Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

TINA-DPE AST Design

N. J. Howarth

Abstract

This document describes the design of an Abstract Syntax Tree and Pretty Printer using C++.

APM.1180.00.03 **Draft** 27th June 1995

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

TINA-DPE AST Design

TINA-DPE AST Design

N. J. Howarth

APM.1180.00.03

27th June 1995

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	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1	**1**	**Introduction**
3	**2**	**The Various Approaches**
3	2.1	The DPL Approach
3	2.1.1	Creating the AST
4	2.1.2	Walking the Tree
4	2.1.3	Comments
4	2.2	The Modula-3 Approach
5	2.3	The Tina-DPE Approach
5	2.3.1	Requirements
5	2.3.2	Useful Features of C++
6	2.3.3	A First Attempt
7	**3**	**Base classes for the AST**
7	3.1	The Node Class
7	3.1.1	Constructors
7	3.1.2	Member Functions
7	3.2	The NodeList Group of Classes
7	3.3	Support Classes
8	3.3.1	The String Class
8	3.3.2	The Indent Class
9	**4**	**Detailed method of AST Construction**
9	4.1	The Nodes
9	4.1.1	The Attribute Node
9	4.1.2	The Binding Node
10	4.1.3	The Block Node
10	4.1.4	The Declaration Node
11	4.1.5	The Handled Node
11	4.1.6	The Handler Node
11	4.1.7	The Identifier Node
12	4.1.8	The Interface Node
12	4.1.9	The Invocation Node
12	4.1.10	The Literal Node
12	4.1.11	The Number Node
12	4.1.12	The Object Node
13	4.1.13	The Operation Node
13	4.1.14	The Signature Node
13	4.1.15	The Terminate Node
14	4.1.16	The Termination Node
14	4.1.17	The TypeBlock Node
14	4.1.18	The TypeConstructor Node
14	4.1.19	The TypeDefinition Node

14 4.1.20 The TypeExpression Node

15 5 The Generic View
15 5.1 The Identifier Node

17 6 Specific Views
17 6.1 Pretty Printer
17 6.2 Scope
17 6.3 Type inferencing

1 Introduction

This document describes the design of an Abstract Syntax Tree (AST) and the rationale behind that design. The document is intended to aid understanding of the AST code and to facilitate amendments and additions to that code.

Two approaches are investigated: that used by DPL, and that taken by Modula-3. The actual approach taken is then discussed. These are described in Chapter 2.

Chapter 3 describes the basic C++ classes on which the particular nodes and other aspects are built.

Chapter 4 describes the construction of the individual nodes within the AST, and how they relate to each other.

Chapter 5 describes the approach to providing generic and specific views of the AST.

Chapter 6 describes the specific views provided for the AST. To date the only specific view described here is that of the “Pretty Printer”.

2 The Various Approaches

2.1 The DPL Approach

2.1.1 Creating the AST

Each node is represented by a structure for the particular type of node. A node is created by invoking a function which returns a pointer to a structure representing that node. The function takes as arguments the various sub-nodes for the particular type of node. For example a *Signature* node can be created using function *node_signature*. The syntax for *signature* is:

```
signature     = operationName [attributes] arguments responses
```

so a *signature* node will have sub-nodes for the operation name, attributes, arguments and responses. The structure and the function invocation which creates it are:

```
struct Signature
     {
     struct AttributeList    *attributes;
     struct Identifier       *name;
     struct DeclarationList  *arguments;
     struct TerminationSet   *responses;
     };
extern struct Signature *node_signature (
               struct AttributeList    *attributes,
               struct Identifier       *name,
               struct DeclarationList  *arguments,
               struct TerminationSet   *responses);
```

The parser works its way to the bottom of each branch, then backtracks up, generating the nodes as it does so, hence the various arguments to the *node_signature* function are known before it is called. Here the function which parses signatures invokes functions which parse the various sub-trees. Each of these returns a pointer to the node for the sub-tree. The function *node_signature* is then invoked with these pointers as arguments. Function *node_signature* then allocates space for a signature node, and copies the node pointers into the structure for the new node.

Other nodes are handled in a similar manner.

Two main types of node are dealt with: those which take an assortment of sub-nodes, of which *signature* is an example, and those which take a list. The latter takes two arguments, pointers to the nodes at the head and tail of the list. Each of these nodes will already have sub-nodes associated with it, depending on the type of node.

2.1.2 Walking the Tree

As when creating the tree, a function is provided for each type of node. For DPL, different tree walks uses different code. For example, a pretty printer is written in DPL, and a scope checker is written in C. Both walk the tree, with no common code, although they perform similar operations.

The pretty printer has a function for each node. This invokes further functions to print the sub-nodes, and adds in any syntax required (e.g. parenthesis etc.). The scope checker performs a similar task, invoking functions to carry out scope checking of sub-nodes.

2.1.3 Comments

The only problem area would appear to be the mapping of syntax to AST nodes. In some cases additional nodes have been created to facilitate the code, and there is not a direct mapping between nodes and items of syntax.

2.2 The Modula-3 Approach

The Modula-3 package provides basic facilities for specifying an Abstract Syntax Tree. It defines a basic node type AST.NODE, and in separate interfaces, specifies a set of standard methods applicable to an AST node. Language specific ASTs are defined by subtyping AST.NODE and providing implementations for the standard methods.

Standard methods and support include:

- init - provides a stub for any initialisation code
- name - returns the name of a node (useful for debugging)
- provision of node information (number of children, pointer to n'th child etc.)
- iterator for node children through all levels
- support for tree walks - visit own children
- support for tree copies
- support for "displaying" a tree node - language specific

The "tree" is actually a graph, consisting of a set of connected nodes which are all instances of subtypes of the object type NODE. Nodes can have attributes, which are ultimately represented as object fields or methods. Typically, an attribute is a reference or connection to some other node in the AST.

An AST for a specific language is specified as a set of interfaces, which share the naming convention LLAST, where LL is a language-specific prefix. Within this set, it is also conventional to specify the AST as a series of views, each of which provides some new nodes (possibly none) and new attributes on nodes defined in other layers.

The declarations of the node types and the specifications of the node attributes are divided into separate interfaces. The node types for each view are defined in an interface named LLAST_VV, where VV is a tag denoting the view. The "fundamental" attributes on these nodes are specified in an interface named LLAST_VV_K, where K is a tag which denotes either the kind of attribute that is being added or indicates a sub-view. For example, F is conventionally used

to indicate attributes represented as object fields, and M indicates methods that are applicable to this view.

Attribute types for the Modula-3 implementation fall into two groups. The first group comprises lexical types, denoting, for example, the characters of an identifier of the characters of a TEXT literal. These types are given concrete definitions by the particular compiler implementation. The second group comprises the children of a node, which are always node types.

The rationale behind this is that when a new node type is created, is has the sum of all the attributes that were specified in the contributing views. The different views, however, are distinct, and not aware of each other.

2.3 The Tina-DPE Approach

2.3.1 Requirements

- a facility to create nodes for each element of the syntax
- nodes should be generated bottom up, so that as each node is created, it knows about its children (if any)
- access to information in the node is available only by invocation, to ensure encapsulation and safety of data
- access to information in the node is view-based
- it should be possible to:
  - iterate throughout the entire tree
  - walk through the children of a node
  - display a node
  - add additional views of a node
- the AST code should be written in C++

2.3.2 Useful Features of C++

The most immediately apparent feature of C++ which will be of help is that of the abstract class. A “node” class can contain pure virtual functions which can then be implemented in a derived class for each type of node. By this means nodes with different behaviour can be treated in the same way.

Taking each of the above requirements in turn:

- A node will be defined for each element of the syntax. This will be derived from a base class “node” which contains (mostly) pure virtual functions. This derivation may be indirect via other generic nodes. Nodes can then be created using the **new** operator.
- Nodes will be generated bottom up, and pointers to subsequent nodes passed to new nodes as they are created. Such subsequent nodes or groups of nodes may comprise an object in their own right.
- All data will be private to the nodes (at some level of derivation) and accessible to the outside world only by using member functions. In many cases the data will not be available directly in any form, but its effect seen by executing a member function.

- Multiple constructors will facilitate application programming by providing increased flexibility.
- The use of “views” can be supported by the use of member functions.
- Member functions will be provided to:
  - iterate throughout the tree
  - walk through the children of a node
  - display a node
- Additional member functions can be added to support an additional view. Initially no commonality will be determined between the views. When such commonality becomes apparent, this may be apportioned to a “generic view” member function for that node.

2.3.3 A First Attempt

Since the syntax from TINA is not yet available, an attempt will be made to use the AST to model DPL. Details of the syntax and semantics of DPL can be found in [ANSA 93].

It should be possible to define the node constructors and destructors, iterators and pretty printer within the new AST structure, and fit this in to the existing DPL compiler, merely changing the calls which create the nodes to invocations of operations in the new C++ classes.

3 Base classes for the AST

This section provides an outline of the construction of an AST node, and identifies the data and member functions required in the base nodes.

There are two major types of classes: those that represent an individual node, and those that represent a list of nodes. A management class is also required to handle the latter group.

3.1 The Node Class

The most basic class within the AST design is the Node class. Each element of the syntax is represented by a node. The basic node class provides a group of constructors and several member functions, most of which are not generally used by the derived classes, which provide their own constructors for their specific member data. However those of the base node have been left in to provide greater generality, and to provide a template for derived classes.

Two data members are provided: a pointer to a list of nodes, and a string, which is generally used to hold the name of the derived class, and hence provide an aid to debugging.

3.1.1 Constructors

- `Node( char *name )` uses the String constructor (see section 3.3.1) to set the given character string into a String class.

3.1.2 Member Functions

- `virtual void display() const` displays the name of the node.
- `int is_id( const char *other )` invokes the compare function on the String object holding the name of the node. This returns a 1 if the String matches the char string given, otherwise it returns a 0.

3.2 The NodeList Group of Classes

A node list consists of a head, which is a pointer to a node, and a tail, which is a pointer to a further nodelist. Items can be added to the head of an existing list by setting the head of a new list to the new item, and the tail to the existing list. An empty list will have both head and tail set to null. The last item in a list will have its tail set to null.

3.3 Support Classes

Certain generic classes provide support for other classes by defining specific data or operations.

3.3.1 The String Class

The String class provides general string handling capability for the nodes. A string is represented by a character buffer and a length, and supports the following operations.

3.3.1.1 Constructors

Two constructors are provided:

- `String::String()` generates an empty string.
- `String::String( const char *s)` generates a string with the length and content of the character string passed as an argument.

3.3.1.2 Destructor

The destructor deletes the buffer created by the constructors.

3.3.1.3 Member Functions

- `void display() const` outputs the string to the standard output device.
- `void operator=( const String &other)` overwrites the existing string with the new string other.
- `int compare( const char *other )` compares the string with a character string, and returns 1 if they are the same, otherwise returns 0.
- `int compare( const String *other )` compares the string with another string, and returns 1 if they are the same, otherwise returns 0.
- `int get_length() const` returns the length of the string.

3.3.2 The Indent Class

The Indent class is used by the Pretty Printer to control indentation. The number of indentation steps is held in the member data steps, and this determines the number of times an indent is printed when the display function is invoked.

3.3.2.1 Constructors

- `Indent()` sets the number of steps to 0.
- `Indent( int n )` sets the number of steps to n.

3.3.2.2 Member Functions

- `void inc()` increments the number of steps.
- `void dec()` decrements the number of steps.
- `void display() const` outputs a new line followed by the current indentation.

4 Detailed method of AST Construction

This section describes the detailed design of the nodes for each element of the syntax, in particular data members and constructors.Member functions are handled under generic and specific views in Chapters 5 and 6.

The individual nodes are described in the following sections in alphabetical order. At the start of the discussion on each node, the relevant syntax is shown. This can then be easily related to the member data and constructors for that node.

The use of optional constructors greatly simplifies the use of these nodes, providing much greater flexibility for the application programmer.

4.1 The Nodes

4.1.1 The Attribute Node

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

An attribute takes a variable number of name and (optional) block pairs. This is represented by an *attributeList* object, which is a list of pointers to *attributeNode* objects.

An *attributeNode* object represents a single *attributeName [attributeBlock]* pair.

The member data for an attribute name consists of pointers to two nodes:

- *identifier*, which represents the *attributeName*
- *block*, which represents the *attributeBlock*

Since the *attributeBlock* is optional, there are two constructors:

`AttributeNode( Node *id )` takes a single argument, which is copied to the *identifier*. *Block* is set to zero.

`AttributeNode( Node *id, BlockNode *bl )` takes two arguments, for the *identifier* and the *block*.

Attributes are not currently used.

4.1.2 The Binding Node

definition = name {name} "=" expression
assignment = name {name} ":=" expression
initialisation = declaration {declaration} ":=" expression

A binding takes a list of at least one declaration, and an expression, and an indication of the type of binding. This is represented as follows:

- *declarations*, a pointer to a list of nodes which represents a list of binding names

- *expression*, a pointer to a node representing the expression
- *type*, a value indicating whether the binding is a constant, initial or variable binding

Since the binding can take either one or more names, indicated by a pointer either to a single node or to a list, there are two constructors:

`BindingNode( DeclarationList *decs, ExpressionNode *exp, e_type t )` takes as arguments a pointer to a list of declarations (names), a pointer to an expression, and a value indicating the type of the binding.

`BindingNode( DeclarationNode *dec, ExpressionNode *exp, e_type t )` takes as arguments a pointer to a node for a single name, a pointer to an expression, and the value indicating the type of the binding.

4.1.3 The Block Node

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

A block consists of one or more expressions contained either in parenthesis "()" or square brackets "[]". This is represented as follows:

- *value*, a value indicating the type of terminations returned, from all expressions (*allExprs*) or from the last expression only (*lastExpr*)
- *order*, a value indicating the ordering of the expression, which can be *empty*, *singleton*, *sequential*, *exclusive*, *unconstrained*, or *concurrent*
- *expressions*, a pointer to a list of nodes which represents a list of expressions.

Since the block can take either none, one or more expressions, there are three constructors:

`BlockNode( e_value val, e_order ordr, ExpressionList *exps)` takes the full complement of parameters, the value, order, and a pointer to a list of expressions.

`BlockNode( e_value val, Node *exps )` takes the value and a pointer to a single expression node. This block must be a singleton and is set up as such in the constructor.

`BlockNode( e_value val )` takes only the value. This represents an empty block, and the order is set up accordingly.

4.1.4 The Declaration Node

declaration = name { name } ":" type_expr

This design follows that of the DPL compiler which in practice only permits a single name within a declaration node. This could easily be modified should it become a requirement for a declaration to include a list of names.

A declaration list takes a variable number of name and type expression pairs. This is represented by an *declarationList* object

An *declarationNode* object represents a single *name typeExpr* pair, although the *typeExpr* is optional.

The member data for an declaration consists of pointers to two nodes:

- *name*, which represents the *name*
- *type*, which represents the *typeExpr*

There are two constructors:

`DeclarationNode( IdentifierNode *i )` takes a single argument, which is copied to the *name*. *type* is set to zero.

`DeclarationNode( IdentifierNode *i, TypeExprNode *n )` takes two arguments, for the *name* and the *type*.

4.1.5 The Handled Node

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

The handled node is represented as follows:

- *handled*, a pointer to a block node
- *handlers*, a pointer to a list of handler nodes,

Since there many be none, one or many handlers, there are three constructors:

`HandledNode( BlockNode *h )` takes a single argument, a pointer to a block node.

`HandledNode( BlockNode *h, HandlerList *hset )` takes a pointer to a block node and a pointer to a list of handlers.

`HandledNode( BlockNode *h, HandlerNode *hset )` takes a pointer to a block node and a pointer to a single handler.

4.1.6 The Handler Node

namedHandler = terminationName arguments block

defaultHandler = "?" block

All handlers are treated in the same way, and the default handler can be distinguished since it has no termination name.

The handler node is represented as follows:

- *block*, a pointer to a block node
- *name*, a pointer to an identifier node
- *args*, a pointer to a list of arguments

There are several options here. All handlers must specify a block. Handlers other than the default handler specify a name, and may optionally have arguments. There are therefore four constructors:

`HandlerNode( BlockNode *b )` just takes a pointer to a block node. This represents the default handler.

`HandlerNode( IdentifierNode *i, BlockNode *b )` takes both a name and a block. This represents a named handler with no arguments.

`HandlerNode( IdentifierNode *i, DeclarationNode *d, BlockNode *b )` takes a name, a block and a single argument.

`HandlerNode( IdentifierNode *i, DeclarationList *d, BlockNode *b )` takes a name, a block, and a list of arguments.

4.1.7 The Identifier Node

The identifier node takes a single argument which is held as a String type. Its constructor is:

```
IdentifierNode( char *name ).
```

An identifier node also has data representing its length, its position in the file which is being compiled, a pointer into the symbol table, and its type.

4.1.8 The Interface Node

interface = "interface" [attributes] ["data" embedded]
"(" {operation} ")"

The interface node is the first of the nodes which has too many constructors to list here. The large number of constructors is due to the fact that both attribute and data members are optional, and there may be none, one or more operations. Also if there are attributes, there may be one or more of these. ince "one or more" is represented either by a pointer to a node, or by a pointer to a list, this leads in total to seventeen constructors.

The data is held as follows:

- *attributes*, a pointer to a list of attributes
- *data*, an object of the class LiteralNode
- *operations*, a pointer to a list of operations

4.1.9 The Invocation Node

invocation = unit "." operationName block

An invocation node consists of the following:

- *unit*, a pointer to a node which might be either a name, an invocation, a block or an object
- *ident*, a pointer to an identifier node representing the operation name
- *block*, a pointer to a block node

The invocation node has only a single constructor:

```
InvocationNode( Node *u, IdentifierNode *i, BlockNode *b ).
```

4.1.10 The Literal Node

The String node takes a single argument which is held as a String type. Its constructor is:

```
LiteralNode( char *str ).
```

4.1.11 The Number Node

The number node takes a single argument which is held as a String type. Its constructor is:

```
NumberNode( char *name ).
```

4.1.12 The Object Node

object = "object" [attributes] ["data" embedded] block

An object always takes a block as a sub-node, and may optionally have attributes and/or embedded data. It is represented as follows:

- *attributes*, a pointer to a list of attributes
- *data*, a LiteralNode holding the data string

- *block*, a pointer to a block node

Since attributes and data are optional, and the attribute element may be a single attribute or a list, there are six constructors:

`ObjectNode( Node *bl )` takes only a block pointer.

`ObjectNode( AttributeList *attr, Node *bl )` takes a list of attributes and a block pointer.

`ObjectNode( AttributeNode *attr, Node *bl )` takes a single attribute and a block pointer.

The final three constructors are similar to the three above but additionally take `char *dt` as a second argument, to set up the data in a String type. The number of attributes *nattrs* is set up in each constructor.

4.1.13 The Operation Node

operation = operationName [attributes] arguments responses
block | "code" embedded

The operation node is another of the nodes which have many options, resulting in thirteen constructors, so these will not be detailed here. An operation node is represented as follows:

- *name*, the operation Name. Each constructor expects a name as an argument.
- *attributes*, a pointer to a list of attributes
- *arguments*, a pointer to a list of arguments
- *responses*, a pointer to a list of responses
- *form*, a pointer to a body node

4.1.14 The Signature Node

signature = operationName [attributes] arguments responses

The signature node is similar to the operation node, but does not take a block or code. It is represented as follows:

- *name*, the operation Name. Each constructor expects a name as an argument.
- *attributes*, a pointer to a list of attributes
- *arguments*, a pointer to a list of arguments
- *responses*, a pointer to a list of responses

4.1.15 The Terminate Node

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

The terminate node has a fixed format. It consists of the following:

- *name*, a pointer to an identifier node
- *block*, a pointer to a block node

It has a single constructor:

`TerminateNode( IdentifierNode *n, BlockNode *b )`.

4.1.16 The Termination Node

response = "->" [terminationName] "(" {declaration} ")"

The termination node consists of the following:

- *name*, a pointer to an identifier node. The name may be blank, but must be specified, to avoid confusion with a single declaration node below.
- *declarations*, a pointer to a list of declarations

There are three constructors, for cases of none, one or more declarations:

`TerminationNode( IdentifierNode *n )`, sets up a node with no declarations.

`TerminationNode( IdentifierNode *n, DeclarationNode *dn )` sets up a node with a single declaration.

`TerminationNode( IdentifierNode *n, DeclarationList *dec )` sets up a node with a list of declarations.

4.1.17 The TypeBlock Node

This is similar to a block node except that as a type block it is more restricted. It holds a pointer to a list of type expressions.

4.1.18 The TypeConstructor Node

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

The type constructor node takes optional attributes (none, one or a list), and one or a list of signatures. It has the following format:

- *attributes*, a pointer to a list of attributes
- *signatures*, a pointer to a list of signatures

There are six constructors for each of the various combinations of options.

4.1.19 The TypeDefinition Node

This is distinguished from the normal definition node to facilitate type inferencing, and consists of a declarations list and a pointer to a type expression node.

4.1.20 The TypeExpression Node

typeExpression = typeName | typeConstructor | typeBlock

The type expression node takes a type, indicating which of the possible expressions is represents, and a pointer to the node of the appropriate type.

5 The Generic View

This section describes the data and member functions provided as part of the generic view for each node. Initially these will mostly be stubs called by the specific view functions. When it becomes clear which aspects of the specific views are generic, they will become part of the generic view.

To date the only generic functions are described below.

5.1 The Identifier Node

Two general-purpose member functions are provided, for comparing an identifier with either a character string, or an object of type String. Each of these invokes the compare function on the String for the current identifier, and returns a 1 if a match is found, otherwise a 0. The two function definitions are:

```
int IdentifierNode::compare( const IdentifierNode *other )
int IdentifierNode::compare( const char *other )
```

6 Specific Views

This section describes the data and member functions for nodes of the AST for specific views.

6.1 Pretty Printer

Each node has a *display* function as one of its member functions. This function will normally display any syntactical parts of the current node, and invoke the *display* function on each of the sub-nodes. The *display* function therefore carries out its own iteration throughout the tree. An invocation of *display* at any level will display the specified node, and all sub-nodes of that node.

Each display function adds any syntax necessary for that node, e.g. the words “object” or “after....handle”, any parenthesis, indentation or operators as appropriate.

6.2 Scope

The scope checker accesses the AST by means of a scope function provided for each node.

6.3 Type inferencing

The type inferencer accesses the AST by means of a function, normally called *findType*, for each node. Some nodes may support several different methods for use by the type inferencer.

References

[ANSA 93]

DPL Programmers' Manual, TR.031.00, APM Ltd., Cambridge U.K., April 1993.