ABSTRACT
Dynamic reconfiguration is the action of modifying a soft-
ware system at runtime. Several works have been using ar-
chitectural specification as the basis for dynamic reconfig-
uration. Indeed ADLs (architecture description languages) let
architects describe the elements that could be reconfig-
ured as well as the set of constraints to which the system
must conform during reconfiguration. In this work, we in-
estigate the ADL literature in order to illustrate how recon-
figuration is supported in four well-known ADLs: π-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 as-
sessing 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, π-ADL, ACME, C2SADL, Dy-
namic Wright

1. INTRODUCTION

The disciplined software engineering relies on software ar-
chitectures to describe systems [3]. For modeling a software
architecture, the academy proposed the architecture descrip-
tion language (ADL) and their toolsets [10, 18, 29]. Also,
the industry has used ADLs to develop systems. Industry
eamples are (i) π-ADL has been used to architect and refine
federated knowledge management systems in Engineering
Ingegneria Informatica - Italy [22]; (ii) and AADL (Archi-
tecture 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 pro-
erties, and behaviour.

Software systems evolve over their life time [5, 21, 26]. Dy-
namic reconfiguration is when the evolution is performed at
runtime with no service disruption. The dynamic reconfig-
uration can be handled by architectural concepts in an ADL.
However among many existing ADLs, only few allow model-
ing dynamic reconfiguration. π-ADL [23], ACME/Plastik [4,
13], C2SADL [16, 27], DAOP-ADL [28], Darwin [15], Dy-
namic Wright [1, 2], Rapide [32], Weaves [25], and xADL [8]
are typical examples. Nevertheless, there is no current con-
sensus 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 litera-
ture; and, (ii) to illustrate how four well-known ADLs sup-
port dynamic reconfiguration: π-ADL, ACME, C2 SADL
and Dynamic Wright. We chose these four languages be-
cause they rely on different paradigms: the higher order
typed π-calculus [23], first order predicate logic [11], compo-
nent- and event-based [31, 16], and, graph grammars and
communicating sequential processes (CSP) [2, 1], respec-
tively. They also complement each other as π-ADL models
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 the 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 π-ADL because it provides both description with behaviours thanks to its π-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 at the type level (IPv6 extends the type of IPv4).

Also, the dynamic reconfiguration can be foreseen or unforeseen [12]. The unforeseen 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, Oalan, 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, sequence, choice, and iteration). Only 3 approaches (CommUnity, Dynamic Wright, and Gerel) provide full support while the other only support basic composition. Finally, Bradbury 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. While ADL surveys have reached good understanding of the types of architectural elements defined prior runtime, fixed sets of 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 toolkit (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 \texttt{behaviour} is ... where ...! (lines 2-17). This architectural behaviour defines the instances of components (lines 3-6) and connectors (lines 7-9). The statement \texttt{where} in behaviour specifies the connections between components and connectors (lines 11-16). The example of component TCP/IP Application is partially showed at lines 29-35: two ports and a behaviour. This 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 the toolset and hyper-code.

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

\footnotesize
\begin{itemize}
  \item 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.
\end{itemize}
architecture TCPIP is abstraction() {  
  behaviour is compose  
  application is TCPIP_Application()  
  and transport is TCPIP_Transport()  
  and net is TCPIP_Net()  
  and link is TCPIP_Link()  
  and app2transp is Conn2Layers()  
  and transp2net is Conn2Layers()  
  and net2link is Conn2Layers()  
} where {  
  application::request unifies app2transp::source  
  and app2transp::sink unifies transport::service  
  and transport::request unifies transp2net::source  
  and transp2net::sink unifies net::service  
  and net::request unifies net2link::source  
  and net2link::sink unifies link::service  
}  

component TCP/IP_Application is abstraction() {  
  port service is ( ... ).  
  port request is ( ... ).  
  behaviour is {  
    processRequest : function(d: DataType): DataType {  
      unobservable.  
    }  
    processResponse : function(d:DataType): DataType {  
      unobservable.  
    }  
  }  
}  

component TCP/IP_Transport is abstraction() { ... }  

component TCP/IP_Net is abstraction() { ... }  

cOMPONENT TCPIP_LINK is abstraction() { ... }  

Component TCP/IP_Link is abstraction() { ... }  

connector Conn2Layers is abstraction() {  
  port source is ( ... )  
  port sink is ( ... )  
  behaviour is ( ... )  
}

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6}
\caption{Implementation of scenario 1 in \(\pi\)-ADL.}
\end{figure}

Summary of \(\pi\)-ADL. With these implementations, some issues are identified on \(\pi\)-ADL support for dynamic reconfiguration:

1. changing types needs external help, e.g. toolset, hyper-code and \(\pi\)-ARL. Toolset and hyper-code are used for capturing, specifying the changes, and applying the changes. While \(\pi\)-ARL is a language to specify refinement 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 another approach with a specific component to specify dynamic reconfiguration;

3. For the third scenario, the \(\pi\)-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 \(\pi\)-ARL approach, we can have better control;

4. Also with \(\pi\)-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 initial purpose, and
component TCP_IPv6 is abstraction() {
  ...
}

component reconfiguration is abstraction (system: Behaviour) {
  behaviours : sequence (Behaviour).
  behaviour is {
    behaviours := decompose system.
    application := behaviours::1::bhvr.
    transport := behaviours::1::bhvr.
    app2transp := behaviours::4::bhvr.
    transp2net := behaviours::5::bhvr.
    net2link := behaviours::6::bhvr.
  }
  compose {
    application and transport and link
    and app2transp and transp2net and net2link
  } where {
    application::request unifies app2transp::source
    and app2transp::sink unifies transport::service
    and transport::request unifies transp2net::source
    and transp2net::sink unifies net::service
    and net::request unifies net2link::source
    and net2link::sink unifies link::service
}
}

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

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).
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 unforeseen reconfiguration. This approach is limited in the sense that components and connectors usually need the dynamic reconfiguration of their behaviour but as ACME/Plastik does not provide 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 scripts that specify dynamic reconfiguration 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.

**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 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 unforeseen reconfiguration. This approach is limited in the sense that components and connectors usually need the dynamic reconfiguration of their behaviour but as ACME/Plastik does not provide 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**
Figure 12: TCP-IP stack system implementation in C2SADL.

```
architecture DecoderStream is
component_instances {
  mpeg instantiates TCP/IP_MpegDecoder;
h263 instantiates TCP/IP_H263Decoder;
transport instantiates TCP/IP_Transport;
internet instantiates TCP/IP_Internet;
link instantiates TCP/IP_Link;
}
connectors {
  application2transport;
  transport2internet;
  internet2link;
}
topology ports {
  top_ports { mpeg filter no_filtering; }
  bottom_ports { transport filter no_filtering; }
}
end DecoderStream;
```

Figure 13: Partial implementation of scenario 1 in C2SADL.

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

Figure 14: Implementation of scenario 2 in C2SADL.

```
\ \ Create a new type of component
component TCP/IP_IPv6 is subtype
all <= all TCP/IP_Internet (...)
...
end TCP/IP_IPv6
```

```
\ \ Create a new instance of component
DecoderStream.AddComponent(TCP/IP_IPv6);
```

```
\ \ Unlink the instance of component IPv4
DecoderStream.Unweld(TCP/IP_IPv4);
```

```
\ \ Link the instance of component IPv6
DecoderStream.Weld(TCP/IP_IPv6);
```

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.

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, 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 TCP/IP-MPEG-Decoder Component is described with two ports (lines 3-4) and a behaviour (lines 6-7). The configuration is described with the Configurator ... EndConfigurator 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 Configurator [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.

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

```c
Style TCP/IP-Style
Component TCP/IP_MPEG-Decoder
Port dataReceivedFromTransport = ... 
Port requiredService = ... 
[describe the port behaviour using a variant of CSP] 
Computation = ... 
[describe the component behaviour using a variant of CSP] 
... 
Connector ConnLayers 
Role source = ... 
Role sink = ... 
Glue = ... 
[describe the connector behaviour using a variant of CSP] 
Constraints 
... 
EndStyle 
... 
EndConfigurator 
```

![Figure 16: Implementation of scenario 1 in Dynamic Wright](image)

4. to specify the operations to perform dynamic reconfiguration (lines 32-35 and 40-43).
5. to use the statements of the external choice (\(\square\)) and successfully terminate (\(\square\)), 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
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. π-ADL and ACME/Plastik do not support both foreseen and unforeseen dynamic reconfigurations. As C2SADL and Dynamic Wright 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 by the Tab. 2. π-ADL and ACME/Plastik provide operations for both levels, however, the two ADLs use external help to specify reconfiguration at the type level. C2SADL and Dynamic Wright provide 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. π-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>
<thead>
<tr>
<th>ADL</th>
<th>Foreseen behaviour</th>
<th>Unforeseen behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>π-ADL</td>
<td>inside of all architectural elements behaviour</td>
<td>ADL statements aided by the ArchWare toolset</td>
</tr>
<tr>
<td>ACME Plastik</td>
<td>special behaviours in the configuration</td>
<td>ADL statements and external language aided by the Plastik framework</td>
</tr>
<tr>
<td>C2SADL</td>
<td>limited support by the ADL</td>
<td>external script in AML</td>
</tr>
<tr>
<td>Dynamic Wright</td>
<td>special behaviours in the configuration</td>
<td>not provided by the ADL</td>
</tr>
</tbody>
</table>

None of the four ADLs has a mechanism for assessing the system. π-ADL can rely on π-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.

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

<table>
<thead>
<tr>
<th>ADL</th>
<th>Foreseen</th>
<th>Unforeseen</th>
</tr>
</thead>
<tbody>
<tr>
<td>π-ADL</td>
<td>ADL statements with external help</td>
<td>ADL statements</td>
</tr>
<tr>
<td>ACME Plastik</td>
<td>ACME/Plastik statements with external help</td>
<td>ACME/Plastik statements</td>
</tr>
<tr>
<td>C2SADL</td>
<td>not provided by the ADL</td>
<td>operations defined in AML</td>
</tr>
<tr>
<td>Dynamic Wright</td>
<td>not provided by the ADL</td>
<td>operations defined in an extension of the ADL</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ADL</th>
<th>Nominally</th>
<th>Dynamic reconfiguration</th>
</tr>
</thead>
<tbody>
<tr>
<td>π-ADL</td>
<td>All architectural elements with π-ADL behaviour</td>
<td>tangled with nominally behaviour</td>
</tr>
<tr>
<td>ACME Plastik</td>
<td>External language used in properties of the architectural elements</td>
<td>ACME/Plastik on-do statement</td>
</tr>
<tr>
<td>C2SADL</td>
<td>Component behaviour based on event communication</td>
<td>Not provided</td>
</tr>
<tr>
<td>Dynamic Wright</td>
<td>Variant of CSP in connector glue and component computation</td>
<td>Configuration special behaviour with variant of CSP</td>
</tr>
</tbody>
</table>

Table 3: ADL support for behaviour specification.
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


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


