Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

DPL Reference Manual

Dave Otway

Abstract

This document defines the syntax of the Distributed Programming Language (DPL) and gives an informal definition of its semantics. Its intended audience are designers and implementors of translators, type libraries, attribute libraries and engineering support environments.

The design principles of the language and guidelines for its use are not provided in this document. These can be found in the ANSA Computational Model and the DPL Programmer's Manual.

APM.1015.01 **Approved** 23 May 1994

Technical Report

Distribution:
Supersedes:
Superseded by:

DPL Reference Manual

DPL Reference Manual

Dave Otway

APM.1015.01

23 May 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Overview

1.1 Purpose

This document defines the syntax of the ANSA Distributed Programming Language (DPL) and gives an informal definition of its semantics.

It is meant to be a reference document that defines the language precisely and unambiguously. It is not written as a tutorial and is not designed to be read in any particular order. Each chapter completely covers a particular aspect of the language and does not attempt to avoid forward references or to define concepts before they are referred to. No attempt is made to justify the design of the language or explain how to use it.

1.2 Audience

The intended audience are designers and implementors of translators, type libraries, attribute libraries and engineering support environments.

The audience is assumed to have a working knowledge of both the ANSA Computational Model and the DPL language.

1.3 Stability

The bulk of the DPL language is regarded as stable. The main areas of uncertainty are the type system, attributes and engineering terminations.

1.4 Related documents

APM.1001: *The ANSA Computational Model*

DPL was designed as a concrete syntax for writing programs that conform to these abstract semantics.

APM.1004: *The ANSA Atomic Activity Model and Infrastructure*

This is an architectural framework for the execution of concurrent distributed computations which maintain and preserve the integrity of distributed object state in the face of failures. As such it places extra requirements on the computational model. DPL does not yet take all of these requirements into account.

APM.1014: *DPL Programmers' Manual*

This is the language tutorial for application programmers.

APM.1020: *Abstract and Automate*

This presents the arguments that constrained the semantics of DPL.

2 Notation

The syntax of DPL is defined using extended Backus Naur Form (BNF) as the meta-syntax.

2.1 Meta-syntax

The meta-syntax is used to define how the language tokens may be composed into programs. It is defined as BNF with the simple extension of enclosing iterative parts of a production rule in curly brackets.

"xyz"	denotes the terminal symbol xyz
xyz	denotes the non-terminal symbol xyz
=	delimits a non-terminal symbol from its production rule
|	delimits alternative parts of a production rule
[ ]	enclose an optional (0 or 1 occurrences) part of a production rule
{ }	enclose an iterative (>=0 occurrences) part of a production rule

2.2 Lexical Meta-Syntax

The lexical meta-syntax is used to define how the language symbols may be composed into language tokens. It is defined as the meta-syntax with the following extensions, which may contain any character in the character set:

***	skips input characters until the closing bracket symbol which matches the immediately preceding opening bracket symbol; nested on every opening bracket symbol and un-nested on every closing bracket symbol while nested
...	skips input characters until the following terminal symbol
0--9	denotes any input character in the range "0" to "9" inclusive
tab	denotes the appropriate control character

These extensions are sufficient to define both types of DPL comments but need additional refinement to define the syntax of embedded blocks [§3.3] and string literals [§3.4].

3 Lexical syntax

Before the text of a program is parsed it must first be scanned to remove the separators and to compose the remaining symbols into a stream of tokens which form the terminal symbols of the DPL language. The set of valid tokens is composed of the lexical tokens defined in §3.2 plus the characters and keywords explicitly defined in §4.1.2 and §4.1.3.

3.1 Separators

Separators are recognised and processed before the token stream is constructed. A separator separates adjacent tokens but is otherwise ignored.

```
separator     =  " " | comment | layout_char

comment       =  "{"  ***  "}"  |  "%" ...  newline

layout_char   =  newline | return | tab | form feed | vertical tab
```

The encoding of layout characters is source character set dependent.

3.2 Lexical tokens

The following productions are used in the definitions of the lexical tokens:

```
digit         =  0--9

letter        =  a--z  |  A--Z  |  "_"

number        =  [ "-" ]  digit { digit }

string        =  """ ...  """    % see §3.4 for escapes
```

The following productions define the syntax of the lexical tokens:

```
embedded      =  "["  ***  "]"   % Square brackets in comments
                                 % and constants are ignored
                                 % see §3.3

identifier    =  letter { letter | digit }

literal       =  number | string
```

3.3 Embedded blocks

Within an embedded block any square brackets enclosed in string constants, character constants or comments must be ignored; this is dependent on the

syntax of the embedded language. Macro definitions and calls may also contain square brackets but as these are impossible to process in incomplete program fragments, the programmer is responsible for ensuring that they are properly balanced in an embedded block.

3.4 String literals

A string literal is a (possibly empty) sequence of characters enclosed in double quotes. This sequence may contain any characters in the source character set except the double quote ", backslash \, or newline character.

These characters, plus others that would be impossible or inconvenient to enter directly in the source program, may be included in string literals by using an escape sequence. Escape sequences are introduced by a backslash character. The effect of an invalid escape sequence is undefined.

3.4.1 Character escape sequences

Character escape sequences are used to represent some common special characters independently of the source character set. They consist of a backslash followed by an escape code. The escape codes and their meanings are:

a	alert (bell)	n	newline	v	vertical tab
b	backspace	r	return	\	backslash
f	form feed	t	tab	"	double quote

Both the backslash and the following escape code are replaced by the appropriate character for the escape code in the target character set.

3.4.2 Numeric escape sequences

Numeric escape sequences allow any character in the target character set to be expressed by writing the numeric value of the character in octal. They consist of a backslash followed by up to three octal digits. A numeric escape sequence terminates when three octal digits have been used or when the first character that is not an octal digit is encountered. The backslash and valid octal digits are replaced by the character in the target character set with that numeric value.

3.4.3 Concatenation

Adjacent string literals are automatically concatenated into a single string literal. This is intended for string literals that will not fit on one source program line.

4 Language syntax

This chapter defines the syntax of the DPL language. The distinguished non-terminal is program.

4.1 Terminal symbols

The set of terminal symbols in the DPL language syntax consists of the lexical tokens, characters and keywords defined below.

4.1.1 Lexical tokens

The following lexical tokens (defined in §3.2) are terminal symbols:

```
embeddedidentifierliteral
```

4.1.2 Characters

The following characters and character pairs are terminal symbols:

```
(  )  [  ]  <  >  ?  .  ,  ;  |  :  =  ||  :=  ->
```

No separators are allowed between the two characters in a pair.

4.1.3 Keywords

The following keywords are terminal symbols:

```
activity  code     handle       object          type
after     data     interface    reterminate
```

4.2 Reserved words

There are no reserved words; all of the keywords can be used as identifiers. This will not confuse a parser but it may confuse a human reader.

4.3 Semantic Indicators

The following productions are strictly redundant; they are used to convey semantic information such as which name space an identifier belongs to.

```
attributeBlock =  block
attributeName  =  identifier
language       =  identifier
name           =  identifier          % of an interface binding
operationName  =  identifier
terminationName=  identifier
```

4.4 Syntax rules

```
activity      =  "activity"  block

argumentList  =  "("  { declaration }  ")"

assignment    =  name { name }  ":="  expression

attributeList =  "<"  { attributeName [ attributeBlock ] }  ">"

block         =  "("  [ expressionList ]  ")"
                | "["  [ expressionList ]  "]"

declaration   =  name { name }  ":"  typeExpression

defaultHandler=  "?"  block

definition    =  name { name }  "="  expression

expression    =  activity | assignment | definition
                | handledBlock | initialisation | interface
                | literal | terminate | type | unit

expressionList=  expression [  {  ";"   expression }
               |  {  ","  expression } |  {  "||"  expression }
               |  {  "|"  expression }  ]

handledBlock  =  "after"  block  "handle"
                   "("  { namedHandler }  [ defaultHandler ]  ")"

initialisation=  declaration { declaration }  ":="  expression

interface     =  "interface"  [ attributeList ]
                   [  "data"  [ language ] embedded ]
                   "("  { signature operationBody }  ")"

invocation    =  unit  "."  operationName  block

namedHandler  =  terminationName  argumentList  block

object        =  "object"  [ attributeList ]
                   [  "data"  [ language ] embedded ] block

operationBody =  block |  "code"  [ language ] embedded

program       =  object

resultList    =  "("  { typeExpression }  ")"

signature     =  operationName [ attributeList ] argumentList
                   "->"  [ terminationName ]  resultList
                 {  "->"    terminationName    resultList  }

terminate     =  "->"  terminationName block
                | "->"  "reterminate"

type          =  "type"  [ attributeList ]  "("  { signature }  ")"

typeExpression=  type | unit

unit          =  name | invocation | block | object
```

5 Scope rules

5.1 Introduction of terms

An **identifier** is a symbol which provides a **name** for an operation, a termination or an interface.

A **binding** associates a name with an interface instance.

A **range** is a textual region of a program which limits the visibility of any bindings defined within it.

The **scope** of a binding is a textual region of a program where all occurrences of a particular interface name refer to that binding.

The **extent** of an instance of an object, interface or binding is the temporal interval in the evaluation of a program for which it is guaranteed to exist.

5.2 Name spaces

The names of operations, terminations and interfaces are interpreted in distinct name spaces and are therefore resolved in different contexts. The same identifier can be used to name an operation, a termination and an interface without any ambiguity; but this may confuse a human reader.

5.2.1 Operation names

Operation names distinguish operations in an interface or type. Operation names must be unique within a particular interface or type, which forms the context in which they are resolved. Different interfaces or types may have an operation with the same name and those operations may have different signatures. Operation names are introduced into a program only by writing them in contexts where an operation name is expected; they cannot be assigned or manipulated.

5.2.2 Termination names

Termination names distinguish terminations. Termination names must be unique within a particular operation signature. Termination names are introduced into a program only by writing them in contexts where a termination name is expected; they cannot be assigned or manipulated.

5.2.3 Interface names

Interface names distinguish bindings to interface instances. Interface names are resolved in the scopes of their bindings. The rest of this chapter is devoted to defining the properties of interface names in more detail.

5.2.4 Compile-time names

Attributes and embedded languages have their own distinct name spaces, which have compile-time contexts that can be changed by attributes. They do not name entities that exist at run-time.

5.3 Bindings

A binding is an association between an interface name and an interface instance. A binding may be either variable or fixed. Variable bindings may be re-assigned to different instances, fixed ones may not.

Bindings are defined once but may be referred to multiple times. The type of a binding is specified when the binding is defined. This type is taken as the type of any interface instance referenced via the binding and is the type to which any interface instance assigned to a variable binding must conform.

5.3.1 Defining occurrences

Variable bindings are defined and established by executing an initialisation expression.

- An initialisation expression simultaneously defines a number of variable bindings of the specified types and then assigns to them the corresponding interfaces delivered by the anonymous termination of the initialising sub-expression. The type of each interface must conform to the type of the corresponding binding.

Fixed bindings are defined and established by executing a definition expression or are defined by an arguments clause and then established by invoking an operation or by handling a termination.

- A definition expression defines a number of fixed bindings and then assigns to them the corresponding interfaces delivered by the anonymous termination of the initialising sub-expression. The type of each binding is defined to be the type of the interface assigned to it.
- The arguments clause of an operation signature in an interface expression defines the names and types of the formal arguments of that operation. Each subsequent invocation of the operation establishes fixed bindings between the formal argument names and the corresponding actual arguments, whose types must conform to the types of the formal ones.
- An arguments clause in a termination handler defines the names and types of the formal arguments of the termination. Each subsequent handling of the termination establishes fixed bindings between the formal argument names and the corresponding actual arguments, whose types must conform to the types of the formal ones.

5.3.2 Referential occurrences

Both variable and fixed bindings are referred to by using their name, which delivers the interface instance to which it is currently bound (except on the left-hand side of an assignment expression, where it refers to the binding itself).

Variable bindings may be re-assigned by referring to them on the left hand side of an assignment expression, which assigns to them the corresponding

interfaces delivered by the anonymous termination of the sub-expression on the right hand side.

5.4 Ranges

A range is a textual region of a program defined by specified syntactic constructs. Where range constructs are nested, the region within the enclosed construct is part of both the enclosed and enclosing ranges.

5.4.1 Open ranges

An open range is one that does not intrinsically affect the visibility of any bindings defined in an enclosing range. Variable and fixed bindings defined in an enclosing range can be referred to from within an open range unless masked by the definition of a binding of the same name in the enclosed range.

Specific open ranges are:

- each signature in a type expression
- each signature / operationBody pair in an interface expression
- each namedHandler in a handledBlock
- every other form of block expect the block of an object; specifically:
  - — the block of an activity expression
  - — the block of a defaultHandler
  - — the block of a handledBlock expression
  - — the block of an invocation
  - — the block of a terminate expression
  - — the block form of a unit expression
- each expression in a concurrent (| |), unconstrained (|) or exclusive (,) expresionList

5.4.2 Closed ranges

A closed range removes the visibility of any variable bindings defined in an enclosing range (i.e. variable bindings defined in an enclosing range cannot be referred to from inside a closed range). The visibility of fixed bindings defined in an enclosing range is not affected.

The only closed range is:

- the block of an object

The purpose of closed ranges is to prevent variables in one object from being referenced from another object; thus enabling objects to be encapsulated and distributed. Fixed bindings present no problems because they can be copied whenever a new instance of a referencing object is created.

5.5 Scopes

The scope of a binding is that textual region of a program in which all occurrences of a particular interface name refer to that binding. The scope of a

binding is the smallest range enclosing the defining occurrence of the binding, but excluding any enclosed ranges which mask the binding as defined below.

5.5.1 Scope of fixed bindings

The scope of a fixed binding excludes any enclosed ranges which:

- contain a definition of a binding with the same name

5.5.2 Scope of variable bindings

The scope of a variable binding excludes any enclosed ranges which:

- contain a definition of a binding with the same name
- are closed

5.5.3 Forward references

There is potentially a textual region at the start of a scope where a name is in scope but the defining occurrence of the binding has not yet been evaluated. Referential occurrences of a binding occurring in this region are known as forward references. Forward references are not permitted in order to avoid the possibility of attempting to evaluate references before they are established.

5.5.4 Self References

A binding is not established until its defining occurrence has been completely evaluated; so any reference to it on the right hand side of its defining expression will be unbound if it is evaluated. Therefore self references are not generally permitted.

However, it is impossible to define self or mutually recursive types and interfaces without such references. Self references inside type or interface definitions do not need to be evaluated and are permitted in the following restricted circumstances:

- only in definition expressions
- where the right hand side consists of a type expression, interface expression, or a block containing only type or interface expressions

A consequence of this restriction is that mutually recursive types or interfaces must be defined in the same definition expression.

5.5.5 Peer references

The expressions in a concurrent, unconstrained or exclusive expression list have no defined evaluation order, so there is no guarantee that a binding which had a defining occurrence in one of the expressions in the list and which had referential occurrences in one or more of the other expressions would be established before one of its references was evaluated. Therefore such peer references are prevented by defining each expression in a non-sequential expression list to be an open range.

For the same reason, references into the body of an activity are prevented by making its body an open range.

5.5.6 Shared references

Each expression in an expression list can reference bindings previously defined in enclosing ranges. For expressions in concurrent or unconstrained expression lists any references shared by multiple expressions are potentially concurrent. Also, references from the body of an activity to bindings previously defined in enclosing ranges are potentially concurrent with shared references from other activities.

Concurrent access to shared fixed bindings poses no problems; but the computational model (APM.1001: *The ANSA Computational Model*) does not guarantee the integrity of concurrently accessed shared variable bindings where at least one activity or sub-activity may be assigning to the binding. It is the programmer's responsibility to serialise access to such bindings. Currently, this can only be done by event counts and sequencers (RC.274: *DPL Nucleus Interface*). In future, atomicity and concurrency control mechanisms will be available (APM.1004: *ANSA Atomic Activity Model and Infrastructure*).

5.6 Extents

Instances of interfaces, objects and bindings must exist for at least as long as a reference can be made to (any part of) them.

The extent of an interface instance is from its creation (by evaluation of its defining interface expression) at least until there are no interface names still bound to it.

The extent of an object instance is from its creation (by evaluation of its defining object clause) at least until there are no interface names still bound to any instance of any of its interfaces and there are no activities being evaluated within it.

A binding may be closed over by referring to it from within an operation in an interface expression. The extent of a binding is at least longer than the lifetime of all activities executing within its scope and the extents of all interface instances which close over it.

Instances of interfaces, objects and bindings must be persistent throughout their extent but they become candidates for garbage collection outside of it.

The extent of an activity is the duration of the evaluation of the block forming its body and the extent of a sub-activity is the duration of the evaluation of the expression forming its body, including the delivery of its outcome to its parent activity. Activities and sub-activities must be persistent throughout their extent but the resources associated with them become candidates for garbage collection outside of it.

6 Type model

6.1 Kinds of type

The type model is primarily about interface types which in more general contexts are abbreviated to types. The three other kinds of type in the type model (expression types, interface list types and operation types) are just a convenient way of structuring and simplifying the description of interface types.

Interface types play the central role in the type model since:

- all bindings have interface types
- type expressions and interface expressions both construct interface types

6.1.1 Expression types

Expression types must be able to cater for invocations and blocks which can deliver multiple results. For consistency all expressions are defined to return an interface list although for some expressions the length of the list is always zero or one.

Expression types must also be able to cater for invocations and blocks which have multiple terminations each with a different interface list. Thus expression types consist of a set of interface list types.

6.1.2 Interface list types

Interface list types are lists of interface types.

For consistency all results are delivered as lists even if the length of the list is zero or one.

There is one exception to this rule; expression lists deliver a list of interface lists as their anonymous termination, but this is always reduced to a single interface list by the immediately enclosing block before any conformance checks are made.

6.1.3 Interface types

Interface types are constructed by type and interface expressions. They define the type of an interface as a set of bindings between operation names and operation types.

6.1.4 Operation types

Operation types (also known as signatures) map their argument types to their result types. That is, they map an interface list type to an expression type.

6.2 Conformance rules

Conformance rules are defined for each of the four kinds of type. These rules are defined in terms of each other. The rules are defined so that, for types S and T, S conforms to T if and only if S provides whatever is required by T. Therefore if S conforms to T, then something of type S may be provided wherever something of type T is required.

6.2.1 Expression conformance

An expression type E1 conforms to an expression type E2 if and only if:

- for each termination in E1 there is a termination with the same name in E2
- the interface list type delivered by each termination in E1 conforms to the interface list type delivered by the corresponding termination in E2

6.2.2 Interface list conformance

An interface list type L1 conforms to an interface list type L2 if and only if:

- the list L1 and L2 are the same length
- each interface type in L1 conforms to the interface type in the corresponding position in L2

6.2.3 Interface conformance

An interface type I1 conforms to an interface type I2 if and only if:

- for each operation in I2 there is an operation with the same name in I1
- for each such operation in I1 the operation type conforms to the operation type of the corresponding operation in I2

6.2.4 Operation conformance

An operation type O1 conforms to an operation type O2 if and only if:

- the interface list type of the arguments of operation O2 conforms to the interface list type of the arguments of operation O1
- the expression type delivered by O1 conforms to the expression type delivered by O2

6.3 Summary

An informal rule-of-thumb for interface type conformance is that of "*no-surprises*", which states that neither the user or provider of an interface should receive anything it was not expecting (i.e. operations or terminations it is not equipped to deal with).

For an interface type S to conform to an interface type T:

- S must have at least the operations of T
- each operation in T must have at most the terminations of the corresponding operation in S

This reversal of direction in the conformance relationship arises because the user of an interface chooses the operation but the provider of an interface chooses the termination.

This rule is then recursively applied to the argument and result types, but note that the direction of conformance for operation arguments is the reverse of that for operation results, since the user of an operation supplies the arguments while the provider of an operation supplies the results.

Note: The type model is under review (see RC.339 : *Revising the DPL Type System*).

7 Evaluation semantics

This chapter defines the evaluation semantics of DPL in the absence of engineering faults [§8] or resource limitations [§9].

The distinguished non-terminal is program.

7.1 Activity

```
activity  =  "activity" block
```

The evaluation of an activity constructor causes a new activity to be created and scheduled to evaluate the activity body. The block forms the body of the new activity. The outcome of an activity constructor is an anonymous termination with no results.

After an activity constructor has been evaluated, a new activity is guaranteed to have been created and scheduled to initiate the evaluation of the body as soon as sufficient resources are available to do so. The language semantics do not guarantee that such resources will ever become available.

The evaluation of the body of the new activity will be independent of all other activities. It will have shared access to all bindings within enclosing ranges, concurrently with any other activity having access to those bindings. All the components of the body will be evaluated as part of the new activity. The outcome of the block forming the body of the activity is discarded.

7.2 Argument list

```
argumentList  =  "(" { declaration } ")"
```

An argument list declares the formal names and types of the arguments to an operation (in the signature of a type or interface) or named handler. When the operation or termination handler is evaluated, fixed bindings are established between the formal argument names and the corresponding interfaces passed in the actual argument list [§7.14 *Invocation*].

7.3 Assignment

```
assignment  =  name { name } ":=" expression
```

An assignment expression changes which interfaces are bound to names in variable bindings.

The variable bindings named in the assignment expression must be in scope. The number of names must equal the length of the interface list in the

anonymous termination of the constituent expression and each interface in the list must conform to the type of the corresponding binding.

The constituent expression is evaluated. If the outcome is anonymous then the variable binding of each name is changed to refer to the interface in the corresponding position in the interface list delivered by the anonymous termination of the constituent expression, which is also the outcome of the assignment expression.

If the outcome is a named termination, all the names retain their existing bindings and the outcome of the assignment is the named termination.

The outcome of an assignment expression is always the outcome of its constituent expression.

7.4 Attribute list

```
attributeList  =  "<" { attributeName [ attributeBlock ] } ">"
```

Attributes are declarative: they specify how the program text within their scope should be interpreted.

Attributes are instructions from the programmer to the program development tool set. They are embedded in the program source so that their application can be tightly scoped and so that the compiler can check that they are being correctly applied.

Attributes may be attached to an object, interface, type or signature clause, and an attribute may have different effects on each kind of clause. The scope of an attribute is the whole of the clause to which it is attached. Default attributes may be subsequently altered or overridden by other attributes.

The name of the attribute must be defined in an attribute library which is in scope in the compile-time environment at the point at which the attribute occurs. The scope of attribute libraries are specified by attributes.

A particular attribute may be designed to be applied at any time during the compilation, linking or execution phases. It may cause the program text (or an internal representation of the program) to be transformed, different code to be generated, different engineering functions to be linked or behaviour to be changed dynamically at run-time. The precise effect of each attribute will be documented with its attribute library.

Some attributes require arguments, which must be provided by an attribute block following the attribute name.

Note: Currently, this `attributeBlock` must be enclosed in round brackets and may only contain an exclusive expression list of literals. More complex forms of `attributeBlock` are for further study.

Attributes may be used for many purposes, such as: enhancing the quality of a program by automatically transforming it; checking that an assertion is correct; making an assertion that a compiler cannot deduce from the source; altering a transparency; controlling the mapping to engineering and technology objects; making engineering or technology choices or specifying management policies. These many uses can be divided into two categories:

(i) changing the semantics of the program

(ii) changing how the program is implemented, configured or managed

In order to preserve the integrity of the computational model, an attribute which changes program semantics must always be defined in terms of source code transformations; although it does not have to be implemented this way.

7.5 Block

```
block  =  "(" [ expressionList ] ") | "[" [ expressionList ] "]"
```

A block selects components from the anonymous termination of its constituent expression list.

If an expression list is present, and its outcome is a named termination, then that termination is the outcome of the block.

If an expression list is present, and its outcome is a list of anonymous terminations, then the outcome of the block is an anonymous termination with an interface list constructed according to the form of brackets used to delimit the block.

- In a block delimited by round brackets "(" and ")" the anonymous termination of the block contains an interface list constructed by concatenating the interface lists from the anonymous terminations of all the expressions in the expression list.
- In a block delimited by square brackets "[" and "]" the anonymous termination of the block is the anonymous termination of the last expression in the expression list.

If no expression list is present, the outcome of the block is an anonymous termination with an empty interface list.

7.6 Declaration

```
declaration  =  name { name } ":" typeExpression
```

A declaration defines the names and types of bindings established by its enclosing construct.

7.7 Default handler

See [§7.11 *Handled block*].

7.8 Definition

```
definition  =  name { name } "=" expression
```

A definition expression creates fixed bindings between names and interfaces.

The number of names in a definition expression must equal the number of interfaces in the anonymous termination of the constituent expression.

The constituent expression is evaluated. If the outcome is anonymous, a fixed binding is established in the current range between each name and the interface in the corresponding position in the anonymous termination of the constituent expression, which is also the outcome of the definition expression.

Each binding has the same type as the interface it refers to. These bindings are defining occurrences and cannot be changed.

If the outcome of the constituent expression is a named termination, the bindings will not be established and the outcome of the definition expression is the named termination.

The outcome of a definition expression is always the outcome of its constituent expression.

7.9 Expression

```
expression  =  activity | assignment | definition
             | handledBlock | initialisation | interface
             | literal | terminate | type | unit
```

An expression constructs the outcome of its enclosing clause from the outcome of its component.

The component of the expression is evaluated. If the outcome is anonymous, it is used in constructing the anonymous outcome of the clause enclosing the expression. If the outcome of the component is a named termination then this is the outcome of both the expression and its enclosing clause.

The outcome of an expression is always the outcome of its component.

7.10 Expression list

```
expressionList  =  expression [ { ";" expression }
                   | { "," expression } | { "||" expression }
                   | { "|" expression } ]
```

An expression list controls the relative order of evaluation of its constituent expressions.

If the outcomes of all the expressions in an expression list are anonymous terminations then the outcome of the expression list is a list of those anonymous terminations in the textual order of the expressions. This list of anonymous terminations is reduced to a single anonymous termination by the block which must always enclose an expression list. If the outcome of any expression is a named termination then the outcome of the expression list is the named termination of one of the expressions.

The order of evaluation of the expressions in an expression list is controlled by an ordering operator. Expression lists with more than two expressions must have all their expressions separated by the same ordering operator.

- The sequential operator ";" specifies that the expressions must be evaluated one at a time in the order in which they are written until an expression delivers a named termination or all the expressions have delivered anonymous terminations. If the outcome of an expression is a named termination then that becomes the outcome of the expression list and any remaining expressions are not evaluated.
- The exclusive operator "," asserts that there are no dependencies between the expressions and specifies that the expressions must be evaluated one at a time in any order until an expression delivers a named termination or

all the expressions have delivered anonymous terminations. The order in which the expressions are evaluated may vary between different evaluations of the expression list. If the outcome of an expression is a named termination then that becomes the outcome of the expression list and any remaining expressions are not evaluated.

- The concurrent operator "||" specifies that the expressions must be evaluated with sufficient apparent concurrency to avoid deadlocks due to any dependencies between the expressions. There is no defined order for the initiation or completion of the evaluation of the expressions and all expressions must be fully evaluated. The evaluation of the expression list is complete when an expression delivers a named termination or all the expressions have delivered anonymous terminations. If more than one expression delivers a named termination then any one of them may be chosen as the outcome of the expression list. This choice is arbitrary and may not be repeatable.
- The unconstrained operator "|" asserts that there are no dependencies between the expressions and specifies that the expressions may be evaluated in any order with any degree of concurrency. The evaluation of the expression list is complete when an expression delivers a named termination or all the expressions have delivered anonymous terminations. If any expression delivers a named termination then not all expressions are guaranteed to be evaluated but any that have started to be evaluated must have their evaluation completed; this enables them to clean up and avoid leaving resources in invalid or unrecoverable states. If more than one expression delivers a named termination then any one of them may be chosen as the outcome of the expression list. This choice is arbitrary and may not be repeatable.
- If there is only one expression in the expression list then the outcome of the expression list is the outcome of that expression.

The evaluation of the expressions in a concurrent (or concurrently evaluated unconstrained) expression list is done by sub-activities of the activity evaluating the expression list.

7.11 Handled block

```
handledBlock   =  "after" block "handle"
                    "(" { namedHandler } [ defaultHandler ] ")"

namedHandler   =  terminationName argumentList block

defaultHandler=   "?" block
```

Note: This syntax may well change to bring it into line with the (as yet unspecified) case expression. Other changes under consideration are:

1) Allow multiple termination names to enable terminations with the same arguments to be processed by the same handler block.
2) Adding a syntactic marker between arguments and block to enable the default handler to be given a specific name without making it a reserved word.
3) A more meaningful pair of keywords.

Resolution of these issues has been postponed until a case expression is defined.

A handled block expression chooses the termination handler to evaluate when the outcome of the initial block is a named termination.

Evaluation of a handled block expression begins with evaluation of the initial block.

If the outcome of the initial block is an anonymous termination then that is the outcome of the handled block expression.

If the outcome of the initial block is a named termination for which there is a named handler (with matching termination name) defined in the handled block expression then the corresponding handler block is evaluated with the formal arguments of the handler bound to the arguments of the named termination from the initial block. The outcome of this handler block is then the outcome of the handled block expression.

If the outcome of the initial block is a named termination for which there is no named handler (with matching termination name) defined in the handled block expression then if a default handler (introduced by "?") is present its block is evaluated and its outcome is the outcome of the handled block expression; otherwise the named termination of the initial block is the outcome of the handled block expression.

Because the default handler block has no formal arguments any arguments of a named termination handled by it are not accessible.

Within any of the handler blocks a termination of the form ->reterminate causes that block and the handled block expression to terminate with the named termination (and its arguments) of the initial block as their outcome.

7.12 Initialisation

```
initialisation = declaration {declaration} ":=" expression
```

An initialisation expression establishes new variable bindings.

The number of names declared must equal the length of the interface list in the anonymous termination of the constituent expression and the type of each interface in the list must conform to the declared type of the corresponding name. An initialisation expression is the defining occurrence of a variable binding, with its declared type, for each declared name.

The constituent expression is evaluated. If the outcome is anonymous then a variable binding is established in the current range for each declared name and bound to the interface in the corresponding position in the anonymous termination of the constituent expression, which is also the outcome of the initialisation expression.

If the outcome is a named termination then no bindings are established and the outcome of the initialisation expression is the named termination. The bindings that consequently failed to be established cannot be subsequently referenced, because the named termination can only be handled in an enclosing range, where they will no longer be in scope.

The outcome of an initialisation expression is always the outcome of its constituent expression.

7.13 Interface

```
interface     =  "interface" [ attributeList ]
                   [ "data" [ language ] embedded ]
                   "(" { signature operationBody } ")"

operationBody =  block | "code" [ language ] embedded
```

An interface expression constructs new interfaces to an object. It creates a new instance of the interface it defines every time it is evaluated. The new interface is to the instance of the (nearest textually enclosing) object that the interface expression is evaluated in.

Evaluation of an interface expression creates a mapping from the operation names to their bodies and closes over all the (fixed and variable) external bindings referenced from within the operation bodies.

A closure retains the ability, when operations are subsequently invoked in the interface, to refer to the external bindings that were in scope when the interface was created. Access to these closed-over bindings is shared and potentially concurrent. This means that:

- bindings defined outside of any interface in an enclosing object are shared between all instances of all interfaces created in the same instance of the enclosing object
- bindings defined in an enclosing operation (or its arguments) are shared between all instances of all interfaces created in the same invocation of the enclosing operation
- an instance of an interface which refers to bindings defined in an enclosing operation (or its arguments) will refer to different instances of those bindings from all instances of all interfaces created by different invocations of the enclosing operation

The code embedded form of operation body enables the body to be coded in the target language of the compiler. The embedded code must conform to the code generation conventions for the translator when accessing arguments, accessing object and interface instance data, or returning results. These conventions are target language and translator specific. If present, the language name is used by the translator to select between multiple target languages or to check that the only available one is being used.

The evaluation of the body of an operation is deferred until the operation is invoked.

The outcome of an interface expression is an anonymous termination with a singleton interface list referring to the new interface instance constructed from the operations defined in the interface expression.

This new interface instance has the same type as all other instances created by the same interface expression, but may have different initial state due to differences in the external variable bindings from which the set of closures which form the operations where constructed.

Different instances of an interface created by the same interface expression, which have different invocation histories, may develop differing instance state (i.e. they may respond differently to identical invocations).

The optional data embedded clause enables target language data to be associated with each interface instance. The form of these data declarations

depends on the target language and translator, but would usually be in a suitable form for embedding in a target language record (or similar) construct. Embedded data can only be accessed from embedded code operation bodies. If present, the language name is used by the translator to select between multiple target languages or to check that the only available one is being used.

An instance of an interface must continue to exist for at least as long as a binding exists to it.

7.14 Invocation

```
invocation  =  unit "." operationName block
```

An invocation invokes an operation in a particular instance of an interface and supplies the actual argument list to be used in the evaluation of the body of the operation.

Evaluation of an invocation begins with the exclusive evaluation (i.e. one at a time in any order) of the unit and the block. The anonymous termination of the unit must be a singleton interface list. The anonymous termination of the block must be an interface list which conforms (in number, position and type) to the formal argument list of the named operation in the type of the interface delivered by the unit.

If the outcomes of both the unit and the block are anonymous then the named operation in the interface delivered by the unit is invoked with the interface list delivered by the block as its actual argument list. Invocation of the operation establishes fixed bindings between the formal argument names and the corresponding actual argument interfaces. Both the invocation clause and the operation being invoked share the same instance of any arguments or results passed between them. The operation is evaluated in the context of the operation closure formed when the interface instance was constructed plus the actual argument bindings. The outcome of the invocation is the outcome of the operation.

If the outcome of either the unit or the block is a named termination then the other one must, because the evaluation order is exclusive, either have been completely evaluated with an anonymous outcome or not evaluated at all. In this case, the operation is not evaluated and the outcome of the invocation is the named termination delivered by the unit or block.

7.15 Named handler

See [§7.11 *Handled block*].

7.16 Object

```
object  =  "object" [ attributeList ]
              [ "data" [ language ] embedded ] block
```

An object clause constructs new objects. It creates a new instance of the object it defines every time it is evaluated.

Evaluation of an object clause establishes a copy of each fixed binding defined in enclosing ranges and referenced from the block forming the body of the

object. The block forming the body of the object is evaluated in the environment formed by these bindings.

The outcome of an object clause is the outcome of its block.

The optional data embedded clause enables target language data to be associated with each object instance. The form of these data declarations depends on the target language and translator, but would usually be in a suitable form for embedding in a target language record (or similar) construct. Embedded data can only be accessed from embedded code operation bodies. If present, the language name is used by the translator to select between multiple target languages or to check that the only available one is being used.

An instance of an object must exist for at least as long as a binding exists to one of its interfaces or an activity is still evaluating (some part of) it.

7.17 Operation body

See [§7.13 *Interface*].

7.18 Program

```
program  =  object
```

The outcome of a program is the outcome of its constituent object. What happens to the outcome of a program is dependent on the programming environment.

7.19 Result list

```
resultList  =  "(" { typeExpression } ")"
```

A result list declares the types of the results returned by an anonymous or named termination of an operation.

7.20 Signature

```
signature  =  operationName [ attributeList ] argumentList
                     "->" [ TerminationName ] resultList
                   { "->"   TerminationName   resultList }
```

A signature clause declares the following details of an operation:

- the name of the operation
- the formal names and types of its arguments
- the names and result types of its terminations

There must be at least one termination, either anonymous or named. If there is more than one termination then only the first one can be anonymous.

7.21 Terminate

```
terminate  =  "->" TerminationName block | "->" "reterminate"
```

Evaluation of a terminate clause with a terminationName and block begins with the evaluation of the block. If the outcome of the block is anonymous then the outcome of the terminate clause is a named termination with the specified name and the results delivered by the anonymous termination of the block.

If the outcome of the block is named, then that is the outcome of the termination clause.

Evaluation of a termination clause with reterminate is only valid in the block of a termination handler. Its outcome is the named termination and arguments that caused the handler to be entered.

7.22 Type

```
type  =  "type" [ attributeList ] "(" { signature } ")"
```

The outcome of a type clause is an anonymous termination with a singleton interface list containing an instance of an interface of type `AbstractType` which represents the type specified by the list of operation signatures.

Note: The type model is being revised.

7.23 Type expression

```
typeExpression  =  type | unit
```

The anonymous outcome of a type expression must be an anonymous termination with a singleton interface list containing an interface conforming to the type `AbstractType`.

A type expression specifies the type of a variable, argument or result.

If the unit form is used it must be capable of being fully evaluated before it is needed for a type conformance check.

Note: This is normally at translation time, but since it is intended to provide type parameterisation this may be at run-time for type arguments.

7.24 Unit

```
unit  =  name | invocation | block | object
```

A unit is a subset of expression [§7.9] that restricts the components allowable in certain contexts. The outcome of a unit is the outcome of its component.

8 Fault model

The ANSA computational model is defined in a perfect world in which there are infinite resources that never break. Unfortunately, however clever the supporting engineering is at masking faults and avoiding resource exhaustion, it can only reduce the frequency of occurrence of faults but never eliminate them entirely.

This recognition of the inevitable (but hopefully rare) failure of the engineering to totally isolate an application from the consequences of faults and resource limitations is where a practical programming language diverges from the purer theory of the computational model.

In order to create a dependable application, an application program must be able to detect and deal with engineering faults. This raises a number of issues:

- how are faults reported
- where can faults occur (and not occur)
- what guarantees are provided on fault occurrence and non-occurrence
- how are faults propagated
- how extensive are the potential consequences of a fault
- how do fault reports effect the type system
- how are faults classified

Fault detection itself is the province of the engineering.

8.1 Fault reports

In order to integrate engineering fault semantics cleanly into the language it is desirable to use existing constructs as much as possible. The only construct in the language suitable for reporting faults is the named termination. This provides all that is necessary for a fault reporting mechanism in that it can discriminate between fault classes, pass specific information about particular instances and initiate alternative actions.

In addition, named terminations enable engineering services written in DPL to report faults.

The potential problems with using named terminations are that the points where named terminations can be raised do not coincide with the points at which faults need to be reported and this will consequently effect the type system. These issues are considered below.

8.2 Fault points

Invocations are the obvious points at which engineering faults may occur. This covers all external faults and faults in local engineering services which are called explicitly, but some other language constructs consume resources that cannot always be pre-allocated in practical systems. These are described below.

[Faults associated with particular “data types” in other languages (such as overflow or divide by zero) are handled by the named terminations of operations in the interface implementing the DPL “data type”, and are not regarded as engineering faults.]

Engineering faults will only be reported at the fault points identified below. Faults occurring anywhere else will be catastrophic [§8.5]. In particular, copying of bindings is assumed to be atomic.

8.2.1 Invocations

Invocations are ideal reporting points for communications faults, partial failures (of remote services) detected by the engineering and insufficient resources to evaluate an operation.

These faults are reported by (named) engineering terminations implicitly defined in the signature of every operation.

Faults detected by explicitly invoked local engineering services can be reported either by explicit or implicit terminations.

8.2.2 Object constructors

Constructing a new instance of an object consumes extra memory; if this is not available then the outcome of the object constructor is a (named) engineering termination.

8.2.3 Interface constructors

Constructing a new instance of an interface consumes extra memory; if this is not available then the outcome of the interface constructor is a (named) engineering termination.

8.2.4 Activity constructors

Constructing a new activity consumes extra resources; if these are not available then the outcome of the activity constructor is a (named) engineering termination.

8.2.5 Sub-activities

A sub-activity is used to evaluate each expression in a concurrent (or concurrently evaluated unconstrained) expression list. A sub-activity consumes extra resources; if these are not available then the outcome of its expression is a (named) engineering termination.

8.3 Fault guarantees

In order to construct dependable applications, a programmer needs to know at each possible engineering fault point exactly what guarantees the occurrence or non-occurrence of the fault has provided.

These guarantees are only valid at the fault points and care must be taken when handling the engineering terminations to avoid confusion with any faults caused by enclosed constructs.

8.3.1 Invocations

When the outcome of an invocation is an invocation engineering fault the operation is guaranteed to have been evaluated at most once.

When the outcome of an invocation is not an invocation engineering fault the operation is guaranteed to have been evaluated exactly once.

8.3.2 Object constructors

When the outcome of an object constructor is an object construction engineering fault the object is guaranteed not to have been constructed, and therefore the evaluation of its body has not been started.

When the outcome of an object constructor is not an object construction engineering fault the object is guaranteed to have been constructed and its body evaluated.

8.3.3 Interface constructors

When the outcome of an interface constructor is an interface construction engineering fault the interface is guaranteed not to have been constructed.

When the outcome of an interface constructor is not an interface construction engineering fault the interface is guaranteed to have been constructed.

8.3.4 Activity constructors

When the outcome of an activity constructor is an activity construction engineering fault the activity is guaranteed not to have been constructed, and therefore the evaluation of its body has not been scheduled or started.

When the outcome of an activity constructor is not an activity construction engineering fault the activity is guaranteed to have been constructed and its evaluation scheduled.

It is not possible to guarantee that the evaluation of an activity will be completed because it is subject to the vagaries of schedulers, resource managers and interaction with other activities. The new activity has the same status as all other activities as soon as the evaluation of its constructor has been completed successfully; this means that the underlying engineering mechanisms have allocated enough resources to *schedule* the new activity and have taken over the responsibility for initiating its execution as soon as sufficient resources are available to do so. In engineering terms this means that a thread has been created and queued but not necessarily a task.

8.3.5 Sub-activities

When the outcome of an expression being evaluated in a sub-activity is a sub-activity construction engineering fault the expression is guaranteed not to have been evaluated.

When the outcome of an expression being evaluated in a sub-activity is not a sub-activity construction engineering fault the expression is guaranteed to have been evaluated.

8.4 Fault propagation

In order to avoid the confusion which may be caused by unhandled engineering faults being propagated out to an enclosing instance of the same construct, these propagated faults may be changed in some way so as to make such propagation obvious.

Note: This is a subject for further study.

8.5 Fault boundaries

A fault may be an isolated incident or a symptom of a spreading malaise. Before attempting corrective action the potential extent of any consequential corruption must be known. The obvious boundaries to the spread of any corruption are the domain (administrative, naming, security, etc.), node, capsule and object.

To a large extent, domains can be protected from one another by interceptors, nodes are naturally protected from each other by their physical separation and capsules are protected from each other by addressing mechanisms; but in most cases protection mechanisms for individual objects are too expensive.

Certain faults within capsules and nodes may leave the capsule or node so corrupted that it can not be relied on to take effective recovery action. These faults are designated as catastrophic and will result in the capsule or node being aborted, leaving the rest of the system to cope with their disappearance.

Note: This is a subject for further study.

8.6 Type implications

Adding engineering terminations has implications for the type model. If they are added explicitly the type of every interface and constructor expression would be changed. Engineering terminations will differ between different implementations.

Type checking explicit engineering terminations is beneficial in that it would ensure engineering conformance but detrimental in that it would restrict portability. Engineering conformance will not change very frequently and does not need to be rechecked for every binding. Applications attempting to recover from engineering faults will be more concerned with fault boundaries and recovery strategies than the exact nature of the faults. Precise details of faults are of more concern to systems administrators and designers.

Therefore the termination names can be chosen to reflect the fault boundaries and possible recovery strategies while the fault details can be passed as argument strings, enabling a standard set of termination names to be

specified. Each interface and constructor expression would then have its type argmented in the same way and there is no need to burden the programmer and type checker by making this explicit.

Note: This is a subject for further study.

8.7 Fault names and classifications

Note: This is a subject for further study.

9 Resource management

The ANSA computational model assumes sufficient resources are available to completely evaluate a program; unfortunately this is not always the case.

Although the engineering infrastructure is responsible for making the required resources available to a program, it cannot always guarantee to do so. Therefore the programmer must take some responsibility for the management and utilisation of resources.

The following sections define this division of responsibility for the three resources necessary to evaluate a program written in the DPL language: memory, processing and communications. Access to physical devices is outside of the computational model and must be provided by an interface (with operations implemented in another language) which interacts according to the model.

9.1 Memory management

Memory is a resource that is constantly being consumed during the evaluation of a program. Although virtual memory spaces can be very large they are still finite. The two problems with the management of the memory resources are:

- how can memory be most effectively recycled
- what happens when it finally runs out

9.1.1 Infrastructure responsibilities

The infrastructure is responsible for allocating all memory except that associated with tasks [§9.2.2].

The ANSA computational model and the DPL language provide no mechanisms for explicitly deleting anything and thus implicitly assume that everything that is created will exist forever. However, the infrastructure can safely delete an entity and recycle its memory resource if it is certain that the disappearance of the entity can never be proved by an application program. This means that entities can be garbage collected outside of their extent [§5.6]. The infrastructure is responsible for this garbage collection both locally and remotely.

Note: Garbage collection of distributed closed graphs is still a current research area.

If the infrastructure cannot prove that a long unused entity is garbage, it can passify it to a cheaper storage medium.

If memory runs out, the infrastructure must generate a fault report [§8.1]. It is responsible for ensuring that such fault reports are only generated at defined fault points [§8.2]. This means that memory required for other entities must be pre-allocated at the previous fault point.

Note: Certain stack technologies make this impossible or very expensive to achieve for invocation links, such faults are catastrophic [§8.5].

9.1.2 Programmer responsibilities

Strictly speaking there are no programmer responsibilities for memory management, but a programmer can aid the garbage collector by ensuring that extents are as short as possible.

Note: Explicit deletion mechanisms and application supplied clean up operations are for further study.

9.2 Processing management

Processing resources are represented by activities and sub-activities.

Activities and sub-activities are both mapped onto threads in the engineering model. A thread represents what is to be evaluated and the resources required to do the evaluation are provided by a task. Tasks are mapped onto whatever processing resources are provided by the underlying technology.

9.2.1 Infrastructure responsibilities

Threads are very small and are pure memory resources so they do not need to be separately managed by the infrastructure; they are just part of the general memory resource and the infrastructure is completely responsible for them.

9.2.2 Programmer responsibilities

Tasks are much more substantial resources which map onto the underlying processor resources and require substantial amounts of memory such as stacks. To avoid wasteful usage, task allocation needs to be limited. Threads only need a task while they are actively being evaluated; they can manage without one while they are queued waiting for evaluation.

To make progress a program needs only one more task than the maximum number of threads that can be blocked awaiting an event count [RC.274: *DPL Nucleus Interface*] at the same time. Any more tasks than this minimum number can be used to increase the real concurrency of the program.

The programmer is responsible for specifying the number of tasks in a capsule using the `nucleus.tasks` operation [RC.274: *DPL Nucleus Interface*] and for ensuring that this is sufficient to avoid deadlock due to lack of tasks.

Note: It is intended to transfer the responsibility for task deadlock avoidance to the infrastructure, while still allowing the programmer to increase the real concurrency above the minimum. This is for further study.

The programmer is responsible for ensuring that threads do not deadlock for reasons other than lack of tasks (see APM.1004: *ANSA Atomic Activity Model and Infrastructure* for suitable algorithms and specifications).

9.3 Communications management

Invocation by one object of an operation in an interface to a non co-located object (i.e. in a different capsule) requires communication resources.

9.3.1 Infrastructure responsibilities

The management of communications resources is completely transparent to the application program and is the responsibility of the infrastructure.

9.3.2 Programmer responsibilities

The programmer is responsible for introducing client and server programs using a trader via the binder interface of the nucleus [RC.274: *DPL Nucleus Interface*].

When an invocation fails due to an engineering fault [§8] then the program may attempt to recover by re-trading. There is no guarantee that this will deliver a binding to the same instance of the service previously in use; in fact it is extremely unlikely to. A new service instance will not contain the previous invocation history.