

# $\pi$ -ADL: An Architecture Description Language based on the Higher-Order Typed $\pi$ -Calculus for Specifying Dynamic and Mobile Software Architectures

Flavio Oquendo

University of Savoie at Annecy

ESIA – LISTIC Lab – Formal Software Architecture and Process Research Group

B.P. 806 – 74016 Annecy Cedex – France

Flavio.Oquendo@univ-savoie.fr

## Abstract

A key aspect of the design of any software system is its architecture. An architecture description, from a runtime perspective, should provide a formal specification of the architecture in terms of components and connectors and how they are composed together. Further, a dynamic or mobile architecture description must provide a specification of how the architecture of the software system can change at runtime. Enabling specification of dynamic and mobile architectures is a large challenge for an Architecture Description Language (ADL). This article describes  $\pi$ -ADL, a novel ADL that has been designed in the ArchWare European Project to address specification of dynamic and mobile architectures. It is a formal, well-founded theoretically language based on the higher-order typed  $\pi$ -calculus. While most ADLs focus on describing software architectures from a structural viewpoint,  $\pi$ -ADL focuses on formally describing architectures encompassing both the structural and behavioural viewpoints. The  $\pi$ -ADL design principles, concepts and notation are presented. How  $\pi$ -ADL can be used for specifying static, dynamic and mobile architectures is illustrated through case studies. The  $\pi$ -ADL toolset is outlined.

**Keywords:** Architecture Description Languages, Specification Languages, Dynamic Architectures, Mobile Architectures,  $\pi$ -Calculus

## 1. Introduction

Software architecture has emerged as an important subdiscipline of software engineering. A key aspect of the design of any software system is its architecture, i.e. the fundamental organisation of the system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [25].

A software architecture can be characterised according two its evolution at runtime:

- static architectures: the architecture does not evolve during the execution of the system;
- dynamic architectures: the architecture can evolve during the execution of the system, e.g. components can be created, deleted, reconfigured, or moved at runtime;
- mobile architectures: components can logically move during the execution of the system.

An architecture description specifies an architecture. An architecture can be described according to different viewpoints. From a runtime perspective, two viewpoints are frequently used in software architecture [25]: the structural viewpoint and the behavioural viewpoint.

The structural viewpoint may be specified in terms of:

- components (units of computation of a system),
- connectors (interconnections among components for supporting their interactions),
- configurations of components and connectors.

Thereby, from a structural viewpoint, an architecture description should provide a formal specification of the architecture in terms of components and connectors and how they are composed together. Further, in the case of a dynamic or mobile architecture, it must provide a specification of how its components and connectors can change or move at runtime.

The behavioural viewpoint may be specified in terms of:

- actions a system executes or participates in,
- relations among actions to specify behaviours,
- behaviours of components and connectors, and how they interact.

A large challenge for an Architecture Description Language (ADL) is the ability to describe static but also dynamic and mobile software architectures from structural and behavioural viewpoints. Indeed, for describing dynamic and mobile architectures, an ADL must be able to describe changing structures and behaviours of components and connectors (including creation/deletion/reconfiguration/ moving) at runtime.

This article describes the  $\pi$ -Architecture Description Language ( $\pi$ -ADL) [43][41] that enables specification of static, dynamic and mobile architectures.  $\pi$ -ADL, a novel ADL that has been designed in the ArchWare<sup>1</sup> European Project, is a formal specification language:

- it is a formal, well-founded theoretically language based on the higher-order typed  $\pi$ -calculus [47];
- it is automated by tools, i.e. a specification and verification toolset providing support for automated checking.

$\pi$ -ADL takes its roots in previous work concerning the use of  $\pi$ -calculus as semantic foundation for architecture description languages [14][13]. Indeed, a natural candidate for expressing (runtime) behaviour would be the  $\pi$ -calculus as it is [36], which provides a general model of computation and is Turing-complete. This means that in  $\pi$ -calculus “every computation is possible but not necessarily easy to express”. In fact, the classical  $\pi$ -calculus is not suitable as an ADL since it does not provide architecture-

<sup>1</sup> The ArchWare European Project is partially funded by the Commission of the European Union under contract No. IST-2001-32360 in the IST-V Framework Program.

centric constructs to easily express architectures in particular with respect to architectural structures. Therefore, a language encompassing both structural and behavioural architecture-centric constructs is needed.  $\pi$ -ADL is this encompassing language, defined as a domain-specific extension of the higher-order typed  $\pi$ -calculus. It achieves Turing completeness and high architecture expressiveness with a simple formal notation.

The remainder of this article is organised as follows. Section 2 introduces  $\pi$ -ADL design principles. Section 3 presents  $\pi$ -ADL concepts and notation. Section 4 illustrates through three case studies how  $\pi$ -ADL can be used for specifying a static architecture, a dynamic architecture, and a mobile architecture. In section 5, we compare  $\pi$ -ADL with related work. To conclude we summarise, in section 6, the main contributions of this article and briefly outlines the  $\pi$ -ADL toolset for supporting architecture specification and verification.

## 2. Design Principles

$\pi$ -ADL provides the core structure and behaviour constructs for describing static, dynamic and mobile software architectures. It is a formal specification language designed to be executable [24] and to support automated verification.

The following general principles guided the design of  $\pi$ -ADL:

- formality:  $\pi$ -ADL is a formal language: it provides a formal system, at the mathematical sense, for describing static, dynamic and mobile architectures and reasoning about them;
- runtime perspective:  $\pi$ -ADL focuses on the formal description of architectures from the runtime viewpoint: the (runtime) structure, the (runtime) behaviour, and how these may evolve over time;
- executability:  $\pi$ -ADL is an executable language (a virtual machine runs specifications of software architectures);
- user-friendliness:  $\pi$ -ADL supports different concrete syntaxes – textual [14][53] and graphical [6] (including UML-based) notations – to ease its use by architects and engineers.

Based on these general principles, the design of  $\pi$ -ADL followed the following language design principles [40][51][52]:

- the principle of correspondence: the use of names are consistent within  $\pi$ -ADL, in particular there is a one to one correspondence between the method of introducing names in declarations and parameter lists;
- the principle of abstraction: all major syntactic categories have abstractions defined over them (in  $\pi$ -ADL, it includes abstractions over behaviours and abstractions over data),
- the principle of type completeness: all types are first-class without any restriction on their use.

In first-class citizenship, in addition to rights derived from type completeness (i.e. where a type may be used in a constructor, any type is legal without exception), there are properties possessed by all values of all types that constitute their civil rights in the language. In  $\pi$ -ADL they are:

- the right to be declared,
- the right to be assigned,

- the right to have equality defined over them,
- the right to persist.

Additionally,  $\pi$ -ADL provides an extension mechanism, i.e. new constructs can be defined on top of the language using user-defined mixfix abstractions. This extension mechanism provides the basis for providing style-based definitions [15].

In ArchWare, a style definition notation [15], built on the  $\pi$ -ADL, provides the style constructs from which the base component-and-connector style and other derived styles can be defined. Conceptually, an architectural style provides:

- a set of abstractions for architectural elements,
- a set of constraints (i.e. properties that must be satisfied) on architectural elements, including legal compositions,
- a set of additional analyses that can be performed on architecture descriptions constructed in the style.

## 3. Concepts and Notation

### 3.1 Core Concepts

$\pi$ -ADL supports description of software architectures from a runtime perspective. In  $\pi$ -ADL, an architecture is described in terms of components, connectors, and their composition. Figure 1 depicts its main constituents.

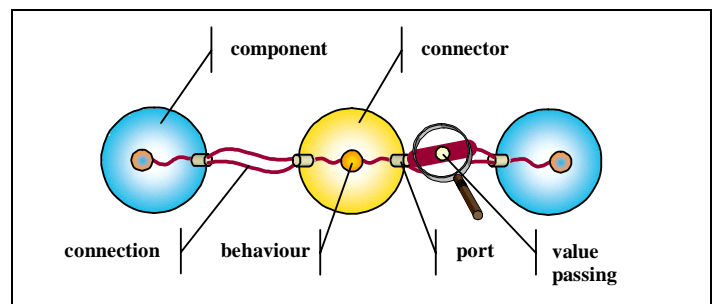


Figure 1. Architectural concepts in  $\pi$ -ADL

Components are described in terms of external ports and an internal behaviour. Their architectural role is to specify computational elements of a software system. The focus is on computation to deliver system functionalities.

Ports are described in terms of connections between a component and its environment. Their architectural role is to put together connections providing an interface between the component and its environment. Protocols may be enforced by ports and among ports.

Connections are basic interaction points. Their architectural role is to provide communication channels between two architectural elements.

A component can send or receive values via connections. They can be declared as output connections (values can only be sent), input connections (values can only be received), or input-output connections (values can be sent or received).

Connectors are special-purpose components. They are described as components in terms of external ports and an internal behav-

our. However, their architectural role is to connect together components. They specify interactions among components.

Therefore, components provide the locus of computation, while connectors manage interaction among components. A component cannot be directly connected to another component. In order to have actual communication between two components, there must be a connector between them.

Both components and connectors comprise ports and behaviour. In order to attach a port of a component to a port of a connector, at least a connection of the former port must be attached with a connection of the later port. A connection provided by a port of a component is attached to a connection provided by a port of a connector by unification or value passing. Thereby, attached connections can transport values (that can be data, connections, or even architectural elements).

From a black-box perspective, only ports (with their connections) of components and connectors and values passing through connections are observable. From a white-box perspective, internal behaviours are also observable.

Components and connectors can be composed to construct composite elements (see Figure 2), which may themselves be components or connectors. Composite elements can be decomposed and recomposed in different ways or with different components in order to construct different compositions.

Composite components and connectors comprise external ports (i.e. observable from the outside) and a composition of internal architectural elements. These external ports receive values coming from either side, incoming or outgoing, and simply relay it to the other side keeping the mode of the connection. Ports can also be declared to be restricted. In that case, constituents of composite elements can use connections of restricted ports to interact with one another but not with external elements.

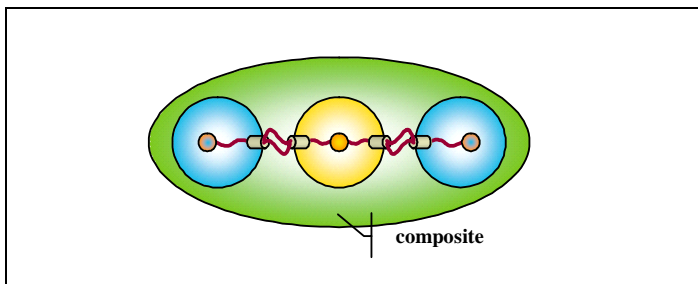


Figure 2. Architectural composition in  $\pi$ -ADL

Architectures are composite elements representing systems. An architecture can itself be a composite component in another architecture, i.e. a sub-architecture.

$\pi$ -ADL provides primitive constructs for supporting the description of all these architectural concepts. In  $\pi$ -ADL, architectures, components, and connectors are formally specified in terms of typed abstractions over behaviours.

### 3.2 Formal System

$\pi$ -ADL is formally defined by a formal transition and type system. The formal system of  $\pi$ -ADL is defined in a layered approach:

- the formal system of the base layer, named  $\pi$ -ADL<sub>B</sub>, provides only connection and behaviour constructs;
- the formal system of the first-order layer, named  $\pi$ -ADL<sub>FO</sub>, extends  $\pi$ -ADL<sub>B</sub> with data and structure constructs: base data types and type constructors;
- the formal system of the higher-order layer, named  $\pi$ -ADL<sub>HO</sub>, extends  $\pi$ -ADL with full first-class citizenship.

The base layer<sup>2</sup> of  $\pi$ -ADL is the language fragment for describing typed behaviours. It provides only a void data type and value, a base type *Behaviour*, and the type constructors *connection* and *abstraction*. This fragment of  $\pi$ -ADL is closely related to the typed  $\pi$ -calculus [48].

The first-order layer of  $\pi$ -ADL extends the base layer with base types and type constructors. Its type system includes the base types *Natural*, *Integer*, *Real*, *Boolean*, *String*, and *Any*<sup>3</sup>; the type constructors *tuple*<sup>4</sup>, *view*, *union*, *quote*, *variant*, and *location*; the collection type constructors *sequence*, *set*, and *bag* and introduces connection mobility.

The higher-order layer of  $\pi$ -ADL extends the first-order layer with full first-class citizenship including behaviour mobility.

This layered definition of  $\pi$ -ADL allows to easily extend the type system with new base types and new type constructors. In that way,  $\pi$ -ADL can be seen as an open family of layered languages having as root the base layer.

### Typing Rules

Behaviours use connections (at ports) to connect, and transmit values to one another via connections (including connections and behaviours themselves). Let us assume that there exists an enumerable infinite set of typed values.

There are different kinds of values:

- behaviours are values that can be enacted,
- connections are values that can be used to engage in communications,
- values<sup>5</sup> are the data that can be exchanged along connections.

There are different kinds of types:

- a behaviour type<sup>6</sup>, the type of all behaviours.
- connection types<sup>7</sup> are the types that can be ascribed to connections,
- value types are the types that can be ascribed to values,

<sup>2</sup> It corresponds to a base  $\pi$ -calculus, without connection re-configuration, defined as the basic formal system to be extended for defining other  $\pi$ -calculi.

<sup>3</sup> Type *Any* is an infinite union type; values of this type consist of a value of any type together with a representation of that type.

<sup>4</sup> The base layer of  $\pi$ -ADL extended with base types and the tuple type constructor corresponds to a polyadic typed value-passing  $\pi$ -calculus.

<sup>5</sup> Connections and behaviours are included among the values.

<sup>6</sup> The type of behaviours is *Behaviour*.

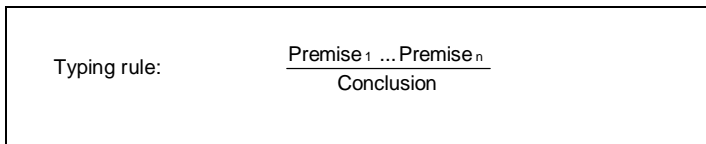
<sup>7</sup> Connection types are also value types.

An assignment of a type  $T$  to a value  $v$  is declared by  $v : T$ . A type environment is a finite set of assignments of types to values, where the values in the assignments are all different. Let  $\Delta$  range over type environments.

Type judgments are of the form  $\Delta \vdash v : T$  where  $v$  may be a data value, a connection or a behaviour:

- a value type judgment  $\Delta \vdash v : T$  asserts that  $v$  has type  $T$  under type assumptions  $\Delta$ ,
- if  $v$  is a behaviour then  $T$  is *Behaviour* (i.e. the behaviour type),
- a behaviour type judgment  $\Delta \vdash b : Behaviour$  asserts that behaviour  $b$  respects the type assumptions in  $\Delta$ .

The  $\pi$ -ADL type system is formalised by means of typing rules. The terms that can be typed using these rules are well-typed terms. Typing rules use the structure of proof rules.



**Figure 3. Typing rule**

A typing rule (see Figure 3) can be read “if the statements in the premises listed above the line are established, then the conclusion below the line is derived”. Thereby:

- if  $Premise_1, \dots, Premise_n$  are well-typed then  $Conclusion$  is well-typed.

Rules with no premises are called axioms. Rules with one or more premises are called inference rules.

Given a type system:

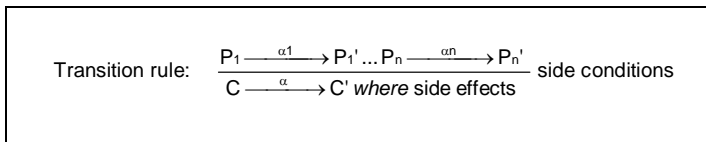
- a valid type judgment is one that can be proved from the axioms and inference rules of the type system,
- a value  $v$  is well typed in  $\Delta$  if there is  $T$  such that  $\Delta \vdash v : T$  is valid ( $v$  is well typed if it is well typed in  $\Delta$ , for some  $\Delta$ ).

Following this approach, the  $\pi$ -ADL type system is completely formalised (see [43] for details).

**Transition Rules**

The semantics of a language can be formalised by means of a transition system.

A transition system is defined by transition rules. Transition rules also use the structure of proof rules.



**Figure 4. Transition rule**

In a transition rule (see Figure 4), premises and conclusions are transition relations. Thereby, if the transition relations labelled by  $\alpha_1 \dots \alpha_n$  can fire, then the transition relation labelled by  $\alpha$  can fire,

i.e. if  $P_1$  can fire  $\alpha_1$  and become  $P_1'$  ... and  $P_n$  can fire  $\alpha_n$  and become  $P_n'$ , then  $C$  can fire  $\alpha$  and become  $C'$ . Side conditions and side effects can be expressed. Side conditions can be seen as pre-conditions on terms expressed in the premises and side effects as post-conditions on terms expressed in the conclusion.

This structural operational semantics represents behaviour (and thereby computation) of the  $\pi$ -ADL by means of a deductive system, expressed by the transition system. Therefore, the formal semantics of the  $\pi$ -ADL is defined by a transition system, with transition rules formalising the operational semantics of the constructs of the language, in line with its type system, formalised by the typing rules. Type soundness asserts that well-typed terms do not give rise to runtime errors under the transition system.

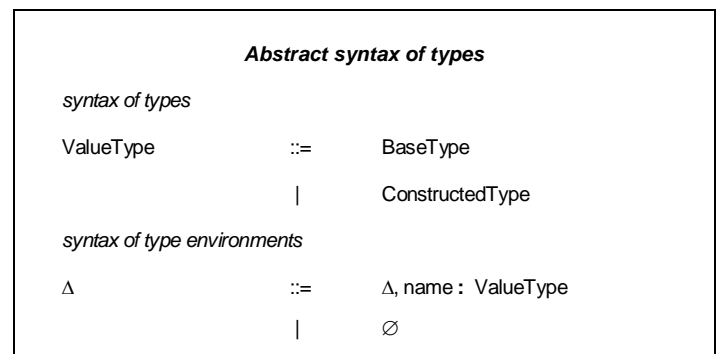
Following this approach, the  $\pi$ -ADL semantics is completely formalised (see [43] for details).

**3.3 Abstract syntax**

The basis for defining the semantics is the abstract syntax of  $\pi$ -ADL, which is defined using the following notation for the abstract production rules:

- keywords are written with bold;
- non-terminals are written without bold;
- a sequence of zero, one or more elements is written:  $Element_{min}, \dots, Element_{max}$ , where the value of  $min$  specifies the minimum number of elements ( $0$  specifies possibly no elements,  $1$  specifies at least one element) and the value of  $max$  specifies the maximum number of elements ( $Element_n$  specifies any number of elements);
- alternative choices are written separated by  $|$ .

The abstract syntax of types for expressing typed behaviours is defined as follows.



**Figure 5. Abstract syntax of value types**

As shown in Figure 5, all types are value types. Value types are base types or constructed types. Type environments are expressed through declarations. A value can be declared as being of a base type or a constructed type.

The abstract syntax of base types is defined in Figure 6.

<b>Abstract syntax of base types</b>	
<i>syntax of base types</i>	
BaseType ::=	<b>Any</b>
	<b>Natural</b>
	<b>Integer</b>
	<b>Real</b>
	<b>Boolean</b>
	<b>String</b>
	<b>Behaviour</b>

**Figure 6. Abstract syntax of base types**

The abstract syntax of constructed types is defined in Figure 7.

<b>Abstract syntax of constructed types</b>	
<i>syntax of constructed types</i>	
ConstructedType ::=	<b>tuple</b> [ ValueType <sub>1</sub> , ..., ValueType <sub>n</sub> ]
	<b>view</b> [ label <sub>1</sub> : ValueType <sub>1</sub> , ..., label <sub>n</sub> : ValueType <sub>n</sub> ]
	<b>union</b> [ ValueType <sub>1</sub> , ..., ValueType <sub>n</sub> ]
	<b>quote</b> [ name ]
	<b>variant</b> [ label <sub>1</sub> : ValueType <sub>1</sub> , ..., label <sub>n</sub> : ValueType <sub>n</sub> ]
	<b>location</b> [ ValueType ]
	<b>sequence</b> [ ValueType ]
	<b>set</b> [ ValueType ]
	<b>bag</b> [ ValueType ]
	<b>inout</b> [ ValueType ]   <b>in</b> [ ValueType ]   <b>out</b> [ ValueType ]
	ValueType <sub>0</sub> , ..., ValueType <sub>n</sub> → <b>Behaviour</b>

**Figure 7. Abstract syntax of constructed types**

The abstract syntax for expressing typed behaviours is defined in Figure 8. Behaviours are expressed using the following  $\pi$ -ADL constructs:

- type,
- value,
- prefix,
- condition,
- choice,
- composition,
- decomposition,
- replication,
- inaction,
- application.

<b>Abstract syntax of behaviours and values</b>	
<i>syntax of behaviours</i>	
behaviour ::=	type . behaviour
	value . behaviour
	prefix . behaviour
	<b>if</b> boolean <b>then</b> { behaviour <sub>1</sub> }
	<b>else</b> { behaviour <sub>2</sub> }
	<b>choose</b> { behaviour <sub>0</sub> ... <b>or</b> behaviour <sub>n</sub> }
	<b>compose</b> { behaviour <sub>0</sub> ... <b>and</b> behaviour <sub>n</sub> }
	<b>decompose</b> behaviour
	<b>replicate</b> behaviour
	<b>done</b> -- inaction
	abstraction ( expression <sub>0</sub> ..., expression <sub>n</sub> ) -- application
prefix ::=	<b>via</b> connectionValue <b>send</b> value
	<b>via</b> connectionValue <b>receive</b> variable : ValueType
	<b>unobservable</b>
	<b>if</b> boolean <b>do</b> prefix
<i>syntax of values</i>	
value ::=	baseValue
	constructedValue
constructedValue ::=	...
	connectionValue
	behaviourValue
	abstractionValue

**Figure 8. Abstract syntax of typed behaviours and values**

### Type

**type** *name is ValueType . behaviour* introduces an alias to type *ValueType* in the scope of a behaviour *behaviour*.

### Value

Declaration **value** *name is value . behaviour* expresses the restriction of the scope of a name *name*, bound to value *value*, to the scope of a behaviour *behaviour*.

### Prefix

Behaviours enact by performing actions. The capabilities for action are expressed via the prefixes. A prefix can be an output or input prefix, a silent prefix or a match prefix.

An output prefix **via** *connection* **send** *value* expresses the capability to send a value *value* via the connection *connection*. An input

prefix **via connection receive value** expresses the capability to receive a value *value* via the connection *connection*. The silent prefix **unobservable** expresses the capability to enact an action invisibly, i.e. internally, silently. The match prefix **if boolean do prefix** expresses the capability of enacting a prefix *prefix* if the condition *boolean* is true, or do nothing otherwise.

Prefix **prefix . behaviour** expresses the capability of a behaviour to enact a prefix *prefix* and then become *behaviour*. Thereby, prefix expresses the capability of sending and receiving values via connections, enacting unobservable actions, or enacting conditional actions. After enacting *prefix*, the behaviour will proceed as *behaviour*.

### Condition

Condition **if boolean then { behaviour<sub>1</sub> } else { behaviour<sub>2</sub> }** expresses the capability of a behaviour to proceed as *behaviour<sub>1</sub>* if the *boolean* is true, otherwise as *behaviour<sub>2</sub>*. When one of the capabilities is exercised, the other is no longer available.

### Choice

Choice **choose { behaviour<sub>0</sub> ... or behaviour<sub>n</sub> }** expresses the capability of a behaviour to choose either the capability of *behaviour<sub>0</sub>* or the capability of *behaviour<sub>1</sub>* ... or the capability of *behaviour<sub>n</sub>*. When one of the capabilities is exercised, the others are no longer available. Thereby, the choice will proceed either as *behaviour<sub>0</sub>*' or as *behaviour<sub>1</sub>*' ... or as *behaviour<sub>n</sub>*', i.e. as a behaviour *behaviour<sub>i</sub>*' after exercising the capability of *behaviour<sub>i</sub>*.

### Composition

Composition **compose { behaviour<sub>0</sub> ... and behaviour<sub>n</sub> }** expresses the capability of a behaviour to parallel compose the capabilities of *behaviour<sub>0</sub>* ... and *behaviour<sub>n</sub>*. *behaviour<sub>0</sub>* ... and *behaviour<sub>n</sub>* can proceed independently and can interact via attached connections. Thereby, the composition will proceed by exercising the capabilities of independent actions in *behaviour<sub>0</sub>*' ... and *behaviour<sub>n</sub>*', or jointly exercising a capability of interaction, i.e. when via an attached connection one behaviour sends a value and the other receives the value yielding an unobservable communication action.

### Decomposition

Decomposition **decompose behaviour** expresses the capability of a behaviour to be decomposed into its constituent behaviours *behaviour<sub>1</sub>* ... and *behaviour<sub>n</sub>* with their unbound connections.

A behaviour is decomposed in its constituent behaviours once it reaches its reduction limit<sup>8</sup>. Behaviours, and in particular composite behaviours, can be decomposed and recomposed. The decomposition of a composite behaviour yields the set of constituent behaviours as they were before composition, i.e. behaviours with their unbound connections. These behaviours can be composed again in the same or another composition.

### Replication

Replication **replicate behaviour** expresses the capability of a be-

haviour to replicate itself infinitely. It can be thought of as an infinite composition defined as **compose { behaviour and replicate behaviour }**.

### Inaction

Inaction **done** is a behaviour that can do nothing. Inaction is not a basic construct: it can be defined as an empty choice.

### Application

In order to apply abstractions, application<sup>9</sup> is used. An application has the form **abstraction ( value<sub>0</sub>, ..., value<sub>n</sub> )**. Applications instantiate abstractions yielding behaviours.

Declaring types and values supports the expression of architectural abstractions. An architectural abstraction can be an architecture, a component or a connector. Interfaces of architectural abstractions are expressed by ports in terms of incoming, outgoing or incoming-and-outgoing connections.

The abstract syntax for expressing architectural abstractions is defined in Figure 9.

<b>Abstract syntax of architectural abstractions</b>	
<i>syntax of architectural abstractions</i>	
architecturalAbstraction ::=	archtype name <b>is</b> abstraction
archtype ::=	<b>architecture</b>   <b>component</b>   <b>connector</b>
abstraction ::=	<b>abstraction</b> ( name <sub>0</sub> : ValueType <sub>0</sub> , ..., name <sub>n</sub> : ValueType <sub>n</sub> ) { type <sub>0</sub> . . . . type <sub>n</sub> . value <sub>0</sub> . . . . value <sub>n</sub> . port <sub>0</sub> . . . . port <sub>n</sub> . <b>behaviour is</b> { behaviour } <b>} assuming</b> { property }
type ::=	<b>type</b> name <b>is</b> ValueType
value ::=	<b>value</b> name <b>is</b> expression
port ::=	<b>port</b> name <b>is</b> restriction { connection <sub>0</sub> . . . . connection <sub>n</sub> <b>} assuming</b> { <b>protocol is</b> property }
restriction ::=	<b>free</b>   <b>restricted</b>
connection ::=	<b>connection</b> name <b>is</b> mode ( ValueType )
mode ::=	<b>in</b>   <b>out</b>   <b>inout</b>

**Figure 9. Abstract syntax of architectural abstractions**

The  $\pi$ -ADL abstract syntax is completely defined in [43]. Its concrete textual syntax (with a tutorial) is presented in [41].

<sup>8</sup> Behaviours must be in normal form with respect to  $\beta$ -reduction of values in order to be decomposed.

<sup>9</sup> The term application is used instead of pseudo-application.

4. Case Studies

We present hereafter three case studies to illustrate how  $\pi$ -ADL can be used to formally describe static, dynamic, and mobile software architectures. In the first case study, we describe a client-server architecture of a data acquisition system. The architecture is static: it does not evolve during the execution of the system. In the second case study, we describe a pipe-filter architecture for a software that computes prime numbers according to the primes sieve of Eratosthenes. The architecture is dynamic: it can evolve during the execution of the system. In the third case study, we describe a client-server architecture with mobile agents. The architecture is mobile: agent components can move from clients to server and back during the execution of the system.

4.1 Describing a Static Architecture

To start, let us describe a client-server architecture of a data acquisition system [45]. The architecture is static: it does not evolve during the execution of the system.

First we will present a black-box description of the architecture focusing on interface (i.e. ports and their connections) of components and connectors. Then we will add the internal behaviour of components and connectors in a white-box description. Finally the encompassing structure (i.e. binding among components and connectors using connection unifications) is described.

The data acquisition system receives pairs of key and data: a key and a data value to be stored under this key. New data values for the same key overwrite old values. Concurrently, it answers requests for the data of a certain key by sending the data value stored under this key.

Figure 10 depicts the data acquisition system as a whole. It is represented as a component, named *DasDef*, having ports<sup>10</sup> *update* and *request*. These ports represent the interaction of the *DasDef* system with its environment.

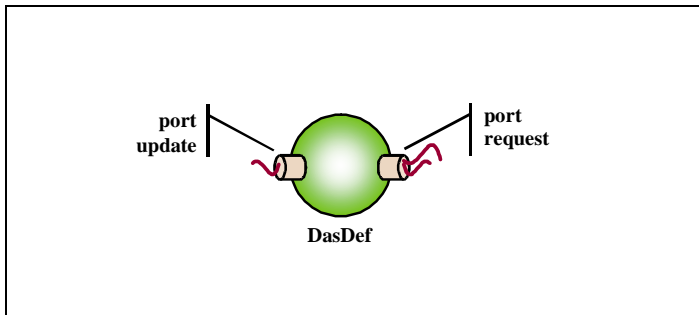


Figure 10. The system as a whole

Using  $\pi$ -ADL, the *DasDef* system abstraction can be formally described as shown in Figure 11.

```

architecture DasDef is abstraction() {
  type Key is Any. type Data is Any. type Entry is tuple[Key, Data].
  port update is { connection in is in(Entry) }.
  port request is { connection key is in(Key).
                  connection data is out(Data) } assuming {
    protocol is { ( via key receive any. true*. via data send any ) * }
  }.
  ...
}
    
```

Figure 11. Ports of the Architecture

In this interface description of the data acquisition system, type *Key* is the set of all possible key values and type *Data* is the set of all possible data values. *Entry* is the tuple type *tuple[Key, Data]*, i.e. the set of all possible entries associating key and data values. Two ports are declared: *update* that comprises the connection *in* for receiving entries and *request* that comprises the connections *key* and *data* for answering requests. The protocol enforced by this port is that requests for the data of a certain key, which are received via the connection *key*, are answered by sending (after processing) a data value via the connection *data*. For each key received there must be a data sent before accepting the next key.

The data acquisition system is composed of a sensor and a data manager. The sensor acts as a client of the data manager that acts as a server managing the sensor's acquired data. The raw data received from the environment first undergoes some processing in the sensor and is then forwarded to the remote data manager that stores the data. Figure 12 outlines the abstract architecture of the system in terms of its components and connectors.

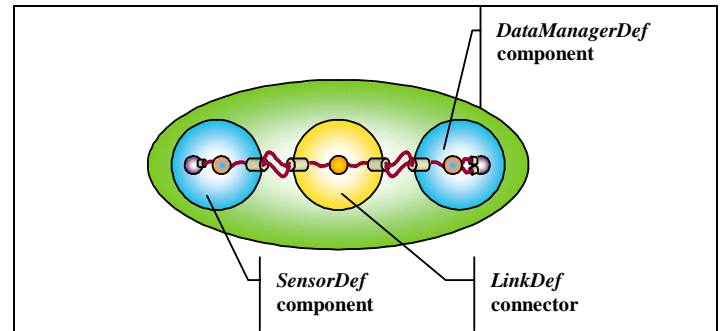


Figure 12: Outline of the architecture

The architecture consists of a sensor component *SensorDef*, a data manager component *DataManagerDef*, and a connector *LinkDef* to connect them together. These components and connector can be formally described in  $\pi$ -ADL as follows.

<sup>10</sup> By syntactic convention, ports that are not explicitly declared as restricted are external free ports.

```

component SensorDef is abstraction() {
  type Key is Any. type Data is Any. type Entry is tuple[Key, Data].
  port incoming is { connection in is in(Entry) }.
  port outgoing is { connection toLink is out(Entry) }.
  ...
} assuming {
  protocol is { ( via incoming::in receive any. true*.
                  via outgoing::toLink send any )* }
}

```

Figure 13. Ports of the component *Sensor*

In component *SensorDef* (see Figure 13), two ports are declared: *incoming* that comprises the connection *in* for receiving entries and *outgoing* that comprises the connection *toLink* for forwarding these entries. The protocol enforced by the two ports is that a value received via the connection *in* is (after processing) forward by sending some (possibly processed) value via the connection *toLink*. For each entry received there must be a value sent before accepting the next entry.

```

component DataManagerDef is abstraction() {
  type Key is Any. type Data is Any. type Entry is tuple[Key, Data].
  port select is { connection key is in(Key).
                  connection data is out(Data)
  } assuming {
    protocol is { ( via key receive any. true*. via data send any )* }
  }.
  port incoming is { connection fromLink is in(Entry) }.
  ...
}

```

Figure 14. Ports of the component *Data Manager*

In component *DataManagerDef* (see Figure 14), two ports are declared: *select* that comprises the connection *key* for receiving key values and the connection *data* for sending the data value stored under this key, and *incoming* for receiving entries.

```

connector LinkDef is abstraction() {
  type Key is Any. type Data is Any. type Entry is tuple[Key, Data].
  port incoming is { connection toLink is in(Entry) }.
  port outgoing is { connection fromLink is out(Entry) }.
  ...
} assuming {
  protocol is { ( via incoming::toLink receive entry : Entry.
                  via outgoing::fromLink send entry )* }
}

```

Figure 15. Ports of the connector *Link*

In connector *LinkDef* (see Figure 15), two ports are declared: *incoming* that comprises the connection *toLink* for receiving entries and *outgoing* that comprises the connection *fromLink* for forwarding these entries. The protocol enforced by the two ports is that entries received via the connection *toLink* are immediately forward by sending it via the connection *fromLink*.

This black-box description of the *DasDef* system architecture can be further detailed to achieve a white-box description of the architecture that encompasses interface, behavioural and then structural aspects.

The behaviour of the components *SensorDef* and *DataManagerDef* and the connector *LinkDef* can be formally described in  $\pi$ -ADL as follows.

```

component SensorDef is abstraction() {
  ...
  behaviour is {
    process is function(d: Data) : Data { unobservable }.
    via incoming::in receive entry : Entry.
    project entry as key, data.
    via outgoing::toLink send tuple(key, process(data)).
    behaviour()
  }
} assuming { ... }

```

Figure 16. Behavior of the component *Sensor*

In component *SensorDef* (see Figure 16), a function *process* : *Data*  $\rightarrow$  *Data* handles the processing for a single datum in the sensor. The behaviour specifies that once an *entry* is received via connection *in*, it is sent containing the same key and a processed data via connection *toLink*. The behaviour is recursively defined. Once an entry is handled, it continues with the same (recursive) behaviour for the next one.

```

component DataManagerDef is abstraction() {
  ...
  database is location(Set(Entry)).
  behaviour is {
    choose { via incoming::fromLink receive entry : Entry.
            database := database' including(entry).
            behaviour()
          or
            via select::key receive queryKey : Key.
            via select::data send (database' selecting(e |
                                project e as key, data. key=queryKey)).
            behaviour() }
  }
}

```

Figure 17. Behaviour of the component *Data Manager*

In component *DataManagerDef* (see Figure 17), the database is modelled as a location that stores a set of tuples of *Entry* type. Given a key, a data value can be retrieved. If there is not yet an item stored under the given key, then the database returns a null value. The behaviour specifies a choice: updating the database or handling requests. When an *entry* is received via connection *fromLink*, it is included in the database. When a *queryKey* is received via connection *key*, its corresponding data (retrieved from the database) is sent via connection *data*. The behaviour is recursively defined. Once an entry or a request is handled, it continues with the same (recursive) behaviour for the next entry or request.

```

connector LinkDef is abstraction() {
  ...
  behaviour is {
    via incoming::toLink receive entry : Entry.
    via outgoing::fromLink send entry.
    behaviour()
  }
  } assuming { ... }
}

```

Figure 18. Behavior of the connector *Link*

In connector *LinkDef* (see Figure 18), the behaviour specifies that entries received via the connection *toLink* are immediately forward by sending it via the connection *fromLink*. The behaviour is recursively defined. Once an entry is handled, it continues with the same (recursive) behaviour for the next entry.

Using the components *SensorDef* and *DataManagerDef* and the connector *LinkDef*, the abstract architecture *DasDef* can be composed in  $\pi$ -ADL as shown in Figure 19, thereby providing the structure of the architecture in terms of attached components and connector.

```

architecture DasDef is abstraction() {
  ...
  behaviour is compose { sensor      is SensorDef()
                        and link      is LinkDef()
                        and dataManager is DataManagerDef()
  }
  where {
    sensor::incoming relays update
    and sensor::outgoing unifies link::incoming
    and link::outgoing unifies dataManager::incoming
    and request relays dataManager::select
  }
}

```

Figure 19. Composition of the *Architecture*

In the architecture, the component instances *sensor* and *dataManager* are connected using the connector *link*. In order to actually

connect them, connections must be unified<sup>11</sup>. Connection *toLink* of port *outgoing* of component *sensor* is unified with connection *toLink* of port *incoming* of connector *link*. Connection *fromLink* of port *outgoing* of connector *link* is unified with connection *fromLink* of port *incoming* of component *dataManager*.

Besides connecting component instances together, the architecture must express the binding between external ports and ports of components. This binding is expressed by connection relay. Connection *in* of external port *update* is relayed to connection *in* of port *incoming* of component *sensor*. Connection *key* of external port *request* is relayed to connection *key* of port *select* of component *dataManager*. Connection *data* of port *select* of component *dataManager* is relayed to connection *data* of external port *request*.

#### 4.2 Describing a Dynamic Architecture

Let us now describe a pipe-filter architecture for a software that computes prime numbers. The description expresses a dynamic architecture according to the primes sieve of Eratosthenes. It is composed of a generator service used to output the numbers 2 to *n*, and a pipeline of filters where each filter crosses out multiples of a prime number.

The component *GeneratorService* (see Figure 20) is used to generate a sequence of natural numbers from *x* to *n*, followed by an *eos* (end-of-stream).

```

component GeneratorService is abstraction(x, n, eos : Natural) {
  port is { connection outGenerator is out(Natural) }.
  behaviour is {
    if (x =< n)
      then { via outGenerator send x.
             behaviour(x+1, n, eos) }
    else { via outGenerator send eos.
           done }
  }
}

```

Figure 20. Component *GeneratorService*

The connector *PipeService* (see Figure 21) buffers numbers between filters in the pipeline.

<sup>11</sup> If connections have the same names in different ports, identifying ports is enough to express unifications (if connection names are different, then they must be explicitly unified).

```

connector pipeService is abstraction (eos : Natural) {
  port is { connection inPipe is in(Natural).
            connection outPipe is out(Natural) }.
  behaviour is {
    via inPipe receive x : Natural.
    via outPipe send x.
    if x <> eos then behaviour(eos).
    else done
  }
}

```

**Figure 21. Connector *PipeService***

The component *FilterService* (see Figure 23) records the first value it gets and subsequently filters out multiples of the value from the numbers it receives and sends non-filtered numbers to the next filter in the architecture. If the next filter does not exist<sup>12</sup> it is first created, composed with a pipe, then the value sent.

```

component FilterService is abstraction(eos : Natural) {
  port is { connection inFilter is in(Natural).
            connection outFilter is out(Natural) }.
  behaviour is {
    crossOutMultiple is abstraction(p, eos : Natural) {
      ...
    }.
    prime is location(Natural).
    via inFilter receive p : Natural.
    prime := p.
    crossOutMultiple(p, eos)
  }
}

```

**Figure 22. Component *FilterService***

In order to filter out multiples of a prime number, the local abstraction *crossOutMultiple* is used (see Figure 23).

```

component FilterService is abstraction(eos : Natural) {
  ...
  crossOutMultiple is abstraction(p, eos : Natural) {
    via inFilter receive x : Natural.
    if (p <> eos)
    then if (x mod p) <> 0
      then if isConnectionUnified(outFilter)
        then { via outFilter send x.
              crossOutMultiple(p, eos) }
        else compose{ ps is PipeService(eos)
                    and fs is FilterService(eos)
                    and cm is { via outFilter send x.
                              crossOutMultiple(p, eos) }
              } where { outFilter unifies inPipe
                        and outPipe unifies inFilter }
      else crossOutMultiple(p, eos)
    else if isConnectionUnified(outFilter)
      then { via outFilter send eos. done }
      else done
  }.
  ...
}

```

**Figure 23. Local abstraction *crossOutMultiple* in component *FilterService***

The architecture composed of the component *GeneratorService* and the pipeline of filters *FilterService* and pipes *PipeService* is described in Figure 24.

```

architecture ComputingPrimes is abstraction (n, eos : Natural) {
  behaviour is compose {
    gs is GeneratorService(2, n, eos)
    and ps is PipeService(eos)
    and fs is FilterService(eos)
  } where { gs::outGenerator unifies ps::inPipe
            and ps::outPipe unifies fs::inFilter
          }
} assuming { parameter is { eos < 2 xor eos > n } }

```

**Figure 24. Architecture *ComputingPrimes***

*ComputingPrimes* is a dynamic architecture. Its number of components and connectors depends on parameter  $n$ , but this function is not known at design time, it is only discovered at runtime.

#### 4.3 Describing a Mobile Architecture

To finish, now let us describe a client-server architecture with mobile agents: a client creates and sends an agent performing some actions to a server site. The server accepts the agent sent

<sup>12</sup> An introspection feature (boolean function *isConnectionUnified*) is used to test if connection *outFilter* is attached, and thereby communication is eventually possible.

from the client. The agent, that initially is located in the client, is moved from the client to the server. It comes back to the client after finishing its work at the server. The architecture is mobile: agent components move from clients to server and back during the execution of the system.

The component *Client* can be specified as shown in Figure 25. This component encompasses the agent component *MobileAgent*.

```

component Client is abstraction(
  sendAgent : out[abstraction[]],
  waitAgent : in[Behaviour],
  agentConnection : out[inout[], inout[]]
){
  component MobileAgent is abstraction(
    agentConnection : out[inout[], inout[]]
  ){
    ...
  }.
  behaviour is {
    unobservable.
    via sendAgent send MobileAgent.
    via waitAgent receive agentBack : Behaviour.
    unobservable.
    done
  }
}

```

Figure 25. Component *Client*

```

component MobileAgent is abstraction(
  agentConnection : out[inout[],inout[]]
){
  port is restricted { connection startAgent is inout()
                    connection endAgent is inout() }.
  behaviour is {
    via agentConnection send (startAgent, endAgent).
    via startAgent receive.
    unobservable.
    via endAgent send.
    done
  }
}

```

Figure 26. Mobile component *MobileAgent*

The component *Server* can be specified as shown in Figure 27.

```

component Server is abstraction(
  inGateForAgent : in[abstraction[]],
  outGateForAgent : out[Behaviour],
  agentConnection : in[inout[], inout[]]
){
  behaviour is {
    replicate via inGateForAgent receive agent : abstraction[],
    compose {
      mobileAgent is agent().
    and hostingAndSendingBack is {
      via agentConnection receive (startAgent : out[],
                                endAgent : in[]).
      via startAgent send.
      via endAgent receive.
      unobservable.
      via outGateForAgent send mobileAgent.
      unobservable.
      done }
    }
  }
}

```

Figure 27. Component *Server*

Now the architecture *mobileAgentClientsToServer* can be composed as shown in Figure 28.

```

architecture MobileAgentClientsToServer is abstraction(
  client : abstraction[ out[abstraction[]],
                       in[Behaviour],
                       out[inout[], inout[]],
  server : abstraction[ in[abstraction[]],
                       out[Behaviour],
                       in[inout[], inout[]] ] {
  port is restricted {
    connection moveToServer is inout(abstraction[]).
    connection moveFromServer is inout(Behaviour).
    connection agentConnection is inout(inout[], inout[]].
  }.
  behaviour is {
    compose { c is replicate client(moveToServer,
                                    moveFromServer, agentConnection)
    and s is server(moveToServer,
                    moveFromServer, agentConnection) }
  }
}

```

Figure 28. Architecture *mobileAgentClientsToServer*

It is worth noting that *mobileAgentClientsToServer* is a mobile architecture (sub-components can move among components) but also a dynamic architecture (the number of clients is not known at design time, but only at runtime).

## 5. Related Work

Different ADLs have been proposed in the literature [35], including: ACME/Dynamic-ACME [22][23], AESOP [21], AML [54], ARMANI [37], CHAM-ADL [26][27], DARWIN [32], META-H [10], PADL [8][9], RAPIDE [46][31], SADL [38][39],  $\sigma\pi$ -SPACE [14][29], UNICON-2 [17], WRIGHT/Dynamic-WRIGHT [2][3], and ZETA [5].

Most of these ADLs assume that architectures are static. Some however supports the description of dynamic aspects of architectures. They are DARWIN, Dynamic-ACME, Dynamic-WRIGHT,  $\sigma\pi$ -SPACE, and RAPIDE. But this support is rather limited. For instance, to the best of our knowledge, none of these languages have enough expressive power to describe dynamic architectures where the changes in the architecture depends on its data input, like in the case study presented in section 4.2. Moreover, none of these ADLs have components as first-class citizens in order to cope with description of mobile architectures, like in the case study presented in section 4.3.

An ADL is defined by both a syntax and a formal, or semi-formal, semantics. Typically, ADLs embody a conceptual framework reflecting characteristics of the domain for which the ADL is intended.

The focus of  $\pi$ -ADL is on the formal modelling of dynamic and mobile architectures (evolvable at design and runtime [44]), and for supporting the computer-aided formal verification [4] and refinement [42] of these models.

Comparing ADLs objectively is a difficult task because their focuses are quite different. Most ADLs essentially provide constructs for a component-and-connector description including topological constraints from a structural viewpoint. The reason for this is probably that structure is certainly the most understandable and visible part of an architecture. But behavioural and qualities are not completely neglected. They are often taken into account (even if partially) in most ADLs. They are certainly an essential part of an architecture description.

$\pi$ -ADL introduces the notion of architectural abstractions, which can be architectures, components and connectors from a runtime perspective. All abstractions are first-class citizens. In  $\pi$ -ADL, architectures, components, and connectors are formally specified in terms of typed abstractions over behaviours.

Furthermore,  $\pi$ -ADL provides a notation to express assumptions as properties of architectures [4]. It combines predicate logic with temporal logic in order to allow the specification of both structural properties and behavioural properties.

Regarding structural properties, the property notation is based on predicate logic. Regarding behavioural properties, it is based on modal  $\mu$ -calculus [28]. The choice of modal  $\mu$ -calculus as the underlying formalism provides a significant expressive power [49]. Moreover, having a unified notation for expressing both structural

and behavioural properties facilitates the specification task of architects, by allowing a more natural and concise description.

The main limitation of most of the ADLs with regard to analysis support is that they address either structural or behavioural properties, but not both at the same time as in  $\pi$ -ADL.

No other ADL provides a comprehensive set of features for describing dynamic and mobile architectures as  $\pi$ -ADL.

$\pi$ -ADL provides a novel ADL that is general-purpose and Turing-complete. It can be seen as a second generation ADL: in first generation ADLs, languages focused mainly in structural aspects and were not complete; in second generation, languages must encompass both structure and behaviour specification and must be complete. A detailed positioning of  $\pi$ -ADL with respect to the state-of-art is given in [19][30].

## 6. Conclusion and Future Work

$\pi$ -ADL supports formal specification (and corresponding verification) of static, dynamic and mobile software architectures. This is a key factor in the architectural design phase. For that reason it differs from the other existing ADLs.

$\pi$ -ADL provides a graphical notation defined as a UML Profile in a model driven architecture framework [12]. This assists in the mapping between semi-formal architecture diagrams and formal descriptions, which can be analysed and refined, by providing the ability to elaborate visually the architecture specification.

A major impetus behind developing formal languages for architectural description is that their formality renders them suitable to be manipulated by software tools. The usefulness of an ADL is thereby directly related to the kinds of tools it provides to support architectural description, but also analysis, refinement, code generation, and evolution [44]. Indeed,  $\pi$ -ADL is supported by a comprehensive toolset for supporting architecture-centric formal development. It is composed of:

- a  $\pi$ -ADL visual modelling tool [7], implemented as an extension of the Objecteering UML Modeller,
- a  $\pi$ -ADL callable compiler and a persistent virtual machine,
- a  $\pi$ -ADL verification tool based on CADP [20][34] and XSB Prolog,
- a  $\pi$ -ADL refinement tool providing a process-centred refinement framework based on the Maude rewriting logic system [11][33],
- a  $\pi$ -ADL-to-Code synthesizer, from which a  $\pi$ -ADL-to-Java code generation tool has been synthesized.

Ongoing work is mainly related to completing the development of the  $\pi$ -ADL toolset in the ArchWare European Project [44].

$\pi$ -ADL has been applied in several realistic case studies and industrial business cases at Thésame (France) and Engineering Ingegneria Informatica (Italy). The pilot project at Thésame aims to architecting agile integrated industrial process systems. The pilot project at Engineering Ingegneria Informatica aims to architecting federated knowledge management systems.  $\pi$ -ADL has also been used by the CERN (Switzerland) for architecting human computer interfaces for monitoring particle accelerator restart.

Future work is mainly related with the formal development of an architecture-centric formal method. This formal method, called the  $\pi$ -Method, like formal methods such as B [1], FOCUS [50], VDM [18], and Z [16], aims to provide full support for formal description and development. Unlike these formal methods that do not provide any architectural support, the  $\pi$ -Method has been built from scratch to support architecture-centric formal software engineering.

## References

- [1] Abrial J.-R.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] Allen R.: *A Formal Approach to Software Architectures*. PhD Thesis, Carnegie Mellon University, 1997.
- [3] Allen R., Douence R., Garland D.: *Specifying and Analyzing Dynamic Software Architectures*. In *Fundamental Approaches to Software Engineering*, LNCS 1382, Springer Verlag, 1998.
- [4] Alloui I., Garavel H., Mateescu R., Oquendo F.: *The ArchWare Architecture Analysis Language: Syntax and Semantics*. Deliverable D3.1b, ArchWare European RTD Project, IST-2001-32360, January 2003.
- [5] Alloui I., Oquendo F.: *Supporting Decentralised Software-intensive Processes using ZETA Component-based Architecture Description Language*. *Enterprise Information Systems*, Joaquim Filipe (Ed.), Kluwer Academic Publishers, 2002.
- [6] Alloui I., Oquendo F.: *The ArchWare Architecture Description Language: UML Profile for Architecting with ArchWare ADL*. Deliverable D1.4b, ArchWare European RTD Project, IST-2001-32360, June 2003.
- [7] Alloui I., Oquendo F.: *Describing Software-intensive Process Architectures using a UML-based ADL*, *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS'04)*, Porto, Portugal, April 2004.
- [8] Bernardo M., Ciancarini P., Donatiello L.: *Architecting Systems with Process Algebras*. Technical Report UBLCS-2001-7, July 2001.
- [9] Bernardo M., Ciancarini P., Donatiello L.: *Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems*, *Proceedings of the 2nd Working IEEE/IFIP Conference on Software Architecture*, Amsterdam, IEEE-CS Press, August 2001.
- [10] Binns P., Engelhart M., Jackson M., Vestal S.: *Domain-Specific Software Architectures for Guidance, Navigation, and Control*. *International Journal of Software Engineering and Knowledge Engineering*. 1996.
- [11] Bolusset T., Oquendo F.: *Formal Refinement of Software Architectures Based on Rewriting Logic*, *ZB2002 International Workshop on Refinement of Critical Systems: Methods, Tools and Experience*, Grenoble, Janvier 2002.
- [12] Brown A.W.: *An Introduction to Model Driven Architecture – Part I: MDA and Today's Systems*. The Rational Edge, February 2004.
- [13] Chaudet C., Greenwood M., Oquendo F., Warboys B.: *Architecture-Driven Software Engineering: Specifying, Generating, and Evolving Component-Based Software Systems*. *IEE Journal: Software Engineering*, Vol. 147, No. 6, UK, December 2000.
- [14] Chaudet C., Oquendo F.: *A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems*. *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*. IEEE Computer Society, Grenoble, September 2000.
- [15] Cimpan S., Leymonerie F., Oquendo F.: *The ArchWare Foundation Styles Library*. Report R1.3-1, ArchWare European RTD Project, IST-2001-32360, June 2003.
- [16] Davies J., Woodcock J.: *Using Z: Specification, Refinement and Proof*. Prentice Hall International Series in Computer Science, 1996.
- [17] DeLine R.: *Toward User-Defined Element Types and Architectural Styles*. *Proceedings of the 2nd International Software Architecture Workshop*, San Francisco, 1996.
- [18] Fitzgerald J., Larsen P.: *Modelling Systems: Practical Tools and Techniques for Software Development*, Cambridge University Press, 1998.
- [19] Gallo F. (Ed.): *Annual Report: Project Achievements in 2002*. Appendix B: *Survey of State-of-the-Art and Typical Usage Scenario for ArchWare ADL and AAL*. Deliverable D0.4.1, ArchWare European RTD Project, IST-2001-32360, February 2003.
- [20] Garavel H., Lang F., Mateescu R.: *An Overview of CADP 2001*. *European Association for Software Science and Technology (EASST) Newsletter*, Vol. 4, August 2002.
- [21] Garland D., Allen R., Ockerbloom J.: *Exploiting Style in Architectural Design Environments*. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, 1994.
- [22] Garland D., Monroe R., Wile D.: *ACME: An Architecture Description Interchange Language*. *Proceedings of CASCON'97*, Toronto, November 1997.
- [23] Garland D., Monroe, R., Wile D.: *ACME: Architectural Description of Component-Based Systems*. *Foundations of Component-Based Systems*, Leavens G.T., and Sitaraman M. (Eds.), Cambridge University Press, 2000.
- [24] Greenwood M., Balasubramaniam D., Cimpan S., Kirby N.C., Mickan K., Morrison R., Oquendo F., Robertson I., Seet W., Snowdon R., Warboys B., Zirintsis E.: *Process Support for Evolving Active Architectures*, *Proceedings of the 9th European Workshop on Software Process Technology*, LNCS 2786, Springer Verlag, Helsinki, September 2003.
- [25] IEEE Std 1471-2000: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, October 2000.
- [26] Inverardi P., Wolf A.: *Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model*. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.
- [27] Inverardi P., Wolf A., Yankelevich D.: *Static Checking of System Behaviors using Derived Component Assumptions*. *ACM Transactions on Software Engineering and Methodology*, Vol. 9, No. 3, July 2000.
- [28] Kozen D.: *Results on the Propositional  $\mu$ -Calculus*. *Theoretical Computer Science* 27:333-354, 1983.
- [29] Leymonerie F., Cimpan S., Oquendo F.: *Extension d'un langage de description architecturale pour la prise en compte des styles architecturaux : application à J2EE*. *Proceedings of the 14th International Conference on Software and Systems Engineering and their Applications*. Paris, December 2001 (In French).
- [30] Leymonerie F., Cimpan S., Oquendo F.: *"État de l'art sur les styles architecturaux : classification et comparaison des langages de description d'architectures logicielles"*, *Revue Génie Logiciel*, No. 62, September 2002 (In French).
- [31] Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D., Mann W.: *Specification and Analysis of System Architecture Using RAPIDE*. *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, April 1995.
- [32] Magee J., Dulay N., Eisenbach S., Kramer J.: *Specifying Distributed Software Architectures*. *Proceedings of the 5th European Software Engineering Conference*, Sitges, Spain, September 1995.
- [33] Martí-Oliet N., Meseguer J.: *Rewriting Logic: Roadmap and Bibliography*. *Theoretical Computer Science*, 2001.
- [34] Mateescu R., Garavel H.: *XTL: A Meta-Language and Tool for Temporal Logic Model-Checking*. *Proceedings of the 1st International Workshop on Software Tools for Technology Transfer*, Aalborg, Denmark, July 1998.

- [35] Medvidovic N., Taylor R.: A Classification and Comparison Framework for Architecture Description Languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California. Irvine, February 1997.
- [36] Milner R.: Communicating and Mobile Systems: The Pi-Calculus. Cambridge University Press, 1999.
- [37] Monroe R.: Capturing Software Architecture Design Expertise with ARMANI. Technical Report CMU-CS-98-163, Carnegie Mellon University, January 2001.
- [38] Moriconi M., Qian X., Riemenschneider R.A.: Correct Architecture Refinement. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995.
- [39] Moriconi M., Riemenschneider R.A.: Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Computer Science Laboratory, SRI International, Technical Report SRI-CSL-97-01, March 1997.
- [40] Morrison R.: On the Development of S-algol. PhD Thesis, University of St Andrews, 1979.
- [41] Oquendo F.: The ArchWare Architecture Description Language: Tutorial. Report R1.1-1, ArchWare European RTD Project, IST-2001-32360, March 2003.
- [42] Oquendo F.: The ArchWare Architecture Refinement Language. Deliverable D6.1b, ArchWare European RTD Project, IST-2001-32360, December 2003.
- [43] Oquendo F., Alloui I., Cimpan S., Verjus H.: The ArchWare Architecture Description Language: Abstract Syntax and Formal Semantics. Deliverable D1.1b, ArchWare European RTD Project, IST-2001-32360, December 2002.
- [44] Oquendo F., Warboys B., Morrison R., Dindeleux R., Gallo F., Gavel H., Occhipinti C.: ArchWare: Architecting Evolvable Software. Proceedings of the 1st European Workshop on Software Architecture, LNCS 3047, Springer Verlag, St Andrews, UK, May 2004.
- [45] Philipps J., Rumpe B.: Refinement of Pipe and Filter Architectures. Proceedings of FM'99, LNCS 1708, 1999.
- [46] RAPIDE Design Team: Guide to the RAPIDE 1.0. Language Reference Manuals, Stanford University, July 1997.
- [47] Sangiorgi D.: Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD Thesis, University of Edinburgh, 1992.
- [48] Sangiorgi D., Walker D.: The Pi-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2001.
- [49] Stirling C.: Modal and Temporal Properties of Processes. Springer Verlag, 2001.
- [50] Stolen K., Broy M.: Specification and Development of Interactive Systems. Springer Verlag, 2001.
- [51] Strachey C.: Fundamental Concepts in Programming Languages. Oxford University Press, Oxford, 1967.
- [52] Tennent R.D.: Language Design Methods based on Semantic Principles. Acta Informatica 8, 1977.
- [53] Verjus H., Oquendo F.: The ArchWare Architecture Description Language: XML Concrete Syntax. Deliverable D1.3b, ArchWare European RTD Project, IST-2001-32360, June 2003.
- [54] Wile D.: AML: An Architecture Meta Language. Proceedings of the 14th International Conference on Automated Software Engineering, pp. 183-190. Cocoa Beach. October 1999.