# Issues of Architectural Description Languages for Handling Dynamic Reconfiguration

Leonardo Minora, Jérémy Buisson, Flavio Oquendo, Thais Batista

▶ **To cite this version:**

Leonardo Minora, Jérémy Buisson, Flavio Oquendo, Thais Batista. Issues of Architectural Description Languages for Handling Dynamic Reconfiguration. 6ème Conférence francophone sur les architectures logicielles (CAL'2012), May 2012, Montpellier, France. pp.69-80. hal-00699895

**HAL Id: hal-00699895**
**https://hal.science/hal-00699895**

Submitted on 21 May 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Issues of Architectural Description Languages for Handling Dynamic Reconfiguration

Leonardo A. Minora
Federal Institute of Education, Science and Technology of Rio Grande do Norte
Natal (RN), Brazil
leonardo.minora@ifrn.edu.br

Jérémy Buisson
UEB / Écoles de St-Cyr Coëtquidan / Université de Bretagne Sud
Guer, France
jeremy.buisson@st-cyr.terre-net.defense.gouv.fr

Flavio Oquendo
UEB / Université de Bretagne Sud
Vannes, France
flavio.oquendo@univ-ubs.fr

Thais V. Batista
Federal University of Rio Grande do Norte
Natal (RN), Brazil
thais@dimap.ufrn.br

## ABSTRACT

Dynamic reconfiguration is the action of modifying a software system at runtime. Several works have been using architectural specification as the basis for dynamic reconfiguration. Indeed ADLs (architecture description languages) let architects describe the elements that could be reconfigured as well as the set of constraints to which the system must conform during reconfiguration. In this work, we investigate the ADL literature in order to illustrate how reconfiguration is supported in four well-known ADLs: $\pi$-ADL, ACME, C2SADL and Dynamic Wright. From this review, we conclude that none of these ADLs: (i) addresses the issue of consistently reconfiguring both instances and types; (ii) takes into account the behaviour of architectural elements during reconfiguration; and (iii) provides support for assessing reconfiguration, e.g., verifying the transition against properties.

## Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

## Keywords

dynamic reconfiguration, software architecture, architecture description language, ADL, $\pi$-ADL, ACME, C2SADL, Dynamic Wright

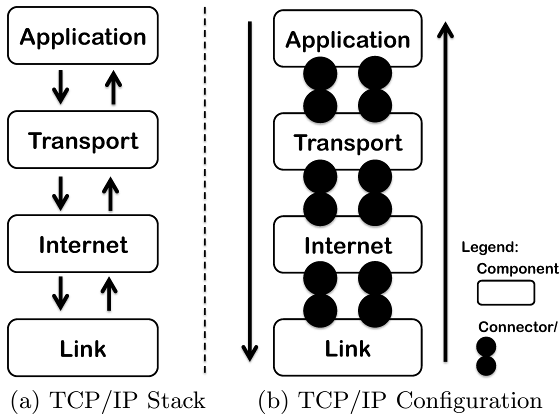
## 1. INTRODUCTION

*CAL'2012* 6ème Conférence francophone sur les architectures logicielles, Montpellier, France

The disciplined software engineering relies on software architectures to describe systems [3]. For modeling a software architecture, the academy proposed the architecture description language (ADL) and their toolsets [10, 18, 29]. Also, the industry has used ADLs to develop systems. Industry examples are (i) $\pi$-ADL has been used to architect and refine federated knowledge management systems in Engineering Ingegneria Informatica - Italy [22]; (ii) and AADL (Architecture Analysis & Design Language) is a standard language for the Society of Automotive Engineers [30]. According to the state-of-the-art [3, 18, 19, 29], the following concepts are relevant to describe software architectures: components and connectors (respectively computational and communication elements), architectural constraints, non functional properties, and behaviour.

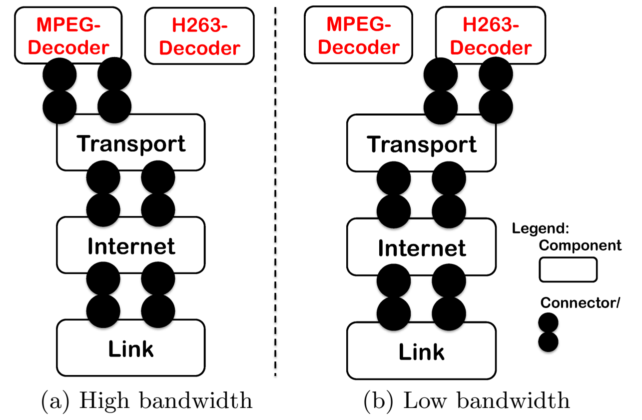
Software systems evolve over their life time [5, 21, 26]. Dynamic reconfiguration is when the evolution is performed at runtime with no service disruption. The dynamic reconfiguration can be handled by architectural concepts in an ADL. However among many existing ADLs, only few allow modeling dynamic reconfiguration. $\pi$-ADL [23], ACME/Plastik [4, 13], C2SADL [16, 27], DAOP-ADL [28], Darwin [15], Dynamic Wright [1, 2], Rapide [32], Weaves [25], and xADL [8] are typical examples. Nevertheless, there is no current consensus about how ADLs should address reconfiguration, e.g., what language constructs should be provided.

In this work, the goal is twofold: (i) to investigate the ADLs support for handling dynamic reconfiguration in the literature; and, (ii) to illustrate how four well-known ADLs support dynamic reconfiguration: $\pi$-ADL, ACME, C2 SADL and Dynamic Wright. We chose these four languages because they rely on different paradigms: the higher order typed $\pi$-calculus [23], first order predicate logic [11], component- and event-based [31, 16], and, graph grammars and communicating sequential processes (CSP) [2, 1], respectively. They also complement each other as $\pi$-ADL models



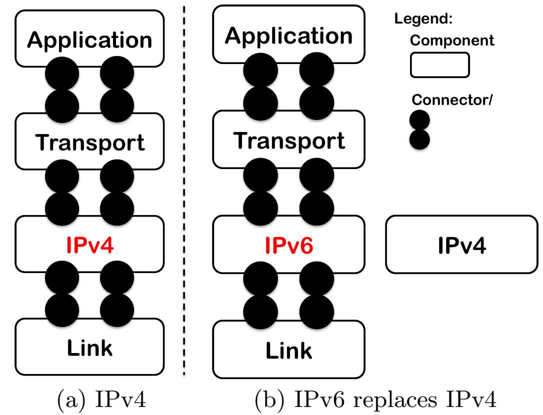
(a) TCP/IP Stack (b) TCP/IP Configuration

**Figure 1: simplified version of the TCP/IP stack system.**



(a) High bandwidth (b) Low bandwidth

**Figure 2: scenario 1 to illustrate changing the component.**



(a) IPv4 (b) IPv6 replaces IPv4

**Figure 3: scenario 2 to illustrate insertion of component.**

the behavior of architectures, ACME focuses on the structure, C2SADL make the attention for components and their concurrent events, and Dynamic Wright supports the definition of structure and behavior.

This work is structured as follows. Section 2 motivates this work thanks to the example of a TCP/IP stack system. Three scenarios illustrate three facets of dynamic reconfiguration. Section 3 presents the basics concepts of software architecture and dynamic reconfiguration. Section 4 details the related works about dynamic reconfiguration at the architecture level. Section 5 compares the four above-mentioned ADLs in the light of our example of Section 2. Section 6 concludes the paper with open issues that we note in the ADL support for dynamic reconfiguration.

## 2. MOTIVATION

Dynamic reconfiguration aims at modifying the software systems at runtime with no service disruption. Critical systems usually want to benefit of a dynamic reconfiguration because any service disruption may have substantial consequences. Examples of critical systems are stock market quotation systems, telecommunications systems, safety systems, and air traffic control systems. However, there is no current consensus about how the software architecture should address dynamic reconfiguration at architecture level. Does regular syntax for dynamic architectures allow the same reconfigurations as specific language constructs? Should the state of components and connectors be taken into account at the architecture level?

To support this work, we rely on a simplified version of the TCP/IP stack (Fig. 1(a)). Each of the four layers (Application, Transport, Internet, and Link) is modeled as a component (Fig. 1(b)).

We consider the three following reconfigurations:

1. *Switch the application component.* Assume two alternatives: MPEG-Decoder and H263-Decoder. MPEG-Decoder is used if *bandwidth* is high (Fig. 2(a)). Otherwise H263-Decoder is selected (Fig 2(b)). Switching between MPEG and H263 starts a new stream. Therefore, no state in the decoder needs be kept. We can therefore replace one component by another.

2. *Insertion of a new component type.* Presume that the Internet Protocol version 6 (IPv6) replace the version 4 (IPv4) (Fig. 3(b)). The IPv6 component has backward compatibility with the IPv4 component. So it has the ports for both IPv4 (for compatibility reason) and for IPv6 (new feature). Therefore, the type of the component at the internet layer is changed: it has 2 additional ports.

3. *Update a component type.* Assume that the algorithm of error control of the Transport component is improved. When the error control algorithm is changed, the new behaviour needs to know the status of each packet (acknowledged, sent, timeout, ...) as well as the content of non-acknowledged packets (in order to resend them). Therefore, only the behaviour part of the component should be replaced. The state of the component should be unchanged.

These scenarios cover a wide spectrum. They can either be programmed at design-time or be discovered at run-time. They feature reconfiguration at both type and instance level.

They target both the structure and the behaviour of the architecture. We investigate the ADLs literature in order to illustrate how reconfiguration is supported in these scenarios.

In this paper, we focus on four ADLs that are representative of the main trends. Each ADLs complement the other. The complement at the sense for describing an initial architecture as well as dynamic reconfiguration. We choose $\pi$-ADL because it provides both description with behaviours thanks to its $\pi$-calculus roots. Whilst the ACME/Plastik is a declarative language with focus on the architecture structure for both descriptions. While C2SADL is used to compose architectures based on component and dynamic reconfiguration based on architectural events. Finally, Dynamic Wright provides support for architectural description based in graph-grammars and dynamic reconfiguration specification with a variant of Communicating Sequential Processes (CSP).

## 3. UNDERLYING CONCEPTS

*Software architecture concepts.* Software architectures describe systems by specifying their elements and the interactions between them [3, 29]. In this work, we use the basics concepts defined in [10, 18, 29]: a *component* is an unit of computation and storage; a *connector* is an unit of communication; and, a *configuration* is a specification of a software architecture in terms of components, connectors, and relationship between them.

Ports, roles, behaviours, constraints, and non-functional properties are other concepts defined in [10, 18, 29] used to describe these architectural elements. Ports and roles are used to define the way of interaction, ports for configurations and components, and roles for connectors. Behaviours are the architectural element internal computation. And the last two concepts, constraints and non-functional properties, are used to denote the assertions, invariants, quality of service that architectural elements are expected to meet.

Also, we consider types and instances of configurations, components, and connectors for modelling a software architectures [18, 29]. The types are abstractions like classes in the Object-Oriented Paradigm which encapsulate both the structure and the behaviours that can be performed on its structure. These types will be used to build the instances and these instances will be used to describe software architectures.

Fig. 4 illustrates in ACME the basics concepts with our example TCP/IP stack system (Sec. 2 Fig. 1(b)) at instance and type level. The types are defined in the **Family** statement block (lines 1-18) while the instance level in **System** (lines 20-30).

*Dynamic reconfiguration concepts.* A dynamic reconfiguration is a set of operations to modify an existing configuration at runtime. At the software architecture level, the operation is defined in terms of the architectural elements at type or instance level. Scenario 1 is an example of reconfiguration at the instance level (MPEG and H263 share the same type); and scenario 2 is an example of reconfiguration

```
1  Family TCPIP_MF {
2    . . .
3    Connector Type TCPIP_Conn2Layers {
4      ProvidedRole source;
5      RequiredRole sink;
6    };
7    Component Type TCPIP_Component {
8      Port Type DataFrom extends ProvidedPort with {};
9      Port Type DataTo extends RevidedPort with {};
10     Property Type Layer = enum{application, transport, internet,
           link}
11   };
12   Component Type TCPIP_Application extends TCPIP_Component
         with {
13     Port dataReceivedFromTransport: DataFrom;
14     Port requiredService: DataTo;
15     Property layer = "application";
16   };
17   . . .
18 }
19
20 System DecoderStream : TCPIP_MF = new TCPIP_MF extended with {
21   Component application : TCPIP_Application;
22   Component transport : TCPIP_Transpot;
23   Component internet : TCPIP_Internet;
24   Component link : TCPIP_Link;
25
26   Connector
27     application2transport, transport2internet, internet2link:
           Conn2Layers;
28   Attchments {
29     application.requiredService to application2transport.source;
30     application2transport.sink to transport.service;
31
32     transport.requiredService to transport2internet.source;
33     transport2internet.sink to internet.service;
34
35     internet.requiredService to internet2link.source;
36     internet2link.sink to link.service;
37   }
38   . . .
39 }
```

**Figure 4: Simplified version of TCP/IP stack system in ACME ADL.**

at the type level (IPv6 extends the type of IPv4).

Also, the dynamic reconfiguration can be foreseen or unforeseen [12]. The foreseen is a programmed dynamic reconfiguration specified at design time. While the unforeseen is an ad-doc defined at runtime. Example of foreseen and unforeseen is the scenario 1 and 3, respectively. In the scenario 1, the architect specify at design time that the system modify itself when the bandwidth change. Whereas the behaviour improvement is built at runtime as defined in scenario 3.

## 4. RELATED WORK

Medvidovic and Taylor [18] created a classification framework based on architectural elements and illustrated them with the following ADLs: ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, Weaves, and Wright. However, this work has no focus on the dynamic reconfiguration. Hence, its discussion about dynamic reconfigurations is limited, e.g. all ADLs were assessed only at the instance level, not at the type level.

Kacem *et al.* [14] described the capabilities of Darwin, ArchJava, Olan, Rapide, Wright, and ACME to support dynamic reconfiguration. They classify the ADLs as configuration and description language. The configuration language sup-

ports the description of a software system and limited dynamic reconfigurations, while the description language supports the specification of modifications to be applied to an existing architecture. Despite this work focus is on dynamic reconfiguration, it also presents some limitations such as the fact that the evolution is applied only at instance level and do not take into account the behaviour of the architectural elements.

Bradbury [7] and Bradbury *et al.* [6] proposed a framework to compare fourteen formal specification approaches which support dynamic software architectures. Amongst the criteria, Bradbury *et al.* considered the basic operations, how they can be composed and whether they apply to variable sets of architectural element types. About the basic operations (addition and removal of components and connectors), Bradbury et al. showed that most approaches support them. Also, they considered the criterion of operations composition (basic[1], sequence, choice, and iteration). Only 3 approaches (CommUnity, Dynamic Wright, and Gerel) provide full support while the other only support basic composition. Finally, Bradury *et al.* mentioned that all approaches support only the types of architectural elements defined prior runtime, fixed sets of architectural elements.

While ADL surveys have reached good understanding of the concepts underlying software architectures (Medvidovic and Taylor [18]), this is not true regarding dynamic reconfiguration at the architecture level. Even Bradbury [7] and Bradbury *et al.* [6], probably the most complete surveys, do not consider operations such as insertion of configuration and behaviour modification. They do not recognize *link* and *unlink* as basic operations. However, they present the examples of these operations on some approaches, like Gerel and C2SADL. Considering the scenarios of section 2, these related works address only the first scenario (partially) but not the two last. Consequently, it is clear that to discuss the three scenarios presented in section 2 is essential to have a better understanding of the dynamic reconfiguration issues. These scenarios consider both instance and type levels, the basic operations include configurations, components, and connectors, as well as their structures and behaviours.

# 5. COMPARISON OF 4 ADL

## 5.1 Applying the example in ADLs

In this subsection, we describe the scenarios defined in Sec. 2 in all four ADLs: $\pi$-ADL (subsection 5.1.1), ACME/Plastik (subsection 5.1.2), C2SADL (subsection 5.1.3), and Dynamic Wright (subsection 5.1.4). For each ADL we show the purpose, the TCP/IP stack system, a solution for three scenarios, and discussion about this implementation.

### 5.1.1 $\pi$-ADL

*Purpose of the ADL.* $\pi$-ADL is based on $\pi$-calculus and it is designed to describe the concurrent and mobile systems [23]. The description of architectural elements is mainly represented by ports and behaviour.

[1] *Basic composite operation* is the ability to group basic operations for reconfigurations.

*Dynamic reconfiguration support.* This ADL provides support for foreseen and unforeseen dynamic reconfiguration. The foreseen can be described in the behaviour of any architectural elements. While unforeseen needs some support, e.g., in the Virtual Machine (VM), in order to obtain a root reference to the reconfigured system. In ArchWare, this issue is addressed by the deep intrication between the toolset (including the visual and textual editors) and the VM thanks to the hyper-code technology [24].

*Initial architecture.* The implementation of TCP/IP stack system is shown in Fig. 5. The configuration is composed at lines 1-18 basically with a behaviour, statement `behaviour` is ... where ...! (lines 2-17). This architectural behaviour defines the instances of components (lines 3-6) and connectors (lines 7-9). The statement `where` in behaviour specifies the connections between components and connectors (lines 11-16). The example of component `TCPIP_Application` is partially showed at lines 20-35: two ports and a behaviour. The component behaviour is implemented with two internal methods (lines 24-25) and the definition of communication between its ports (lines 27-33).

*Scenario 1.* The implementation of scenario 1 in $\pi$-ADL is composed by a component that performs the dynamic reconfiguration (Fig. 6). This component has two parameters: the application component and the system component. Both are represented as a behaviour. The application component is a MPEG-Decoder or H263-Decoder.

Still, this component is structured with seven “variables” and a behaviour, lines 2-4 and 5-27, respectively. The variables are used for behaviours of components and connectors. To perform the modification, the behaviour component use the following steps: [2] decomposition the system behaviour (line 6); assign the behaviours to variables (lines 8-14); compose a new system behaviour (lines 16-26) with a replaced application component (line 20).

*Scenario 2.* The implementation for the unforeseen scenario 2 is shown in Fig. 7. This code is applied in a system at runtime to deploy the two new component “type”. First, the component type is a new protocol *IPv6* (lines 1-3), and the second is a component to perform dynamic reconfiguration (lines 4-32). In order to execute this dynamic reconfiguration two steps are applied: (i) to obtain the root behaviour of the system, and (ii) to invoke the computation of the reconfiguration component. Both operations are aided by toolset and hyper-code.

The code of the reconfiguration component is described using the following steps: (i) to decompose the behaviours

[2] Decomposition, modification, and composition is a process provided by the ArchWare Virtual Machine. This facility is possible: take a snapshot of the system or a specific component/connector, both represented as a sequence of behaviour; make a modification; and, compose again the behaviour with new composition. When it is composed, the ArchWare Virtual Machine automatically updates the system.

```
1   archictecture TCPIP is abstraction() {
2     behaviour is compose {
3       application is TCPIP_Application()
4       and transport  is TCPIP_Transport()
5       and net        is TCPIP_Internet()
6       and link       is TCPIP_Link()
7       and app2transp is Conn2Layers()
8       and transp2net is Conn2Layers()
9       and net2link   is Conn2Layers()
10    } where {
11      appliation::request    unifies app2transp::source
12      and app2transp::sink   unifies transport::service
13      and transpost::request unifies transp2net::source
14      and transp2net::sink   unifies net::service
15      and net::request       unifies net2link::source
16      and net2link::sink     unifies link::service
17    }
18  }
19
20  component TCPIP_Application is abstraction() {
21     port service is { ... }.
22     port request is { ... }.
23     behaviour is {
24        processRequest is function(d: DataType):DataType {
              unobservable}.
25        processResponse is function(d:DataType):DataType {
              unobservable}.
26
27        choose {
28           via service::wait receive entryData: DataType.
29           via request::call send processRequest(entryData).
30        or
31           via request::response receive replyData: DataType.
32           via service::reply send processResponse(replyData).
33        }
34     }
35  }
36
37  component TCPIP_Transport is abstraction() { ... }
38
39  component TCPIP_Internet is abstraction() { ... }
40
41  component TCPIP_Link is abstraction() { ... }
42
43  connector Conn2Layers is abstraction() {
44    ...
45    port source is { ... }
46    port sink is { ... }
47
48    behaviour is { ... }
49  }
```

**Figure 5: Partial specification of the TCP/IP stack system in π-ADL.**

of the system in components and connectors (lines 9); (ii) to assign to "variables" the instances of components that represent application, transport, and link (lines 11-13); (iii) to attribute connectors to "variables" (lines 15-17); (iv) to compose a new system behaviour (lines 19-30) with a new instance of the protocol IPv6 component (line 21). As in the scenario 1, when performing a new composition, the behaviour of the system is automatically updated.

*Scenario 3.* In the scenario 3 it is also used the process of decomposition, modification, and composition, as showed above. We use the ArchWare Virtual Machine facility to get a reference to the root of the target Transport component. After, we make a modification into the behaviour of the component. At last, the ArchWare Virtual Machine synchronize the system with this modification.

```
1   component reconfiguration is abstraction(application:
        Behaviour, system: Behaviour) {
2     behaviours : sequence[Behaviour].
3     transport : Behaviour. net : Behaviour. link : Behaviour.
4     app2transp : Behaviour. transp2net : Behaviour. net2link :
          Behaviour.
5     behaviour is {
6       behaviours := decompose system.
7
8       transport := behaviours::1::bhvr.
9       net := behaviours::2::bhvr.
10      link := behaviours::3::bhvr.
11
12      app2transp := behaviours::4::bhvr.
13      transp2net := behaviours::5::bhvr.
14      net2link := behaviours::6::bhvr.
15
16      compose {
17        application and transport and net and link
18        and app2transp and transp2net and net2link
19      } where {
20        appliation::request    unifies app2transp::source
21        and app2transp::sink   unifies transport::service
22        and transpost::request unifies transp2net::source
23        and transp2net::sink   unifies net::service
24        and net::request       unifies net2link::source
25        and net2link::sink     unifies link::service
26      }
27    }
28  }
```

**Figure 6: Implementation of scenario 1 in π-ADL.**

*Summary of π-ADL.* With these implementations, some issues are identified on π-ADL support for dynamic reconfiguration:

1. changing types needs external help, e.g. toolset, hyper-code and π-ARL. Toolset and hyper-code are used for capturing, specifying the changes, and applying the changes. While π-ARL is a language to specify refinements in the system;
2. instances modifications can be specified in architectural elements or also with toolset and hyper-code help. At first, usually the dynamic reconfiguration is tangled with nominally behaviour. However, we used other approach with a specific component to specify dynamic reconfiguration;
3. For the third scenario, the π-ADL do not provide a mechanism to design the intermediate states for updating the instances with the improvement of behaviour. However, if we use the π-ARL approach, we can have better control;
4. Also with π-ARL is possible build constraints and non-functional properties for architectural elements. However, it's not used to assess dynamic reconfiguration.

### 5.1.2 ACME ADL and Plastik

*Purpose of the ADL.* ACME/Armani is a declarative language based on the first-order predicate logic [11, 20]. Its initial purpose was to define a common interchange language for architecture design tools. Also, its statements are designed to describe the architectural structures at instance and type levels. The extensions [4, 13] to support dynamic reconfiguration preserves these initials purpose, and

```
1  component TCPIP_IPv6 is abstracion() {
2    ...
3  }
4  component reconfiguration is abstraction(system: Behaviour)
       {
5    behaviours : sequence[Behaviour].
6    application: Behaviour. transport : Behaviour. link :
         Behaviour.
7    app2transp : Behaviour. transp2net : Behaviour. net2link :
         Behaviour.
8    behaviour is {
9      behaviours := decompose system.
10
11     application := behaviours::0::bhvr.
12     transport := behaviours::1::bhvr.
13     link := behaviours::3::bhvr.
14
15     app2transp := behaviours::4::bhvr.
16     transp2net := behaviours::5::bhvr.
17     net2link := behaviours::6::bhvr.
18
19     compose {
20       application and transport and link
21       and net is TCPIP_v6
22       and app2transp and transp2net and net2link
23     } where {
24       appliation::request    unifies app2transp::source
25       and app2transp::sink   unifies transport::service
26       and transpost::request unifies transp2net::source
27       and transp2net::sink   unifies net::service
28       and net::request       unifies net2link::source
29       and net2link::sink     unifies link::service
30     }
31   }
32 }
```

**Figure 7: Implementation of scenario 2 in π-ADL.**

```
1  System StreamDecoderSystem : TCPIP_MF, PlastikMF {
2    Component mpeg-decoder : TCPIP_MpegDecoder = new
         TCPIP_Application extend with {
3      property decoder-type = "MPEG";
4      property algorithm = ...;
5    }
6    Component h263-decoder : TCPIP_H263Decoder = new
         TCPIP_Application extend with { ... }
7  // other instances of component and connector
8    ...
9  // initial attachments
10   ...
11 // programmed dynamic reconfiguration to low bandwidth
12   on (link.bandwidth == 'low') do {
13     detach mpeg-decoder.requiredService to application2transport.
           source;
14     detach transport2application.sink to mpeg-decoder.
           dataReceivedFromTransport;
15     attachments {
16       h263-decoder.requiredService to application2transport.source
           ;
17       transport2application.sink to h263-decoder.
             dataReceivedFromTransport;
18     }
19   }
20 // programmed dynamic reconfiguration to high bandwidth
21   on (link.bandwidth == 'high') do {
22     detach h263-decoder.dataTo to application2transport.source;
23     detach transport2application.sink to h263-decoder.
           dataReceivedFromTransport;
24     attachments {
25       mpeg-decoder.requiredService to application2transport.source
           ;
26       transport2application.sink to mpeg-decoder.
             dataReceivedFromTransport;
27     }
28   }
29 }
```

**Figure 8: Implementation of scenario 1 in ACME/-Plastik.**

are named as ACME/Plastik. Thereafter, any configuration and dynamic reconfiguration specified using this extension (ACME/Plastik) are based on the structure of an architecture.

*Dynamic reconfiguration support.* ACME/Plastik permits to describe foreseen and unforeseen dynamic reconfiguration. To unforeseen reconfiguration it is possible to describe as a specific behaviour of a configuration with the **on** (*conditional_expression*) **do** {*operations*} statement. Unforeseen reconfiguration can be expressed in a script that will be applied in the system at runtime aided by a toolset provided by the Plastik framework. Both, *operations* and script are composed with ACME statements and its ACME/-Plastik extension. The *conditional_expression* is composed in the Armani language [20].

*Initial architecture.* The example of the TCP/IP stack system in ACME is showed in Fig. 4. The types are specified in the **Family** statement while the instances are in the **System** statement. For types, a connector (lines 3-6) and two components (lines 7-11 and 12-16) are implemented as example. The configuration is composed by the component instances (lines 21-24), connector instances (lines 26-27), and the connections between them (lines 28-37).

*Scenario 1.* The implementation in ACME/Plastik of scenario 1 is showed in Fig. 8. The configuration is named with `StreamDecoderSystem` and it extends the `TCPIP_MF` and `PlastikMF`. The `TCPIP_MF` is showed in Fig. 4 while `Plastik_MF` in [4, 13]. Such configuration consists of: the insertion of two new components (type and instance) in lines 2-5 and line 6; the instantiation of components and connectors (omitted with comment and ... in lines 7-8); the linking of components and connectors (omitted with comment and ... in lines 7-8); and the specification of two dynamic reconfiguration situations, lines 12-19 and 21-28.

Both dynamic reconfigurations use the same similar description. First, the conditional expressions are `link.bandwidth == 'low'` and `link.bandwidth == 'high'`. Second, the *operations* are described as two unlinking (lines 13-14, 22-23) and one linking (lines 15-18, 24-27) statements. These *operations* specified the replacement of an instance of the component.

*Scenario 2.* The scenario 2, an unforeseen insertion of a new component type, is describe with an ACME/Plastik script in Fig. 9. For this scenario, the script is composed by: the insertion of a type and instance of a component (lines 1-3), the unlinking of the old component instance (lines 5-6) and the linking of the new component (lines 7-10).

```
1  Component ipv6 : TCPIP_IPv6 = new TCPIP_Internet extended
       with {
2    ...
3  }
4
5  detach ipv4.requiredService to internet2link.source;
6  detach link2internet.sink to ipv4.dataReceivedFromLink;
7  attachments {
8    ipv6.requiredService to internet2link.source;
9    link2internet.sink to ipv6.dataReceivedFromLink;
10 }
```

**Figure 9: Implementation of scenario 2 in ACME/-Plastik**

```
1  Component Type TCPIP_Transport extends TCPIP_Component with {
2    ...
3    Property behaviour = {
4      \\ description of behaviour in other languages, e.g. CSP
5    }
6    ...
7  }
```

**Figure 10: Implementation of scenario 3 in ACME/-Plastik**

*Scenario 3.* The unforeseen upgrade of a behaviour in scenario 3 can be described by the script illustrated in Fig. 10. This script defines a new version of a component with an improvement on its behaviour. The behaviour in ACME/Plastik is expressed as a property and it can be also described in other languages, such as external formal languages, see lines 3-5 in Fig. 10.

*Summary of ACME/Plastik.* The dynamic reconfiguration issues for ACME can be summarized as follows:

1. Dynamic reconfiguration in terms of structure is supported by ACME/Plastik. In terms of behaviour, dynamic reconfiguration is described using any external language embedded inside the property element. Usually, this is used for foreseen reconfiguration. This approach is limited in the sense that components and connectors usually needs the dynamic reconfiguration of their behaviour but as ACME/Plastik does not provides elements for behaviour specification, the architect has to rely on an external language. E.g. if the `TCPIP_Transport` component needs to decide if it has a buffer or not, and if it uses a error detecting or not, and so on. These situations have to be described using an external language (in general formal languages such as CSP).
2. ACME/Plastik does not provide a mechanism to control the intermediate states of a reconfiguration. E.g. the implementation of the scenario 3, the architect cannot define the approach used to update the instances.
3. For unforeseen reconfiguration ACME/Plastik relies on external scripts. This approach requires an external interpreted associated to the ADL to interpret the external script in order to reconfigure the system.

### 5.1.3 C2SADL

```
1  ...
2  component TCPIP_Transport is
3    interface // define the component ports
4      top_domain is // ports to link high level connectors
5        out
6          ...
7        in
8          ReceiveData(package: ApplicationPackage);
9      bottom_domain is // ports to link low level connectors
10       out
11         SendToInternt(package: TransportPackage);
12       in
13         ...
14   methods // define interfaces of internal behaviours
15     function pack(data: ApplicationPackage) : TransportPackage
           );
16     ...
17   behaviour // describe the external behaviour
18     ...
19     received_messages ReceiveData;
20       invoke_methods Pack;
21       always_genarate SendToInternet;
22       ...
23 end TCPIP_Transport
24 ...
```

**Figure 11: Components of TCP-IP stack system definition in C2 IDL.**

*Purpose of the ADL.* Initially, the purpose of C2 was to describe the architecture of a software based on a Graphical User Interface (GUI) [31]. Therefore, the ADL is based on hierarchically concurrent components. C2 also allows the use of messages to notify the architectural elements. C2 is subdivided in 2 languages: C2 IDL (Interface Description Language) for describing the components types, and C2 ADL to specify the configurations. For components it is possible to specify top and bottom ports, to declare internal methods, and to specify the behaviour of its ports. The implementation of the internal methods is done by the developer in the source code generated by a toolset. With C2 ADL it is possible to define instances of components and connectors, as well as links between them.

*Dynamic reconfiguration support.* The Architectural Modification Language (AML) was created to extend C2 for supporting dynamic reconfiguration [16, 27]. AML is a declarative language that uses the structure view of an existing configuration and messages to notify this configuration about a dynamic reconfiguration. This language provides statements to build the scripts that specify dynamic reconfiguration. However, in these scripts it is not possible to determine the moment to apply a dynamic reconfiguration. Because that, human intervention is needed via a toolset to trigger dynamic reconfiguration.

*Initial architecture.* The TCP/IP stack system is described in C2 IDL in Fig. 11 and C2 ADL in Fig. 12.

*Scenario 1.* For scenario 1, for changing the component according to the low or high bandwidth state, it is necessary two C2SADL scripts. In Fig. 13 is described the script for dynamic reconfiguration when bandwidth is low. This script uses two statements, a `Unweld` for unlinking

```
architecture DecoderStream is
   component_intances {
      mpeg instantiates TCPIP_MpegDecoder;
      h263 instantiates TCPIP_H263Decoder;
      transport instantiates TCPIP_Transport;
      internet instantiates TCPIP_Internet;
      link instantiates TCPIP_Link;
   }
   connectors {
      application2transport;
      transport2internet;
      internet2link;
   }
   topology {
      connector application2transport {
         top_ports { mpeg filter no_filtering; }
         bottom_ports { transport filter no_filtering; }
      }
      connector transport2internet {
         top_ports { transport filter no_filtering; }
         bottom_ports { internet filter no_filtering; }
      }
      connector internet2link {
         top_ports { internet filter no_filtering; }
         bottom_ports { link filter no_filtering; }
      }
   }
end DecoderStream;
```

**Figure 12: TCP-IP stack system implementation in C2 ADL.**

```
\\ Bandwidth is low
DecoderStream.Unweld(mpeg, application2transport);
DecoderStream.Weld(h263, application2transport);
```

**Figure 13: Partial implementation of scenario 1 in C2SADL.**

the *mpeg* instance and `Weld` for linking the *h*263 instance. Other AML's statements are `AddComponent`, `AddConnector`, `RemoveComponent`, and `RemoveConnector`. All these statements are used with a defined configuration, e.g. line 2 and 3 in Fig. 13 with the configuration named as `DecoderStream`. Despite of the fact that AML statements are based on services of the ArchStudio tool suite, there are services that are not provided by the AML language, e.g. `start()` and `stop()`.

*Scenario 2.* The second scenario, the unforeseen insertion of component instance and type, is inferred as shown in Fig. 14. Inference because both work about AML [16, 27] stated that it is possible to make insertion and deletion of types. However, they only show examples how to build dynamic reconfiguration at the instance level. Medvidovic *et al.* [17] mention types and subtypes but at design level.

Dynamic reconfiguration in Fig. 14 specifies: the insertion of a new type of component (lines 2-5), the creation a new instance of component (line 9), the unlinking of the instance of IPv4 (lines 12-13), and the linking of the instance of IPv6 (lines 16-17). The component type name is inferred by a toolset before performing dynamic reconfiguration.

```
\\ Creata a new type of component
component TCPIP_IPv6 is subtype
  all <= all TCPIP_Internet (...)
  ...
end TCPIP_IPv6

// Create a new instance of component
//   The name of the type is implicitly discovered
DecoderStream.AddComponent(TCPIP_IPv61);

// Unlink the instance of component IPv4
DecoderStream.Unweld(trasport2internet, TCPIP_IPv41);
DecoderStream.Unweld(TCPIP_IPv41, internet2link);

// Link the instance of component IPv6
DecoderStream.Weld(trasport2internet, TCPIP_IPv61);
DecoderStream.Weld(TCPIP_IPv61, internet2link);
```

**Figure 14: Implementation of scenario 2 in C2SADL.**

```
component TCPIP_Transport is
   ...
   behaviour // describe the external behaviour
      ...
      received_messages ReceiveData;
         invoke_methods VerifyData, Pack;
         always_genarate SendToInternet;
      ...
end TCPIP_Transport
...
```

**Figure 15: Implementation of scenario 3 in C2SADL.**

*Scenario 3.* The third scenario is an improvement of the behaviour that can be modeled by the `behaviour` statement via a sequence of invocations to internal methods. Example of this change, see line 6 in Fig. 15, the invocation is modified to verify the received and packed data before sending a transport package to the internet layer. If the change is an improvement of an internal method, it is not possible to describe with C2SADL.

*Summary of C2SADL.* After the C2SADL implementation of the three scenarios, we identified these issues:

1. ArchStudio tool provides some services for supporting dynamic reconfiguration, e.g. to start and to stop architectural elements. However, these services are not available in AML. Thus, the dynamic reconfiguration specification in AML is limited.

2. foreseen dynamic reconfiguration needs human intervention because C2SADL do not provide statements for the system apply dynamic reconfiguration.

3. it does not support the specification of dynamic reconfiguration inside of the architectural elements. Because that, the foreseen dynamic reconfiguration is implemented only at the implementation level;

4. it does not provide a mechanism for controlling and monitoring the intermediate states of a dynamic reconfiguration;

5. it also does not support the assessment of dynamic reconfiguration.

### 5.1.4 Dynamic Wright

*Purpose of the ADL.* Dynamic Wright has focus on the structure and behaviour of an architecture [2]. Dynamic Wright supports the description of the architectural elements types structures and behaviours[3], and initial configuration. Structure, initial configuration, instances, and links are described in a declarative form, while behaviours are specified in a graph-based and a variant of the Communicating Sequential Process (CSP) form.

*Dynamic reconfiguration support.* Dynamic Wright only supports foreseen dynamic reconfiguration. It is built as a special behaviour of the initial configuration. It has the same base of a "nominally" behaviour of architectural elements. The extension described in [2] proposes additional statements and control events. The statements are **new**, **remove**, `attach`, and **detach**. The control events are used to define the specific moment to perform a dynamic reconfiguration.

As Dynamic Wright does not support unforeseen reconfiguration, the scenarios 2 and 3 cannot be implemented.

*Initial architecture.* The initial configuration of the TCP/-IP stack system is illustrated in Fig. 16. The component and connector types are specified in the `Style ... EndStyle` (lines 1-16) statements. The implementation of the `TCPIP_MPEG−Decoder` Component is described with two ports (lines 3-4) and a behaviour (lines 6-7). The configuration is described with the `Configuror ... EndConfiguror` statement. The initial configuration instances and connections are specified at lines 18-27.

*Scenario 1.* Fig. 16 also illustrates the specification of scenario 1. It uses the **Configuror** *[initial configuration ]***where** *[behaviours]* Statement. *behaviours* are special named behaviours of the configuration for specifying dynamic reconfiguration. In this case, there are two specification: lines 29-36 for low bandwidth and lines 37-44 for high bandwidth. The following steps were used to define the first and the second dynamic reconfiguration situations:

1. to define a name (lines 29 and 37).
2. to specify the control events that define the moment for performing the operations of dynamic reconfiguration (lines 30 and 38).
3. to determine the type of configuration to use (lines 31 and 39). Dynamic Wright use the term `Style` for this purpose.

[3] In Dynamic Wright it is possible to specify the behaviour for component ports and computations, and connector roles and glues.

```
1  Style TCPIP−Style
2    Component TCPIP_MPEG−Decoder
3      Port dataReceivedFromTransport = ...
4      Port requiredService = ...
5        [describe the port behaviour using a variant of CSP]
6      Computation = ...
7        [describe the component behaviour using a variant of CSP]
8    ...
9    Connector Conn2Layers
10     Role source = ...
11     Role sink = ...
12     Glue = ...
13       [describe the connector behaviour using a variant of CSP]
14   Constraints
15     ...
16 EndStyle
17 Configuror DecoderStream
18   Style TCPIP−Style
19    new.mpeg : TCPIP_MPEG−Decoder
20    → new.h263 : TCPIP_H263−Decoder
21    ... [other component instances]
22    → new.app2trans : Conn2Layers
23    → new.trans2net : Conn2Layers
24    → new.net2link : Conn2Layers
25    → attach.mpeg.requiredService.to.app2trans.source
26    → attach.app2trans.sink.to.transport.service
27    ... [other attachments]
28   where
29    WaitForBandwidthLow = (
30    link.control.bandwidthDown → mpeg.control.off → h263.control
        .on
31    → Style TCPIP−Style
32      detach.mpeg.requiredService.to.app2trans.source
33      → attach.h263.requiredService.to.app2trans.source
34      → detach.trans2app.sink.to.mpeg.dataReceivedFromTransport
35      → attach.trans2app.sink.to.h263.dataReceivedFromTransport
36    ) □ §
37    WaitForBandwidthHigh = (
38      link.control.bandwidthUp → h263.control.off → −> mpeg.
          control.on
39      → Style TCPIP−Style
40        detach.h263.requiredService.to.app2trans.source
41        → attach.mpeg.requiredService.to.app2trans.source
42        → detach.trans2app.sink.to.h263.
              dataReceivedFromTransport
43        → attach.trans2app.sink.to.mpeg.
              dataReceivedFromTransport
44     ) □ §
45 EndConfiguror
```

**Figure 16: Implementation of scenario 1 in Dynamic Wright**

4. to specify the operations to perform dynamic reconfiguration (lines 32-35 and 40-43).
5. to use the statements of the external choice (□) and successfully terminate (§), lines 36 and 44.

*Summary of Dynamic Wright.* The following issues are identified after the implementation of the scenario 1:

1. it does not provide operations to define dynamic reconfiguration for the type level of architectural elements;
2. this ADL provides a refined mechanism of control events to determine the moment to perform a dynamic reconfiguration. Although, it does not provide a mechanism to control and monitor the intermediate states. e.g. if the application of the scenario 1 is a banking system, the state must be copied from the instance of the

**Table 1: Foreseen and unforeseen dynamic reconfiguration support in ADLs.**

| | Foreseen | Unforeseen |
|---|---|---|
| $\pi$-ADL | inside of all architectural elements behaviour | ADL statements aided by the ArchWare toolset |
| ACME Plastik | special behaviours in the configuration | ADL statements and external language aided by the Plastik framework |
| C2SADL | limited support by the ADL | external script in AML |
| Dynamic Wright | special behaviours in the configuration | not provided by the ADL |

replaced component to the new component. Dynamic Wright does not provide operations for this purpose.

3. despite of providing the specification of behaviour for components and connectors, it is not possible to define dynamic reconfiguration for this architectural elements;

4. similarly to the other ADLs, Dynamic Wright does not provide a mechanism for the assessment of dynamic reconfiguration.

## 5.2 Summary of comparison

The summary of the four ADLs support for dynamic reconfigurations is shown in Tab. 1. $\pi$-ADL and ACME/Plastik are the only ADLs that support both foreseen and unforeseen dynamic reconfigurations. As C2SADL with the AML language do not provide a way to specify an internal initiation of dynamic reconfiguration, foreseen dynamic reconfiguration cannot be automatically triggered. Dynamic Wright supports only foreseen reconfiguration by defining the special behaviour of a configuration.

The four ADLs do not address the issue of consistent reconfiguration (reconfiguration at type and instance levels) as shown the Tab. 2. $\pi$-ADL and ACME provides operations for both levels, however the two ADLs use external help to specify reconfiguration at the type level. C2SADL AML provides operations to modify instances. In spite of the toolset provides the services for both levels. Dynamic Wright does not provide operations to specify dynamic reconfiguration for the type of architectural elements.

The support for specifying the behaviour of the architectural elements can be subdivided in two categories: nominally and dynamic reconfiguration. Nominally is a set of functions that the architectural elements can invoke. Dynamic reconfiguration is the specification of a behaviour to compose a foreseen dynamic reconfiguration. The ADLs support for both is resumed in Tab. 3. $\pi$-ADL supports both in the architectural element behaviour. ACME/Plastik nominally behaviour can be considered in two ways: a property of architectural element described with an external language, e.g. CSP; and, port/rule communication for components and connectors; for dynamic reconfiguration, ACME/Plastik provides the on-do clause. Dynamic Wright glue and

**Table 2: Dynamic reconfiguration support for modification on type and instance level.**

| | Type | Instance |
|---|---|---|
| $\pi$-ADL | ADL statements with external help | ADL statements |
| ACME Plastik | ACME/Plastik statements with external help | ACME/Plastik statements |
| C2SADL | not provided by the ADL | operations defined in AML |
| Dynamic Wright | not provided by the ADL | operations defined in an extension of the ADL |

**Table 3: ADL support for behaviour specification.**

| | Nominally | Dynamic reconfiguration |
|---|---|---|
| $\pi$-ADL | All architectural elements with $\pi$-ADL behaviour. | tangled with nominally behaviour. |
| ACME Plastik | External language used in properties of the architectural elements. | ACME/Plastik on-do statement. |
| C2SADL | Component behaviour based on event communication. | Not provided. |
| Dynamic Wright | Variant of CSP in connector glue and component computation. | Configuration special behaviour with variant of CSP. |

computation clauses define the nominally behaviour of connectors and components; a configuration has a special behaviour for dynamic reconfiguration. C2SADL has no behaviour for dynamic reconfiguration but nominally component behaviour is described by events communication.

None of the four ADLs has a mechanism for assessing the system. $\pi$-ADL can rely on $\pi$-AAL for defining and analyzing the architectural constraints and non-functional properties. While ACME/Plastik has Armani. even though it has no formal semantics, and therefore, it is not liable to rigorous analysis [33]. C2SADL provides type verification for events communication. Dynamic Wright provides a refined mechanism of control events to determine the moment to perform a dynamic reconfiguration. Although, the assessment of dynamic reconfiguration is not provided on four ADLs.

# 6. CONCLUSION

In this paper we analysed the support of ADLs for handling dynamic reconfigurations. We started with a motivation example and some reconfiguration scenarios and we described the example and the scenarios using four well-known ADLS. We used the example and the specifications to discuss the following issues: (i) how each ADL addresses the issue of consistently reconfiguring both instances and types; (ii) how each ADL takes into account the behaviour of the architectural elements during reconfiguration; and (iii) how each ADL supports the assessing of dynamic reconfiguration.

In comparison to related works, we analysed some important issues for the dynamic reconfiguration support at the architectural level: foreseen and unforeseen changes; instance and type level modifications; structure and behaviour for all architectural elements; definition of nominally and dynamic reconfiguration behaviour for all architectural elements; and, the analysis of dynamic reconfiguration.

We can conclude that some issues still remain open as no ADL provides support for all of them together: how to apply the changes made in type level to their respective instances? how to control the set of intermediate states of a software system during a dynamic reconfiguration? how to assess dynamic reconfiguration?

# 7. ACKNOWLEDGMENTS

Special thanks to Collège Doctoral International (CDI)/Université Européenne de Bretagne (UEB) for financial support this work.

# 8. REFERENCES

[1] R. Allen, R. Douence, and D. Garlan. Specifying dynamism in software architectures. In *In Proceedings of the Workshop on Foundations of Component-Based Systems*, pages 11–22, 1997.

[2] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998. An expanded version of a the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997.

[3] L. Bass, P. C. Clements, and R. Kazman. *Software Architecture in Pratice*. Addison-Wesley Professional, 2 edition, 2003.

[4] T. V. Batista, A. T. A. Gomes, G. Coulson, C. Chavez, and A. F. Garcia. On the interplay of aspects and dynamic reconfiguration in a specification-to-deployment environment. In *Proceedings of the 2nd European Conference on Software Architecture*, ECSA '08, pages 314–317, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[6] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Techinical Report 2004-477, Software Technology Laboratory, School of Computing, Queen's University, 2004.

[7] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A Survey of Self-Management in Dynamic Software Architecture Specifications. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems*, pages 28–33. ACM, 2004.

[8] E. M. Dashofy, A. van Der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. *Proceedings of the first workshop on Self-healing systems - WOSS '02*, page 21, 2002.

[9] P. Fingar. Component-based frameworks for e-commerce. *Communication of the ACM*, 43:61–67, October 2000.

[10] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1997.

[11] D. Garlan, R. Monroe, and D. Wile. Acme: Architectural description of component-based systems. *Foundations of component-based systems*, pages 47–68, 2000.

[12] A. Gomes, T. V. Batista, A. Joolia, and G. Coulson. Architecting Dynamic Reconfiguration in Dependable Systems. In R. de Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 237–261. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2007.

[13] A. Joolia, T. V. Batista, G. Coulson, and A. T. A. Gomes. Mapping adl specifications to an efficient and reconfigurable runtime component platform. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 131–140, 2005.

[14] M. H. Kacem, M. Jmaiel, A. H. Kacem, and K. Drira. Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 189–195, 2005.

[15] J. Magee and J. Kramer. Self organising system structuring. In *Joint proceedings of the second international software architecture workshop (ISAW) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, ISAW '96, pages 35–38, New York, NY, USA, 1996. ACM Press.

[16] N. Medvidovic. Adls and dynamic architecture changes. *Joint proceedings of the second international software architecture workshop (ISAW) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops* -, pages 24–27, 1996.

[17] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '96, pages 24–32, New York, NY, USA, 1996. ACM.

[18] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[19] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 178 –187, 2000.

[20] R. Monroe. Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, January 2001. Version 2.3.

[21] R. Morrison, D. Balasubramaniam, G. Kirby,

K. Mickan, B. Warboys, R. M. Greenwood, I. Robertson, and B. Snowdon. A framework for supporting dynamic systems co-evolution. *Automated Software Engineering*, 14(3):261–292, July 2007.

[22] F. Oquendo. Formally refining software architectures with $\pi$-arl: a case study. *SIGSOFT Softw. Eng. Notes*, 29:1–26, September 2004.

[23] F. Oquendo. $\pi$-adl: An architecture description language based on the higher-order typed $\pi$-calculus for specifying dynamic and mobile software architectures. *Software Engineering Notes*, 29(4):1–14, 2004.

[24] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWare: Architecting Evolvable Software. *Software Architecture*, 3047/2004:257–271, 2004.

[25] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

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

[27] P. Oreizy and R. N. Taylor. On the role of software architectures in runtime system reconfiguration. In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, pages 61–70, may 1998.

[28] M. Pinto, L. Fuentes, and J. M. Troya. Daop-adl: An architecture description language for dynamic component and aspect-based development. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[29] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[30] Society Automo Engineers International (SAE). *Aerospace Standard AS5506 - Architecture Analysis & Design Language (AADL)*, November 2004.

[31] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. Whitehead, E. James, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, June 1996.

[32] J. Vera, L. Perrochon, and D. C. Luckham. Event-based execution architectures for dynamic software systems. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA)*, WICSA, pages 303–318, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[33] P. Waewsawangwong. A constraint architectural description approach to self-organising component-based software systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 81–83, Washington, DC, USA, 2004. IEEE Computer Society.