Advanced Networked Systems Architecture

ANSA

24 Hills Road
CAMBRIDGE
United Kingdom CB2 1JP

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: +44 223 323010
TELEX: 817343 BLUCAM G

What is an Architecture ?

or

A short stab at Brain Pain

Number: AO.26.01

Date: 3rd September 1986 12:25 pm

Area: AO (Architectural Overview)

Status: P (Discussion Paper)

Classification: C (**COMMERCIAL-IN-CONFIDENCE**)

Distribution: T (ANSA Team)

Author: HJW (John Winterbotham)

CONTENTS

1 INTRODUCTION

An architecture has been informally considered to consist of a 'framework, some components, and some rules'. To quote [1] -

> To understand the problem further, we must establish a reasonable definition of what we mean by the term 'architecture'. Within the ANSA activities we take an architecture to be a framework within which system designs may be placed and whose structure they adopt, a set of generic components that may (or must) be used in the systems, a set of generic relations (interfaces) that may (or must) be used between these components, and some rules about all of these that guide and constrain their use, and the addition of new parts of the framework or new components.

This paper lays a more rigorous foundation, that can form the basis for a **specification** of ANSA, and provide the theoretic structure within which designers can develop and specify conforming specialisations and systems.

2 WHAT IS AN ARCHITECTURE ?

In order to specify an architecture we must define a number of things. Initially we must specify a notation (syntax) that will be used to record and define the various parts of the architecture, and the meanings (semantics) of the notational elements. Together these form an **algebra**. Once we are able to record the architecture, we must address the issues of what will be recorded. There are four parts to the definition of the architecture - the **types** of things within it, the **state** that contains instances of these types and so contains the architectural and system description, the **predicates** that evaluate logical expressions concerning the definition, and the **rules** which apply the predicates and logical expressions constraining the type and state, and so allow statements about what is, or is not, part of ANSA conforming systems.

3 ALGEBRA DEFINITION

This section defines the algebra that is used to record the architecture.

At present the algebra used is an informal mixture of set theory and first order predicate calculus. The syntax employs fairly conventional notation with use of the logical and mathematical character sets available on the office automation system in use. Later changes to the notation are likely to make it machine readable, and possibly to move to an existing formalism such as VDM or LOTOS.

3.1 Symbols

_	A separating symbol within strings to aid readability.
()	Contains a clause of an expression for clarity and preciseness
[]	Contains a mapping
<>	Contains a list
{a}	A set of elements of type a
|	Indicates and separates alternatives, syntactic OR
¬	Unary NOT
=	Logical EQUALS
∨	Logical OR
∧	Logical AND
⇒	Logical IMPLICATION
⊨	Logical SEQUENT
a→b	A mapping from a to b
a∈b	a is a member of set b
∉	is not a member
a⊂b	a is a proper subset of b
⊄	is not a proper subset
a⊆b	a is a subset of b
⊈	is not a subset
∀a:b	For all a, b is true
∃a:b	There exist a such that b is true
∃!	There exists exactly one ...
a::=b	Symbol a *becomes* (is replaced by) expression b. (EG in definitions of **types.**
a:=b	Data unit a is *assigned* the value of expression b
a⇔b	a is EQUIVALENT to b
a≙b	a is DEFINED as b
a×b	The CROSS-PRODUCT of a and b

Other symbols not currently in use -
↓←↑⇌⇆∦∢∻≃≈∝±∓∇∞≤≥⋘⋙≜
∋∄⊣⊢⊊⊃⊅⊇⊉⊋∪∩∴∵⇐⇔∅≡

3.2 Notational conventions

In order to assist in reading the specifications, a number of conventions are adopted -

a) All names of TYPES are entirely in upper case characters.

b) General references to instances of a type will use the type name, but be entirely in lower case.

c) All data structures (within the state) will have names that start with a capital letter, and are otherwise in lower case.

d) Certain types will be assumed and have generally understood meanings; these will not be further defined, and this will be indicated by *italicising* their names, which will otherwise conform to these conventions.

e) Predicate names will be entirely lower case, and will be written in function form (see 6 below).

f) Comments are preceeded by '--' and end at the right hand end of the line.

3.3 The Logic of Partial Functions

In most practical application of logic, there will be many cases in which a simple TRUE / FALSE (boolean) result is inadequate, and we must allow for the indeterminate or undecidable outcome. This is discussed in detail in [4], especially page 9 and section 3.3. The basic alteration to the conventional two valued logic is an extension to the type BOOLEAN -

BOOLEAN::= {TRUE, INDETERMINATE, FALSE}

and the consequences of this on the remainder of the logic.

The effects of using a ternary logic have NOT been evaluated for this notation, with the exception of a brief examination of the differences between **implication** and **sequent**, that is discussed in section 7 below.

4 TYPE DEFINITION

The concepts and practice of type definition often causes confusion and conflict of opinion. This paper follows the views expounded in [2], especially section 3, which the reader is recommended to study.

4.1 Types as sets

In brief, the insight of [2] is that the definition of a type may be regarded as the definition of a **set of possible values**, while instances of that type are **members of the set** of possible values. To quote - "The phrase *having a type* is then interpreted as *membership* in the appropriate set.". All of the various type manipulations then become equivalent to set operations on the various sets of possible values. To quote again - "Since types are sets, subtypes simply correspond to subsets. Moreover, the semantic assertion *T1 is a subtype of T2* corrresponds to the mathematical condition $T1 \subseteq T2$."

This view of types is therefore a good match to the existing notation with which we represent the architecture, and indeed is so good as to raise the question of whether a 'type system' is required at all.

4.2 Do we need types ?

With the great similarity between type and set definitions, we must justify the use of typing concepts.

We believe that both approaches are necessary in order to retain and make explicit the defferences between -

> **Type statements** that are about sets of possible (allowed) values, and
> **Data statements** that are about selecting **values** from the type.

For example, and ignoring the notational conventions,

```
days     =   {Mon, Tue, Wed, Thurs, Fri}
a_days   =   {Mon, Tue, Wed, Thurs, Fri}
```

from which very little can be deduced about the semantics of 'days' and 'a_days'. However if we write -

```
days     := {Mon, Tue, Wed, Thurs, Fri} --Note this is assignment
A_DAYS ::= {Mon, Tue, Wed, Thurs, Fri} --Note this is replacement
```

we can easily see that 'A_DAYS' is a type definition of days of the week on which attendance is allowed , while 'days' is the set of days actually attended in a week (IE the set of days drawn from A_DAYS on which attendance was recorded.) From this we conclude that retaining the distinction serves a practical purpose.

4.3 How to define a type

Various notational styles are used for defining types, according to their appropriateness to the type being defined, and the whim of the definer.

Types are instances of the type TYPE, and TYPE may therefore be defined using the same notational schemes as may be used for any other types. Thus the following definitions are equivalent -

(TYPE ::= ATOMIC_TYPE | ENUMERATED_TYPE |
BOUNDED_TYPE | *ANY_OTHER_TYPE*)⇔

(TYPE := ATOMIC_TYPE ∪ ENUMERATED_TYPE ∪
BOUNDED_TYPE ∪ *ANY_OTHER_TYPE*)⇔

(TYPE := {t}: (t ∈ ATOMIC_TYPE∨
t ∈ ENUMERATED_TYPE∨
t ∈ BOUNDED_TYPE∨
t ∈ *ANY_OTHER_TYPE*))

where -

ATOMIC_TYPE ::= {*STRING , BOOLEAN, ,INTEGER, etc*}

ENUMERATED_TYPE := {value1 , value2 , value3, ...}

BOUNDED_TYPE := {x ∈ SOME_TYPE }:
(This predicate expression must be TRUE)

ANY_OTHER_TYPE ::= -- Well these are hardly sufficient on their own, so we must allow for other things we haven't yet thought about.

If this hasn't confused everyone I may get away without specifying a more formal definition !

5 STATE DEFINITION

Any algebra for defining architectures must be capable of defining the types of the components of the architecture (see section 4), but this is not sufficient. We must also have somewhere to specify **instances** of these types. The **state** is the repository for these instances (see [3] Chapter 4, p64), and so can be said to 'contain' the instances of the architecture.

However the state is itself an instance of a data store having a structure as well as value. The structure of the state is therefore given by a type definition (type **STATE**).

ANSA therefore must define type STATE (possibly while leaving some parts of it to be defined by the designer), and some of the data values that the architects require of all conforming system designs.

5.1 Type STATE

The type definition of STATE defines the required information about the names, properties, dimensions, and objects that make up conforming architectures.

STATE ::= {props: [PROPERTY_NAME → ANSA_PROPERTY ∪ *USER_PROPERTY*],

dims: {ANSA_PROPERTY_NAME},

objects: {OBJECT_NAME → *OBJECT*}

}

NOTE the notation name: type-expression *is used to name elements and to define their types.*

PROPERTY_NAME::= ANSA_PROPERTY_NAME ∪ *USER_PROPERTY_NAME*

ANSA_PROPERTY_NAME ::= {"Comms-layer", "HI-layer", "Release", "Structure-layer", "Abstraction-level"}

ANSA_PROPERTY ::= --Not sure what this is !

OBJECT_NAME ::= {n} : (n ∈ ANSA_NAME) ∧ (is_an_object(n))

ANSA_NAME ::= {n} : (n ∈ *NAME*) ∧ (ansa_conforming(n))

-- Et cetera !

5.2 State values in conforming designs

Where ANSA wishes to define values for part of the state, it may do so via the definition of requisite types (as for ANSA_PROPERTY_NAME above), via rules (*there shall exist at least 1 ANSA conforming directory service within the design, ...*), or via the definition of a **state template** which contains the required values, and which the designer copies and adds to as required. This raises the question of how the algebra represents the **initialisation** of the state.

state_template:: STATE -- With initialisation of some or all parts.

This question of initialisation is unanswered at present.

6 PREDICATE DEFINITION

A **predicate** is a function that returns a value that is a member of the set boolean. It may take many arguements -

PREDICATE::= [{ANY_TYPE} → BOOLEAN]

6.1 Notation

There are several notations in use for writing down predicates. However we envisage that many of the predicates that we will wish to define will take many arguements (>2 anyway), which rules out all notations except that used conventionally for functions -

PREDICATE::= [{ARGUEMENT_TYPE} → BOOLEAN]

or

predicate_name (arguements, ...) → (result ∈ BOOLEAN)

This notation is not quite so reader friendly, in that the implied ordering of arg1 and arg2 in the notation -

arg1 predicate_name arg2 → (result ∈ BOOLEAN)

is not available to be used (or abused) in the definition. However this advantage is outweighed by the disadvantages of needing multiple notational schemes, and so we use the single function style notation.

6.2 Operators

Some predicates are so commonly used that they are defined as **operators.** Many operators are derived from the set and predicate theoretic basis upon which we are building (EG $\vee$ $\wedge$ $\forall$ $\exists$ $\in$, etc), and are written in the form of a more or less standard symbol connecting the 1 or 2 arguements to which the operators apply. This is a notational convenience that we reserve for the generally recognised operators, and do not use when defining the architecture.

6.3 Predicate body definition

In order to actually define a predicate (which is a function really) we need a notation in which to express both the typing of the arguements and value, and the expressions by which the value may be derived from the arguements.

In this work we adopt the notation described in [4] section 1.2, which is as follows.

The definition of a predicate is given in two parts, the **signature** and the **body**. The signature provides information about the types of the arguements and the type of the value (always BOOLEAN in the case of predicates) into which the arguements are mapped. The body of the definition follows, and is composed of a formal predicate invocation (with formal arguements), a 'defined as' symbol ($\triangleq$) and the expressions that evaluate to the value returned.

For example (from [4] page 15) -

is_common_divisor:	$N \times N \times N1 \rightarrow$ BOOLEAN	--Signature
is_common_divisor(i,j,d) $\triangleq$	d divides i $\wedge$ d divides j	--Body

7 RULES DEFINITION

Rules are logical expressions that captures some aspect of the constraints on the expression of the architecture. Since any rule may be either TRUE (adhered to), INDETERMINATE, or False (not adhered to), it can be represented as a PREDICATE or a LOGICAL EXPRESSION in the logic of partial functions. As a basic method of evaluating the adherance to the rule these differ little. The main difference is that when expressed as a predicate, it may be parameterised and used in the succinct expression of other rules.

7.1 Form of rules

Two forms of logical expression appear to be candidates for the writing of rules, **implication** (⇒) and **sequent** (⊨). These can be shown to be interchangeable when using classical boolean logic, but are not interchangeable when using the logic of partial functions ([4] page 9). This can be illustrated -

	A⇒B	B=	T	?	F
A=	T		T	?	F
	?		T	?	?
	F		T	T	T

	A⊨B	B=	T	?	F
A=	T		T	?	F
	?		?	?	?
	F		?	?	?

NOTE: the '?' indicates the indeterminate condition required in the Logic of Partial Functions. Note also that the bottom row of the sequent table is uncertain as no definitions have been found for this world. The values given (indeterminate) have been chosen on the basis that the descriptions found appear to say nothing about the value of the sequent when A is FALSE.

When defining rules we wish to incorporate an expression or expressions that bound the applicability of a further expression that evaluates to the value of the rule. When such a bounding expression is not TRUE (IE the rule is not applicable), the evaluation of the rule is meaningless. If we used the **implication** to build the rules, they would evaluate to TRUE whenever the boundaries were indeterminate and the condition true, or outside the boundaries (these entries in the table are in bold). This is a rather permissive rule and would be misleading in some circumstances as conformance would appear to be true when in fact the conditions were not met.

We therefore prefer the sequent form (with the strong interpretation of the bottom row given above), and so define the form of a rule as -

RULE::= BOUNDS ⊨ CONDITION

BOUNDS:= {BOUNDARIES}

BOUNDARIES, CONDITION ∈ *LOGICAL_EXPRESSION*

This may also be interpreted in quantified form -

rule::= ∀b∈bounds : condition -- with some modification for LPF

8 EXAMPLE SPECIFICATION

To be added in a later edition.

9 REFERENCES

[1] *On the Dimensionality of Architectures* AO.14 Section 2

[2] "On Understanding Types, Data Abstraction, and Polymorphism. Luca Cardelli and Peter Wegner". Computer Surveys and University of St Andrews Annual Lecture Course 1985/86.

[3] "Software Development A Rigorous Approach" Cliff B. Jones Prentice / Hall

[4] "Systematic Software Development Using VDM" Cliff B. Jones Prentice / Hall