and Verification of Component-Based Systems*, pages 33–48, Singapore.
- Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University.
- Allen, R., Douence, R., and Garlan, D. (1998). Specifying and analysing dynamic software architectures. *Journal of Fundamental Approaches to Software Engineering*, 11(6):21–37.
- Barais, O. (2002). Approche statique, dynamique et globale de l'architecture d'applications réparties. Master report, Ecole Mine de Douai, Laboratoire d'Informatique fondamentale de LILLE.
- Bellissard, L., Atallah, S. B., Boyer, F., and Riveill, M. (1996). Component-based programming and application management with olan. In *Proceedings of Workshop on Distributed Computing Systems*, pages 579–595.
- Dulay, J., Kramer, N., and Magee, J. (1993). Structuring parallel and distributed programs. *IEEE Software Engineering Journal*, 8(2):73–82.
- Garlan, D., Monroe, R., and Wile, D. (1997). Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183.
- Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93.
- Rapide (1997). Guide to the rapide 1.0 language reference manuals. Technical report, Group Computer Systems Lab Stanford University.
- Riveill, M. and Senart, A. (2002). Aspects dynamiques des langages de description d'architecture logicielle. *L'Objet RTSI - L'objet. Coopération et systèmes à objets*, 8(3).