ISSN 1744-1986

Technical Report N° 2005/07

Using PADL to specify AFrames

Jon G. Hall
Lucia Rapanotti

28th April 2005

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Using PADL to specify AFrames

Jon G. Hall Lucia Rapanotti
Computing Research Centre
The Open University

Abstract

In this short technical note we show how PADL – the process algebraic architectural description language of Bernardo, Ciancarini and Donatiello – can be used to specify AFrames. AFrames exist to structure the machine in a Problem Frames development, and are important in that framework as they allow architectural expertise to be captured and reused therein. Because of the close proximity of PADL to other Architectural Description Languages, we assert that this work opens the Problem Frames framework to standard architectural abstractions.

1 Introduction

Architectural description languages (ADLs, [1]) are used to describe software system architectures. ADLs consist of components and connectors, i.e., respectively, units of functionality and combinators thereof to build more complex functionality. ADLs will typically describe the topology of a software system, i.e., its structure, as well as its behaviour.

Many architectures have been described in ADLs. To open AFrames, and so Problem Frames, to the large repository of architectural knowledge contained in the ADL literature we show how AFrames relate to one such notation, PADL. PADL ([2], [3]) provides a formalisation of the notion of an architectural type using PA, a CCS-like process algebra ([10]), that, through a CCS-like notion of bisimulation equivalence, allows the compositional and efficient reasoning about its well-formedness. We choose PADL for its formal basis in process algebra, and its amenability to manipulation of its textual descriptions.

Architectural Frames (AFrames), introduced in [4], [11] and formalised in [5], allow the naming and characterisation of architectural styles so that they can

domain[1] of a problem diagram, a component of the Problem Frames approach [8]. An AFrame represents the fixed components of an architectural style together with those parts that are free to be designed.

2 AFrames

Architectural Frames (AFrames) were introduced in [11] [5] as a new element of the Problem Frames framework. The intention behind AFrames is to provide a practical tool for sub-problem decomposition and recomposition that allows the Problem Frames practitioner to separate and address, in a systematic fashion, the concerns arising from the intertwining of problems and solutions. The rationale behind AFrames is the recognition that solution structures can be usefully employed to inform problem analysis. AFrames where interpreted in [5] as rule application in the *software problem calculus* defined therein, and from whence the formalisation of AFrames stems.

AFrames capture the development expertise contained in architectural artefacts: it is a particular combination of architectural expansion, including choice of solution architecture, and progression, that has already proven useful for a particular class of problems. AFrames work with problem classes, characterised in [11], [5] by problem frames [8] and, following the multi-paradigm basis of that work, allow the description of domains (and, in particular, components) in any suitable language. An important restriction on the description language is that it must be able to distinguish control from observation of events therein, and such that every event has a unique controlling source [12]. This latter constraint we call *control correctness*, for ease of reference.

The following example of an AFrame is adapted from [5]: in that paper, a *ModeSwitch architecture* – one that decides which of two solution components, each satisfying a sub-requirement of the original requirement, should have priority given that the sub-requirements are in conflict. In essence, the architecture implements a simple switch that routes to the environment the output of the component appropriate to the mode selected by an environment input. To define such an architecture as an AFrame we write

$$ModeSwitchFrame[ModeSwitch^{output}_{mode,c1,c2}](C1^{c1}, C2^{c2})^{output}_{mode}$$

in which *ModeSwitchFrame* is the name of the AFrame, *ModeSwitch* is a (fixed) connector with (natural language) definition

1 The full form of AFrame allows the characterisation of other forms of architecture, such as that present in an organisation, a team or a training structure.

> On receiving the *mode1* command from the environment on channel *mode*, relay the output of *C1* to the *output* channel. On receiving the *mode2* command from the environment on channel *mode*, relay the output of *C2* to the *output* channel.

Input channels to the architecture/component are written superscripted – in this case there in a single input (*mode*) to the *ModeSwitch* component which is also the single input to the whole architecture; output channels are written subscripted; channels that appear as decoration on two domains are shared – both *c1* and *c2* are shared with the *ModeSwitch* component; input and output channels that decorate the whole expression are exposed to the environment. The *C1* and *C2* are 'to-be-found' components: the use of AFrames is always in the context of a problem, i.e., requirements in a real-world context, and AFrames expose the to-be-found components – in this case *C1* and *C2* – to the design process so that a system satisfying the requirements can be developed. The application of an AFrame leaves the software problem solver with more, hopefully simpler, software problems to solve: there will be one for each component to be found in the AFrame.

For this AFrame, the problem context is the following:

> The problem description must contain domains that provide the required mode information and accept the output from the architecture.

An AFrame is thus a development tool that can be used to capture a developer's experience of design; in the example of the *ModeSwitchFrame*, it might be assumed that the developer has had experience of developing such switches, and that s/he has engineered the *ModeSwitchFrame* to deal with a general case. The definition of an AFrame's components need not be provided formally; in the example above, natural language is used to capture the behaviour of the *ModeSwitch* components.

3 PADL

PADL, a process algebraic architectural description language inspired by ADL WRIGHT [1] and DARWIN [9]. It allows classes of architectures to be defined as types. Once defined, an architecture can be used to verify compatibility and conformity to ensure, respectively, that an architectural component is well-connected – meaning that it can communicate with its environment in an expected way – and that actual parameters to an architectural type are consistent with the formal parameters of that type – meaning that actual and formal parameters are bisimulation equivalent, with respect to the component and connector interactions.

An example of a PADL description is given in [3]. The example is of a two-place buffer (composed in the pipe and filter style) and whose type is defined as

archi_type *PipeFilter*

which names the architectural type (*PipeFilter*). Elementary component and connector types are defined as (PA) terms, with indications of input (observes) and output (controls) for channels:

archi_elem_types

elem_type *FilterType*

behavior *Filter* ≙ *accept_item.Filter'* +
fail.repair.Filter
Filter' ≙ *accept_item.Filter"* +
serve_item.Filter +
fail.repair.Filter
Filter" ≙ *serve_item.Filter'* +
fail.repair.Filter

interactions input *accept_item*
output *serve_item*

elem_type *PipeType*
behavior *Pipe* ≙ *accept_item,*
(forward_item1.Pipe +
forward_item2.Pipe)

interactions input *accept_item,*
output *forward_item1*,*forward_item2*

with the **interactions** indicating which channels the components and connectors expose to other elementary types.

There is one more component to a PADL definition of an architectural type, the topology, which determines the topology of the architecture through the **attachments** between its components, together with its **interactions** with the outside world. For the *PipeFilter* example, the topology is defined as:

archi_topology

archi_elem_instances *F0, F1, F2 : FilterType*
P : PipeType

archi_interactions input *F0.accept_item,*
output *F1.serve_item,*
F2.serve_item

archi_attachments
from *F0.serve_item* **to** *P.accept_item*
from *P.forward_item1* **to** *F1.accept_item*
from *P.forward_item2* **to** *F2.accept_item*

There is a graphical notation that is used to illustrate an architecture. It is

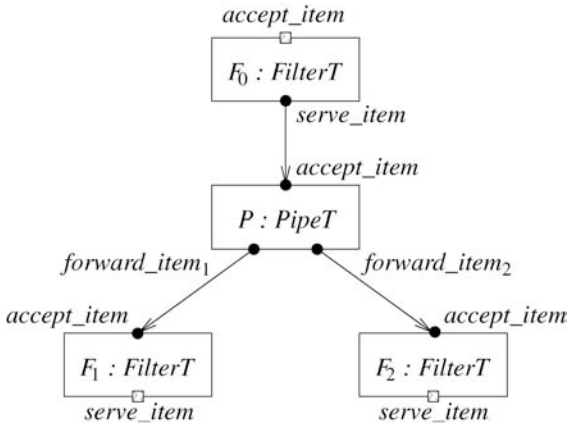

Fig. 1. Flow graph of the *PipeFilter* (from [3])

highly reminiscent of a problem diagram[2] in the problem frames framework – essentially, they both represent domains and their interconnections. For the problem above, the *flow graph* is shown in Figure 1. Considering the boxes as given domains, arcs (with arrows indicating flow) as the sharing of phenomena, we can, by blurring the eyes a little, imagine a collection of domains. Of course, blurring the eyes does not constitute a relationship, but it does inspire us to look more closely. There are differences:

- there is no machine domain in a flow graph as there must be in a context diagram;
- outputs, even on the same channel, can come from different components, as is the case with *serve_item* in the figure; in contrast, in a context diagram (as already mentioned) each event must be from a single domain;
- the links between components and connectors in a flow graph is through the association of one process algebraic event with another; in a context diagram events are shared between domains;
- there is no requirements ellipse.

The machine domain is the subject of design in a problem diagram. The lack of a machine, i.e., the object of the design process, occurs because a PADL-defined architecture describes a completed design for that architecture. We will see later how this restriction can be lifted.

The second difference is more material and will motivate the definition of a transformation of a PADL-defined architecture to allow it to be seen as an AFrame.

The third difference is the source of another, easier, transformation on event

[2] It would be more precise to say it is reminiscent of a context diagram. But a problem diagram is just a context diagram with a requirements ellipse.

labels.

The other thing missing from the flow graph is a requirements ellipse, this is related to the fact that a PADL-defined architecture is indicative. We can imaging that the PADL-defined architecture was produced so that certain requirements were discharged.

4 Relating PADL and AFrames

We consider as part of the Domain and Requirements Description Language (DRDL, [6]) for the problem diagrams, the process algebra PA defined as terms of the language

$$E ::= 0 \mid a.\, E \mid E/L \mid E[\varphi] \mid E + E \mid E \parallel_S E \mid A$$

defined in [3]. The behaviour of terms in this language will not concern us in this paper, but the reader can rest assured that, should they have a process algebraic background, they will recognise each of the operators behaviours as either that of CCS [10] or of CSP [7].

Including PA in DRDL for a problem diagram means that, in particular, domains can be described therein. Of course, we choose PA so that the transformation between an AFrame and its PADL equivalent does not include complex inter-notational transformations between elements of the DRDL. As an example of the use of PA to describe a domain, consider the description of the *ModeSwitch* domain which might be described as:

$$\begin{aligned}
ModeSwitch &\triangleq mode1.Mode1 + mode2.Mode2\\
Mode1 &\triangleq acceptC1Event.output.Mode1\\
&\quad + acceptC2output.Mode1\\
&\quad + mode1.Mode1 + mode2.Mode2\\
Mode2 &\triangleq acceptC2output.output.Mode2\\
&\quad + acceptC1output.Mode2\\
&\quad + mode1.Mode1 + mode2.Mode2
\end{aligned}$$

We first show how the AFrame

$$Name[KI_{oI}^{cI}, \ldots, \mathrm{Kn}_{on}^{cn}]()_o^c$$

over PA translates as a PADL architecture. This identifies a subset of PADL expressions that have an AFrame representation.

We assume that the AFrame has no to-be-found components, there is no place in PADL for to-be-found components. Moreover, we will assume that each *Ki* has PA description *PAKi* – that for the *ModeSwitch* component appears above – and that its name is *Name*. Form the PADL description:

archi_type *Name*

which names the architectural type (*PipeFilter*). Elementary component and connector types are defined as CCS expressions, with indications of input (observes) and output (controls) for channels:

archi_elem_types
elem_type *K1Type*
behavior *PAK1*
interactions input *o1*
output *c1*

...

elem_type *KnType*
behavior *PAKn*
interactions input *on,*
output *cn,*

and topology

archi_topology
archi_elem_instances *K1 : K1Type*
...
Kn : KnType
archi_interactions input *o*
output *c*
archi_attachments
from *Ki.a* **to** *Kj.a*

whenever *a* is an element of *ci* and *oj*.

Through this transformation, we see the close proximity of AFrames and PADL types. The translation is quite natural as AFrames include all of the information needed to construct a PADL type. The main difficulty is that AFrames have no contained idea of a type, and so we must construct the **elem_types** 'on the fly'.

4.1 The AFrame subset of PADL

The subset of PADL defined by AFrames includes architectural types equivalent to most PADL types, either directly – because they are members of the subset – or indirectly – because there is a member of the subset that can be reached through transformation of the original PADL term. In the latter case, the basis of this translation ensures that the PADL type is 'control correct': we give a PADL semantics preserving transformations within PADL that results in a PADL type equivalent – through the transformation in the previous subsection – to an AFrame.

Assume we have a PADL type introduced through the statement:

archi_type *PADLType*

with **archi_elem_instances** *K1*,...,*Kn*, with topology including

archi_interactions input *o*
output *c*

and attachments

archi_attachments
from *Ki.a* **to** *Kj.b*

each with type *KTypei* create an AFrame with signature

$$PADLType[K1^{c1}_{o1}, ..., Kn^{cn}_{on}]()^{c}_{o}$$

Let *KTypei* be the PA expression formed by replacing each action name *a* in *Ki* and each name *b* in *Kj* with the action name *ab* where **from** *Ki.a* **to** *Kj.b* appears as an **archi_attachment**; because of our assumption of a sufficiently large name space, such relabelling preserves behaviour up to bisimulation. Let the descriptions associated with *Ki* be *KTypei*, the *ci* and *oj* being the set of actions formed from all actions *ab* of the above type.

4.2 Control Correction

The restriction for single control can be removed by inserting a 'control correcting' component into a PADL type. The role of the component is to accepts all multiple outputs on the same channel as inputs and to route them through a to single output. Provided this component can be placed after any initial choices have been made, and provided that its internal behaviour is not visible, then the resulting PADL type will be bisimulation equivalent to the original. For the example *PipeFilter* above, we would introduce the 'control corrector' type elementary type *CCType* such that:

elem_type *CCType*
behavior $CCT \triangleq accept\_item1.serve\_item1.CCT +$
$+ accept\_item2.serve\_item.CCT$

interactions input *accept_item1, accept_item2*
output *serve_item*

introduce a single **archi_elem_instance** *C : CCType*, to join *F1* and *F2*'s outputs together. The topology must be changed to connect *C* and to allow it to be the conduit for the original outputs thus:

archi_attachments
…
from *F1.serve_item* **to** *C.accept_item*
from *F2.serve_item* **to** *C.accept_item*

c1 = *{serve_itemaccept_item}*
c2 = *{forward_item1accept_item, forward_item1accept_item}*
c3 = *{serve_itemaccept_item1}*
c4 = *{serve_itemaccept_item2}*
c = *{serve_item}*

o1 = *{accept_item}*
o2 = *{serve_itemaccept_item2}*
o3 = *{forward_item1accept_item}*
o4 = *{forward_item2accept_item}*
o = *{serve_itemaccept_item1, serve_itemaccept_item2}*

archi_interactions input …
output *C.serve_item*

4.3 Example

For the PADL type defined above, applying the transformations defined above, we have the following AFrame:

$$PipeFilter[F0^{c0}_{o0}, F1^{c1}_{o1}, F2^{c2}_{o2}, P^{c3}_{o3}, C^{c4}_{o4}]()^{c}_{o}$$

in which

$c1$={*serve_itemaccept_item*}
$c2$={*forward_item1accept_item, forward_item2accept_item*}
$c3$={*serve_itemaccept_item1*}
$c4$={*serve_itemaccept_item2*}
c={*serve_item*}

$o1$={*accept_item*}
$o2$={*serve_itemaccept_item2*}
$o3$={*forward_item1accept_item*}
$o4$={*forward_item2accept_item*}
o={*serve_itemaccept_item1, serve_itemaccept_item2*}

with the DRDL description for *F0* being

F0 ≙ *accept_item.F0‘ + fail.repair.F0*
F0’ ≙ *accept_item.F0” + serve_itemaccept_item.F0 + fail.repair.F0*
F0’ ≙ *serve_itemaccept_item.F0’ + fail.repair.F0*

that for *F1* being

F1 ≙ *forward_item1accept_item.F1’ + fail.repair.F1*
F1’ ≙ *forward_item1accept_item.F1” + serve_itemaccept_item1.F1 + fail.repair.F1*
F1’ ≙ *serve_itemaccept_item1.F1’ + fail.repair.F1*

that for *F2* being

F2 ≙ *forward_item2accept_item.F2’ + fail.repair.F2*

$$F2' \triangleq forward\_item2accept\_item.F2'' + serve\_ itemaccept\_item2.F2 + fail.repair.F2$$
$$F2' \triangleq serve\_itemaccept\_item2.F2' + fail.repair.F2$$

The *Pipe* is

$$Pipe \triangleq serve\_itemaccept\_item.(forward\_item1accept\_item.Pipe + forward\_item2accept\_item.Pipe)$$

and the control corrector is

$$C \triangleq serve\_itemaccept\_item1.serve\_item.C + serve\_itemaccept\_item2.serve\_item.C$$

5. Discussion and Conclusions

We have explored the relationship that exists between AFrames and PADL types, showing, in the process, how a description in one can be transformed into a description in the other. The relationship is such that most PADL types are related to an AFrame, if not in their original form then after having simple, behaviour preserving, transformations applied to it.

PADL is typical of ADLs, and is closely related to other ADLs such as Wright [1] and Darwin [9] which, in turn are related and/or have inspired other forms of ADL. This opens the door for the incorporation of many architectures and architectural styles – particularly those already encoded in PADL – into the Problem Frames approach.

In [5], Hall and Rapanotti define a problem calculus in which AFrames are use to transform problems. Through this technical note we therefore admit traditional architectures, as described in ADLs, as transformations into that problem calculus.

6. Acknowledgements

We are pleased to acknowledge the financial support of IBM, under the Eclipse Innovation Awards, and the EPSRC, Grant number EP/C007719/1. Thanks also go to our colleagues in the Computing Research Centre at the Open University, particularly Anne De Roeck, Bashar Nuseibeh and Michael Jackson.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connectors. ACM Transactions On Software Engineering and Methodology **6**(3):213-249. 1997.

[2] M. Bernardo, P. Ciancarini, and L. Donatiello. On the Formalization of Architectural Types with Process Algebras. In: Proceedings of SIGSOFT 2000 (FSE-8). Pages: 140--148. 2000.

[3] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. ACM Trans. Softw. Eng. Methodol. **11**(4):386--426. 2002.

[4] J. G. Hall and L. Rapanotti. Problem Frames for Socio-Technical Systems. In: Requirements Engineering for Socio-Technical Systems. Pages: 318--339. ISBN: 159140507-6. 2004.

[5] J. G. Hall and L. Rapanotti. A Framework for software problem analysis. Department of Computing Technical Report: 2005/05. 2005.

[6] J. G. Hall, L. Rapanotti, and M. A. Jackson. Problem Frame Semantics for Software Development. Journal of Software and Systems Modelling. Available from Springer-Online. 2005.

[7] C. A. R. Hoare. Communicating Sequential Processes. Prentice-Hall International. ISBN: 0131532898. 1985.

[8] M. A. Jackson. Problem Frames: Analyzing and Structuring Software Development Problem. Addison-Wesley Publishing Company. ISBN: 020159627X. 2001.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In: Proceedings of the 5th European Software Engineering Conference (ESEC 1995). Pages: 137-153. 1995.

[10] R. Milner. Communication and Concurrency. Prentice-Hall. ISBN: 0131150073. 1989.

[11] L. Rapanotti, J. G. Hall, M. A. Jackson, and B. Nuseibeh. Architecture-driven Problem Decomposition. In: Proceedings of the 12th IEEE International Requirements Engineering Conference (RE'04). Pages: 73--82. 2004.

[12] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. ACM Transactions on Software Engineering and Methodology **6**(1):1-30. 1997.