Methods for Testing and Specification (MTS); Abstract Test Description Language (ATDL)

Technical Report

Keywords:
ASN.1, methodology, MTS, testing, ATDL

Gong Yue

Samsung Electronics Co., Ltd.
Samsung China Research Center

SAMSUNG Electronics Research Institute
Communications House South Street
TW18 4QE City : STAINES
UNITED KINGDOM - GB

February 2006

**That's one small step for (a) man,
one giant leap for mankind.**

— Neil Armstrong

**If I have seen further it is by standing
upon the shoulders of Giants.**

— Sir Isaac Newton

1. Scope 1

2. References 1

3. Definitions and abbreviations 2

3.1. Definitions 2
3.2. Abbreviations 6

4. Introduction 8

4.1. General 8
4.1.1. The core language and presentation formats 9
4.1.2. Differences between TTCN-3 and ATDL 10
4.2. ATDL overview 11
4.2.1. ATDL views 11
4.2.2. Statement diagram 11
4.2.3. Implementation view 11
4.3. Static view 12
4.3.1. Relationships 12
4.4. ATDL grammars 13
4.5. Unanimity of the specification 13
4.6. Conformance 13
4.7. Comparison of ATDL, C++ and Java 14

5. Basic language elements 16

5.1. General 16
5.2. Parameterization 17
5.2.1. Static and dynamic parameterization 17
5.2.2. Formal and actual parameter lists 18
5.2.3. Empty formal parameter list 18
5.2.4. Nested parameter lists 18
5.3. Parameter semantics 18
5.3.1. In and inout parameters 19
5.3.2. Template parameters 20
5.3.3. Out parameters 20
5.3.4. String parameters 20
5.3.5. Array parameters 20
5.3.6. Open array parameters 20
5.4. Scope rules 22
5.5. Identifiers and keywords 22
5.6. Division of text 22
5.7. General drawing rules 23
5.7.1. Comments 23
5.7.2. Diagram area 24
5.7.3. Diagram heading 24
5.7.4. Usage of semicolons 24
5.7.5. Usage of task symbols 25
5.8. Variables declarations 25
5.8.1. Declaration of variables within <create request symbol>s 25
5.8.2. Declaration of variables within <default symbol>s 25
5.8.3. Declaration of variables within <reference symbol>s 25
5.9. Special terminal symbols 25
5.9.1. Separators 26
5.9.2. Operators 26

6. Abstract Object Definition Language 26

6.1. Conventions for the syntax description 27
6.2. AODL keywords 27
6.3. GORBA/AODL basics 27
6.4. Modules 28
6.5. Defining group types 28
6.6. Defining co-class types 29
6.6.1. Co-class type inheritance 29
6.6.2. Required interface types 30
6.6.3. Supported interface types 30
6.6.4. Co-class diagrams 30
6.7. Declaring exception types 31
6.8. Defining co-interface types 31
6.8.1. Co-interface type inheritance 31
6.8.2. Defining operational co-interface types 32
6.9. Importing from modules 32
6.10. Templates for sending messages 33
6.11. Summary 34
6.11.1. Benefits of AODL 34

7. Declaring ATDL/AODL signals 35

7.1. Declaring messages 35
7.2. Declaring operations 36
7.2.1. Procedure signatures 36
7.2.2. Operation attribute 37
7.2.3. parameter lists 37
7.2.4. Declaring parameters 37
7.2.5. Value returning remote procedures 37
7.2.6. Raises expressions 38

8. Declaring ATDL/AODL constants 38

8.1. Constant expressions 38

9. ATDL/AODL operators 39

9.1. Additive Operators 40
9.1.1. Unary arithmetic operators 41
9.2. String operators 41
9.3. Multiplicative operators 41
9.4. Relational operators 42
9.4.1. The class operator 43
9.5. Boolean logical operators 43
9.5.1. Conditional logical operator 44
9.6. Bitwise operators 44
9.7. Shift operators 45
9.8. Rotate operators 46
9.9. Primary expressions 46
9.10. Typecast expressions 46

10. ATDL/AODL types and values 47

10.1. Simple generic types 48
10.2. Basic types and values 48
10.2.1. Integral types and values 49
10.2.2. Character types and values 49
10.2.3. Real types and values 50
10.2.4. Boolean type and value 51

10.2.5. Objid type and values ... 51
10.2.6. Ordinal types ... 52
10.2.7. AODL specific native types ... 52
10.2.8. ATDL specific verdict types ... 52
10.2.9. Basic string types and values ... 52
10.3. Sub-typing of basic types ... 54
10.3.1. Value Set constructors ... 55
10.3.2. Length restriction ... 56
10.3.3. Subrange type ... 56
10.4. Structured types and values ... 57
10.4.1. Parameterized type ... 57
10.4.2. Sequence type and values ... 58
10.4.3. Choice type and values ... 59
10.4.4. Set type and values ... 59
10.4.5. Enumerated type and values ... 60
10.5. Array type and values ... 60
10.5.1. Dynamic arrays ... 61
10.5.2. Array constants ... 61
10.6. Sets of types ... 62
10.7. Variant types ... 62
10.8. Changes to ASN.1 ... 62
10.9. Miscellaneous productions ... 63
10.10. Pre-defined ATDL/AODL types ... 63
10.10.1. Useful simple basic types ... 63
10.10.2. Useful character string types ... 65

11. Modules ... 66

11.1. Module diagram ... 66
11.2. Naming of modules ... 66
11.3. Module parameters ... 66
11.3.1. Default values for module parameters ... 67
11.4. Module definitions part ... 67
11.5. Module control part ... 67
11.5.1. Termination of test cases ... 67
11.5.2. Controlling execution of test cases ... 68
11.5.3. Test case selection ... 68
11.5.4. Use of timers in control ... 68
11.5.5. Control diagram ... 68
11.6. Groups ... 69
11.6.1. Group members ... 70
11.6.2. Host support for groups ... 70
11.6.3. Unique group names ... 70
11.6.4. Declaring groups ... 71
11.6.5. Group diagram ... 71
11.7. Importing from modules ... 71
11.7.1. Rules on using import ... 72
11.7.2. Recursive import ... 72
11.7.3. Importing single definitions ... 72
11.7.4. Import on demand ... 72
11.7.5. Importing groups ... 72
11.7.6. Handling name clashes on import ... 73
11.7.7. Import definitions from non-ATDL modules ... 73

12. Test configurations ... 73

12.1. Test configurations at specification level ... 73
12.1.1. Defining association contracts ... 73

12.1.2. Abstract test system interface ... 74
12.1.3. Configuration diagrams ... 75

12.2. Test configurations at instance level ... 76

12.2.1. Channel communication model ... 76
12.2.2. Restrictions on connections ... 77

12.3. Defining interface types ... 77

12.3.1. Interface diagrams ... 77
12.3.2. The message-based interface types ... 78
12.3.3. Operational interfaces ... 79
12.3.4. Interface inheritance ... 80
12.3.5. Declaring exception types ... 80

13. Defining classes ... 81

13.1. Defining class types ... 81

13.1.1. Scope of a class type name ... 82
13.1.2. Passive object ... 83
13.1.3. Final classes ... 83
13.1.4. Class inheritances ... 83
13.1.5. Ancestor interfaces ... 84

13.2. Class members ... 84

13.3. Declaring properties ... 84

13.3.1. Signal handlers ... 85

13.4. Declaring fields ... 85

13.4.1. Static fields ... 86
13.4.2. Initialization of fields ... 86

13.5. Visibility of class members ... 86

13.6. Virtual classes ... 87

13.6.1. Method template ... 88
13.6.2. Incarnating ... 88
13.6.3. Method template instantiation ... 88

13.7. Declaring methods ... 88

13.7.1. Method implementations ... 89
13.7.2. Method binding ... 89
13.7.3. Inheritance, overriding, and hiding ... 91
13.7.4. Overloading methods ... 92
13.7.5. Destructors ... 93
13.7.6. Raises expressions ... 93

13.8. Declaring constructors ... 94

13.8.1. Constructor body ... 95
13.8.2. Constructor overloading ... 95
13.8.3. Default constructor ... 95
13.8.4. Raises expressions ... 95

13.9. Class references ... 95

13.10. Coordinating threads ... 95

13.10.1. Synchronized fields ... 96
13.10.2. Synchronized methods ... 96

13.11. Exceptions ... 97

13.11.1. Self-exceptions ... 97
13.11.2. Compile-Time Checking of Exceptions ... 97
13.11.3. Unchecked exceptions ... 97
13.11.4. The exceptions handling ... 98

14. Declaring variables ... 98

14.1. Kinds of variables ... 98

15. Declaring templates ... 99

15.1. Declaring message templates 100
15.1.1. Templates for receiving messages 100
15.2. Parameterization of templates 100
15.2.1. Parameterization with matching attributes 100
15.2.2. Templates reference 100
15.3. Template matching mechanisms 101
15.4. Modified templates 102
15.4.1. General 102
15.4.2. Parameterization of modified templates 102
15.4.3. In-line modified templates 102
15.5. Changing template fields 103
15.6. Value of Operation 103
15.7. Matching incoming values 103
15.7.1. In-line matching operators 103
15.7.2. Matching specific values 104
15.7.3. Constructed value 105
15.7.4. Instead of Value 106
15.7.5. Inside Values 107
15.7.6. Attributes of values 107
15.7.7. Matching character pattern 108

16. Routines and method templates 108

16.1. Functions 108
16.2. Test cases 109
16.2.1. Test case diagram 110
16.2.2. Parameterization of test cases 110
16.3. Overloading test cases and functions 111
16.4. Altsteps 111
16.4.1. Parameterization of altsteps 111
16.4.2. Altstep diagram 112
16.4.3. Invocation of altsteps 112
16.5. Method templates 113
16.5.1. Method template definition 113
16.5.2. Method template explicit incarnation 114
16.5.3. Name resolution in method templates 114

17. Overview of program statements and operations 116

17.1. Statement block 117
17.1.1. Statement diagrams 118
17.1.2. Statements 118
17.1.3. Unreachable Statements 118
17.2. Kinds of conversion 118
17.2.1. Identity conversions 118
17.2.2. Widening primitive conversions 119
17.2.3. Narrowing primitive conversions 119
17.2.4. Widening reference conversions 119
17.2.5. Narrowing reference conversions 120
17.2.6. Charstring conversions 120
17.2.7. Forbidden Conversions 120
17.3. Assignment conversion 120
17.4. Method invocation conversion 121
17.5. Casting conversion 121
17.6. Type compatibility and identity 121
17.6.1. Type identity 121
17.6.2. Type compatibility 121

18. Basic program statements 123
18.1. Local variable declaration statements 123
18.2. The task statements 123
18.2.1. The Write statement 124
18.2.2. External actions 124
18.2.3. Expression statements 124
18.3. The If-else statement 124
18.3.1. The if statement with else branch 125
18.3.2. Control icons 126
18.4. The Choice statement 126
18.5. The in-line expressions 127
18.5.1. The labeled statement 127
18.5.2. The if statement without else branch 128
18.5.3. The For statement 128
18.5.4. The While statement 129
18.5.5. The Do-while statement 130
18.6. The Break statement 131
18.7. The Continue statement 132
18.8. The Stop execution statement 132
19. Behavioural program statements 133
19.1. Alternative behaviour 133
19.1.1. Graphical notation 134
19.1.2. Execution of alternative behaviour 135
19.1.3. Selecting/deselecting an alternative 135
19.1.4. Guard condition 136
19.1.5. Else branch in alternatives 136
19.1.6. ATDL test events 136
19.1.7. Re-evaluation of alt statements 137
19.1.8. Invocation of altsteps as alternatives 137
19.2. The Continue statement 137
19.3. The Return statement 137
19.4. The Raise statement 138
19.4.1. Raise a self-exception 139
19.4.2. Re-raising exceptions 139
19.5. Exception handling 139
19.5.1. The Try statement 139
19.5.2. The Catch clause 140
19.5.3. Catch a remote-exception 141
19.5.4. The Timeout exception 141
19.5.5. The catch all handler 141
19.5.6. The catch any clause 142
19.6. Test verdict operations 142
19.6.1. Test case verdict 143
19.6.2. Verdict values and overwriting rules 143
19.7. Default Handling 144
19.7.1. The default mechanism 144
19.7.2. Default references 145
19.7.3. The activate operation 145
19.7.4. The deactivate operation 145
20. Expressions 146
20.1. Boolean expressions 146
20.1.1. Conditional ? operator 146
20.2. Primary expressions 147

20.2.1. Self ... 147
20.2.2. Parenthesized expressions ... 147
20.3. Typecast expressions ... 147
20.3.1. Value typecasts ... 147
20.3.2. Variable typecasts ... 147
20.4. Component instance creation expressions ... 148
20.4.1. Initializing the test component ... 148
20.4.2. Component instance ... 149
20.5. Field access expressions ... 149
20.5.1. Field access using an object reference ... 150
20.5.2. Accessing inherited members ... 150
20.6. Method invocation expressions ... 150
20.6.1. Invocation of functions ... 151
20.6.2. Execution of test cases ... 151
20.6.3. Determining the method ... 152
20.6.4. Choose the most specific method ... 153
20.7. References for data objects ... 153
20.7.1. Array references ... 153
20.7.2. Record references ... 153
20.7.3. String references ... 154
20.8. Assignments ... 154
20.8.1. Assignment rules for array types ... 155
20.8.2. Assignment rules for string types ... 156
21. Object-based programming ... 156
21.1. Class templates ... 156
21.1.1. Class template definition ... 158
21.1.2. Class template instantiation ... 158
21.1.3. Template arguments for non-generic type parameters ... 160
21.1.4. Member methods of class templates ... 160
21.1.5. Static members of class templates ... 160
21.1.6. Class template incarnations ... 161
21.1.7. Class template partial incarnations ... 163
21.1.8. Name resolution in class templates ... 163
21.1.9. Groups and class templates ... 163
21.2. Threads and operations ... 163
21.2.1. Defining thread classes ... 164
21.2.2. The Priority field (informative) ... 166
21.2.3. The Running operation ... 167
21.2.4. The Start thread method ... 167
21.2.5. The Stop thread method ... 168
21.2.6. The Done operation ... 169
21.2.7. The MTC, System , Sender and Self operations ... 169
22. Communication operations ... 169
22.1. Connection Points ... 170
22.1.1. Simple connectable object ... 170
22.2. Interface references ... 171
22.2.1. Interface typecast ... 171
22.3. General format of communication operations ... 171
22.3.1. General format of the sending operations ... 171
22.3.2. General format of the receiving operations ... 172
22.4. Message-based communication ... 172
22.4.1. The Send operation ... 173
22.4.2. The Receive operation ... 173
22.4.3. The Trigger operation ... 174

22.5. Operation templates **175**
22.5.1. Templates for invoking procedures 175
22.5.2. Templates for accepting operation invocations 176
22.5.3. In-line assignments for invoking operations 176
22.6. Procedure-based communication **177**
22.6.1. The Call operation 177
22.6.2. Determining the method 179
22.6.3. The Synchronize operation 179
22.7. Interceptors **180**
22.8. Channel controlling operations **181**
22.8.1. The Bind method 181
22.8.2. The Release method 182
22.8.3. The Clear channel operation 182
22.8.4. The Start channel operation 183
22.8.5. The Stop channel operation 183
22.8.6. Use of any and all with channels 183

23. Timers and operations **183**

23.1. Timers as parameters **184**
23.2. Timer class methods **184**
23.2.1. The Start timer operation 185
23.2.2. The Stop timer method 186
23.2.3. The Read timer method 186
23.2.4. The Running timer operation 187
23.2.5. The Timeout operation 187
23.2.6. Summary of use of any and all with timers 187

24. Specifying attributes **188**

24.1. Display attributes **189**
24.2. Encoding of values **189**
24.2.1. Encode attributes 189
24.2.2. Variant attributes 189
24.2.3. Special strings 189
24.2.4. Invalid encodings 190
24.3. Extension attributes **190**
24.4. Scope of attributes **190**
24.5. Overwriting rules for attributes **190**
24.6. Changing attributes of imported language elements **190**

25. The System module **191**

25.1. The Group System.lang **191**
25.1.1. The Class TNumber 191
25.1.2. The Class TInteger 191
25.1.3. The Class TFloat 194
25.1.4. The Class TDouble 195
25.1.5. The Class TBitString 196
25.1.6. The Class TBitStringBuffer 197
25.1.7. The Class TOctetString 198
25.1.8. The Class TOctetStringBuffer 198
25.1.9. The Class THexString 199
25.1.10. The Class THexStringBuffer 200
25.1.11. The Class TCharString 201
25.1.12. The Class TCharStringBuffer 201
25.2. The Group System.io **202**
25.2.1. The Class DataInputStream 202
25.3. Predefined functions **203**

25.3.1. Number of elements in a structured type ... 203
25.3.2. The IsPresent function ... 203
25.3.3. The IsChosen function ... 203
25.3.4. The LowerBoundary function ... 203
25.3.5. The UpperBoundary function ... 203

26. ATDL BNF and static semantics ... 204

26.1. ATDL grammars ... 204
26.1.1. ATDL terminals ... 204
26.1.2. Meta-language for graphical grammar ... 205
26.1.3. Static and dynamic objects ... 205

26.2. ATDL syntax BNF productions ... 206
26.2.1. ATDL Module ... 206
26.2.2. Module Definitions Part ... 207
26.2.3. Control Part ... 214
26.2.4. Type ... 217
26.2.5. Value ... 217
26.2.6. Parameterisation ... 219
26.2.7. With Statement ... 219
26.2.8. Statement Blocks ... 219
26.2.9. Behavior Statements ... 220
26.2.10. Basic Statements ... 221
26.2.11. Miscellaneous productions ... 224

27. An INRES example ... 225

1. Scope

The present document defines the Core Language of ATDL. ATDL can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of CORBA based platforms, APIs etc. ATDL is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

While the design of ATDL has taken the eventual implementation of ATDL translators and compilers into consideration the means of realization of executable test suites (ETS) from abstract test suites (ATS) is outside the scope of the present document.

2. References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] Stroustrup, Bjarne. *The C++ Progamming Language,* 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

[3] ISO/IEC 9646-1 : *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 1: General concepts*, 1994.

(See also ITU-T Recommendation X.290 : 1995)

[4] ISO/IEC 9646-2 : *Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 2: Abstract test suite specification*, 1994. (See also ITU-T Recommendation X.291 : 1995)

[5] ISO/IEC 9646-3 (1998): "Information technology - Open Systems Interconnection - Conformance testing methodology and framework - Part 3: The Tree and Tabular Combined Notation (TTCN)".

[6] ISO/IEC 646, *Information technology - ISO 7-bit coded character set for information interchange*, 1991.

[7] ISO/IEC 10646-1, *Information technology - Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane*, 1993.

[8] ITU-T Recommendation X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".

[9] ITU-T Recommendation X.681: " *Information Technology - Abstract Syntax Notation One (ASN.1) - Part 2: Information Object Specification*", 1994.

[10] ITU-T Recommendation X.682: "Information technology - Abstract Syntax Notation One (ASN.1): Constraint specification".

[11] ITU-T Recommendation X.683: "Information technology - Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications".

[12] ISO/IEC 8859-1: "Information technology - 8-bit single-byte coded graphic character sets - Part 1: Latin alphabet No. 1".

[13] ITU-T (CCITT) Recommendation X.722, *Guidelines for the definition of Managed Objects*, January 1992. (Also ISO/IEC 10165-4.)

[14] ITU-T Recommendation X.690: "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)".

[15] [MSC96] ITU-T Recommendation Z.120: *Message Sequence Chart (MSC)*. ITU-TS, Geneva, October 1996.

[16] ITU-T Recommendation X.660 (1992): "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: General procedures".

[17] [SDL92] ITU Rec. Z.100, *Specification and Description Language SD*L, ITU-T (03/1993).

[18] ITU Recommendation Z.100, *Specification and Description Language SDL*, ITU-TS, Geneva, November 1999.

[19] ISO/IEC 6429 (1992): "Information technology - Control functions for coded character sets".

[20] [TRI] ETSI TR 102 043: *Methods for Testing and Specification (MTS); The TTCN-3 Runtime Interface(TRI); Concepts and definition of the TRI*, April 2002.

[21] [OSI] ISO/IEC 7498-1 : *Information technology - Open Systems Interconnection - Basic Reference Model - Part 1: The Basic Model*, 1995. (Also ITU-T Rec. X.200 : 1994.)

[22] [ROSE] ITU-T Recommendation X.880, I*nformation technology — Remote Operations: Concepts, model and notation*, 1994. (See also ISO/IEC 13712-1:1995)

[23] [ODP-3] ITU-T Recommendation X.903, *Basic Reference Model of Open Distributed Processing, 'Part3: Architecture'*, February 1995.

(See also ISO/IEC 10746-3)

[24] [ODL] Object Management Group, *Draft of ITU Object Definition Language*, (obtained via ftp://ftp.omg.org/pub/docs/telecom/98-02-10).

[25] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Object Management Group, July 1995.

[26] [COMP] Object Management Group, *CORBA Components*, Version 3.0, June 2002.

[27] IEEE Standard for Binary Floating-Point Arithmetic, ANIS/IEEE Std 754-1985.

[28] [INRES] Dieter Hogrefe, *OSI formal specification case study: the INRES protocol and service*, IAM-91-012, 1991.

[29] ITU-T (CCITT) Recommendation X.722, *Guidelines for the definition of Managed Objects*, January 1992. (Also ISO/IEC 10165-4.)

[30] [UML] Object Management Group, *OMG Unified Modeling Language Specification 1.3 (draft)*, OMG Document Number: ad/99-02-01, January 1999.

3. Definitions and abbreviations

3.1. Definitions

For the purposes of the present document, the terms and definitions given in ISO/IEC 9646-1 [3] and ISO/IEC 9646-3 [4] and the following apply:

actual parameter: A specific value corresponding to a formal parameter.

argument: see actual parameter.

association contract: A description of a connection among instances of components.

Boolean expression: An enumeration whose values are **true** and **false**.

branch: An element in a statement diagram or an ETSC diagram in which a single node leads to more than one possible outgoing path, each with its own guard condition.

BTSC diagram: A diagram that shows object interactions arranged in time sequence.

call: To invoke an operation.

channel: A tuple of interface object references that is an instance of an association contract or an exception.

class: The descriptor for a set of objects that share the same fields, methods, contracts, and behavior.

class member: The fields and methods of a class are called its *member*s.

class method: A class method is a method that operates on classes instead of objects.

co-class: An abstract or physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of co-interfaces.

co-class diagram: A diagram that shows the organizations and dependencies among co-class types.

co-class instance: A co-class instance is an instance that originates from a co-class.

co-interface: A point within a testing environment where the occurrence of test events is to be controlled and observed, as defined in an Abstract Test Method.

communication operation: Operations such as **send** and **call** are collectively known as communication operations. These operations shall only be used in ATDL test cases and functions.

compatible type: ATDL is strongly typed, and the language requires type compatibility. Variables, constants, constraints etc. have compatible types if they resolve to the same base type and, in the case of assignments, matching etc., no sub-typing (e.g., ranges, length restrictions) is violated.

component: Threads, passive objects and co-objects are referred to collectively as components.

component type: A model element that describes behavioral and structural features. Kinds of components include class, thread class and co-class. Classes are the most general kind of component. Most properties of classes apply to components, in general, with certain restrictions for each kind of component.

component instance diagram: A component instance diagram is a graph of instance symbols representing component instance and channels representing contract instance.

component reference: Thread object references, object references and co-object references are referred to collectively as *component references.*

configuration diagram: A configuration diagram presents either a test configuration at specification level, which contains a set of inter-connected test components, as well as their required relationships given in a particular context, or it presents a test configuration at instance level with a collection of component instances and their relationships.

constructor: A component-scope method that creates and initializes an instance of a component.

control icons: Optional symbols that provides convenient shortcut notation for various control patterns.

Coordination Signal (CS): An item of structured information which may be transferred from one Test Component to another at a cp-interface.

cp-interface: An interface within a testing environment, assigned to two Test Components in a Test Component Configuration, where CSs may be exchanged synchronously or asynchronously between these Test Components.

creation: The instantiation and initialization of an object or other instance.

deployment diagram: A diagram that shows the configuration of run-time groups and the co-class instances and objects that live on them.

dependency: A relationship between two components in which one component (the client) requires the presence of another component (the server) for its correct functioning or implementation.

destructor: A destructor is a special method that destroys the object where it is called and de-allocates its memory.

ETSC diagram: A diagram that shows an ETSC graph.

event: The specification of a noteworthy occurrence that has a location in time and space.

exception: A message raised in response to behavioral faults by the underlying execution machinery.

expression: A string that encodes a statement to be interpreted by a given language.

formal parameter: The specification of a variable that can be changed, passed, or returned. A parameter may include a name, type, and direction.

function: Functions are methods which may return a value.

group: A general-purpose mechanism for organizing elements into groups. Groups may be nested within other groups.

group instance: A group instance is an instance of a component group.

guard condition: A condition that must be satisfied in order to enable an associated guarded statement to fire.

guarded statement: A guarded statement will perform specified statements when an optional specified event occurs and specified guard conditions are satisfied.

implementation: A definition of how something is constructed or computed. For example, a thread class or class is an implementation of a co-class; a method is an implementation of an operation.

import: A stereotype of the permission dependency in which the names of the elements in the supplier module are added to the namespace of the client module.

inheritance: The mechanism by which more specific elements incorporate structure and behavior defined by more general elements.

in-line reference: an inline expression whose constituent is a single block or TSC reference.

interface: A place within a testing environment, assigned to two Test Components in a Test Component Configuration, where signals may be exchanged synchronously or asynchronously between these Test Components.

interface typecast: An interface typecast dynamically queries a given object and returns an interface reference to the object.

join: A place in a statement diagram, or ETSC diagram at which two or more statement blocks or TSC references combine to yield one statement block or TSC reference.

length restriction: Where a length restriction is used, the set of values for a type defined by this restriction shall be a true subset of the value defined by the base type.

location: The physical placement of a run-time entity, such as an object or a co-class instance, within a distributed environment. In ATDL, location is discrete and the units of location are groups.

lock: The built-in mechanism of a thread that prevents the execution of other threads from using that object instance.

main thread component [MTC]: The single Test Thread in a Test Component Configuration responsible for creating and controlling Parallel Test Threads and computing and assigning the test verdict.

merge: A place in a state diagram, or ETSC diagram where two or more alternate control paths come together. Antonym: branch.

message: The specification of an asynchronous communication between objects.

message-based communication: A message stimulus is an asynchronous communication between two instances that conveys information with the expectation that action will ensue.

message-based interface: An interface serves to name a collection of messages.

message event: An event that is the receipt by an object of a message sent to it.

method: Test cases and functions are referred to collectively as *methods.*

module: A self-contained collection of ATDL objects. All referenced objects are either explicitly defined in the Module, are imported from other sources or are defined as external objects in the module.

object: A discrete entity with a well-defined boundary and identity that encapsulates state and behavior; an instance of a class.

operation: An operation is a specification of a transformation or query that an object may be called to execute.

operation attribute: The operation attribute specifies which invocation semantics the communication service must provide for invocations of a particular operation.

operational co-interface: A named set of operations that characterize the behavior of a co-class.

operational cp-interface: An *operational cp-interface* defines methods that can be implemented by a class or thread.

ordinal type: An ordinal type defines an ordered set of values in which each value except the first has a unique *predecessor* and each value except the last has a unique *successo*r.

parallel thread component [PTC]: Test thread object created by the main thread component.

procedure: Operations, testcases and functions are referred to collectively as *procedures.*

procedure template: Method templates and generic operations are referred to collectively as *procedure templates.*

realization: The relationship between a specification and its implementation.

receive: To handle a message instance passed from a sender object.

record type: The sequence, set and choice types are collectively referred to as record types.

reference: A denotation of a model element, commonly called a pointer. Reference is a relationship between elements at the same level of detail or between elements in different containers. For a reference to be possible, the component performing the reference must have visibility to the component being referenced.

root type: basic type, structured type, special data type, special configuration type or special default type to which the user-defined ATDL type can be traced back.

routine: Test cases, altsteps and functions are referred to collectively as *routines.*

self-exception: A self-exception can carry an error message, from an object to itself. It transfers control to the innermost *exception handler* that can handle exceptions of the given component.

send: To create a message instance by a sender object and to transfer it to a receiver object in order to convey information.

signal: The conveyance of information from one object (or other instance) to another. A signal may be a message or the call of an operation.

signature: The name and formal parameter properties of a signal, such as an operation or message. A signature may include optional return types (for operations, not for messages).

snapshot: A collection of objects, channels, and values that forms the configuration of a system at an instance during its execution. ATDL operational semantics assume that the status of any of the events cannot change during the process of trying to match one of a set of alternatives.

specification: A declarative description of what something is or does. For example, a thread class or a co-interface is a specification. Contrast: implementation.

statement block: A statement block is a sequence of statements and local declaration statements within braces.

statement diagram: A diagram that shows a statement graph. Contrast: statement block.

static field: Class variables and constant fields are referred to collectively as static fields.

strong typing: strict enforcement of type compatibility by type name equivalence with no exceptions.

synchronize event: The event of receiving a call for an operation that is implemented by the object in which the operation is called.

task: An executable atomic computation that results in a change in the state of the model or the return of a value.

template: ATDL templates are specific data structures for testing; used to either transmit a set of distinct values or to check whether a set of received values matches the template specification.

testcase: See ISO/IEC 9646-1 [3].

thread class: A thread class is a class whose instances are thread objects.

thread-local field: A thread-local variable is also called a thread-local field.

thread object: An object that owns a thread of control and can initiate control activity; an instance of a thread class.

timeout event: An event that denotes the passage of a given amount of time.

type: A type declaration specifies an identifier that denotes a type.

view: A projection of a model, which is seen from on perspective or vantage point and omits entities that are not relevant to this perspective.

virtuality: As with fields, methods in ATDL may be decorated with visibility modifiers and virtuality modifiers. The virtuality modifiers are *static*, *virtual*, and *abstract*.

visibility: An enumeration whose value (public, protected, or private) denotes whether the model element to which if refers may be seen outside its enclosing namespace.

3.2. Abbreviations

For the purposes of the present document the following abbreviations apply:

ANSI	American National Standards Institute
AODL	Abstract Object Definition Language
API	Application Programming Interface
ARI	ATDL Runtime Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
ASP	Abstract Service Primitive
ATDL	Abstract Test Description Language
ATDL/gr	ATDL Graphic Representation
ATS	Abstract Test Suite
ATSC	Abstract Test Sequence Chart
BER	Basic Encoding Rules
BTSC	Basic Test Sequence Chart
BNF	Backus-Nauer Form
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CP	Coordination Point
CPU	Central Processing Unit
EBNF	Extended Backus-Nauer Form
ETSC	Extended Test Sequence Chart
ETSI	European Telecommunications Standard Institute
FIFO	First In First Out
GDMO	Guidelines for the Definition of Managed Objects
GORBA	Generic Object Request Broker Architecture
IDL	Interface Definition Language
ISO	International Standardization Organization
ITU	International Telecommunications Union
IUT	Implementation Under Test
MSC	Message Sequence Chart
MTC	Master Thread Component
NaN	Not-a-Number
ODL	Object Definition Language
ODP	Open Distributed Processing
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Interconnection
PDU	Protocol Data Unit
(P)ICS	(Protocol) Implementation Conformance Statement
(P)IXIT	(Protocol) Implementation eXtra Information for Testing
PTC	Parallel Thread Component
ROS	Remote Operation Service
SDL	Specification and Description Language
SUT	System Under Test
TRI	TTCN-3 Runtime Interface
TSC	Test Sequence Chart
TSO	Test Suite Operation
TTCN	Tree and Tabular Combined Notation
TTCN-3	The Testing and Test Control Notation version 3
UML	Unified Modeling Language

4. Introduction

The compact test notation, *Abstract Test Description Language* (ATDL), defines a framework and methodology for conformance testing [3] of implementations of OSI and ITU protocols. ATDL is a general-purpose, concurrent, component-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. ATDL is strongly related to TTCN [1, 5] but is organized rather differently, with a number of aspects of TTCN omitted and a few ideas from other languages included.

ATDL is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Its benefits include easy-to-read code, quick compilation, and the use of multiple module files for modular programming.

The ATDL is a compact notation being developed for the specification of test cases at a level that is abstracted from the architecture of any real test system that these test cases may eventually be run on. This thesis specifies requirements on what a test suite standard may specify about a conforming realization of the test suite, including the operational semantics of ATDL test suites.

4.1. General

TTCN was designed from the beginning with OSI-based protocol conformance testing in mind and even TTCN-2[5] was not adequate for various kinds of testing. Therefore a more flexible and powerful test description language was called for. In October 2000, TTCN-3 [1] was approved in ETSI. TTCN-3 is a significant improvement over TTCN-2 in precision, expressiveness and capability to meet emerging testing needs.

The test features provided with ATDL allow you to submit your application to a robust test campaign. Each feature uses a different approach to the software testing problem, from the use of test drivers stimulating the code under test, to source code instrumentation testing internal behavior from inside the running application.

ATDL retained the proven features of TTCN-3 but was designed to provide the new features listed above. ATDL is designed to bring object-oriented programming techniques to software testing and is applicable to both object-oriented and procedural source code.

ATDL is a general-purpose, concurrent, class-based and object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. ATDL is strongly related to TTCN but is organized rather differently, with a number of aspects of TTCN omitted and a few ideas from other languages included. It allows the use of different graphical presentation (display) formats. Apart from the tabular (conformance testing) presentation format known from TTCN-2, the development of an SDL-like (Specification and Description Language) graphical presentation format appears to be of special interest and therefore is part of our work.

ATDL is a flexible, powerful and object-oriented language applicable to the specification of all types of reactive system tests over a variety of communication interfaces. ATDL retained the proven features of TTCN-3 but was designed to provide many new features. ATDL is on syntactical (and methodological) level a drastic change to TTCN-3, however, the main concepts of TTCN-3 have been retained and improved and new concepts have been included, so that ATDL will be applicable for a broader class of systems. Improved concepts are, e.g., the integration of ASN.1 [8], replacement of the test component construct with the thread class construct, and replacement of abstract test system interface and address type concepts of TTCN-3 with multiple co-interface co-class type concepts.

The major contribution of this thesis is the design of a test description programming language with explicit support for interactions with an external environment. AODL allows you to create and use interfaces and co-interfaces in your application. Interfaces are a way extending the single-inheritance model of the ATDL by allowing a single class to implement more than one interface, and by allowing several classes descended from different bases to share the same interface. Co-interfaces are useful when sets of operations, such as streaming, are used across a

broad range of objects. Co-interfaces are also a fundamental aspect of the COM (Component Object Model) and CORBA [25] distributed object models.

Compared to TTCN-3, the ATDL defined herein has been extended in the areas of object-oriented data, harmonization of a number of features to make the language simpler. The overall design goal of simplicity through generality has been achieved for the extended language, and the conformance of the textual core language with the principles of abstraction, correspondence and data type completeness has been preserved for the extended language.

ATDL has brought together many of the divergent, yet similar programming languages in the ITU domain [1, 8, 15, 17, 29] and CORBA [25] into a single model. The goals of the unification efforts were to keep it simple, to cast away elements of existing ASN1, SDL, TTCN and MSC that didn't work in practice, to add elements from other methods that were more effective, and to invent new only when an existing solution was not available.

It should be made clear that ATDL not a radical departure from MSC, TTCN, ODL, or SDL, but rather the legitimate successor to all of them. ATDL is more expressive yet cleaner and more uniform than MSC, SDL, and TTCN. This means that there is value in moving to ATDL, because it will allow projects to model things they could not have done before. Users of most other methods and modeling languages will gain value by moving to the ATDL, since it removes the unnecessary differences in notation and terminology that obscure the underlying similarities of most of these approaches. ATDL has then laid the foundations for a new version of TTCN-3 by drawing together a range of ideas under one umbrella. It finished by presenting the concrete textual grammar of two new proposed language features: AMSC and AODL.

The basic notion is relatively simple. ATDL is, in some sense, a combination of an Interface Definition Language and a "regular" programming language. The OMG has adopted a series of technical services (CORBA and CORBA Services) and a way to exchange designs between analysis and design tools (UML). What is lacking is the application architecture, technology and specification mechanism that specializes these designs and services for distributed component testing applications. The Generic ORB Architecture (GORBA) and ATDL provide this layer.

4.1.1. The core language and presentation formats

A main advantage of the graphical representation is its clear graphical layout, which immediately gives an intuitive understanding of the described behavior. Using graphical representation format may considerably improve the readability of test cases and make them more understandable. ATDL has succeeded in further broadening the horizons of the TTCN-3 model. ATDL is an evolution from SDL, MSC, UML, and many other sources. The ATDL graphical notation is a melding of graphical syntax from various sources, with a number of symbols removed and with a few new symbols added. This thesis has succeeded in bringing UML ideas to ATDL and in making these ideas more efficiently implement-able under Generic ORB Architecture.

There are several new concepts that are included in ATDL, including

1) statement diagrams

2) EMSC diagrams

3) configuration diagrams at specification level and instance level, and

4) interfaces, co-interfaces and co-classes.

Many of these ideas were present in various individual methods and theories but ATDL brings them together into a coherent whole.

In addition to the pure textual format, ATDL will define at least three graphical presentation formats: a tabular conformance testing presentation format that resembles the tabular form of TTCN-2, a graphical presentation format that resembles the graphical form of SDL [SDL92], and an MSC presentation format that supports the presentation but also development of ATDL test cases on MSC level [MSC96]. All of them can be mappable readily to and from the ATDL textual

grammar for the corresponding concepts. In particular, the graphical form of ATDL and the textual form of ATDL are equivalent at semantic level.

4.1.2. Differences between TTCN-3 and ATDL

The advantages of language stability began to be outweighed by the need to update ATDL to support and better match other languages that are frequently used in combination with ATDL. Also modern tools and techniques have made it practical to generate software more directly from ATDL specifications, but further significant gains could be acquired by incorporating better support for this use in ATDL. While ATDL is largely an upgrade of TTCN-3, it was decided that some incompatibility with TTCN-3 was justified; otherwise the resulting language would have been too large, too complex and too inconsistent. This sub-clause provides information about the changes.

Changes have been made in a number of areas, which focus on simplification of the language, and adjustment to new application areas:

a) Adjustment of syntactical conventions to other languages with which ATDL is used;
b) Harmonization of the concepts of thread, class and co-class to be based on “component”;
c) Interface descriptions that conform to the CORBA object model and the UML meta-model;
d) Direct containment of groups and components in groups;
e) Replacement of the test component construct with the thread class construct;
f) New model for data;
g) Constructs to support the use of ASN.1 with ATDL previously in TTCN-2.

Compared to TTCN-3 as defined in [1], the ATDL defined herein has been extended in the areas of object-oriented data, harmonization of a number of features to make the language simpler, and features to enhance the usability of ATDL with other languages such as ASN.1, CORBA, ITU-T ODL (Z.130) and UML. Other minor modifications have been included. Though care has been taken not to invalidate existing TTCN-3 documents; some changes may require some descriptions to be updated to use the ATDL.

The major extensions were in the area of object orientation. While TTCN-3 is component based in its underlying model, some language constructs had been added to allow ATDL to more completely and uniformly support the object paradigm:

a) Thread class, class and co-class types;
b) Thread class, class, co-class and group instances based on types;
c) Parameterization of types by means of context parameters of a constructor;
e) Specialization of types, and redefinition of virtual functions and test cases.

ATDL supports three kinds of class types. Co-class types are supported for backward compatibility with ITU-ODL. All ITU-ODL object templates are ATDL co-classes. Class types are supported for backward compatibility with object-orientated programming languages. All Java classes are ATDL classes. Thread class types are supported for backward compatibility with TTCN. All TTCN test components can be easily mapped to ATDL thread classes.

Unlike Java, which was designed to restrain programmers into making "correct" object-oriented design choices, ATDL can adapt to any paradigm.

Some of the definitions of TTCN-3 were extended considerably, e.g. **port** definition. It should be noted that the extensions were used for utilizing the power introduced by the object oriented extensions, e.g. to use implementation and inheritance for interfaces and/or co-interfaces.

On the syntactic level, ATDL is case-sensitive. Keywords of ATDL that are not keywords of TTCN-3 are:

bind, break, cardinal, choice, class, co, constructor, continue, destructor, final, implements, inherited, interface, members, operation, overload, private, protected, public, raises, real, release, requires, sequence, supports, synchronize, synchronized, thread, try, uses, virtual, wide.

The following keywords of TTCN-3 are not keywords in ATDL:

address, and4b, check, component, connect, disconnect, execute, getcall, getreply, goto, interleave, length, map, match, message, mixed, not4b, nowait, on, or4b, param, port, procedure, record, repeat, reply, runs, signature, subset, superset, to, union, universal, unmap, valueof, xor4b.

A small number of constructs of TTCN-3 are not available in ATDL: port type definitions, component type definitions, test system interface definitions, representation of configuration. These constructs were not compatible with the CORBA object model and the UML meta-model, and the overhead of keeping them in the language and tools did not justify their retention.

4.2. ATDL overview

This chapter presents a brief walkthrough of ATDL concepts and diagrams using a simple example. The purpose of the chapter is to organize the high-level ATDL concepts into a small set of views and diagrams that present the concepts visually. It shows how the various concepts are used to describe a system and how the views fit together.

4.2.1. ATDL views

There is no sharp line between the various concepts and constructs in ATDL, but, for convenience, we divide them into several views. A view is simply a subset of ATDL modeling constructs that represents one aspect of a system.

Structural classification describes the things in the system and their relationships to other things.

Table 1: ATDL Views and Diagrams

Major Area	Views and Diagrams		Main Concepts
structural	static view: configuration diagram at specification level		component, cp-interface, co-interface, inheritance, dependency, realization
	implementation view: co-class diagram		co-class, co-interface, dependency, realization
	instance and deployment view: configuration diagram at instance level		group instance, component instance, channel, dependency, location
dynamic	activity view	statement diagram	alternative, join, guarded statement, event
		ETSC diagram	control icons, in-line reference
	interaction view	BTSC diagram	task, instance, signal, activation

Dynamic behavior describes the behavior of a system over time. Behavior can be described as a series of changes to snapshots of the system drawn from the static view. Dynamic behavior views include the activity view, ETSC view and basic interaction view.

4.2.2. Statement diagram

The operational semantics of ATDL is based on the interpretation of flow graphs. In this report flow graphs are introduced (see statement diagram), the construction of flow graphs representing ATDL module control, test cases, and functions is described, and the construction of flow graphs representing the handling of messages, remote procedure calls, replies to remote procedure calls and exceptions is described.

4.2.3. Implementation view

The implementation view models the co-classes in a system i.e., the software units from which the application is constructed. The implementation is displayed on co-class diagrams. There are

three user interfaces: one each for customers using a kiosk, clerks using the on-line reservation system, and supervisors making queries about ticket sales. The co-class diagram shows the kinds of co-classes in the system; a particular configuration of the application may have more than one copy of a co-class.

A small circle with a name is a co-interface. A solid line from a co-class to a co-interface indicates that the co-class provides the services listed in the co-interface. A dashed arrow from a co-class to a co-interface indicates that the co-class requires the services provided by the co-interface. For example, subscription sales and group sales are both provided by the ticket seller co-class; subscription sales are accessible from both kiosks and clerks, but group sales are only accessible from a clerk.

4.3. Static view

The static view is the foundation of ATDL. The elements of the static view of a model are the concepts that are meaningful in an application, including all kinds of concepts found in systems.

4.3.1. Relationships

Relationships among components are association contract, inheritance, and various kinds of dependency, including realization and usage (see Table 2).

The contract relationship describes semantics connections among individual objects of given components. Contracts provide the connections with which instances of different components can interact. The remaining relationships relate the descriptions of components themselves.

The inheritance relationship relates general descriptions of parent components (super-classes) to more specialized child components (subclasses). Generalization facilitates the description of components out of incremental declaration pieces, each of which adds to the description inherited from its ancestors. The inheritance mechanism constructs complete descriptions of components from incremental descriptions using generalization relationships.

Table 2: Kinds of Relationships

Relationship	Function	Graphical notation
association contract	A description of a connection among instances of components	
dependency	A relationship between two components, A situation in which one component requires another for its correct functioning	
inheritance	A relationship between a more general description and a more specific variety of the general thing, used for inheritance	
realization	Relationship between a specification and its implementation	

The realization relationship relates a specification to an implementation. The realization relationship connects a component, such as a class, to another model element, such as an interface, that supplies its behavioral specification but not its structure or implementation.

A cp-interface or a co-interface is a specification of behavior without implementation; a class includes implementation structure. One or more classes may realize an interface, and each class implements the operations found in the interface or co-interface.

4.4. ATDL grammars

ATDL gives a choice of two different syntactic forms to use when representing a system: a Graphic Representation (ATDL/gr), and a textual Phrase Representation (ATDL/pr). As both are concrete representations of the same ATDL semantics, they are equivalent at semantic level.

A subset of ATDL/pr is common with ATDL/gr. This subset is called common textual grammar.

Figure 1 shows the relationships between ATDL/pr, ATDL/gr and the concrete grammars.

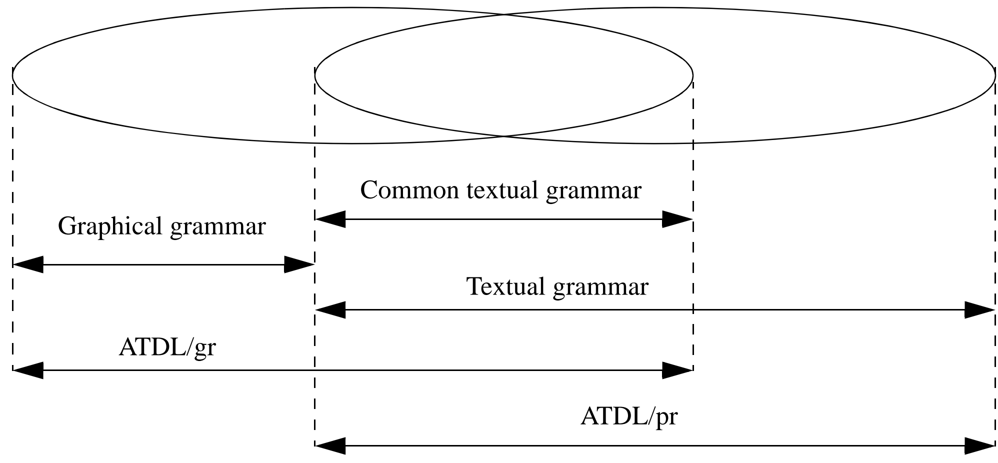


Figure 1. ATDL grammars

Each of the concrete grammars has a definition of its own syntax and of its relationship to the semantics of ATDL. This approach will ensure that ATDL/pr and ATDL/gr are equivalent.

The syntax productions of ATDL are specified in section 26.2. As an aid to clarifying the ATDL description, many syntax productions are embedded in the text of the body of this document. To aid readability some productions will appear in several places in the text. The syntax productions embedded within the text are intended to be identical copies of the corresponding productions from section 26.2, but if there is any conflict section 26.2 shall take precedence.

4.5. Unanimity of the specification

The language is specified syntactically and semantically in terms of a textual description in the body of the current document (clauses 5 to 24) and in a formalized way in Chapter 26. In each case, when the textual description is not exhaustive, the formal description completes it. If the textual and the formal specifications are contradictory, the latter shall take precedence.

4.6. Conformance

The present document does not specify levels of implementation for the language. However, for an implementation claiming to conform to this version of the language, all implemented features of the present document shall be consistent with the requirements given in the present document.

NOTE: This does not prevent any conformant implementation to realize extra features not specified in the present document.

4.7. Comparison of ATDL, C++ and Java

As a Java programmer, you already have the basic idea of object-oriented programming, and the syntax of ATDL no doubt looks familiar to you. This makes sense since ATDL is semantically a superset of Java. However, there are a surprising number of differences between ATDL and Java. These differences are intended to be significant improvements, and if you understand the differences you'll see why ATDL is such a beneficial programming language.

1. ATDL has both kinds of comments like Java and C++ do.

2. Class definitions are roughly the same form in ATDL and Java as in C++, but there's no closing semicolon.

3. ATDL and Java, like C++, have primitive types for efficient access. All the primitive types have specified sizes that are machine independent for portability. Type-checking and type requirements are much tighter in ATDL and Java.

4. The ATDL **wide char** type uses the international 16-bit Unicode character set, so it can automatically represent most national characters.

5. ATDL uses modules and groups in place of namespaces. The name issue is taken care of by putting everything into a class and by using a facility called "groups" that performs the equivalent namespace breakup for class names.

6. There are no ATDL and Java pointers in the sense of C++. When you create an object with **create**, you get back a reference.

7. ATDL has constructors that are similar to constructors in C++ and Java. You get a default constructor if you don't define one, just like in C++ and Java.

8. ATDL has method overloading that works virtually identically to C++ function and Java method overloading.

9. There's no **goto** in ATDL and Java. The one unconditional jump mechanism is the **break** *label* or **continue** *label*, which is used to jump out of the middle of multiply-nested loops.

10. ATDL and Java use a singly-rooted hierarchy, C++ appears to be the only object-oriented language that does not impose a singly rooted hierarchy.

11. Java has no templates or other implementation of parameterized types. Templates are a wonderful feature of ATDL and C++. The collections in Java don't have the same kind of efficiency as template implementations would allow.

12. ATDL and Java have built-in multi-threading support. Mutual exclusion occurs at the level of objects using the **synchronized** keyword as a type qualifier for methods. Only one thread may use a **synchronized** method of a particular object at any one time. Put another way, when a **synchronized** method is entered, it first "locks" the object against any other **synchronized** method using that object and "unlocks" the object only upon exiting the method. There are no explicit locks; they happen automatically.

13. Instead of controlling blocks of declarations like C++ does, the access specifiers in ATDL and Java are placed on each definition for each member of a class. Without an explicit access specifier, an element defaults to "friendly," which means that it is accessible to other elements in the same scope unit (equivalent to them all being C++ **friend**s) but inaccessible outside the scope unit.

14. Nested classes. In C++, nesting a class is an aid to name hiding and code organization (but C++ namespaces eliminate the need for name hiding). ATDL grouping and Java packaging provides the equivalence of namespaces, so that isn't an issue.

15. Inheritance in ATDL has the same effect as in C++, but the syntax is different. ATDL uses the **extends** keyword to indicate inheritance from a base class and the **inherited** keyword to specify methods to be called in the base class that have the same name as the method

you're in. (However, the **inherited** keyword in ATDL allows you to access methods only in the parent class, one level up in the hierarchy.) The base-class constructor is also called using the **inherited** keyword.

16. Inheritance in ATDL doesn't change the protection level of the members in the base class. Also, overridden methods in a descendant class cannot reduce the access of the method in the ancestor class.

17. ATDL provides the **interface** keyword, which creates the equivalent of an abstract base class filled with abstract operations and with no data members. This makes a clear distinction between something designed to be just an interface and an extension of existing functionality via the **extends** keyword.

18. There's no **virtual** keyword in Java because all non-**static** methods always use dynamic binding. The reason **virtual** exists in ATDL and C++ is so you can leave it off for a slight increase in efficiency when you're tuning for performance.

19. ATDL and Java don't provide multiple inheritance, at least not in the same sense that C++ does. The **interface** keyword takes care of combining multiple interfaces.

20. Run-time object type checking functionality in ATDL is quite similar to that in Java and C++. The compiler automatically invokes the dynamic casting mechanism without requiring extra syntax.

21. Exception specifications in ATDL and Java are vastly superior to those in C++. Instead of the C++ approach of calling a function at run-time when the wrong exception is thrown, ATDL and Java exception specifications are checked and enforced at compile-time. In addition, overridden methods must conform to the exception specification of the base-class version of that method: they can raise the specified exceptions. This provides much more robust exception-handling code.

22. ATDL and Java have method overloading, but no operator overloading.

23. The **const** constructs in ATDL and C++ are absent in Java by convention. If you want the equivalent of C++'s pass-by-value in Java, you call **clone( )** to produce a local copy of the argument (although the **clone( )** mechanism is somewhat poorly designed). There's no copy-constructor that's automatically called.

24. Since ATDL and Java can be too restrictive in some cases, ATDL and Java solve this with *external methods* that allow you to call a function written in another language.

25. Generally, ATDL and Java are more robust, via:

- Object handles initialized to **null** (a keyword)
- Handles are always checked and exceptions are raised for failures
- Clean, relatively fool-proof exception handling
- Simple language support for multi-threading

ATDL is a fun language. Java and C++ programmers should have a relatively easy time learning it, and will find that they enjoy using it.

5. Basic language elements

5.1. General

The top-level unit of ATDL is a module. A module cannot be structured into sub-modules. A module can import definitions from other modules. Modules can have parameter lists to give a form of test suite parameterization similar to the PICS and PIXIT parameterization mechanisms defined in ISO/IEC 9646-2.

Table 3: Overview of ATDL language elements

ATDL Language element	Associated keyword	Associated graphical symbol
Structural concepts		
ATDL module definition	**module**	<module symbol>
Group definitions	**group**	<group symbol>
Altstep heading definitions	**altstep**	<reference symbol>
Test thread definitions	**thread**	<thread class symbol>
Class definitions	**class**	<class symbol>
Object template definitions	**co-class**	<coclass symbol>
Component inheritance definitions	**extends**	<component extends symbol>
Communication co-interface definitions	**co-interface**	<interface symbol>
Communication cp-interface definitions	**interface**	<interface symbol>
Interface inheritance definitions	**extends**	<interface extends symbol>
Required interface definitions	**requires**	<dependency symbol>
Supported interface definitions	**supports**	<channel symbol>
Textual object declarations		
Import of definitions from other module	**import**	<text symbol>
Timer declarations	**timer**	<text symbol>
Data type definitions	**type**	<text symbol>
Constant definitions	**const**	<text symbol>
Variable declarations	**var**	<text symbol>
Exception definitions	**exception**	<text symbol>
Signature definitions	**operation**	<interface symbol 1>
External function/constant definitions	**external**	<text symbol>
Data/signature template definitions	**template**	<text symbol>
Dynamic object declarations		
Constructor definitions	**constructor**	<frame symbol>
Function definitions	**function**	<frame symbol>
Altstep definitions	**altstep**	<frame symbol>
Test case definitions	**testcase**	<frame symbol>

The control part of a module calls the test cases and controls their execution. Program statements (such as **if-else** and **do-while**) can be used to specify the selection and execution order of individual test cases. The concept of global variables is not supported in ATDL.

ATDL has a number of pre-defined basic data types as well as structured types such as sequences, sets, choices, enumerated types and arrays. Imported ASN.1 types and values may be used with ATDL.

A special kind of data structure called a template provides parameterization and matching mechanisms for specifying test data to be sent or received over the test channels. The operations on these channels provide both message-based and procedure-based communication capabilities. Procedure calls may be used for testing implementations which are not message based.

Dynamic test behaviour is expressed as test cases. ATDL program statements include powerful behaviour description mechanisms such as alternative reception of communication, timer events and default behaviour. Test verdict assignment and logging mechanisms are also supported.

Finally, ATDL language elements may be assigned attributes such as encoding information and display attributes. It is also possible to specify (non-standardized) user-defined attributes.

5.2. Parameterization

Parameters are used for modules, types, operations, templates, functions, testcases and altsteps.

5.2.1. Static and dynamic parameterization

ATDL supports *value* parameterization according to the following limitations:

a) language elements which cannot be parameterized are: **const**, **var**, **timer**, **control**, **group** and **import**;

Table 4: Overview of parameterizable ATDL language elements

Keyword	Value Parameterization	Associated graphical symbol
module	Static at start of run-time	*Values of:* all basic types and all user-defined types
type	Static at compile-time	*Values of:* all basic types and all user-defined types
template	Dynamic at run-time	*Values of:* all basic types, all user-defined types and **template.**
function	Dynamic at run-time	*Values of:* all basic types, all user-defined types, component type, **interface** type, **default**, **template** and **timer**.
altstep	Dynamic at run-time	*Values of:* all basic types, all user-defined types, component type, **interface** type, **default**, **template** and **timer**.
testcase	Dynamic at run-time	*Values of:* all basic types and of all user-defined types and **template**.
operation	Dynamic at run-time	*Values of:* all basic types and all user-defined types
NOTE 1: **sequence of**, **set of**, **enumerated,** and **subtype** definitions do not allow parameterization. NOTE 2: Examples of syntax and specific use of parameterization with the different language elements are given in the relevant clauses in the present document.		

b) the language element **module** allows *static* value parameterization to support test suite parameters i.e. this parameterization may or may not be resolvable at compile-time but shall be resolved by the commencement of run-time (i.e. *static* at run-time). This means that, at run-time, module parameter values are globally visible but not changeable;

c) all user-defined **type** definitions (including the structured type definitions such as **set**, **sequence** etc.) support *static* value parameterization i.e. this parameterization shall be resolved at compile-time;

d) the language elements **template**, **operation**, **testcase**, **altstep** and **function** support *dynamic* value parameterization (i.e. this parameterization shall be resolvable at run-time).

A summary of which language elements can be parameterized and what can be passed to them as parameters is given in Table 4.

ATDL concrete textual grammar

55 FormalCrefPar ::= FormalValuePar | FormalTemplatePar

168 FormalPar&Type ::= FormalValuePar | FormalTimerPar | FormalTypePar
| FormalTemplatePar | FormalInterfacePar

375 Direction ::= "in" | "out" | "inout"

376 FormalValuePar ::= [Direction] ValueParIdentifier {":" ValueParIdentifier}* Type

377 ValueParIdentifier ::= Identifier

378 FormalTypePar ::= [Direction] ValueParIdentifier

379 FormalInterfacePar ::= ["inout"] InterfaceParIdentifier InterfaceTypeIdentifier

380 InterfaceParIdentifier ::= Identifier

381 FormalTimerPar ::= ["inout"] "timer" TimerParIdentifier

382 TimerParIdentifier ::= Identifier

383 FormalTemplatePar ::= ["in"] "template" TemplateParIdentifier Type

384 TemplateParIdentifier ::= Identifier

5.2.2. Formal and actual parameter lists

The number of elements and the order in which they appear in an actual parameter list shall be the same as the number of elements and their order in which they appear in the corresponding formal parameter list. Furthermore, the type of each actual parameter shall be compatible with the type of each corresponding formal parameter.

Concrete textual grammar

77 ActualCrefParList ::= "(" ActualCrefPar {"," ActualCrefPar}* ")"

78 ActualCrefPar ::= [VarIdentifier AssignmentChar] TemplateInstance | Type

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type, the TemplateInstance production shall resolve to one or more SingleExpressions */

174 ActualParList ::= "(" ActualPar {"," ActualPar}* ")"

175 ActualPar ::= TimerRef | TemplateInstance | Type | Channel | ComponentRef

5.2.3. Empty formal parameter list

If the formal parameter list of the ATDL language elements **function**, **testcase**, **operation**, **altstep** or **external function** is empty then the empty parentheses can be included both in the declaration and in the invocation of that element. In all other cases the empty parentheses shall be omitted.

5.2.4. Nested parameter lists

Generally, all parameterized entities specified as an actual parameter shall have their own parameters resolved in the actual parameter list.

5.3. Parameter semantics

By default, all actual parameters of basic types, basic string types, user-defined structured types, channel type and component type are passed by value. This may optionally be denoted by the keyword `in`. To pass parameters of the mentioned types by reference the keywords **out** or **inout** shall be used.

Timers and channels are always passed by reference and are identified by the keywords **timer** and **interface**. The keyword **inout** may optionally be used to denote passing by reference.

Passing parameters by reference has the following limitations:

a) only the formal parameter lists to **altsteps** called explicitly, **function**s, **operation**s and **testcase**s may contain pass-by-reference parameters;

b) the actual parameters shall only be variables (e.g. not constants or templates).

Actual parameters that are passed by value may be variables as well as constants, templates etc.

5.3.1. In and inout parameters

Most parameters are either **in** parameters (the default) or variable (**inout**) parameters. In parameters are passed *by value*, while variable parameters are passed *by reference*. To see what this means, consider the following functions.

```
function DoubleByValue(MyPar1 Word) return Word       // MyPar1 is an in parameter
{
        MyPar1 := 2 * MyPar1;
        return MyPar1;
}
function DoubleByRef(inout MyPar1 Word) return Word  // MyPar1 is a variable parameter
{
        MyPar1 := 2 * MyPar1;
        return MyPar1;
}
```

These functions return the same result, but only the second one — *DoubleByRef* — can change the value of a variable passed to it. Suppose we call the functions like this:

```
        var I Word, J Word, V Word, W Word;
        I := 4;
        V := 4;
        J := DoubleByValue(I);          // J = 8, I = 4
        W := DoubleByRef(V);            // W = 8, V = 8
```

After this code executes, the variable *I*, which was passed to DoubleByValue, has the same value we initially assigned to it. But the variable *V*, which was passed to DoubleByRef, has a different value.

An **in** parameter acts like a local variable that gets initialized to the value passed in the test case or function call. If you pass a variable as an **in** parameter, the test case or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

If a routine's declaration specifies an **inout** parameter, you must pass an assignable expression — that is, a variable, field, or indexed variable — to the routine when you call it. To use our previous examples, DoubleByRef(7) produces an error, although DoubleByValue(7) is legal.

5.3.2. Template parameters

A template (**template**) parameter is like a local constant or read-only variable. Template parameters are similar to **in** parameters, except that you can't assign a value to a template parameter within the body of a test case or function, nor can you pass one as an **inout** parameter to another routine.

Using **template** parameter allows the compiler to optimize code for structured- and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

5.3.3. Out parameters

An **out** parameter, like a variable parameter, is passed by reference. With an **out** parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The **out** parameter is for output only; that is, it tells the function or test case where to store output, but doesn't provide any input. You should use **out** parameters when you pass an uninitialized variable to a function or test case.

5.3.4. String parameters

When you declare routines that take string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

testcase Check(S **charstring**[20]);   // syntax error

causes a compilation error. But

type TString20 ::= **charstring**[20];

testcase Check(S TString20);

is valid.

5.3.5. Array parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

testcase Sort(A **sequence**[1..10] **of** Smallint);   // syntax error

causes a compilation error. But

type TDigits ::= **sequence**[1..10] **of** Smallint;

testcase Sort(A TDigits);

is valid. For most purposes, however, *open array parameters* are a better solution.

5.3.6. Open array parameters

Open array parameters allow arrays of different sizes to be passed to the same function or test case. To define a routine with an open array parameter, use the syntax *sequence of type* (rather than *sequence* [*X..Y*] *of type*) in the parameter declaration.

For example,

function Find(A **sequence of char**) **return** Smallint;

declares a function called *Find* that takes a character array of any size and returns an integer.

The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The example above creates a function that takes any array of *char* elements,

including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

type TDynamicCharArray ::= **sequence of char**;
function Find(A TDynamicCharArray) **return** Smallint;

Within the body of a routine, open array parameters are governed by the following rules.

- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters. They cannot be passed to *SetLength*.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

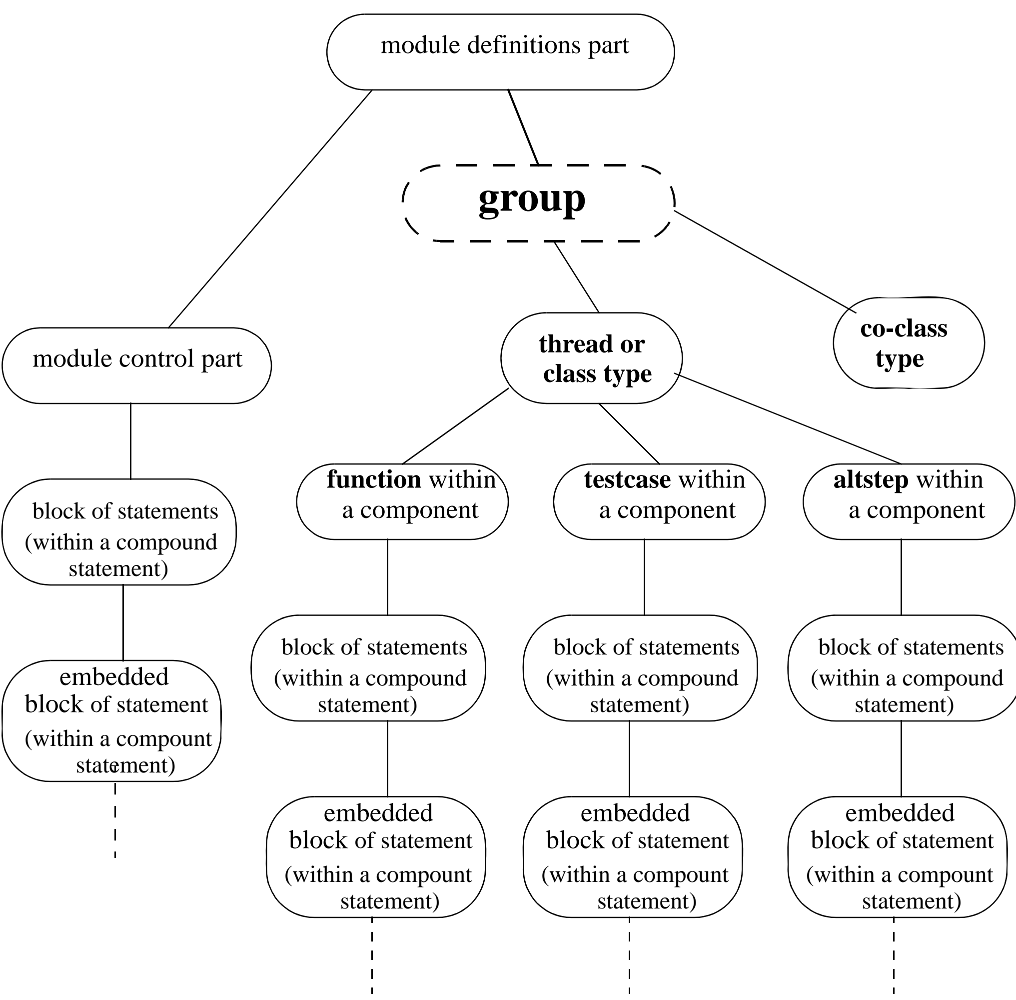


Figure 2. Hierarchy of scope units

5.4. Scope rules

ATDL provides six basic units of scope:

a) module definition part; b) control part of a module; c) groups;

d) component and interface types; e) functions, altsteps and test cases;

f) "blocks of statements and declarations" within compound statements.

Each unit of scope consists of (optional) declarations. The scope units control part of a module, functions, test cases, altsteps and "blocks of statements and declarations" within compound statements may additionally specify some form of behaviour by using the ATDL program statements and operations (see §17).

Definitions made in the module control part have local visibility, i.e. can be used within the control part only.

Test cases, altsteps and functions are individual scope units without any hierarchical relation between them, i.e. declarations made at the beginning of their body have local visibility and may be used in the given test case, **altstep** or function only (e.g. a declaration made in a test case is not visible in a function called by the test case or in an **altstep** used by the test case).

Compound statements, e.g. **if-else**-, **while**-, **do-while**-, or **alt**-statements include "blocks of statements and declarations". They may be used within the control part of a module, test cases, altsteps, functions, or may be embedded in other compound statements, e.g. an **if-else**-statement that is used within a **while**-loop.

The "blocks of statements and declarations" of compound statements and embedded compound statements have a hierarchical relation both to the scope unit including the given "block of statements and declarations" and to any embedded "block of statements and declarations". Declarations made within a "block of statements and declarations" have local visibility.

The hierarchy of scope units is shown in Figure 2. Declarations of a scope unit at a higher hierarchical level are visible in all units at lower levels within the same branch of the hierarchy. Declarations of a scope unit in a lower level of hierarchy are not visible to those units at a higher hierarchical level.

5.5. Identifiers and keywords

ATDL identifiers are case sensitive and ATDL keywords shall be written in all lowercase letters (see §26). ATDL keywords shall not be used neither as identifiers of ATDL objects nor as identifiers of objects imported from modules of other languages.

ATDL identifiers are case sensitive and may only contain lowercase letters (a-z) uppercase letters (A-Z) and numeric digits (0-9). Use of the underscore (_) symbol is also allowed. An identifier shall begin with a letter (i.e. not a number and not an underscore).

5.6. Division of text

The Recommendation is organized by topics described by an optional introduction followed by titled enumeration items for:

a) *Concrete textual grammar* – Both the common textual grammar used for ATDL/pr and ATDL/gr and the grammar used only for ATDL/pr. This grammar is described by the textual syntax, static conditions, and well-formedness rules for the textual syntax.

b) *Concrete graphical grammar* – Described by the graphical syntax, static conditions and well-formedness rules for the graphical syntax, and some additional drawing rules(to those in 5.7).

c) *Semantics* – Gives meaning to a construct, provides the properties it has, the way in which it is interpreted and any dynamic conditions which have to be fulfilled for the construct to behave well in the ATDL sense.

d) *Graphical notation* – This section contains a detailed description of the graphical notation for the concept. The notation section usually includes one or more diagrams to illustrate the concept.

e) *Examples*

Where a topic has an introduction followed by a titled enumeration item, then the introduction is considered to be an informal part of the Recommendation presented only to aid understanding and not to make the Recommendation complete.

5.7. General drawing rules

The size of the graphical symbols can be chosen by the user.

The metasymbol ***is_followed_by*** implies a <flow line symbol>.

Line symbols may consist of one or more straight line segments.

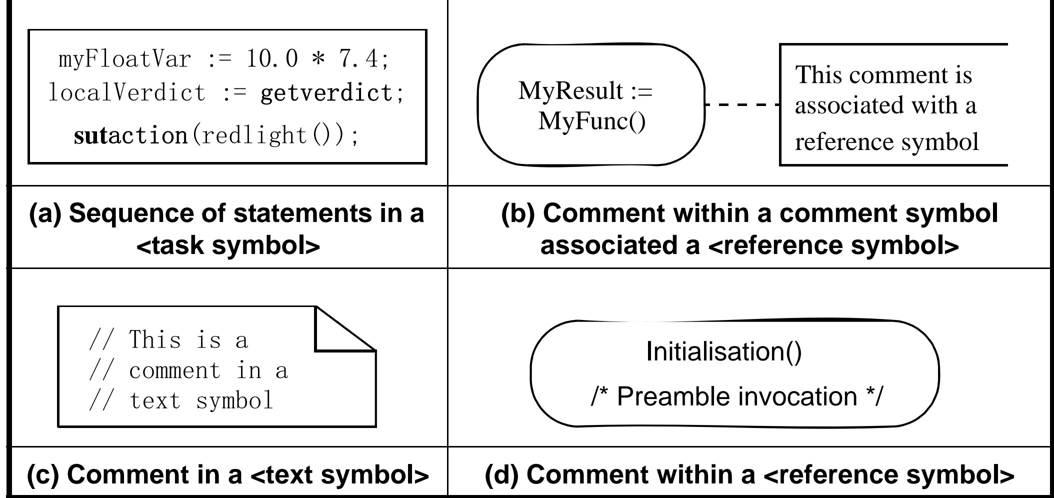
An arrowhead is required on a <flow line symbol>, when it enters another <flow line symbol>, a <try out-connector symbol>, a <decision outlet symbol>, or an <alt outlet symbol>. In other cases, arrowheads are optional on <flow line symbol>s. The <flow line symbol>s are horizontal or vertical.

Vertical mirror images of <internal input symbol>, <internal output symbol>, <message in symbol>, <message out symbol>, <procedure in symbol>, <exception out symbol>, <exception in symbol>, and <comment symbol> are allowed.

The right-hand argument of the metasymbol ***is_associated_with*** must be closer to the left-hand argument than to any other graphical symbol. The syntactical elements of the right-hand argument must be distinguishable from each other.

Text within a graphical symbol must be read from left to right, starting from the upper left corner. The right-hand edge of the symbol is interpreted as a newline character, indicating that the reading must continue at the leftmost point of the next line (if any).

Figure 3. Examples for the effects of the general drawing rules



5.7.1. Comments

A comment is a notation to represent comments associated with symbols or text. Comments written in free text may appear anywhere in an ATDL specification.

In the *Concrete textual grammar*, two forms of comments are used.

Block comments shall be opened by the symbol pair /* and closed by the symbol pair */.

EXAMPLE 1:

```
/* This is a block comment
spread over two lines */
```

Block comments shall not be nested.

Line comments shall be opened by the symbol pair // and closed by a <*newline*>.

510 <comment area> ::= <comment symbol> **contains** FreeText
is_connected_to <dashed association symbol>

One end of the <dashed association symbol> must be connected to the middle of the vertical segment of the <comment symbol>.

A comment in a <comment symbol> can be provided in form of free text, i.e. the comment delimiter "/*", "*/" and "//" of the core language need not to be used.

ATDL/gr provides three possibilities to put comments into ATDL diagrams:

a) Comments may be put into Graphical symbols following the symbol inscription and using the syntax for comments of the ATDL textual language (Figure 3 (d)).

b) Comments in the syntax for comments of the ATDL textual language can be put into text symbols and freely placed in the ATDL diagram area (Figure 3 (c)).

c) The comment symbol can be used to associate comments to graphical symbols. A <comment symbol> can be connected to any graphical symbol by means of a <dashed association symbol>. The <comment symbol> is considered as a closed symbol by completing (in imagination) the rectangle to enclose the text. It contains comment text related to the graphical symbol.

A comment in a comment symbol can be provided in form of free text, i.e. the comment delimiter "/*", "*/" and "//" of the textual language need not to be used (Figure 3 (b)).

5.7.2. Diagram area

Each Graphical control, test case, altstep and function diagram shall have a frame symbol (also called diagram frame) to define the diagram area. All symbols and text needed to define a complete and syntactically correct ATDL diagram shall be made inside the diagram area.

5.7.3. Diagram heading

Each ATDL/gr diagram shall have a diagram heading. The diagram heading shall be placed in the upper left-hand corner of the diagram frame.

The diagram heading shall uniquely identify each ATDL/gr diagram type. The general rule to achieve this is to construct the heading from the keywords testcase, altstep or function followed by the ATDL signature of the test case, altstep or function that should be presented graphically. For an ATDL/gr control diagram, the unique heading is constructed from the keyword **control** followed by the module name.

5.7.4. Usage of semicolons

All ATDL/gr symbols with the exception of the <task symbol> and <save symbol> shall include only one statement in ATDL core textual language. Only a <task symbol> or a <save symbol> may include a sequence of ATDL statements (see §5.7.5).

Semicolons shall separate the statements in a sequence of statements within a <task symbol>. The semicolon is optional for the last statement in the sequence.

5.7.5. Usage of task symbols

The following ATDL declarations, statements and operations are specified within task symbols: local declarations, assignments, **bind**, **write**, **release**, channel controlling and **sutaction**.

A <task symbol> may contain several task statements. It is not necessary to use a separate <task symbol> for each declaration, statement or operation.

5.8. Variables declarations

ATDL allows the declaration and initialization of variables at the beginning of statement blocks. ATDL/gr uses the syntax of the ATDL core language for declarations in several symbols. The type of a symbol depends on the specification of the initialization, e.g. a variable of type **default** that is initialized by means of an **activate** operation shall be specified within a default symbol.

5.8.1. Declaration of variables within <create request symbol>s

Variable declarations of a component type that are initialized by means of component instance creation expressions shall be made within a <create request symbol>. In contrast to declarations within <task symbol>s, each declaration that is initialized by means of a component instance creation expression shall be presented in a separate <create request symbol>.

5.8.2. Declaration of variables within <default symbol>s

Variable declarations of type **default** that are initialized by means of **activate** operations shall be made within a default symbol. In contrast to declarations within <task symbol>s, each declaration that is initialized by means of an **activate** operation shall be presented in a separate <default symbol>.

5.8.3. Declaration of variables within <reference symbol>s

Variable declarations that are initialized by means of a function invocation expression, shall be made within <reference symbol>s. In contrast to declarations within <task symbol>s, each declaration that is initialized by means of a function invocation expression, shall be presented in a separate <reference symbol>.

Figure 4. Examples for declarations in ATDL/gr

var MyComp CompType := CompType.**create**	**var** MyFloatVar **float**; **const** MyConst **cardinal** := 6; **var** MyDefault **default** := **null**
(a) Variable declaration within a <create request symbol>	**(b) Sequence of declarations within a <task symbol>**
var MyVar **integer** := MyFunction()	**var** MyDefault **default** := **activate**(MyAltstep())
(c) Variable declaration within a <reference symbol>	**(d) Variable declaration within a <default symbol>**

5.9. Special terminal symbols

ATDL special terminal symbols are listed in this section.

5.9.1. Separators

The following nine ASCII characters are the ATDL *separators* (punctuators):

( ) { } [ ] ; , .

5.9.1.1. Statement terminator symbols

In general all ATDL language constructs (i.e. definitions, declarations, statements and operations) are terminated with a semi-colon (;). The semi-colon is optional if the language construct ends with a right-hand curly brace (}) or the following symbol is a right-hand curly brace (}), i.e. the language construct is the last statement in a block of statements, operations and declarations.

5.9.2. Operators

The following 27 tokens are the ATDL *operators*, formed from ASCII characters:

:= > < ! **not** **and** **or** **xor** == <= >= != && ||

+ - * / **mod** & | ^ rem << >> <@ @>

6. Abstract Object Definition Language

The AODL grammar is a subset of the proposed ATDL grammar, in other words, AODL is an "internal" language of the ATDL. AODL is a declarative language, it is an extension of ASN.1 [8], with additional constructs to support the operation invocation mechanism, and it can be mappable readily to and from ITU-ODL. AODL supports some features that are not (currently) covered by OMG-IDL and ASN1. The lack of a truly asynchronous method invocation model in CORBA is one of its shortcomings. In AODL, management events are behaviour unconnected with operations; they reflect the active, autonomous nature of managed objects.

The proposal introduces the concept of AODL which provides a level of abstraction needed to express the semantics of telecommunication components and to combine (compose) components, a capability beyond the OMG-IDL. AODL enhances IDL in that it can describe object semantics as well as interfaces. AODL (and semantic definition) is required to support the definition and standardization of common engineering objects, domain specific engineering objects and corporate standard engineering objects.

This submission provides a proposed specification for a Generic Object Request Broker Architecture which satisfies the request: an interoperable framework capable of supporting those engineering objects as application components within a specific vertical market. The GORBA Architecture is based on and dependent on the support of the CORBA infrastructure, services and facilities. Without distributed objects, engineering objects would not be possible.

By default pseudo-object types are not CORBA Objects. The rationale for that decision is that mandating CORBA Object semantics for all pseudo-object types is overkill. If value types had existed at the time CORBA 1.0 was defined pseudo objects (e.g. TypeCodes) could have been expressed cleanly as values too, forcing these kinds of data objects to support all the apparatus of a CORBA Object is an unnecessary burden. The CORBA pseudo-interfaces are conceptually very similar to the operational cp-interfaces of ATDL. It is our belief that the concepts provided by this submission would enable the OMG to eliminate almost all the pseudo-IDL (PIDL) contained in the CORBA specification.

An interesting application of our work is to assign meaning to connector specifications as they appear, for instance, we could enhance the Interface Description Languages (IDLs) used in middleware technologies by supplying automatically derived behavior descriptions in addition to the static method signatures in use today.

6.1. Conventions for the syntax description

Table 5 defines the meta-notation used to specify the extended BNF grammar for AODL/ATDL. (It henceforth just called BNF):

Table 5: The ATDL/AODL Syntactic Meta-notation

::=	is defined to be
abc xyz	abc followed by xyz
|	Alternatively
[abc]	0 or 1 instances of abc
{abc}*	0 or more instances of abc
{abc}+	1 or more instances of abc
(...)	textual grouping
{ ... }	textual grouping
Abc	the non-terminal symbol Abc
ABC	a terminal symbol ABC
"ABC"	a terminal symbol ABC

6.2. AODL keywords

The identifiers listed in Table 6 are reserved for use as keywords and may not be used otherwise, unless escaped with a leading underscore. Keywords obey the rules for identifiers and must be written exactly as shown in the above list.

and	enumerated	inout	octetstring	return
bitstring	exception	integer	of	sequence
boolean	extends	interface	operation	set
cardinal	external	language	optional	supports
char	false	members	or	template
charstring	float	mod	out	true
choice	group	modifies	raises	type
class	hexstring	module	real	variant
co	import	noblock	recursive	void
const	in	not	rem	wide
else	infinity	objid	requires	xor

Table 6: List of AODL terminals which are reserved words

6.3. GORBA/AODL basics

Abstract Object Definition Language (AODL) is an extension to the semantics of the OMG-IDL [25]. These stem from the ITU-ODL [24] and include multiple co-interface and co-class definitions, co-class group definitions, and message-based co-interface descriptions.

AODL is an extension of the ASN1, with additional constructs to support the operation invocation mechanism. It is a declarative language. It supports ATDL syntax for constant, type, template, and operation declarations; it does not include any algorithmic structures or variables.

AODL syntax and semantics have the following characteristics:

a) Every attempt has been made to ensure that AODL is architecturally consistent with evolving component architectures, including the CORBA Component Model [26].

b) AODL is a derivation of OMG IDL and ASN1, uses a subset of ATDL expression syntax, and incorporates many ITU-ODL (Object Definition Language) concepts and terms. AODL is a combination of the syntactic and semantic content of these languages.

c) AODL is a declarative language. It support:

1) ATDL syntax for constant, type, and operation declarations;
2) concepts derived from ITU-ODL to represent information model concepts [9];
3) side effect free expressions with syntax derived from ATDL;
4) concepts derived from TTCN-3 for declarative constraints; and
5) additional constructs required to model engineering objects and domains.

d) There is an isomorphic mapping between AODL and the GORBA interface repository.

6.4. Modules

In AODL, a specification breaks down into one or several modules. The basic element for an AODL compiler, the **module**, can be shared by several specifications, thereby facilitating their coherence. It is preferable, as much as possible, to build up the module names with the number of the standard it refers to and its name or acronym.

Module names are composed of an ATDL identifier followed by an optional object identifier.

The module names may have a worldwide scope of action in which case it is obviously impossible to ensure uniqueness. In order to clear up this ambiguity, the module is registered in a worldwide registration tree, where we deal with object identifiers. If the module is registered, its object identifier is inserted in curly brackets on the left-hand side of its name.

The object identifier *DefinitiveIdentifier* uniquely and unambiguously identifies a module. The object identifier *DefinitiveIdentifier* contains at least two arcs (*DefinitiveObjectIdComponent*). This restriction is imposed by the BER encoding rules, which encode together the two first arcs of the registration tree.

AODL syntax definition

```
1   AODL_Module ::= "module" AODL_ModuleId "{" [ModuleDefinitionsPart] "}"
2   AODL_ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
3   DefinitiveIdentifier ::= Dot ObjectIdType "{" DefinitiveObjIdComponentList "}"
4   DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+
5   DefinitiveObjIdComponent ::= NameForm | Number | NameForm "(" Number ")"
6   ModuleIdentifier ::= Identifier
7   ModuleDefinitionsPart ::= {ModuleDefinition [SemiColon] }+
8   ModuleDefinition ::= SupportingDef | ImportDef | GroupDef | InterfaceDef | CoclassDef
9   SupportingDef ::= TypeDef | ParameterizedTypeDef | ConstDef | TemplateDef | ExceptionDef
```

6.5. Defining group types

A co-class group type specification comprises two high-level parts. The first is associated with the declaration of the co-class group type's identifier. The second part is the group definition body, which comprises a specification of contained co-classe types and co-class group types.

AODL syntax definition

```
10  GroupDef ::= GroupHeading "{"
                      [SupportingDefSpec]
                      [InterfaceDefSpec]
                      [CoclassDefSpec]
                      MemberComponentList
11  GroupHeading ::= "group" GroupIdentifier
12  GroupIdentifier ::= Identifier
13  SupportingDefSpec ::= { SupportingDef SemiColon }*
14  InterfaceDefSpec ::= {InterfaceDef SemiColon }*
15  CoclassDefSpec ::= {CoclassDef SemiColon }*
```

16 MemberComponentList ::= “members” MemberComponentDef {“,” MemberComponentDef}* “;”

17 MemberComponentDef ::= GroupIdentifier | CoclassName

6.6. Defining co-class types

A class may be declared abstract and must be declared abstract if it is incompletely implemented; such a class can be extended by subclasses. A co-class is a class that is incomplete, or to be considered incomplete. Only co-classes may have abstract methods (operations), that is, methods that are declared but not yet implemented.

A co-class specification comprises two high level parts. The first supports inheritance, and is associated with the declaration of the identifier of the co-class. The second part is the co-class template body, which comprises the main sub-parts of the template as follows: 1) a specification of required and supported interfaces and co-interfaces. 2) an initialization specification.

A co-class defines both the public view of a type of object and its private implementation. Data members, properties, and operations make up a co-class definition. Properties describe an object’s state. Operations are the actions that may be performed on an object. Events are notifications that something interesting has happened. A co-class may be an event source that sends notifications, an event sink that receives notifications, or both.

A subclass of a co-class that is not itself abstract may be instantiated, resulting in the execution of a constructor for the co-class and, therefore, the execution of the property initializers for instance variables of that class.

It is a compile-time error to declare a co-class type such that it is not possible to create a subclass that implements all of its operations. This situation can occur if the class would have as members two operations that have the same operation signature but different return types.

ATDL concrete textual grammar

```
90  CoclassDef ::= CoclassHeading “{“
                   [ClassPropertiesList]
                   [InterfaceDefSpec]
                   SupportedInterfaceList
                   [RequiredInterfaceList]
91  CoclassHeading ::= “co” “class” CoclassIdentifier [CoclassHeritage]
92  CoclassIdentifier ::= Identifier
110 ClassPropertiesList ::= { (SupportingDef | ClassFieldDef | DefaultAltstepDef) SemiColon}*
```

AODL syntax definition

```
18  CoclassDef ::= CoclassHeading “{“ [ClassPropertiesList]
                   [UsesClause]
                   [InterfaceDefSpec]
                   SupportedInterfaceList
                   [RequiredInterfaceList]
19  CoclassHeading ::= “co” “class” CoclassIdentifier [CoclassHeritage]
20  CoclassIdentifier ::= Identifier
21  UsesClause ::= “uses” CoclassIdentifier {“,” CoclassIdentifier}*
22  CoclassName ::= [GlobalModuleId Dot] CoclassIdentifier
23  ClassPropertiesList ::= {SupportingDef SemiColon}*
```

6.6.1. Co-class type inheritance

Co-class type inheritance is intended to support specification reuse and to provide a mechanism for defining compatibility via sub-typing relationships.

AODL syntax definition

```
24  CoclassHeritage ::= “(“ CoclassIdentifier “)”
```

6.6.2. Required interface types

The second comprises the (declared) required interfaces which specifies interface types used by instances of the co-class type to perform their functions and provide their services.

ATDL concrete textual grammar

95 RequiredInterfaceList ::= “requires” InterfaceType {“,” InterfaceType}* SemiColon

AODL syntax definition

25 RequiredInterfaceList ::= “requires” InterfaceType {“,” InterfaceType}* SemiColon

6.6.3. Supported interface types

The (declared) supported interfaces of a co-class type are the interfaces listed as supported in the co-class type specifications. Instances of interface types declared as supported may be offered by instances of descendant classes of the co-class types being defined.

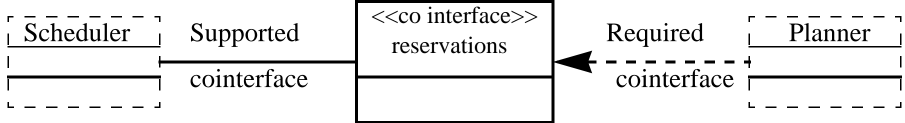
ATDL concrete textual grammar

94 SupportedInterfaceList ::= “supports” InterfaceType {“,” InterfaceType}* SemiColon

AODL syntax definition

26 SupportedInterfaceList ::= “supports” InterfaceType {“,” InterfaceType}* SemiColon

Figure 5. Supported co-interface type and required co-interface type



6.6.4. Co-class diagrams

A co-class diagram is a graph of co-classes connected by contract relationships. A co-class diagram shows the contracts among software components, including the components that specify them (e.g., implementation classes) and the artifacts that implement them (e.g., source code files, binary code files, executable files, scripts).

A diagram containing co-class types may be used to show static dependencies, such as compiler dependencies between programs, which are shown as dashed arrows (dependencies) from a client co-class to a supplier co-class that it depends on in some way.

A co-class diagram has only a type form, not an instance form. To show co-class instances, use a deployment diagram (possibly a degenerate one without groups).

ATDL concrete graphical grammar

96 <coclass diagram> ::= <coclass symbol>
(CoclassIdentifier <class properties area> ConstructorHeading)
[**is_connected_to** <component extends area>]
[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

If a co-class is the realization of a co-interface, the shorthand notation of a circle attached to the <coclass symbol> by a line segment may be used. Realizing a co-interface means that the implementation subclass of the co-class supply all the operations in the co-interface, at least. Using a co-interface means that the implementation elements in the co-class require no more

operations from a supplier co-class that the ones listed in the co-interface (but the client may depend on other interfaces, as well).

6.7. Declaring exception types

Exception declarations permit the declaration of struct-like data structures which may be returned to indicate that an exceptional condition has occurred during the performance of a request.

Each exception is characterized by its ExceptionIdentifier, an exception type identifier, and the type of the associated return value (as specified by the ExceptionMember in its declaration). If an exception is returned as the outcome to a request, then the value of the exception identifier is accessible to the programmer for determining which particular exception was raised.

If an exception is declared with members, a programmer will be able to access the values of those members when an exception is raised. If no members are specified, no additional information is accessible when an exception is raised.

AODL syntax definition

```
27  ExceptionDef ::= “exception” ExceptionIdentifier “{“ {ExceptionMember}* “}”
28  ExceptionIdentifier ::= Identifier
29  ExceptionMember ::= ExceptionTypeIdentifier Type [SemiColon]
30  ExceptionTypeIdentifier ::= Identifier
31  ExceptionName ::= [GlobalModuleId Dot] ExceptionIdentifier
```

6.8. Defining co-interface types

As with classes, we use the term co-interface to refer specifically to an interface listed as supported in the co-class specifications, just in case your development tool distinguishes these from some other type of interface. A co-class cannot implement a co-interface.

AODL syntax definition

```
32  InterfaceDef ::= MsgInterfaceDef | CoOpInterfaceDef
33  MsgInterfaceDef ::= MessageInterfaceHeader MessageAttribs
34  MsgInterfaceHeader ::= “co” “interface” MsgInterfaceTypeIdentifier [MsgInterfaceHeritage]
35  MsgInterfaceTypeIdentifier ::= Identifier
36  CoOpInterfaceDef ::= CoOpInterfaceHeader OperationAttribs
37  CoOpInterfaceHeader ::= “co” “interface” CpOpInterfaceTypeIdentifier [CoOpInterfaceHeritage]
38  CoOpInterfaceTypeIdentifier ::= Identifier
39  InterfaceTypeIdentifier ::= MsgInterfaceTypeIdentifier | CpOpInterfaceTypeIdentifier
40  InterfaceType ::= [ComponentType Dot] InterfaceTypeIdentifier
```

Semantics

AODL co-interface references are semantically equivalent to CORBA object references.

6.8.1. Co-interface type inheritance

The rules which apply to co-interface inheritance are those specified for OMG-IDL [25], with extensions to deal with messages.

Consistent with OMG-IDL, co-interface inheritance is the equivalent of simple inclusion of all attributes (including operational co-interface attributes), operations and messages from the base co-interface into the derived co-interface. This inclusion involves all attributes, operations and messages of the base co-interface, including those obtained by inheritance from other cointerface specifications. Hence a derived co-interface should always be capable of providing the services of the base co-interface.

It should be noted that a message-based co-interface should not inherit from an operational cointerface and vice versa.

AODL syntax definition

41 MsgInterfaceHeritage ::= “extends“ MsgInterfaceIdentifier {“,” MsgInterfaceIdentifier}*

42 CoOpInterfaceHeritage ::= “extends“ CoOpInterfaceIdentifier {“,” CpOpInterfaceIdentifier}*

message-based co-interface

43 MessageAttribs ::= “{“ {MessageList [SemiColon]}+ “}”

44 MessageList ::= [Direction] MessageIdentifier Type

45 MessageIdentifier ::= Identifier

6.8.2. Defining operational co-interface types

An operational cointerface is a collection of operations used to specify a service of a co-class.

AODL syntax definition

46 OperationAttribs ::= “{“ {OperationDef [SemiColon]}+ “}”

ATDL concrete textual grammar

147 OperationAttribs ::= “{“ {OperationDef [SemiColon]}+ “}”

Semantics

A co-interface declaration introduces a new reference type whose members are attributes and operations. This type has no implementation, but otherwise unrelated classes can implement it by providing logical implementations for its operations.

An operational co-interface comprises a set of interrogation and announcement signatures, one for each operation type in the interface template.

6.9. Importing from modules

It is possible to re-use definitions specified in different modules using the **`import`** statement.

AODL syntax definition

47 ImportDef ::= “import” ModuleId (ImportSpec | “{“ {ImportSpec [SemiColon]}* “}”) [“recursive”]

48 ImportSpec ::= ImportAllSpec | ImportGroupSpec | ImportInterfaceSpec | ImportClassSpec
| ImportTypeDefSpec | ImportTemplateSpec | ImportConstSpec

49 ImportAllSpec ::= [DefKeyword] Dot “*”

50 ModuleId ::= GlobalModuleId [“language” FreeText]

51 ModuleName ::= GlobalModuleId | LocalModuleId

52 GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]

53 LocalModuleId ::= ModuleIdentifier {Dot GroupIdentifier}*

54 ImportGroupSpec ::= “group” GroupIdentifier {“,” GroupIdentifier}*

55 ImportInterfaceSpec ::= “interface” InterfaceIdentifier {“,” InterfaceIdentifier}*

56 ImportClassSpec ::= “class” CoclassIdentifier {“,” CoclassIdentifier}*

57 ImportTypeDefSpec ::= “type” TypeIdentifier {“,” TypeIdentifier}*

58 ImportTemplateSpec ::= “template” TemplateIdentifier {“,” TemplateIdentifier}*

59 ImportConstSpec ::= “const” ConstIdentifier {“,” ConstIdentifier}*

60 ExtendedAlphaNum ::= */* REFERENCE - A character from any character set defined in ISO/IEC 10646 */*

61 FreeText ::= “ “ “ {ExtendedAlphaNum}* “ “ “

UML graphical notation

An import dependency is shown as a dashed arrow from the module gaining access to the module supplying elements.

6.10. Templates for sending messages

A template used in a sending operation defines a complete set of field values comprising the message to be transmitted over a test channel. At the time of the sending operation, the template shall be fully defined i.e., all fields shall resolve to actual values and no matching mechanisms shall be used in the template fields, neither directly nor indirectly.

// Given the message definition

type MyMessageType ::= **sequence**

{ field1 Cardinal, field2 **charstring**, field3 **boolean**}

// a message template could be

template MyTemplate MyMessageType :=
{field1 := 1, field2 := "My string", field3 := **true** }

// and a corresponding send operation could be

MyPCO.**send** (MyTemplate);

AODL syntax definition

62 TemplateDef ::= “template” BaseTemplate [DerivedDef] “:=” TemplateBody

63 BaseTemplate ::= TemplateIdentifier [FormalCrefParList] (MessageIdentifier | Operation)

64 TemplateIdentifier ::= Identifier

65 DerivedDef ::= “modifies” TemplateRef

66 FormalCrefParList ::= “(“ FormalCrefPar {“,” FormalCrefPar}* “)”

67 FormalCrefPar ::= FormalValuePar | FormalTemplatePar

/* STATIC SEMANTICS - FormalValuePar shall resolve to an *in* parameter */

68 TemplateBody ::= TemplateValue | FieldSpecList

69 FieldSpecList ::= “{“ [FieldSpec {“,” FieldSpec}*] “}”

70 FieldSpec ::= FieldReference “:=” TemplateBody

71 FieldReference ::= StructFieldIdentifier | ArrayOrBitRef | OperationParIdentifier

72 OperationParIdentifier ::= ValueParIdentifier

73 ArrayOrBitRef ::= “[“ FieldOrBitNumber “]”

74 FieldOrBitNumber ::= SingleConstExpression

/* STATIC SEMANTICS - SingleConstExpression will resolve to a value of cardinal type */

75 TemplateValue ::= SingleConstExpression | Omit | TemplateRefWithPara

76 Omit ::= “omit”

77 TemplateRefWithPara ::= [GlobalModuleId Dot] TemplateIdentifier [ActualCrefParList] | TemplateParIdentifier

78 ActualCrefParList ::= “(“ ActualCrefPar {“,” ActualCrefPar}* “)”

79 ActualCrefPar ::= Value

Semantics

Constraints that are referenced for send events shall not include wildcards (*i.e.*, AnyValue (?) or AnyOrOmit (*)) unless these are explicitly assigned specific values on the **send** event line in the behaviour description.

6.11. Summary

The TTCN style of programming has proven extremely popular because it is easy to use and avoids the need for TTCN programmers to learn a separate interface definition language. However, TTCN lacks interoperability with other languages and it is not currently supported over standard protocols. The AODL subset of ATDL described in this chapter is intended to unify the ease-of-programming of TTCN with support for cross-language operation (through AODL) and support for standard protocols (through IIOP).

To encourage convergence between the TTCN and CORBA programming communities, it is important to define a solution that is both fully compatible with current TTCN semantics and fully compatible with OMG IDL, IIOP, and the CORBA object model.

The subset of ATDL that meets these goals is referred to as AODL. This section describes the subset of ATDL that can be mapped to IDL and can run over GIOP.

All the standard ATDL predefined types except for **verdicttype** are supported as part of AODL. An AODL *remote co-interface* defines an ATDL interface that can be invoked remotely.

At run time, when a reference to an AODL remote co-interface is passed across a remote cointerface, the class of the actual object that is passed must be either a stub class or a remote implementation class.

It is not required that methods in ATDL classes be included into AODL.

6.11.1. Benefits of AODL

Our submission directly addresses requirements from the computational language of the Reference Model for Open Distributed processing [23]. The concepts and designs proposed have been tested in industrial-strength product. They are known to deliver significant benefits for the application developer.

The AODL is the language used to describe ODP objects and domain models. AODL provides a level of abstraction needed to develop components not currently found within the OMG's Interface Definition Language (IDL). A co-class definition written in AODL defines not only the component object interface but also the contract between cooperation components.

AODL provides syntaxes to describe static and dynamic aspects of the computational viewpoint of ODP-systems. Aspects which can be expressed by using the ATDL/AODL syntax include:

a) **Standard inheritance**: Co-class objects support the core object concepts of multiple interface inheritance, including implementation inheritance. All semantics defined for a co-class apply to all subtypes of that co-class.

b) **Replaceable**: The unit of implementation is defined by **co class**. It must be possible to relocate or replace a co-class with another implementation of the co-class, transparent to all clients.

c) **Reusability**: The Generic ORB Architecture (GORBA) lays the foundation for specifying and implementing interoperable abstract test suites. It is anticipated that this foundation will accelerate consensus building and OMG testing-domain task forces.

One key factor in reusability is understanding and communicating the design of existing co-classes. Implementation without design is not reusable. The precise co-class semantics, the contract between cooperating co-classes, provided by GORBA facilitates reusable components.

d) **Scaleable**: The GORBA is implemented as a composition of AODL specifications. Every effort has been made to avoid architecturally mandated limits to scale-ability.

e) **Ease of development and deployment**: The GORBA architecturally separates co-class building from co-class assembly and reuse. Co-class assembly and reuse is an inherently easy method for building and deploying object models.

f) **Off-the-shelf Models**: It is the realization of tailor-able, replaceable, reusable, interoperable, off-the-shelf domain model implementations that attain the cycle-time objective. There is a direct correspondence, maintained at run time, between the component model and component instances. Component instances are directly and dynamically controlled by component specifications.

g) **Application Integration**: The GORBA supports implementation inheritance, ensuring a common implementation semantic across all ORBs and enabling plug-and-play replace-ability between different implementations of a domain model. New or separate object models are integrated with an existing object model by the GORBA and associated tools.

h) **Legacy Applications**: Until the objective of plug-and-play domain models has been achieved, we can view all forms of application software as "legacy". Legacy software will be in many different forms, this proposed specification cannot address all possible scenarios. The integration of legacy applications requires two steps: 1) Define the legacy application in AODL 2) Implement the engineering objects using ATDL.

i) **Generality and Desktop Integration**: Based on existing and proposed CORBA interoperability standards, GORBA objects can interoperate with application components implemented on Microsoft OLE/COM, Internet/JAVA, or other CORBA implementations.

j) **Proof of Commonality**: ASN.1 [8] has already been used in a variety of domain specifications, AODL is an extension of ASN1 grammar productions.

7. Declaring ATDL/AODL signals

A signal is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is a *signal handler*, and can be written by the application developer. Signals let application developers customize the behavior of components without having to change the classes themselves.

7.1. Declaring messages

The specification of an asynchronous communication between components. ATDL is able to send and receive complex messages over the communication channels defined by the test configuration. These messages may be those explicitly concerned with testing the SUT or with the internal co-ordination and control messages specific to the relevant test configuration.

In TTCN-2 these messages are the Abstract Service Primitives (ASPs), the Protocol Data Units (PDUs) and co-ordination messages. The core language of ATDL is generic in the sense that it does not make any syntactic or semantic distinctions of this kind.

Complex messages may be defined as **sequence** types. For example:

```
type MyMessageType ::= sequence
{
        field1 FieldType1,
        field2 FieldType2,
        :
        fieldN FieldTypeN
}
```

Messages can, of course, be sub-structured.

Concrete textual grammar

145 MessageList ::= [Direction] MessageIdentifier Type

146 MessageIdentifier ::= Identifier

7.2. Declaring operations

Operation signatures are needed for synchronous communication. An operation may either be invoked in the SUT (i.e., the test system performs the call) or invoked in the test system (i.e., the SUT performs the call).

For both operations called from the SUT and operations called from the test system the complete **operation** signature shall be defined in the ATDL module.

An operation call will result in the called party performing either a reply (the normal case) or raising an exception. The actions resulting from an accepted operation call are defined by the receiving party.

An operation declaration consists of:

1) An optional operation attribute that specifies which invocation semantics the communication system should provide when the operation is invoked. Operation attributes are described in "operation attribute".

2) The type of the operation's return result; the type may be any type that can be defined in AODL. If no **return** is specified then the function is void. An explicit keyword for void is **null** in AODL.

3) An identifier that names the operation in the scope of the co-interface in which it is defined. The name together with the list of formal parameter types, is called the matching template of the operation. Note that the direction of the parameters is as seen by the *caller* rather than the *callee*.

4) An optional **raises** expression that indicates which exceptions may be raised as a result of an invocation of this operation.

Note that an operation declared in an operational interface must not be declared external or synchronized, or a compile-time error occurs, because those keywords describe implementation properties rather than interface properties. However, an operation declared in an operational interface may be implemented by a method that is declared external or synchronized in a class that implements the operational interface.

7.2.1. Procedure signatures

Procedure signatures (or signatures for short) are needed for procedure-based communication. Procedure-based communication may be used for the communication within the test system, i.e. among test components, or for the communication between the test system and the SUT. In the latter case, a procedure may either be invoked in the SUT (i.e. the test system performs the call) or invoked in the test system (i.e. the SUT performs the call). For all used procedures, i.e. procedures used for the communication among test components, procedures called from the SUT and procedures called from the test system, complete **operation** `signature` shall be defined in the ATDL interface.

ATDL concrete textual grammar

182 OperationDef ::= [OpAttribute] "operation" OperationIdentifier [FormatParList] [ReturnType] [RaisesExpr]

183 OperationIdentifier ::= Identifier

185 Operation ::= [ModuleName Dot] OperationIdentifier

AODL syntax definition

80 OperationDef ::= [OpAttribute] "operation" OperationIdentifier [FormatParList] [ReturnType] [RaisesExpr]

81 OperationIdentifier ::= Identifier

82 Operation ::= [GlobalModuleId Dot] OperationIdentifier

83 ReturnType ::= "return" Type

7.2.2. Operation attribute

ATDL supports *blocking* and *non-blocking* procedure-based communication. Operation definitions for non-blocking communication shall use a **noblock** keyword, shall only have **in** parameters (see §7.2.4) and shall have no return value (see §7.2.5), but may raise exceptions (see §7.2.6). By default, operation definitions without **noblock** keyword are assumed to be used for blocking procedure-based communication.

ATDL concrete textual grammar

184 OpAttribute ::= “noblock” | “template”

AODL syntax definition

84 OpAttribute ::= “noblock” | “template”

7.2.3. parameter lists

A specification of the values that a language element receives. A parameter list is an ordered list of formal parameter declarations. The list may be empty.

A parameter list is a comma-separated list of parameter declarations enclosed in parentheses.

ATDL concrete textual grammar

54 FormalCrefParList ::= FormalCrefPar {“,” FormalCrefPar}*

167 FormalParList ::= “(“ FormalPar&Type {“,” FormalPar&Type}* “)”

AODL syntax definition

85 FormalParList ::= “(“ FormalPar&Type {“,” FormalPar&Type}* “)”

7.2.4. Declaring parameters

Signature definitions may have parameters. Within an **operation** definition the parameter list may include parameter identifiers, parameter types and their direction i.e. **in**, **out**, or **inout**. A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. Note that the direction of the parameters is as seen by the *called* party rather than the *calling* party. The directional attributes are:

- **in** - the parameter is passed from client to server.
- **out** - the parameter is passed from server to client.
- **inout** - the parameter is passed in both directions.

AODL syntax definition

86 FormalPar&Type ::= FormalValuePar | FormalTemplatePar

87 Direction ::= “in” | “out” | “inout”

88 FormalValuePar ::= [Direction] ValueParIdentifier {“,” ValueParIdentifier}* Type

89 ValueParIdentifier ::= Identifier

90 FormalTemplatePar ::= [“in”] “template” TemplateParIdentifier Type

91 TemplateParIdentifier ::= Identifier

7.2.5. Value returning remote procedures

A remote procedure may return a value after its termination. The type of the return value shall be specified by means of a **return** clause in the corresponding signature definition.

7.2.6. Raises expressions

A raises expression specifies which exceptions may be raised as a result of an invocation of the operation or method. For example:

operation MyRemoteProc (**in** Par1 Byte, **out** Par2 **float**, **inout** Par3 Byte) **return** Byte;
    raises (Exception1, Exception2);

The *ExceptionNames* in the raises expression must be previously defined exceptions.

ATDL concrete textual grammar

186 RaisesExpr ::= "raises" "(" ExceptionName {"," ExceptionName}* ")"

AODL syntax definition

92 RaisesExpr ::= "raises" "(" ExceptionName {"," ExceptionName}* ")"

8. Declaring ATDL/AODL constants

Constants can be declared and used in module headers, module control, test cases and functions. Constant definitions are denoted by the keyword **const**. The value of the constant shall be assigned at the point of declaration. For example:

const MyConst1 Cardinal := 1;
const MyConst2 **boolean** := **true**, MyConst3 **boolean**:= **false**;

ATDL concrete textual grammar

47 ConstDef ::= "const" SingleConstDef {"," SingleConstDef}*

48 SingleConstDef ::= ConstIdentifier Type ":=" ConstantExpression

49 ConstIdentifier ::= Identifier

AODL syntax definition

93 ConstDef ::= "const" SingleConstDef {"," SingleConstDef}*

94 SingleConstDef ::= ConstIdentifier Type ":=" ConstantExpression

/* STATIC SEMANTICS - The value of the ConstantExpression shall be of the same type as the stated type for the constant */

95 ConstIdentifier ::= Identifier

8.1. Constant expressions

A *constant expression* is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; bit strings; octet strings; hexadecimal strings; the special constants *True*, and *False*; and expressions built exclusively from these elements with operators, and typecasts. Constant expressions cannot include variables, or function calls, except calls to the ATDL predefined functions.

ATDL concrete textual grammar

476 ConstantExpression ::= SingleConstExpression | CompoundConstExpression

477 SingleConstExpression ::= SingleExpression

479 CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression

AODL syntax definition

96 ConstantExpression ::= SingleConstExpression | CompoundConstExpression

97 SingleConstExpression ::= SingleExpression

98 CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression

9. ATDL/AODL operators

ATDL supports a number of predefined operators that may be used in the terms of ATDL expressions. The predefined operators fall into seven categories:

a) arithmetic operators; b) string operators; c) relational operators;

d) logical operators; e) bitwise operators; f) shift operators.

Table 7: List of ATDL/AODL operators

Category	Operator	Symbol or Keyword	AODL
Arithmetic operators	addition	**+**	yes
	subtraction	**-**	yes
	multiplication	*****	yes
	division	**/**	yes
	modulo	**mod**	yes
	remainder	**rem**	yes
String operators	concatenation	+	yes
Relational operators	equal	**==**	yes
	less than	**<**	yes
	greater than	**>**	yes
	not equal	**!=**	yes
	greater than or equal	**>=**	yes
	less than or equal	**<=**	yes
Logical operators	logical negation	**!**	yes
	logical conjunction	**&**	yes
	logical disjunction	|	yes
	logical exclusive or	^	yes
	conditional and	&&	no
	conditional or	||	no
	conditional ?	?:	no
	value set membership	**in**	no
Bitwise operators	bitwise negation	**not**	yes
	bitwise and	**and**	yes
	bitwise or	**or**	yes
	bitwise xor	**xor**	yes
Shift operators	bitwise shift left	**<<**	yes
	bitwise shift right	**>>**	yes
Rotate operators	rotate left	<@	yes
	rotate right	@>	yes

These operators are listed in Table 7.

The *unary operators* (taking one operand) include +, -, !, **not**, and typecast operators. All other operators are *binary* (taking two operands), except that + and – can function as either unary or

binary. A unary operator always precedes its operand. A binary operator is placed between its operands.

In complex expressions, rules of precedence determine the order in which operations are performed. The precedence of these operators is shown in Table 8. Within any row in this table, the listed operators have equal precedence. An operator with higher precedence is evaluated before an operator with lower precedence. If more than one operator of equal precedence appears in an expression, the operations are evaluated from left to right.

Parentheses may be used to group operands in expressions, in which case a parenthesized expression has the highest precedence for evaluation.

Table 8: Precedence of Operators

Priority	Operator type	Operator
highest		**(…)**
	UnaryBinary	**+, - *, /**, **mod**, **rem**
	Binary	**+, -, &**
	Unary	**not**
	Binary	**and**
	Binary	**xor**
	Binary	**or**
	Binary	**<<, >>, <@, @>**
	Binary	**<, >, <=, >=**
	Binary	**==, !=**
	Unary	**!**
	Binary	**&**
	Binary	**^**
	Binary	**|**
Lowest		

ATDL concrete textual grammar

482 SingleExpression ::= ConditionalExpression [? SimpleExpression Colon ConditionalExpression]

AODL syntax definition

99 SingleExpression ::= SimpleExpression {LogicOp SimpleExpression}*

/* OPERATIONAL SEMANTICS - If both SimpleExpressions and the LogicalOp exist then the SimpleExpressions shall evaluate to specific values of compatible types */

9.1. Additive Operators

The arithmetic operators represent the operations of addition, subtraction, multiplication, division and modulo. Operands of these operators shall be of type *integer* (including derivations of *integer*) or **float** (including derivations of **float**) or **real** (including derivations of **real**), except for **mod** and **rem** which shall be used with *integer* (including derivations of *integer*) types only.

Table 9: Binary additive operators

Operator	Operation	Operand types	Result type
+	addition	integer, cardinal, float, real	integer, cardinal, float, real
-	subtraction	integer, cardinal, float, real	integer, cardinal, float, real

With **integer** types the result type of arithmetic operations is **integer**. With **cardinal** types the result type of arithmetic operations is **cardinal**. With float types the result type of arithmetic operations is **float**. With real types the result type of arithmetic operations is **real**.

ATDL concrete textual grammar

491 AdditiveExpression ::= MultiplicativeExpression { (“+” | “-”) MultiplicativeExpression}*

AODL syntax definition

100 AdditiveExpression ::= MultiplicativeExpression { (“+” | “-”) MultiplicativeExpression}*

9.1.1. Unary arithmetic operators

In the case where plus (+) or minus (-) is used as the unary operator the rules for operands apply as well. The result of using the minus operator is the negative value of the operand if it was positive and vice versa.

Table 10: Unary arithmetic operators

Operator	Operation	Operand types	Result type
+	sign identity	integer, cardinal, float, real	integer, cardinal, float, real
-	sign negation	integer, cardinal, float, real	integer, cardinal, float, real

ATDL concrete textual grammar

493 UnaryExpression ::= [("+" | "-")] Primary | CastExpression

AODL syntax definition

101 UnaryExpression ::= [("+" | "-")] Primary | CastExpression

9.2. String operators

The predefined string operators perform concatenation of values of compatible string types. The operation is a simple concatenation from left to right. No form of arithmetic addition is implied. The result type is the root type of the operands. For example,

'1111'B + '0000'B + '1111'B gives '111100001111'B

If only one operand expression is of type **charstring**, then charstring conversion is performed on the other operand to produce a **charstring** at run time. The result is a reference to a newly created charstring object that is the concatenation of the two operand charstrings. The characters of the left-hand operand precede the characters of the right-hand operand in the newly created string.

For example,

"The square root of 2 is " + Math.sqrt(2)

produces the result:

"The square root of 2 is 1.4142135623730952"

9.3. Multiplicative operators

The operators *, /, **mod** and **rem** are called the *multiplicative operator*s. They have the same precedence and are syntactically left-associative (they group left-to-right). The type of each of the operands of a multiplicative operator must be a primitive numeric type.

The result of performing the division operation (/) on two:

1) *integer* values gives the whole *integer* value resulting from dividing the first *integer* by the second (i.e., fractions are discarded);

2) **float** values gives the **float** value resulting from dividing the first **float** by the second (i.e., fractions are not discarded);

3) **real** values gives the **real** value resulting from dividing the first **real** by the second (i.e., fractions are not discarded).

The value of *x* **mod** *y* is the value of x/*y* rounded in the direction of zero to the nearest integer.

The **rem** operator returns the remainder obtained by dividing its operands. In other words, *x* **rem** *y* = *x* – (*x* **mod** *y*) * *y*.

Table 11: Binary multiplicative operators

Operator	Operation	Operand types	Result type
*	multiplication	integer, cardinal, float, real	integer, cardinal, float, real
/	real division	integer, cardinal, float, real	float, real
mod	integer division	integer, cardinal	integer, cardinal
rem	remainder	integer, cardinal	integer, cardinal

A runtime error occurs when *y* is zero in an expression of the form x/y, *x* **mod** *y*, or *x* **rem** *y*.

ATDL concrete textual grammar

492 MultiplicativeExpression ::= UnaryExpression {MultiplyOp UnaryExpression}*

497 MultiplyOp ::= "*" | "/" | "mod" | "rem"

AODL syntax definition

102 MultiplicativeExpression ::= UnaryExpression {MultiplyOp UnaryExpression}*

103 MultiplyOp ::= "*" | "/" | “mod” | “rem”

9.4. Relational operators

The predefined relational operators represent the relations of equality (==), less than (<), greater than (>), non-equality to (!=), greater than or equal to (>=) and less than or equal to (<=). Operands of equality and non-equality may be of arbitrary but compatible types with the exception of the **enumerated** type, in which case operands shall be instances of the same type. All other relational operators shall have operands only of ordinal type (including derivatives of ordinal type) or float, real (including derivations of float, real). Expressions that use these operators must always evaluate to a **boolean** value.

Relational operators are used to compare two operands. Expressions that use these operators must always evaluate to a **boolean** value. For example, *I* == *J* is **true** just in case *I* and *J* have the same value, and *I* **!=** *J* is **true** otherwise. In all cases the two operands shall be of compatible type , except that a float and an integer can be compared.

Table 12: Relational operators

Operator	Operation	Operand types	Result type
==	equal	any compatible types	Boolean
!=	not equal	any compatible types	Boolean
>	greater than	ordinal types, float, real	Boolean
<	less than	ordinal types, float, real	Boolean
>=	greater than or equal	ordinal types, float, real	Boolean
<=	less than or equal	ordinal types, float, real	Boolean

An order among the values of type **char** or **wide char** is defined by the integer value of their encoding, i.e., the relational operators ==, <, >, !=, >= and <= can be used to compare values of type **char** or **wide char**.

Two **charstring** or **wide charstring** values are equal only, if they have equal lengths and the characters at all positions are the same. For values of **bitstring, hexstring** or **octetstring** types the same equality rule applies with the exception, that fractions which shall equal at all positions are bits, hexadecimal digits or pairs of hexadecimal digits accordingly.

ATDL concrete textual grammar

486 EqualityExpression ::= RelationalExpression [("==" | "!=") RelationalExpression]

487 RelationalExpression ::= ShiftExpression [("<" | ">" | ">=" | "<=") ShiftExpression]
| ShiftExpression “instanceof” RestrictedType
| ShiftExpression "in" ShiftExpression

AIDL syntax definition

104 EqualityExpression ::= RelationalExpression [("==" | "!=") RelationalExpression]

105 RelationalExpression ::= ShiftExpression [("<" | ">" | ">=" | "<=") ShiftExpression]

9.4.1. The class operator

The type of a ShiftExpression operand of the instanceof operator must be a reference type or the null type; otherwise, a compile-time error occurs. The RestrictedType mentioned after the instanceof operator must denote a reference type; otherwise, a compile-time error occurs.

At runtime, ShiftExpression must be an instance of the class denoted by RestrictedType or one of its descendants, or be **null**; otherwise an exception is raised. If the declared type of ShiftExpression is unrelated to RestrictedType—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results.

9.5. Boolean logical operators

The predefined **boolean** operators include the logical NOT operator “!”, logical AND operator “&”, logical exclusive XOR operator “^”, and inclusive OR operator “|”. Their operands shall be of type `boolean`. The result type of logical operations is `boolean`.

The logical NOT is the unary operator that returns the value **true** if its operand was of value **false** and returns the value **false** if the operand was of value **true**.

The logical AND returns the value **true** if both its operands are **true**; otherwise it returns the value **false**.

The logical OR returns the value **true** if at least one of its operands is **true**; it returns the value **false** only if both operands are **false**.

The logical XOR returns the value **true** if one of its operands is **true**; it returns the value **false** if both operands are **false** or if both operands are **true**.

ATDL concrete textual grammar

483 ConditionalExpression ::= LogicalExpression {(“&&” | “||”) LogicalExpression}*

484 LogicalExpression ::= SimpleExpression {LogicalOp SimpleExpression}*

485 SimpleExpression ::= ["not"] EqualityExpression

499 LogicOp ::= “&” | “^” | “|”

AODL syntax definition

106 SimpleExpression ::= ["not"] EqualityExpression

107 LogicOp ::= “&” | “^” | “|”

9.5.1. Conditional logical operator

The && operator is like logical and "&", but evaluates its right-hand operand only if the value of its left-hand operand is true. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value.

Each operand of && must be of type boolean, or a compile-time error occurs. The type of a conditional-and expression is always boolean.

At run time, the left-hand operand expression is evaluated first; if its value is false, the value of the conditional-and expression is false and the right-hand operand expression is not evaluated. If the value of the left-hand operand is true, then the right-hand expression is evaluated and its value becomes the value of the conditional-and expression. Thus, && computes the same result as & on boolean operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

The || operator is like logical or "|", but evaluates its right-hand operand only if the value of its left-hand operand is false. It is syntactically left-associative (it groups left-to-right). It is fully associative with respect to both side effects and result value.

Each operand of || must be of type boolean, or a compile-time error occurs. The type of a conditional-or expression is always boolean.

At run time, the left-hand operand expression is evaluated first; if its value is true, the value of the conditional-or expression is true and the right-hand operand expression is not evaluated. If the value of the left-hand operand is false, then the right-hand expression is evaluated and its value becomes the value of the conditional-or expression. Thus, || computes the same result as | on boolean operands. It differs only in that the right-hand operand expression is evaluated conditionally rather than always.

9.6. Bitwise operators

The predefined bitwise operators perform the operations of bitwise **not**,bitwise **and**, bitwise **or** and bitwise **xor**.

Their operands shall be of type **integer, cardinal, bitstring**, **hexstring**, **octetstring**. In the case of **and**, **or** and **xor** the operands shall be of compatible types. The result type of the bitwise operators shall be the root type of the operands.

The bitwise **not** unary operator inverts the individual bit values of its operand. For each bit in the operand a 1 bit is set to 0 and a 0 bit is set to 1. That is:

not '1'B gives '0'B

not '0'B gives '1'B

The bitwise **and** operator accepts two operands of equal length. For each corresponding bit position, the resulting value is a 1 if both bits are set to 1, otherwise the value for the resulting bit is 0. That is:

'1'B **and** '1'B gives '1'B
'1'B **and** '0'B gives '0'B
'0'B **and** '1'B gives '0'B
'0'B **and** '0'B gives '0'B

The bitwise **or** operator accepts two operands of equal length. For each corresponding bit position, the resulting value is 0 if both bits are set to 0, otherwise the value for the resulting bit is 1. That is:

'1'B **or** '1'B gives '1'B
'1'B **or** '0'B gives '1'B
'0'B **or** '1'B gives '1'B

'0'B **or** '0'B gives '0'B

The bitwise **xor** operator accepts two operands of equal length. For each corresponding bit position, the resulting value is 0 if both bits are set to 0 or if both bits are set to 1, otherwise the value for the resulting bit is 0. That is:

'1'B **xor** '1'B gives '0'B
'0'B **xor** '0'B gives '0'B
'0'B **xor** '1'B gives '1'B
'1'B **xor** '0'B gives '1'B

ATDL concrete textual grammar

489 BitwiseExpression ::= SubResult {BitOp SubResult}*

490 SubResult ::= ["not"] AdditiveExpression | “complement” ValueList

498 BitOp ::= “and” | “xor” | “or”

AODL syntax definition

108 BitwiseExpression ::= SubResult {BitOp SubResult}*

109 SubResult ::= ["not"] AdditiveExpression

/* OPERATIONAL SEMANTICS - If the not operator exists, the operand shall be of type bitstring, octetstring or hexstring. */

110 BitOp ::= “and” | “xor” | “or”

9.7. Shift operators

The predefined shift operators perform the shift left (<<) and shift right (>>) operations. Their left-hand operand shall be of type **bitstring, hexstring, octetstring, cardinal** or **integer**. Their right hand operand shall be of type **cardinal**. The result type of these operators shall be the same as that of the left operand.

The shift operators behave differently based upon the type of their left-hand operand. If the type of the left-hand operand is:

a) **bitstring** then the shift unit applied is 1 bit;

b) **hexstring** then the shift unit applied is 1 hexadecimal digit;

c) **octetstring** then the shift unit applied is 1 octet.

d) *integer type* then the operations $x << y$ and $x >> y$ shift the value of x to the left or right by y bits, which is equivalent to multiplying or dividing x by 2^y; the result is of the same type as x. For example, if *N* stores the value 01101 (decimal 13), then N << 1 returns 11010 (decimal 26).

The shift left (<<) operator accepts two operands. It shifts the left-hand operand by the number of shift units to the left as specified by the right-hand operand. Excess shift units (bits, hexadecimal digits or octets) are discarded. For each shift unit shifted to the left, a zero ('0'B, '0'H, or '00'O determined according to the type of the left-hand operand) is inserted from the right-hand side of the left operand.

The shift right (>>) operator accepts two operands. It shifts the left-hand operand by the number of shift units to the right as specified by the right-hand operand. Excess shift units (bits, hexadecimal digits or octets) are discarded. For each shift unit shifted to the right, a zero ('0'B, '0'H, or '00'O determined according to the type of the left-hand operand) is inserted from the left-hand side of the left operand.

ATDL concrete textual grammar

488 ShiftExpression ::= BitwiseExpression [ShiftOp BitwiseExpression]

500 ShiftOp ::= "<<" | ">>" | "<@" | "@>"

AODL syntax definition

111 ShiftExpression ::= BitwiseExpression [ShiftOp BitwiseExpression]

112 ShiftOp ::= “<<“ | “>>” | "<@" | "@>"

9.8. Rotate operators

The predefined rotate operators perform the rotate left (<@) and rotate right (@>) operators. Their left-hand operand shall be of type **bitstring**, **hexstring**, **octetstring**, **charstring** or **wide charstring**. Their right-hand operand shall be of type **cardinal**. The result type of these operators shall be the same as that of the left operand.

The rotate operators behave differently based upon the type of their left-hand operand. If the type of the left-hand operand is:

a) **bitstring** then the rotate unit applied is 1 bit;

b) **hexstring** then the rotate unit applied is 1 hexadecimal digit;

c) **octetstring** then the rotate unit applied is 1 octet;

d) **charstring** or **wide charstring** then the rotate unit applied is one character;

The rotate left (<@) operator accepts two operands. It rotates the left-hand operand by the number of shift units to the left as specified by the right-hand operand. Excess shift units (bits, hexadecimal digits, octets, or characters) are re-inserted into the left-hand operand from its right-hand side.

The rotate right (@>) operator accepts two operands. It rotates the left-hand operand by the number of shift units to the right as specified by the right-hand operand. Excess shift units (bits, hexadecimal digits, octets, or characters) are re-inserted into the left-hand operand from its left-hand side.

9.9. Primary expressions

AODL syntax definition

113 Primary ::= Value | "(" Expression ")"

9.10. Typecast expressions

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression’s type. For example, **integer**(“A”) casts the character *A* as an integer.

If it is necessary to convert values of one type to values of another type, where the types are not derived from the same root type, then typecast expressions shall be used.

Typecast expressions provide functionalities similar to TTCN-3 *predefined operations.* TTCN-3 now supports a number of *predefined operations*: **int2char**, **char2int**, **int2unichar**, **unichar2int, bit2int**, **hex2int**, **oct2int**, **str2int**, **int2bit, int2hex, int2oct,** and **int2str** [1].

ATDL concrete textual grammar

495 CastExpression ::= Type “(“ SingleExpression “)”

AODL syntax definition

114 CastExpression ::= Type “(“ SingleExpression “)”

10. ATDL/AODL types and values

ATDL is a “strongly typed” language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing.

ATDL supports a number of *predefined* basic types. These basic types include ones normally associated with a programming language, such as **integer**, **cardinal**, **boolean** and string types, as well as some ATDL specific ones such as **objid** and **verdicttype**. It also allows the user to construct his own types from the predefined types. Structured types such as **sequence of** types, **sequence** types and **enumerated** types can be constructed from these basic types.

Table 13: Overview of ATDL/AODL types

Class of type	Keyword	Sub-type	AODL Used
Simple basic generic types	**integer**	range, list, length	yes
	cardinal	range, list, length	yes
	real	range, list, length	yes
Simple basic concrete types	**char**	range, list	yes
	wide char	range, list	yes
	float	range, list	yes
	boolean	list	yes
	objid	list	yes
	verdicttype	list	no
Basic string types	**bitstring**	list, length	yes
	hexstring	list, length	yes
	octetstring	list, length	yes
	charstring	range, list, length	yes
	wide charstring	range, list, length	yes
Structured types	**sequence**	list	yes
	sequence of	list, length	yes
	set	list	yes
	set of	list, length	yes
	enumerated	list	yes
	choice	list	yes
Special generic types	**variant**	list	yes
Native type	**external**	no	yes
Default types	**default**	no	no

The ATDL special type **default** may be used for the default handling (see clause 19.7).

The ATDL types are summarized in Table 13.

ATDL concrete textual grammar

24 TypeDef ::= “type” TypeIdentifier “::=” Type

25 TypeIdentifier ::= Identifier

308 Type ::= BasicType | ConstrainedType | StructuredType | ReferencedType | RestrictedType

325 Value ::= LiteralValue | StringValue | ReferencedValue | TemplateValue&Attributes

326 LiteralValue ::= BooleanValue | ChoiceValue | IntegerValue | FloatingPointLiteral | CharValue
| ObjectIdentifierValue | EnumeratedValue | VerdictValue | NullValue

349 ReferencedValue ::= [GlobalModuleId Dot] ValueReference [ExtendedFieldReference]

350 ValueReference ::= ConstIdentifier | ValueParIdentifier | ModuleParIdentifier | VarIdentifier

374 NullValue ::= “null”

AODL syntax definition

115 TypeDef ::= “type” TypeIdentifier “::=” Type

116 TypeIdentifier ::= Identifier

117 Type ::= BasicType | ConstrainedType | StructuredType | ReferencedType | “variant”

118 Value ::= LiteralValue | StringValue | ReferencedValue

119 LiteralValue ::= BooleanValue | ChoiceValue | IntegerValue | FloatingPointLiteral | CharValue
| ObjectIdentifierValue | EnumeratedValue

120 ReferencedValue ::= [GlobalModuleId Dot] ValueReference [ExtendedFieldReference]

121 ValueReference ::= ConstIdentifier | ValueParIdentifier

10.1. Simple generic types

ATDL supports simple *generic* and *concrete* types.

ATDL is a *strongly typed* language, it is mandatory that every *generic* type has a fixed length that is known at compile time (called early or static binding,). Fixed-length types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations. Strong typing helps detect errors at compile time. Only template methods may have generic type as parameters or generic return types. If a procedure that is not template contains a generic type, then a compile-time error occurs.

Simple generic types include integer, cardinal and real types. For this purpose, data objects of *integer* type are assumed to be defined as signed **sequence of** {byte}, data objects of *cardinal* type are assumed to be defined as unsigned **sequence of** {byte}, data objects of *real* type are assumed to be defined as signed **sequence of** { floating-point byte}.

ATDL supports both Late binding at runtime, and Early binding at compile-time. Early binding is highly recommended, both for compile-time type checking and because it is much faster than late (dynamic) binding. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the test system to set up calls before they are invoked.

The generic type without any type constraints is called raw generic type. Raw generic types are only permitted within virtual class declarations. The exception types cannot be generic: Generic exception or error types are disallowed because the exception handling mechanism is a runtime mechanism and the underlying system does not know anything about generic types.

10.2. Basic types and values

Simple types, which include *ordinal* types, *string* types, *float* types and *real* types.

ATDL concrete textual grammar

309 BasicType ::= OrdinalType | “float” | RealType | StringType | “objid” | VerdictType | BooleanType

AODL syntax definition

122 BasicType ::= OrdinalType | “float” | RealType | StringType | “objid” | “external” | BooleanType

123 ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]

124 ExtendedFieldReference ::= { (Dot StructFieldIdentifier | ArrayOrBitRef) }+

125 TypeReference ::= ParameterizedType | TypeIdentifier

126 ParameterizedType ::= TypeIdentifier TypeActualParList

10.2.1. Integral types and values

An *integer* type represents a subset of the whole numbers. The generic integral types are **integer** and **cardinal**. A **integer** type denotes a type with distinguished values which are the positive and negative whole numbers, including zero. A **cardinal** type denotes a type with distinguished values which are the positive whole numbers, including zero.

An *integer literal* may be expressed in decimal. A decimal numeral is either the single ASCII character 0, representing the integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9, representing a positive integer.

ATDL concrete textual grammar

313 IntegerType ::= (“integer” | “cardinal”) [LengthRestriction]

330 IntegerValue ::= Number

351 Number ::= (NonZeroNum {Num}*) | “0”

352 NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

AODL syntax definition

127 IntegerType ::= (“integer” | “cardinal”) [LengthRestriction]

/* STATIC SEMANTICS - The length restriction may only be omitted when used as generic-type template parameter or return type associated with a template operation declaration. */

128 IntegerValue ::= Number

129 Number ::= (NonZeroNum {Num}*) | “0”

130 NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

10.2.2. Character types and values

The fundamental character types are **char** and **wide char**. **Char** type is a type whose distinguished values are characters of the version of ISO/IEC 646 [6] complying to the International Reference Version (IRV) as specified in clause 8.2 of ISO/IEC 646 [6]. **Wide char** type is a type whose distinguished values are single characters from ISO/IEC 10646 [7].

Values of the type **char** may be given enclosed in single quotes (‘) and followed by the pair of characters (’) and an optional type suffix ‘**C**’, or calculated using a predefined conversion function with the cardinal value of their encoding as argument.

Relational operators equality (==)and non-equality (!=) can be used to compare values of type **char**.

Values of the type **wide char** may be given enclosed in single quotes (‘)and followed by the pair of characters (’) and an optional type suffix ‘**C’**, or calculated using a predefined conversion function with the cardinal value of their encoding as argument, or by translating the ASCII characters \u followed by four hexadecimal digits to the Unicode character with the indicated hexadecimal value according to ISO/IEC 10646 [7].

Relational operators equality (==) and non-equality (!=) can be used to compare values of type **wide char**.

ATDL concrete textual grammar

316 CharType ::= “char”

317 WideChar ::= “wide” “char”

355 CharValue ::= “‘” Char “’” [“C”]

363 Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For charstring a character from the character set defined in ISO/IEC 646. For wide charstring a character from any character set defined in ISO/IEC 10646 */

AODL syntax definition

131 CharType ::= “char”

132 WideChar ::= “wide” “char”

133 CharValue ::= “‘” Char “’” [“C”]

134 Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For charstring a character from the character set defined in ISO/IEC 646. For wide charstring a character from any character set defined in ISO/IEC 10646 */

10.2.3. Real types and values

A real type defines a set of numbers that can be represented with floating-point notation. The ATDL types float and real are IEEE 754 single-precision and ASN.1 REAL type, respectively.

The **float** type represents IEEE 32-bit single-precision floating type. See *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Standard 754-1985 [27], for a detailed specification. Use this type whenever possible, since it results in the best performance for the underlying system.

Floating point numbers are represented as: $<mantissa> \times <base>^{<exponent>}$, and a type suffix.

Where *<mantissa>* a positive or negative integer, *<base>* a positive integer (in most cases 2, 10 or 16) and *<exponent>* a positive or negative integer. A floating-point literal is of type float if it is suffixed with an ASCII letter F or f; otherwise its type is real and it can optionally be suffixed with an ASCII letter R or r.

The floating-point number representation is restricted to a base with the value of 10. Floating point values can be expressed by using either:

- the normal notation with a dot in a sequence of numbers like, 1.23 (which represents $123*10^{-2}$), 2.783 (i.e. 2783×10^{-3}) or -123.456789 (which represents $-123456789 \times 10^{-6}$); or
- by two numbers separated by E where the first number specifies the mantissa and the second specifies the exponent, for example 12.3E4 (which represents 12.3×10^{4}) or -12.3E-4 (which represents -12.3×10^{-4}).

The IEEE 754 standard includes not only positive and negative sign-magnitude numbers, but also positive and negative zeros, positive and negative *infinities*, and a special *Not-a-Number* (hereafter abbreviated NaN). The NaN value is used to represent the result of certain operations such as dividing zero by zero.

The largest positive finite float literal is 3.40282347e+38f. The smallest positive finite nonzero literal of type float is 1.40239846e-45f. The type real in ATDL can model arbitrarily long but finite decimals. A compile-time error occurs if a nonzero fix-length floating-point literal is too large, so that on rounded conversion to its internal representation it becomes an IEEE 754 infinity. An ATDL program can represent infinities without producing a compile-time error by using the predefined keywords “infinity” and “-infinity”.

A compile-time error occurs if a nonzero fix-length floating-point literal is too small, so that, on rounded conversion to its internal representation, it becomes a zero. A compile-time error does not occur if a nonzero floating-point literal has a small value that, on rounded conversion to its internal representation, becomes a nonzero denormalized number.

ATDL concrete textual grammar

315 RealType ::= “real” [LengthRestriction]

344 FloatingPointLiteral ::= (FloatDotNotation | FloatENotation) [FloatTypeSuffix]

345 FloatDotNotation ::= Number Dot DecimalNumber

346 FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number

347 Exponential ::= “e” | “E”

348 FloatTypeSuffix ::= “f” | “F” | “r” | “R”

353 DecimalNumber ::= {Num}*

354 Num ::= “0” | NonZeroNum

AODL syntax definition

135 RealType ::= “real” [LengthRestriction]

/* STATIC SEMANTICS - The length restriction may only be omitted when used as generic-type template parameter or return type associated with a template operation declaration. */

136 FloatingPointLiteral ::= (FloatDotNotation | FloatENotation) [FloatTypeSuffix]

137 FloatDotNotation ::= Number Dot DecimalNumber

138 FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number

139 Exponential ::= “e” | “E”

140 FloatTypeSuffix ::= “f” | “F” | “r” | “R”

141 DecimalNumber ::= {Num}*

142 Num ::= “0” | NonZeroNum

10.2.4. Boolean type and value

A **boolean** type denotes a type consisting of two distinguished values.

Values of boolean type shall be denoted by **true** and **false**.

ATDL concrete textual grammar

314 BooleanType ::= “boolean”

328 BooleanValue ::= "true" | "false"

AODL syntax definition

143 BooleanType ::= “boolean”

144 BooleanValue ::= “true” | “false”

10.2.5. Objid type and values

A **objid** type denotes a type whose distinguished values are the set of all object identifiers conforming to clause 6.2 of ITU-T Recommendation X.660 [16]. Hyphens in object identifiers are replaced with underscores.

ATDL concrete textual grammar

331 ObjectIdentifierValue ::= “{“ ObjIdComponentList “}”

/* STATIC SEMANTICS - ReferencedValue shall be of type object Identifier */

332 ObjectIdComponentList ::= {ObjIdComponent}+

333 ObjIdComponent ::= NameForm | NumberForm | NameAndNumberForm

334 NumberForm ::= Number | ReferencedValue

/* STATIC SEMANTICS - ReferencedValue shall be of type integer and hava a non negative Value */

335 NameAndNumberForm ::= Identifier NumberForm

336 NameForm ::= Identifier

AODL syntax definition

145 ObjectIdentifierValue ::= “{“ ObjIdComponentList “}”

/* STATIC SEMANTICS - ReferencedValue shall be of type object Identifier */

146 ObjectIdComponentList ::= {ObjIdComponent}+

147 ObjIdComponent ::= NameForm | NumberForm | NameAndNumberForm

148 NumberForm ::= Number | ReferencedValue

/* STATIC SEMANTICS - ReferencedValue shall be of type integer and have a non negative Value */

149 NameAndNumberForm ::= Identifier NumberForm

150 NameForm ::= Identifier

10.2.6. Ordinal types

Ordinal types include *integer*, *cardinal, char*, *wide char, enumerated*, and *sub-range* types. An ordinal type defines an ordered set of values in which each value except the first has a unique *predecessor* and each value except the last has a unique *successo*r. Further, each value has an *ordinalit*y, which determines the ordering of the type. For integer types, the ordinality of a value is the value itself; for all other ordinal types except sub-ranges, the first value has ordinality 0, the next value has ordinality 1, and so forth. If a value has ordinality n, its predecessor has ordinality n–1 and its successor has ordinality n+1.

Several predefined functions operate on ordinal values and type identifiers. For example, *UpperBoundary*(*char*) returns 127 because the highest value of ISO/IEC 646 type *char* is 127.

ATDL concrete textual grammar

311 OrdinalType ::= IntegerType | CharType | WideChar | EnumType

AODL syntax definition

151 OrdinalType ::= IntegerType | CharType | WideChar | EnumType

10.2.7. AODL specific native types

AODL provides a declaration for use by object adapters to define an opaque type whose representation is specified by the language mapping for that object adapter. The syntax is:

TypeDef ::= TypeIdentifier "::=" **external**

A native type may be used to define operation parameters and results. However, there is no requirement that values of the type be permitted in remote invocations, either directly or as a component of a structured type.

The native type declaration is provided specifically for use in object adapter interfaces, which require parameters whose values are concrete representations of object implementation instances. It is strongly recommended that it not be used in service or application interfaces. The native type declaration allows object adapters to define new primitive types without requiring changes to the AODL language or to AODL compiler.

10.2.8. ATDL specific verdict types

Verdict type is a type for use with test verdicts consisting of 5 distinguished values.

Values of **verdicttype** shall be denoted by **pass**, **fail**, **inconc**, **none** and **error.**

ATDL concrete textual grammar

337 VerdictValue ::= "pass" | "fail" | "inconc" | "none" | "error"

10.2.9. Basic string types and values

ATDL supports the following basic string types:

a) **bitstring**: a type whose distinguished values are the ordered sequences of zero, one, or more bits. Values of type **bitstring** shall be denoted by an arbitrary number (possibly zero) of the bit digits: 0 1, preceded by a single quote (') and followed by the pair of characters 'B.

b) **hexstring:** a type whose distinguished values are the ordered sequences of zero, one, or more hexadecimal digits, each corresponding to an ordered sequence of four bits.

Values of type **hexstring** shall be denoted by an arbitrary number (possibly zero) of the hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F, preceded by a single quote (') and followed

by the pair of characters 'H; each hexadecimal digit is used todenote the value of a semi-octet using a hexadecimal representation.

c) **octetstring:** a type whose distinguished values are the ordered sequences of zero or a positive even number of hexadecimal digits (every pair of digits corresponding to an ordered sequence of eight bits).

Values of type **octetstring** shall be denoted by an arbitrary, but even, number (possibly zero) of the hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F, preceded by a single quote (') and followed by the pair of characters 'O; each hexadecimal digit is used to denote the value of a semi-octet using a hexadecimal representation.

ATDL concrete textual grammar

312 StringType ::= “bitstring” | CharStringType | WideCharString | “octetstring” | “hexstring”

327 StringValue ::= Bstring | CharStringValue | Ostring | Hstring

356 Bstring ::= " ’ " {Bin | Wildcard}* " ’ " “B”

357 Bin ::= "0" | "1"

358 Hstring ::= " ’ " {Hex | Wildcard}* " ’ " “H”

359 Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"

360 Ostring ::= " ’ " {Oct | Wildcard}* " ’ " “O”

361 Oct ::= Hex Hex

AODL syntax definition

152 StringType ::= “bitstring” | CharStringType | WideCharString | “octetstring” | “hexstring”

153 StringValue ::= Bstring | CharStringValue | Ostring | Hstring

154 Bstring ::= " ’ " {Bin}* " ’ " “B”

155 Bin ::= “0” | “1”

156 Hstring ::= " ’ " {Hex}* " ’ " “H”

157 Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"

158 Ostring ::= " ’ " {Oct}* " ’ " “O”

159 Oct ::= Hex Hex

10.2.9.1. Char string types

A char string represents a sequence of characters.

a) **charstring:** are types whose distinguished values are zero, one, or more characters of the version of ISO/IEC 646 [6] complying to the International Reference Version (IRV) as specified in clause 8.2 of ISO/IEC 646 [6].

b) **wide charstring:** The character string type preceded by the keyword **wide** denotes types whose distinguished values are zero, one, or two characters from ISO/IEC 10646 [7].

Values of **charstring** type shall be denoted by an arbitrary number (possibly zero) of characters from the relevant character set, preceded and followed by double quote (").

Multibyte character sets — especially double-byte character sets (DBCS) — are widely used for Asian languages. In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called *wide characters* and *wide character strings*. The first 256 Unicode characters map to the ASCII character set.

ATDL supports single-byte and Unicode characters and character strings through the c*har*, *wide char*, *charstring*, and *wide charstring* types.

ATDL implementations first should recognize *Unicode escapes* in their input, translating the ASCII characters \u followed by four hexadecimal digits to the Unicode character with the indicated hexadecimal value, and passing all other characters unchanged.

If an eligible \ is not followed by u, then it is treated as a *RawInputCharacter* and remains part of the escaped Unicode stream. If an eligible \ is followed by u, or more than one u, and the last u is not followed by four hexadecimal digits, then a compile-time error occurs.

ATDL implementations next should divide the sequence of Unicode input characters into lines by recognizing *line terminators*, that is, the Carriege Return and Line Feed characters. Neither of the characters Carriege Return and Line Feed is ever considered to be an *InputCharacter*; each is recognized as constituting a *LineTerminator.*

Certain characters can be represented in character by escape sequences like \n (newline); these sequences look like two characters, but represent only one. The complete set of escape sequences is shown in Table 14.

Table 14: List of ATDL escape sequences

Character	Description	Character	Description
\r	carriage return (see [19])	\n	newline (see ISO/IEC 6429 [19])
\”	double quote	\t	horizontal tab (see [19])
\\	backslash	\’	single quote

ATDL concrete textual grammar

318 CharStringType ::= “charstring”

319 WideCharString ::= “wide” “charstring”

339 CharStringValue ::= Cstring | Quadruple

340 UnicodeInputCharacter ::= UnicodeEscape | Char

341 UnicodeEscape ::= \ UnicodeMarker Hex Hex Hex Hex

342 UnicodeMarker ::= {u}+

343 InputCharacter ::= UnicodeInputCharacter

362 Cstring ::= " " " {InputCharacter | Wildcard | “\”}* " " "

AODL syntax definition

160 CharStringType ::= “charstring”

161 WideCharString ::= “wide” “charstring”

162 CharStringValue ::= Cstring | Quadruple

163 UnicodeInputCharacter ::= UnicodeEscape | Char

164 UnicodeEscape ::= \ UnicodeMarker Hex Hex Hex Hex

165 UnicodeMarker ::= {u}+

166 InputCharacter ::= UnicodeInputCharacter

/* STATIC SEMANTICS - The InputCharacter shall not be Carriage Return or Line Feed character */

167 Cstring ::= " " " {InputCharacter | “\”}* " " "

10.3. Sub-typing of basic types

User-defined types shall be denoted by the keyword **type**. With user-defined types it is possible to create sub-types (such as lists, ranges and length restrictions) on simple basic and basic string types according to Table 13.

ATDL concrete textual grammar

39 ConstrainedType ::= BasicType [SubTypeSpec]

40 SubTypeSpec ::= SimpleValueSet | LengthRestriction

AODL syntax definition

168 ConstrainedType ::= BasicType [SubTypeSpec]

169 SubTypeSpec ::= | LengthRestriction
/* STATIC SEMANTICS - The value shall be of the same type as the field being subtyped */

10.3.1. Value Set constructors

A value set is a collection of values of the same type. The values have no inherent order, nor is it meaningful for a value to be included twice in a value set.

A value set constructor denotes a set of values. For example,

(5, 6, 7, 8)

denotes the value set whose members are 5, 6, 7, and 8. The value set constructor

(5..8)

could also denote the same value set.

The **in** operator tests value set membership:

if ['a' **in** MyValueSet] { do something } ;

Value sets can be used in several kinds of statements:

a) a value set denotes a **set of** type value;

b) can be used in control flow statements;

c) used in a receiving operation defines a data template against which an incoming message is to be matched (see clause 15.7.3).

ATDL concrete textual grammar

41 SimpleValueSet ::= SimpleValueList | IntegerRange

AODL syntax definition

170 SimpleValueSet ::= SimpleValueList | IntegerRange

10.3.1.1. Value Set operators

The following operators take value sets as operands.

Table 15: Value Set operators

Operator	Operation	Operand types	Result type
+	union	value set	value set
-	difference	value set	value set
*	intersection	value set	value set
complement	complement	value set	value set

A value set *O* is in *X* + *Y* if and only if *O* is in *X* or *Y* (or both). *O* is in *X* – *Y* if and only if *O* is in *X* but not in *Y*. *O* is in *X* * *Y* if and only if *O* is in both *X* and *Y*. For example,

(1,2,3)+(2,3,4) == (1,2,3,4)

(1,2,3)*(2,3,4) == (2,3)

10.3.1.2. Lists of values

ATDL/AODL permits the specification of a list of distinguished values of any given type as listed in Table 13. The values in the list shall be of the root type and shall be a true subset of the values defined by the root type. The subtype defined by this list restricts the allowed values of the subtype to those values in the list.

ATDL concrete textual grammar

42 SimpleValueList ::= "(" SingleConstExpression {"," SingleConstExpression}* ")"

AODL syntax definition

171 SimpleValueList ::= "(" SingleConstExpression {"," SingleConstExpression}* ")"

10.3.2. Length restriction

ATDL permits the specification of length restrictions on integer, cardinal and string types. The length boundaries are of different complexity depending on the string type with which they are used. In all cases, these boundaries shall evaluate to *cardinal values* (or derived *cardinal values*).

Type	Units of Length
integer or cardinal	Bytes
real	Bytes
bitstring	Bits
hexstring	Hexadecimal digits
octetstring	Octets
character strings	Characters
sequence of	Elements of its base type
set of	Elements of its base type

Table 16: Units of length used in field length specifications

Table 16 specifies the units of length for different string types.

For the upper bound the keyword **infinity** may also be used to indicate that there is no upper limit for the length. The upper boundary shall be greater than or equal to the lower boundary.

For example:

type MyByte ::= **bitstring** [8]; // Exactly length 8

type MyNibbleOrByte ::= **bitstring** [4..8]; // Minimum length 4, maximum length 8

In the context of templates, length restrictions can also be specified on values of array type, thus limiting the number of their elements.

Length specifications shall not conflict, i.e., a restriction on a type (set of values) that is already restricted shall specify a subrange of values of its base type.

ATDL concrete textual grammar

46 LengthRestriction ::= "[" SingleConstExpression [".." UpperBound] "]"

AODL syntax definition

172 LengthRestriction ::= "[" SingleConstExpression [".." UpperBound] "]"

/* STATIC SEMANTICS - LengthRestriction will resolve to a value of cardinal type. LengthRestriction shall only be used with String types, Integer types or to limit sequence of type */

10.3.3. Subrange type

ATDL permits the specification of a range of values of type *ordinal* and **real** or **float** (or derivations of these types). The subtype defined by this range restricts the allowed values of the subtype to the values in the range including the lower boundary and the upper boundary. In the case of **char** and **wide char** types, the boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g. the given position shall not be

empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range. For example:

type MyIntegerRange ::= Smallint (0 .. 255);

For values of ordinal type (or derivations of these types), it is possible to mix lists and subranges.

ATDL concrete textual grammar

43 IntegerRange ::= "(" LowerBound ".." UpperBound ")"

44 LowerBound ::= SingleConstExpression | Minus "infinity"

45 UpperBound ::= SingleConstExpression | "infinity"

AODL syntax definition

173 IntegerRange ::= "(" LowerBound ".." UpperBound ")"

/* STATIC SEMANTICS - IntegerRange shall only be used with ordinal types */

174 LowerBound ::= SingleConstExpression | Minus "infinity"

175 UpperBound ::= SingleConstExpression | "infinity"

10.3.3.1. Infinite ranges

In order to specify an infinite integer or float range, the keyword **infinity** may be used instead of a value indicating that there is no lower or upper boundary. The upper boundary shall be greater than or equal to the lower boundary.

10.3.3.2. Mixing lists and ranges

For values of type **integer, cardinal, char, wide char, real** and **float** (or derivations of these types) it is possible to mix lists and ranges.

10.4. Structured types and values

The **type** keyword is also used to specify structured types such as **sequence of** types, **sequence** types, and **choice** types.

Values of these types may be given using an explicit assignment notation or a shorthand initializer. It is not allowed to mix the two value notations in the same (immediate) context.

ATDL concrete textual grammar

310 StructuredType ::= SequenceType | SequenceOfType | SetType | SetOfType | ChoiceType

AODL syntax definition

176 StructuredType ::= SequenceType | SequenceOfType | SetType | SetOfType | ChoiceType

10.4.1. Parameterized type

Type parameterization [11] allows dummy type identifiers which act as placeholders for any type. This means that a type can be left open by the ATDL descriptor as long as it is resolvable at compile-time. The actual type is only known when the type parameter is actually used. This is a generalization of the PDU meta-type concept of TTCN-2.

ATDL concrete textual grammar

26 TypeDefFormalParList ::= "(" FormalValuePar {"," FormalValuePar}* ")"

324 TypeActualParList ::= "(" SingleConstExpression {"," SingleConstExpression}* ")"

AODL syntax definition

177 TypeDefFormalParList ::= "(" FormalValuePar {"," FormalValuePar}* ")"

/* STATIC SEMANTICS - FormalValuePar shall resolve to an *in* parameter */

178 TypeActualParList ::= "(" SingleConstExpression {"," SingleConstExpression}* ")"

10.4.2. Sequence type and values

ATDL supports ordered structured types known as **sequence** (also called `struct'). The elements of a sequence type may be any of the base types or user-defined types such as other sequences, or sets. The values of a sequence shall be compatible with the types of the sequence fields. The element identifiers are local to the sequence and shall be unique within the sequence.

type MyRecordType ::= **sequence**
{
field1 **integer**[2]**,**
field2 MyOtherRecordType **optional,**
field3 **charstring**
}

The field identifiers *field1*, *field2*, and *field3* are the *field designators* for *MyRecordType*, and they behave like variables. The *MyRecordType* type declaration, however, does not allocate any memory for the *field1*, *field2*, and *field3* fields; memory is allocated when you instantiate the record. A **sequence** value is assigned on an individual element basis or using an value list.

EXAMPLE 2:

const MyStructValue MyStructType:= {MyIntegerValue, {'11001'B, **true**}, "A string"};

Sequences may be defined with no fields (i.e., as empty structs). For example:

type MyEmptyStruct ::= **sequence** { }

For optional fields it allowed to omit the value using the omit parameter symbol.

Elements of nested sequences are referenced by *StructId.FieldId* pairs.

ATDL concrete textual grammar

27 SequenceType ::= “sequence” [TypeDefFormalParList] “{“ [StructFieldDef {“,” StructFieldDef}*] “}”

28 StructFieldDef ::= StructFieldIdentifier Type [SubTypeSpec] [“optional”]

29 StructFieldIdentifier ::= Identifier

AODL syntax definition

179 SequenceType ::= “sequence” [TypeDefFormalParList] “{“ [StructFieldDef {“,” StructFieldDef}*] “}”

180 StructFieldDef ::= StructFieldIdentifier Type [SubTypeSpec] [“optional”]

181 StructFieldIdentifier ::= Identifier

10.4.2.1. Optional elements in a struct

Optional elements in a **sequence** shall be specified using the **optional** keyword.

10.4.2.2. Sequence constants

To declare a sequence constant, specify the value of each field with the field assignments separated by commas. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the sequence type declaration. A constant that is of sequence type shall not contain variables (including module parameters) as field values, either directly or indirectly.

ATDL concrete textual grammar

480 FieldConstExpressionList ::= FieldExpressionList

473 FieldExpressionSpec ::= FieldReference AssignmentChar Expression

AODL syntax definition

182 FieldConstExpressionList ::= “{“ FieldConstExpressionSpec {“,” FieldConstExpressionSpec}* “}”

183 FieldConstExpressionSpec ::= FieldReference AssignmentChar ConstantExpression

10.4.3. Choice type and values

The constructor **choice** gives the choice (also called `union') between several alternatives:

type Afters ::= **choice** (Cardinal) {cheese [0] **charstring**,
dessert [1] **charstring** }

A choice is denoted by an identifier, which is a word beginning with a lower-case letter. This identifier is not encoded and it makes it possible to build unambiguous abstract values; for this reason, the identifiers of a **choice** type must be distinct. Their names should be as self-explicit as possible so that the role of each alternative could be easily understood.

The **choice** type models two pieces of information: the chosen alternative (the identifier) and the value associated with this alternative. A BER decoder will rely on the received tag to determine the alternative that has been chosen and decode the value according to this alternative. The tags of the **choice** alternatives must therefore be distinct. Indeed, the **choice** type does not exist `as is'; it is only a collection of several types among which one of them is chosen to be encoded with its associated tag. The TagTypec production must be a previously defined ordinal type.

Name scope rules require that the element descriptors in a particular choice be unique. If the TagType is an EnumType, the identifier for the enumeration is in the scope of the choice; as a result, it must be distinct from the element descriptors.

A value of type **choice** features the identifier of the chosen alternative followed by the symbol ":", and a value complying with the type of this alternative. For example:

const mine Afters := dessert:"profiteroles"

The initializer notation for setting values shall not be used for values of **choice** types.

The **optional** keyword shall not be used with choice types.

Concrete textual grammar

31 ChoiceType ::= "choice" "(" OrdinalType ")" "{" ChoiceFieldDef {";" ChoiceFieldDef}* "}"

32 ChoiceFieldDef ::= StructFieldIdentifier TaggedType [SubTypeSpec]

33 TaggedType ::= "[" (SingleConstExpression | "else") "]" Type

329 ChoiceValue ::= StructFieldIdentifer Colon Value

AODL syntax definition

184 ChoiceType ::= "choice" "(" OrdinalType ")" "{" ChoiceFieldDef {";" ChoiceFieldDef}* "}"

185 ChoiceFieldDef ::= StructFieldIdentifier TaggedType [SubTypeSpec]

186 TaggedType ::= "[" (SingleConstExpression | "else") "]" Type

/* STATIC SEMANTICS - The value of the ConstantExpression shall be of the same type as the TagTypeSpec. */

187 ChoiceValue ::= StructFieldIdentifier Colon Value

10.4.4. Set type and values

ATDL supports unordered structured types known as **set**. Set types and values are similar to structs except that the ordering of the **set** fields is not significant.

The field identifiers are local to the set and shall be unique within the set (but do not have to be globally unique).

ATDL concrete textual grammar

30 SetType ::= "set" [TypeDefFormalParList] "{" [StructFieldDef {"," StructFieldDef}*] "}"

AODL syntax definition

188 SetType ::= "set" [TypeDefFormalParList] "{" [StructFieldDef {"," StructFieldDef}*] "}"

10.4.4.1. Optional elements in a set

Optional elements in a **set** shall be specified using the **optional** keyword.

10.4.5. Enumerated type and values

ATDL supports **enumerated** types. Enumerated types are used to model types that take only a distinct named set of values. Such distinct values are called enumerations. Each enumeration shall have an identifier. Operations on enumerated types shall only use these identifiers and are restricted to assignment, equivalence and ordering operators. Enumeration identifiers shall be unique within the enumerated type (but do not have to be globally unique) and consequently visible within the context of the given type only.

Each enumeration may optionally have an assigned integer value, which is defined after the name of the enumeration in parenthesis. Each assigned integer number shall be distinct within a single **enumerated** type. For each enumeration without an assigned integer value, the system successively associates an integer number in the textual order of the enumerations, starting at the left-hand side, beginning with zero, by step 1 and skipping any number occupied in any of the enumerations with a manually assigned value.

The integer value also may be used by the system to encode/decode enumerated values. This, however is outside of the scope of the present document (with the exception that ATDL allows the association of encoding attributes to ATDL items).

ATDL concrete textual grammar

36 EnumType ::= “enumerated” “{“ NamedValue {“,” NamedValue}* “}”

37 NamedValue ::= NamedValueIdentifier [“(“ Number “)”]

38 NamedValueIdentifier ::= Identifier

338 EnumeratedValue ::= NamedValueIdentifier

AODL syntax definition

189 EnumType ::= “enumerated” “{“ NamedValue {“,” NamedValue}* “}”

190 NamedValue ::= NamedValueIdentifier [“(“ Number “)”]

191 NamedValueIdentifier ::= Identifier

192 EnumeratedValue ::= NamedValueIdentifier

10.5. Array type and values

An array (also called `sequence of' type) represents an indexed collection of elements of the same type (called the *base typ*e). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once.

Arrays can be allocated *statically* or *dynamicall*y. Array indexes are expressions which shall evaluate to cardinal values. By default, indexing of ATDL arrays shall start with the digit 0 (zero).

Array dimensions shall be specified using constant expressions which shall evaluate to a cardinal value. Array dimensions may also be specified using ranges. In such cases the lower and upper values of the range define the lower and upper index values. For example,

type MyArrayType ::= **sequence** [10] **of** Smallint ; // is an array of exactly 10 integers
type MyArrayType **record** [0..10] **of** Smallint; // is an array of a maximum of 10 integers

Sequences of struct types allow the possibility to specify multi-dimensional arrays. For example:

// Given

type MyStructType ::= **sequence** { field1 Smallint, field2 MyOtherStruct , field3 **charstring** }
// An array of MyStructType could be
type MyStructArray ::= **sequence** [10] **of** MyStructType;

// A reference to a particular element would look like this

MyStructArray[1].field1 := 1;

A multidimensional array is an array of arrays. For example,

type TMatrix ::= **array** [10] of **array** [50] **of float**;

is equivalent to

type TMatrix ::= **array**[10][50] **of float**;

Whichever way *TMatrix* is declared, it represents an array of 500 float values.

The standard functions *LowerBoundary* and *UpperBoundary* operate on array type identifiers and variables. They return the low and high boundarys of the array's first index type. The standard function *SizeOf* returns the number of elements in the array's first dimension.

ATDL concrete textual grammar

34 SequenceOfType ::= "sequence" [{LengthRestriction}+] "of" Type [SubTypeSpec]

AODL syntax definition

193 SequenceOfType ::= "sequence" [{LengthRestriction}+] "of" Type [SubTypeSpec]

10.5.1. Dynamic arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the *SetLength* function. For example,

var MyFlexibleArray ::= **sequence of float**;

declares a one-dimensional dynamic array of floats. The declaration does not allocate memory for *MyFlexibleArray*. To create the array in memory, call *SetLength*. For example, given the declaration above,

SetLength(MyFlexibleArray, 20);

allocates an array of 20 floats, indexed 0 to 19.

Once a dynamic array has been allocated, you can pass it to the standard functions *SizeOf*, *UpperBoundary*, and *LowerBoundary*. *SizeOf* returns the number of elements in the array, *UpperBoundary* returns the array's highest index (that is, *SizeOf*–1), and *LowerBoundary* returns 0. In the case of a zero-length array, *UpperBoundary* returns –1 (with the anomalous consequence that *UpperBoundary* < *LowerBoundary*).

10.5.2. Array constants

To define a multidimensional array constant, enclose the values of each dimension in a separate set of curly braces, separated by commas. For example,

type TCube ::= **sequence** [2][2][2] **of** Smallint;
const Maze TCube := {{{0, 1}, {2, 3}}, {{4, 5}, {6,7}}};

When the value list notation is used, the first value in the list is assigned to the first element, the second list value is assigned to the second element etc. No empty assignment is allowed (e.g. two commas, the second immediately following the first or only with white space between them), elements to be left out from the assignment shall be explicitly skipped or omitted in the list.

ATDL concrete textual grammar

481 ArrayConstExpression ::= "{" [ConstantExpression {"," ConstantExpression}*] "}"

AODL syntax definition

194 ArrayConstExpression ::= "{" [ConstantExpression {"," ConstantExpression}*] "}"

10.6. Sets of types

ATDL supports the specification of sequences and sets whose elements are all of the same type. These are denoted using the keyword **of**. These sequences and sets do not have element identifiers and can be considered similar to an ordered array and an unordered array respectively.

The value notation for **sequence of** and **set of** shall be a value list notation or an indexed notation for an individual element.

When the value list notation is used, the first value in the list is assigned to the first element, the second list value is assigned to the second element etc. No empty assignment is allowed (e.g. two commas, the second immediately following the first or only with white space between them), elements to be left out from the assignment shall be explicitly skipped or omitted in the list.

ATDL concrete textual grammar

35 SetOfType ::= “set” [{LengthRestriction}+] “of” Type [SubTypeSpec]

AODL syntax definition

195 SetOfType ::= “set” [{LengthRestriction}+] “of” Type [SubTypeSpec]

10.7. Variant types

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type variant, which represent values that can change type at runtime. Variants, as they are called, offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time.

Only class templates (§21.1) or method templates may have generic type parameters. Class and method templates are a wonderful feature of ATDL. In many ordinary object-oriented languages, one cannot create a type-safe container. Type safe containers are not the only good thing about class templates. Class templates in ATDL are a very nice way of achieving static polymorphism. Although it is more typical in ordinary object-oriented languages to gain this kind of polymorphism using abstract base classes; there are some distinct advantages to using class templates. For example, there is no virtual overhead. i.e. no extra time or memory is spent managing the dynamic binding of normal virtual functions.

10.8. Changes to ASN.1

1) **Addition of ‘underscore’ to ASN.1 identifiers**: AODL extends the definition of identifier name to allow the use of underscores as well as hyphens.

AODL syntax definition

196 Identifier ::= Alpha {AlphaNum | “_”}*

197 Alpha ::= UpperAlpha | LowerAlpha

198 AlphaNum ::= Alpha | Num

199 UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

200 LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

2) **Alternative syntax for ASN.1 keywords**: ASN.1 currently defines the following base types with space separators between the two keywords: OCTET STRING, BIT STRING, CHARACTER STRING, OBJECT IDENTIFIER. AODL extends the ASN.1 syntax to include an alternative set of keywords containing no space characters.

The consequence of the proposed change is that AODL modules conforming to the ITU-T Recommendation X.680 [ASN1] series specifications can be written in (or transformed too) a syntax which can be directly used within other high level specification languages.

10.9. Miscellaneous productions

201 Dot ::= "."

202 Minus ::= “-”

203 SemiColon ::= ";"

204 Colon ::= “:”

205 AssignmentChar ::= “:=”

10.10. Pre-defined ATDL/AODL types

10.10.1. Useful simple basic types

The pre-defined integer types are *Integer* and *Cardinal*; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. In general, arithmetic operations on integers return a value of type *Integer* which is equivalent to the 32-bit *Longint*. The pre-defined floating-point types are float and Double, representing the single-precision 32-bit and double-precision 64-bit format IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985.

The values of the integer types are integers in the following ranges:

- For *Integer*, from –2147483648 to 2147483647, inclusive
- For *Cardinal*, from 0 to 4294967295, inclusive

10.10.1.1.Signed and unsigned short byte integers

These types supports integer values of the range from –128 to 127 for the signed and from 0 to 255 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on a single byte within the system independently from the actual representation form used.

Type definitions for these types are:

type Shortint ::= **integer** [1] **with** { **variant** "8 bit" };

type Byte ::= **cardinal** [1] **with** { **variant** "unsigned 8 bit" };

10.10.1.2.Signed and unsigned small integers

These types support integer values of the range from -32768 to 32767 for the signed and from 0 to 65535 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on two bytes within the system independently from the actual representation form used.

Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the current document.

Type definitions for these types are:

type Smallint ::= **integer** [2] **with** { **variant** "16 bit" };

type Word ::= **cardinal** [2] **with** { **variant** "unsigned 16 bit" };

10.10.1.3. Signed and unsigned long integers

These types support integer values of the range from -2147483648 to 2147483647 for the signed and from 0 to 4294967295 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on four bytes within the system independently from the actual representation form used.

Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the current document.

Type definitions for these types are:

type Longint ::= **integer** [4] **with** { **variant** "32 bit" };

type Longword ::= **cardinal** [4] **with** { **variant** "unsigned 32 bit" };

10.10.1.4. Signed and unsigned long long integers

These types support integer values of the range from -9223372036854775808 to 9223372036854775807 for the signed and from 0 to 18446744073709551615 for the unsigned type. The value notation for these types are the same as the value notation for the integer type. Values of these types shall be encoded and decoded as they were represented on eight bytes within the system independently from the actual representation form used.

Encoding of values of these types may be the same or may differ from each other and from the encoding of the integer type (the root type of these useful types) depending on the actual encoding rules used. Details of encoding rules are out of the scope of the current document.

Type definitions for these types are:

type Longlongint ::= **integer** [8] **with** { **variant** "64 bit" };

type Longlongword ::= **cardinal** [8] **with** { **variant** "unsigned 64 bit" };

10.10.1.5. IEEE 754 floats

These types support the ANSI/IEEE Standard 754 [27] for binary floating-point arithmetic. The type IEEE 754 float supports floating-point numbers with base 10, exponent of size 8, mantissa of size 23 and a sign bit. The type IEEE 754 double supports floating-point numbers with base 10, exponent of size 11, mantissa of size 52 and a sign bit. The type IEEE 754 extfloat supports floating-point numbers with base 10, minimal exponent of size 11, minimal mantissa of size 32 and a sign bit. The type IEEE 754 extdouble supports floating-point numbers with base 10, minimal exponent of size 15, minimal mantissa of size 64 and a sign bit.

Values of these types shall be encoded and decoded according to the IEEE 754 definitions. The value notation for these types are the same as the value notation for the float type (base 10).

Precise encoding of values of this type depends on the actual encoding rules used. Details of encoding rules are out of the scope of the current document.

Type definitions for these types are:

type IEEE754float ::= **float with** { **variant** "IEEE754 float" };

type Double ::= **real** [8] **with** { **variant** "IEEE754 double" };

type IEEE754extfloat ::= **real** [6] **with** { **variant** "IEEE754 extended float" };

type IEEE754extdouble ::= **real** [10] **with** { **variant** "IEEE754 extended double" };

10.10.2. Useful character string types

10.10.2.1. UTF-8 character string "utf8string"

This type supports the whole character set of the ATDL type **wide charstring** (see clause 10.2.9). Its distinguished values are zero, one, or more characters from this set. Values of this type has entirely (e.g. each character of the value individually) be encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [7]. The value notation for this type is the same as the value notation for the **wide charstring** type.

The type definition for this type is:

type utf8string **wide charstring** ::= **with** { **variant** "UTF-8" };

10.10.2.2. BMP character string "bmpstring"

This type supports the Basic Multilingual Plane (BMP) character set of ISO/IEC 10646 [7]. The BMP represents all characters of plane 00 of group 00 of the Universal Multiple-octet coded Character Set. Its distinguished values are zero, one, or more characters from the BMP. Values of this type have entirely (e.g. each character of the value individually) be encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [7]). The value notation for this type is the same as the value notation for the **wide charstring** type.

The type "bmpstring" supports a subset of the TTCN-3 type **universal charstring.**

The type definition for this type is:

type bmpstring ::= **wide charstring** ('\u0000'.. 'uFFFF') **with** { **variant** "UCS-2" };

10.10.2.3. ISO/IEC 8859 character string "iso8859string"

This type supports all characters in all alphabets defined in the multiparty standard ISO/IEC 8859 [12]. Its distinguished values are zero, one, or more characters from the ISO/IEC 8859 character set. Values of this type has entirely (e.g. each character of the value individually) be encoded and decoded according to the coded representation as specified in ISO/IEC 8859 (an 8-bit coding). The value notation for this type is the same as the value notation for the **wide charstring** type.

The type "iso8859string" supports a subset of the ATDL type **wide charstring**.

In each ISO/IEC 8859 alphabet the lower part of the character set table (positions 02/00 to 07/14) is compatible with the ISO/IEC 646 [6] character set. Hence all extra language specific characters are defined for the upper part of the character table only (positions 10/00 to 15/15). As the "iso8859string" type is defined as a subset of the TTCN-3 type wide charstring, any coded character representation of any ISO/IEC 8859 alphabets can be mapped into an equivalent character (a character with the same coded representation when encoded on 8 bits) from the Basic Latin or Latin-1 Supplement character tables of ISO/IEC 10646 [7].

The type definition for this type is:

type iso8859string ::= **wide charstring** ('\u000' .. '\u00FF') **with** { **variant** "8 bit" };

11. Modules

The principal building blocks of ATDL are modules. For example, a module may define a fully executable test suite or just a library. A module consists of a (optional) definitions part, and a (optional) module control part.

Concrete textual grammar

1 ATDL_Module ::=ModuleHeading "{" [ModuleDefinitionsPart] [ModuleControlPart] "}"

2 ModuleHeading ::= "module" ATDL_ModuleId [ModuleParList]

11.1. Module diagram

A module is shown as a large rectangle with a small rectangle(a "tab") attached on one corner. The name of the module may be placed within the tab.

The ATDL module defines a test suite and the associated collection of statement diagrams, which again define traces of test events. A module may contain a collection of referenced definitions, together with any declarations needed for the test suite.

An ATDL system cannot usually be described easily as a single independent piece of text or on a single diagram. The language therefore supports the partitioning of the specification and use of ATDL from elsewhere.

Concrete graphical grammar

11 <module diagram> ::= <frame symbol> **contains**
(ModuleHeading {{<module text area>}*
{<group diagram>}* {<group reference area>}*
<component interaction area> } **set**)
[**is_followed_by** <control part area>]

12 <module text area> ::= <text symbol> **contains**
{(SupporingDef | ImportDef | ExtFunctionDef) [SemiColon]}*

13 <group reference area> ::= <reference symbol> **contains** GroupHeading

15 <control part area> ::= <reference symbol> **contains** ("control" ATDL_ModuleId)

11.2. Naming of modules

Module names are of the form of an ATDL identifier followed by an optional object identifier.

NOTE: The module identifier is the informal text name of the module.

Concrete textual grammar

3 ATDL_ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]

4 DefinitiveIdentifier ::= Dot ObjectIdType "{" DefinitiveObjIdComponentList "}"

5 DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+

6 DefinitiveObjIdComponent ::= NameForm | Number | NameForm "(" Number ")"

7 ModuleIdentifier ::= Identifier

11.3. Module parameters

The **module** parameter list defines a set of values that are supplied by the test environment at run-time. During test execution these values shall be treated as constants.

Concrete textual grammar

8 ModuleParList ::= "(" ModulePar {"," ModulePar}* ")"

9 ModulePar ::= ["in"] ModuleParIdentifier Type [":=" ConstantExpression]

10 ModuleParIdentifier ::= Identifier

11.3.1. Default values for module parameters

It is allowed to specify default values for module parameters. This shall be done by an assignment in the module parameter list. A default value can be a literal value only and can merely be assigned at the place of the declaration of the parameter. If the test system does not provide an actual run-time value for the given parameter, the default value shall be used during test execution, otherwise the actual value provided by the test system.

11.4. Module definitions part

The module definitions part specifies the top-level definitions of the module and may import identifiers from other modules. Scope rules for declarations made in the module definition part and imported declarations are given in clause 5.4. Those language elements which may be defined in an ATDL module are listed in Table 3. The module definitions may be imported by other modules.

NOTE: ATDL does not support the declaration of variables in the module definitions part. This means that global variables cannot be defined in ATDL. However variables defined in a test component may be used by all test cases, functions etc. running on that component and variables defined in the control part provide the ability to keep their values independently of test case execution.

Concrete textual grammar

21 ModuleDefinitionsPart ::= {ModuleDefinition [SemiColon] }+

22 ModuleDefinition ::= (SupportingDef | TemplateDef | ImportDef | GroupDef | InterfaceDef
| FunctionDef | TestcaseDef | AltstepDef | ExtFunctionDef | ClassDef
| CoclassDef | ThreadClassDef | ClassTemplateDef) [WithStatement]

23 SupportingDef ::= TypeDef | ConstDef | ExceptionDef

11.5. Module control part

The module control part describes the execution order (possibly repetitious) of the actual test cases. A test case shall be defined in the module definitions part and called in the control part.

Test cases are defined in the module definitions part while the module control part manages their execution. All variables, timers etc. (if any) defined in the control part of a module shall be passed into the test case by parameterization if they are to be used in the behaviour definition of that test case i.e. ATDL does not support global variables or timers of any kind.

At the start of each test case the test configuration shall be reset. This means that all components and channels conducted by **create**, **bind**, etc. operations in a previous test case were destroyed when that test case was stopped (hence are not 'visible' to the new test case).

11.5.1. Termination of test cases

A test case terminates with the termination of the MTC. On terminating of the MTC (explicitly or implicitly) all running parallel test components shall be terminated by the test system.

The final verdict of a test case is calculated based on the final local verdicts of the different test components according to the rules defined in clause 19.6. The actual local verdict of a test component becomes its final local verdict when the test component terminates itself or is stopped by itself, another test component or by the test system.

11.5.2. Controlling execution of test cases

Program statements, limited to those defined in Table 21 may be used in the control part of a module to specify such things as the order in which the test cases are to be executed or the number of times a test case may be run.

If no programming statements are used then, by default, the test cases are executed in the sequential order in which they appear in the module control.

Test cases return a single value of type **verdicttype** so it is possible to control the order of execution depending on the outcome of a test case.

11.5.3. Test case selection

Boolean expressions may be used to select and deselect which test cases are to be executed. This includes, of course, the use of functions that return a **boolean** value.

Another way to execute test cases as a group is to collect them in a class and execute that class from the module control with a try statement.

11.5.4. Use of timers in control

Timer may be used to supervise execution of a test case. This may be done using an explicit timeout in the **try** statement. If the test case does not end within this duration, the result of the test case execution shall be a fail verdict and the test system may terminate the test case. The timer used for test case supervision is a system timer and need not be neither declared nor started.

EXAMPLE 3:

```
var  MyTObject TObject := TObject.Create;

try (7E-3) {
        MyTObject.MySimpleTestCase1();
        MyTObject.MySimpleTestCase2();
        MyTObject.MySimpleTestCase3();
        }

TSystemTimer.catch (timeout) {
        setverdict(fail);
        stop;
        }
```

Timer operations may also be used explicitly to control test case execution.

Concrete textual grammar

228 ModuleControlPart ::= “control” “{“ ModuleControlBody “}” [WithStatement] [SemiColon]

229 ModuleControlBody ::= [ControlStatementOrDefList [“stop”] | “stop”]

230 ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+

231 ControlStatementOrDef ::= ControlStatement | ClassLocalInst | ConstDef

400 ControlStatement ::= TestcaseInstance | FunctionInstance | AltConstruct | LoopConstruct
| DecisionConstruct | ActivateStatement | DeactivateStatement
| ChoiceConstruct | AltstepInstance | BasicStatement | TaskStatement

425 BasicStatement ::= TimerStatement | BreakStatement | ContinueStatement | TryStatement

11.5.5. Control diagram

An ATDL control diagram provides a graphical presentation of the control part of an ATDL module. The heading of a control diagram shall be the keyword **control** followed by the module name. Attributes associated to the ATDL module control part shall be specified within a <text

symbol> in the control diagram. The principle shape of an ATDL control diagram and the corresponding ATDL textual description are sketched in Figure 6.

Concrete graphical grammar

232 <control diagram> ::= <frame symbol> **contains** ((“control” ATDL_ModuleId)
({<control text area>}* <control graph area>) **set**)

233 <control text area> ::= <text symbol> **contains**
({ATDLComments}* [MultiWithAttrib] { ATDLComments}*)

234 <control graph area> ::= <statement start symbol> **is_followed_by** <control block area>

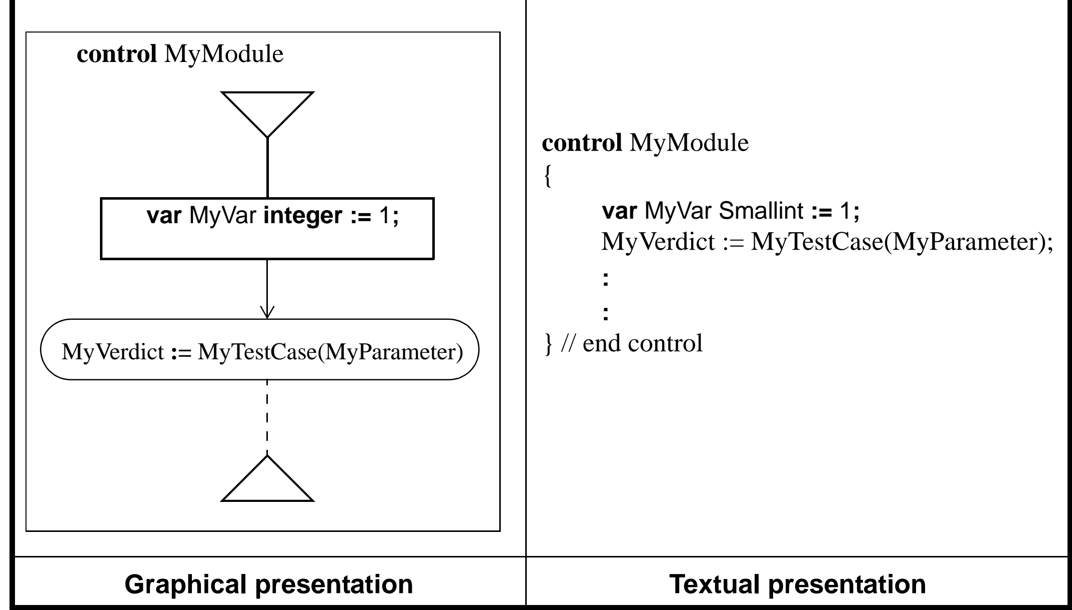
235 <control block area> ::= [<control statement block area> **is_followed_by**]
(<stop symbol> | <statement end symbol>)

236 <control statement block area> ::= <control statement area>
[**is_followed_by** <control statement block area>]

401 <control statement area> ::= <testcase instance area> | <function instance area>
| <fgr inline expression area> | <default area>
| <altstep instance area> | <choice area>
| <decision area> | <task area>

426 <basic statement area> ::= <timer statement area> | <continue area> |
<break area> | <try statement area>

Figure 6. Principle shape of an ATDL control diagram and corresponding textual language



11.6. Groups

In the module definitions part definitions can be collected in named groups. Groups are arranged in a refinable structure. Groups may be nested i.e. groups may contain other groups. There is no limit to the number of hierarchical levels for groups.

Grouping is done to aid readability and to add logical structure to the test suite if required. Groups and nested groups have no scoping except in the context of group identifiers and attributes given to a group by an associated **with** statement.

A group type defines containers for one or more component type or group definitions.

Concrete textual grammar

80 GroupDef ::= GroupHeading “{“
{ SupportingDef SemiColon }*

[InterfaceDefSpec]
[ComponentDefSpec]
MemberComponentList "}"

ATDL modules are organized as sets of groups. Each group has its own set of names for types, which helps to prevent name conflicts. The naming structure for groups is hierarchical. The members of a group are class and interface types. If ATDL code is to be widely distributed, unique group names should be chosen.This can prevent the conflicts that would otherwise occur if two development teams happened to pick the same module name and these modules were later to be used in a single program.

11.6.1. Group members

A *group* can have members of either or both of the following kinds: a) Sub groups of the group b) Types declared in the module definition part of the module. A group may not contain a type declaration and a subgroup of the same name, or a compile-time error results.

Concrete textual grammar

83 SupportingDefSpec ::= { (SupportingDef | FunctionDef | TestcaseDef | AltstepDef) SemiColon }*

84 InterfaceDefSpec ::= {InterfaceDef SemiColon }*

85 ComponentDefSpec ::= { (CoclassDef | ClassDef | ThreadClassDef) SemiColon }*

86 MemberComponentList ::= "members" MemberComponentDef {"," MemberComponentDef}* ";"

87 MemberComponentDef ::= GroupIdentifier | ComponentTypeIdentifier

11.6.2. Host support for groups

Each ATDL host determines how modules, groups, and subgroups are created and stored; which top-level group names are in scope in a particular compilation; and which groups are accessible.

The groups may be stored in a local file system in simple implementations of ATDL. Other implementations may use a distributed file system or some form of database to store ATDL source and/or binary code.

As an extremely simple example, all the ATDL groups and source and binary code on a system might be stored in a single directory and its subdirectories. Each immediate subdirectory of this directory would represent a top-level group.

Under this simple organization of groups, an implementation of ATDL would transform a group name into a pathname by concatenating the components of the group name, placing a file name separator (directory indicator) between adjacent components. For example, if this simple organization were used on a UNIX system, where the file name separator is /, the group name:

org.etsi.ttcn.tci;

would be transformed into the directory name:

org/etsi/ttcn/tci.

11.6.3. Unique group names

Developers should take steps to avoid the possibility of two published groups having the same name by choosing *unique group names* for groups that are widely distributed. This allows groups to be easily and automatically installed and catalogued.

You form a unique group name by first having (or belonging to an organization that has) a module name, such as Org.ETSI; and use this as a prefix for your group names, using a convention developed within your organization to further administer group names.

Such a convention might specify that certain directory name components be division, department, project, machine, or login names.

11.6.4. Declaring groups

A group declaration appears within a module to indicate the group to which the module belongs. A module that has no group declaration is part of an unnamed group. A *group declaration* in a module specifies the name of the group to which the module belongs.

Concrete textual grammar

81 GroupHeading ::= “group” GroupIdentifier

82 GroupIdentifier ::= Identifier

11.6.4.1. Unnamed group

An ATDL module may support one unnamed group. Unnamed groups are provided by ATDL principally for convenience when developing small or temporary applications or when just beginning development.

11.6.5. Group diagram

An ATDL group diagram provides a graphical presentation of an ATDL group. The heading of a group diagram shall be the keyword **group** followed by the group identifier. Each diagram must be owned by exactly one group, which may be nested within (and therefore owned by) another group. A group may contain subordinate groups and ordinary component types.

Concrete graphical grammar

88 <group diagram> ::= <frame symbol>
contains {GroupHeading {{<group text area>}*
[<component interaction area>] {<group reference area>}*}**set**}

89 <group text area> ::= <text symbol> **contains** {(SupportingDef | InterfaceDef) [SemiColon]}*

11.7. Importing from modules

It is possible to re-use definitions specified in different modules using the **import** statement. ATDL has no explicit export construct thus, by default, all module definitions in the module definitions part may be imported. An **import** statement can be used anywhere in the module definitions part. It shall not be used in the control part.

If the object identifier is provided as part of the module name (from which the definitions are imported from) in the import statement, this object identifier shall be used to identify the correct module.

All definitions that are imported from one module shall be referenced in one **import** statement only.

If an imported definition has attributes (defined by means of a **with** statement) then the attributes shall also be imported.

ATDL concrete textual grammar

211 ImportDef ::= “import” ModuleId (ImportSpec | “{“ {ImportSpec [SemiColon]}* “}”) [“recursive”]

212 ImportSpec ::= ImportAllSpec | ImportGroupSpec | ImportInterfaceSpec | ImportConstSpec
ImportComponentSpec | ImportTypeDefSpec | ImportTemplateSpec |
| ImportTestcaseSpec | ImportFunctionSpec | ImportAltstepSpec

213 ImportAllSpec ::= [DefKeyword] Dot “*”

214 ModuleId ::= ModuleName [“language” FreeText]

216 GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]

218 DefKeyword ::= “type” | “const” | “class” | “interface” | “template” | “testcase” | “function” | “altstep”

219 ImportGroupSpec ::= “group” GroupIdentifier {“,” GroupIdentifier}*

220 ImportInterfaceSpec ::= “interface” InterfaceIdentifier {“,” InterfaceIdentifier}*

221 ImportComponentSpec ::= “class” ComponentTypeIdentifier {“,” ComponentTypeIdentifier}*

222 ImportTypeDefSpec ::= “type” TypeIdentifier {“,” TypeIdentifier}*

223 ImportTemplateSpec ::= “template” TemplateIdentifier {“,” TemplateIdentifier}*

224 ImportConstSpec ::= “const” ConstIdentifier {“,” ConstIdentifier}*

225 ImportTestcaseSpec ::= “testcase” TestcaseIdentifier {“,” TestcaseIdentifier}*

226 ImportFunctionSpec ::= “function” FunctionIdentifier {“,” FunctionIdentifier}*

227 ImportAltstepSpec ::= “altstep” AltstepIdentifier {“,” AltstepIdentifier}*

11.7.1. Rules on using import

On using import the following rules shall be applied:

a) Only top-level definitions in the module may be imported. Definitions which occur at a lower scope (e.g. local constants defined in a function) shall not be imported;

b) Only direct importing from the source module of a definition (i.e. the module where the actual definition for the identifier referenced in the **import** statement resides) is allowed;

c) A definition is imported together with its name and all local definitions.

d) A definition is imported together with all information of referenced definitions that are necessary for the usage of the referenced definition.

e) As default, the identifiers of referenced definitions are not automatically imported. If the identifiers of the referenced definitions are wished to be implicitly imported, the **recursive** directive (see §11.7.2) shall be used.

11.7.2. Recursive import

The ATDL default import mechanism imports referenced definitions without their identifier. This means, a referenced definition cannot be used within the importing module for, e.g. declaring a variable or for being sent over a port. Even though this default import mechanism avoids cluttering up of the name space of the importing module, in some cases it is desired to import all referenced definitions together with their identifiers. In ATDL, the **recursive** keyword provides this feature.

11.7.3. Importing single definitions

Single definitions may be imported.

11.7.4. Import on demand

All definitions of a module definitions part may be imported using the “*” symbol next to the module name. If all definitions of a module is imported by using the “*” symbol, no other form of import (import of single definitions, import of the same kind etc.) shall be used for the same **import** statement. A *type-import-on-demand declaration* allows all types declared in the group named by a fully qualified name to be imported as needed. For example,

import MyModule.*;

import Org.ETSI.*

11.7.5. Importing groups

Groups of definitions may be imported. The effect of importing a group is identical to an **import** statement that lists all importable definitions (including sub-groups) of this group.

11.7.6. Handling name clashes on import

All ATDL modules shall have their own name space in which all definitions shall be uniquely identified. Name clashes may occur due to import e.g. import from different modules, import of groups or import of recursive definitions. Name clashes shall be resolved by prefixing the imported definition (which causes the name clash) by the identifier of the module from which it is imported. The prefix and the identifier shall be separated by a dot (.).

In cases where there are no ambiguities the prefixing need not (but may) be present when the imported definitions are used. When the definition is referenced in the same module where it is defined, the module identifier of the module (the current module) also may be used for prefixing the identifier of the definition.

11.7.7. Import definitions from non-ATDL modules

In cases when definitions are imported from other sources than ATDL modules, the language specification shall be used to denote the language (may be together with a version number) of the source (e.g. module, package, library or even file) from which definitions are imported. It consists of the **language** keyword and a subsequent textual declaration of the denoted language.

12. Test configurations

ATDL allows the (static and dynamic) specification of concurrent test configurations (or configuration for short).

Within every configuration there shall be one (and only one) main test thread class (MTC). Test components that are not MTCs are called parallel test components or PTCs. The MTC shall be created automatically at the start of each test case execution. The behavior defined in the body of the test case shall execute on this component. During execution of a test case other components can be created dynamically by the explicit use of the constructor operation.

Test case execution shall end when the MTC terminates. All other PTCs are treated equally i.e., there is no explicit hierarchical relationship among them and the termination of a single PTC terminates neither other components nor the MTC.

12.1. Test configurations at specification level

ROS [ROSE] defines a number of concepts and constructs to describe the interaction between objects that follow the request/reply interaction paradigm. Such objects are called ROS-objects. A pair of ROS-objects must have an association between them to serve as a context for the invocation and performance of operations. ROS defines a **connection package** as two special operations, called **bind** and **unbind**, that are available, as an option, to an application designer to dynamically establish and release, respectively, the association between two ROS-objects. In addition to the means by which an association is established between two ROS-objects, the association is governed by an **association contract**.

There are no restrictions on the number of association **contract**s a component type may maintain. One-to-many association **contract**s are also allowed at specification level.

12.1.1. Defining association contracts

ATDL components communicate by sending and receiving signals. Table 17 shows the different communication possibilities depending on the components involved.

A required interface or co-interface is one that a component needs to support *as a client* in order to provide its services. The (declared) supported interfaces of a class are the interfaces listed as supported in the class specifications. The (declared) supported co-interfaces of a co-class are the co-interfaces listed as supported in the co-class specifications. Instances of interface (cointerface) templates declared as supported may be offered by instances of classes (coclasses) being defined.

A supported co-interface is an object's server co-interface (in the client-server sense), i.e., the operations that it can perform. Co-interfaces listed as being supported on a co-class type are the only co-interfaces for which instances may exist on the co-objects. The declared supported co-interfaces of a base co-class are considered as supported co-interfaces of the sub-co-class and may be instantiated by the object instance of the sub-co-class as well.

A class may be declared to *directly implement* one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that its direct ancestor classes and direct ancestor interfaces do. This (multiple) interface inheritance allows objects to support (multiple) common behaviors without sharing any implementation.

If a co-class uses a co-interface of another co-class, the contract may be shown as a dashed line with an arrowhead on the co-interface symbol (Figure 5). If a co-class is the realization of a co-interface, the shorthand notation of a circle attached to the co-class symbol by a line segment may be used.

In ATDL, operational cp-interfaces and/or co-interfaces and message-based cp-interfaces and/or co-interfaces are defined as the client's view on the server. The service is defined naturally in terms of information sources and sinks. However, all of these definitions presuppose a directionality, or point of view, namely that of the client.

Table 17: Cp-interface and co-interface

client	server	possible contracts	contract objects
components	co-class	required/supported	co-interface
components	class	required/implemented	cp-interface
components	thread class	required/implemented	cp-interface

Each signature is specified as a source if information flows from the server to the client, and as a sink if it flows in the opposite direction. In the abstract class definition of the server, the interface is listed as a supported interface; while on the client, the interface is listed as a required interface.

When a co-class is implemented, its supported co-interface is mapped onto an implemented interface in the implementing class.

Concrete graphical grammar

148 <required interface area> ::= <dependency symbol>
is_connected_to (<component area> <interface area>)

150 <channel symbol> ::= <channel symbol 1> | <channel symbol 2> | <channel symbol 3>

151 <channel symbol 1> ::= <solid association symbol>

12.1.2. Abstract test system interface

The ATDL co-classes replace abstract test system interface and address data type definitions in TTCN-3. An SUT may consist of several entities which have to be addressed individually. The address data type in TTCN-3 is a type for use with port operations to address SUT entities.

In a real test environment ATS need to communicate with the SUT. Instead, a set of well-defined test system co-classes is associated with each test case. A test system co-class definition is a list of all possible co-interfaces through which the test case is connected to the SUT. A set of co-class definitions is used to define the test system interface because, conceptually, co-class definitions and test system interface definitions have the same form.

```
co class MyISDNTestSystemInterface
{
	supports MyBchannelB1, MyBchannelB2, MyDchanneld1;
	:
```

}

12.1.3. Configuration diagrams

A configuration diagram given at specification level shows a test configuration, i.e. a set of interconnected test components with well-defined communication interfaces and/or co-interfaces and an explicit test system interface which defines the borders of the test system.

Concrete textual grammar

125 ComponentTypeIdentifier ::= ThreadClassIdentifier | ClassInstance | CoclassIdentifier

126 ClassInstance ::= ClassIdentifier | ClassTemplateInstance

127 ComponentType ::= [ModuleName Dot] ComponentTypeIdentifier

Concrete graphical grammar

16 <component interaction area> ::= {<component area> | <component dependency area>

17 <component area> ::= <component reference area> | <component diagram>

18 <component diagram> ::= <thread class diagram> | <class diagram> | <coclass diagram>

20 <component reference area> ::= <reference symbol> **contains** ComponentType
[**is_connected_to** <component extends area>]
[**is_connected_to** { <required interface area>+ } **set**]
[**is_connected_to** { <supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

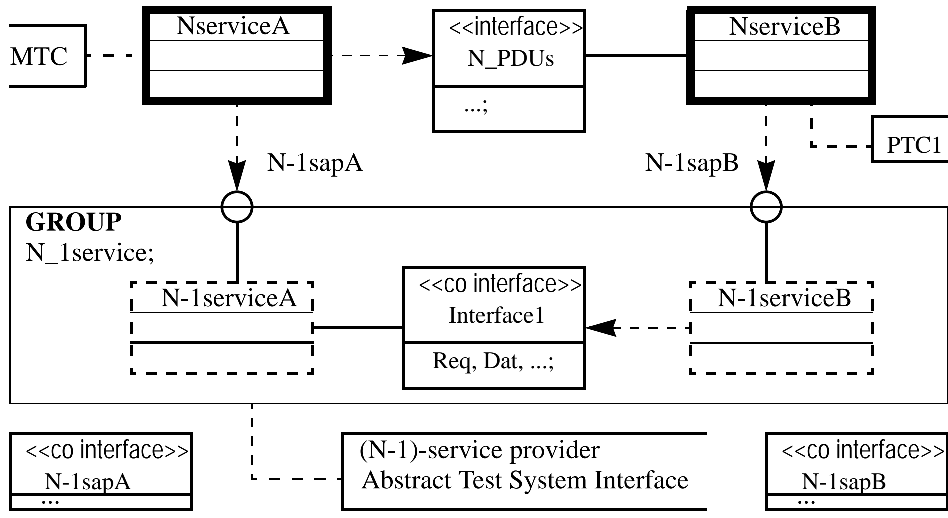


Figure 7. A typical ATDL Configuration Diagram at specification level

12.1.3.1. The usage dependency

A usage dependency is a situation in which one component requires the presence of another component for its correct implementation or functioning. All the components must exist at the same level of meaning i.e., they do not involve a shift in the level of abstraction or realization. A usage may be stereotyped further to indicate the exact nature of the dependency, such as calling an operation of another class or instantiating an object of another class. A usage dependency is indicated by a <dependency symbol>. The arrowhead is on the server (independent) component, and the tail is on the client (dependent) component.

Concrete graphical grammar

19 <component dependency area> ::= <dependency symbol>
is_connected_to (<component area> <component area>)

12.2. Test configurations at instance level

Test configurations can be graphically represented by component instance diagram. Unlike a basic message sequence chart, a component instance diagram explicitly shows the relationships among the component instances. On the other hand, a component instance diagram does not show time as a separate dimension.

ATDL allows the (dynamic) specification of concurrent test configurations. A configuration diagram given at instance level shows a test configuration instance, i.e. a collection of component object boxes and lines mapping to component instances and channels, respectively.

The component instance diagram contains group instances. The group instances may contain run-time instances, such as co-class instances and objects. The model may show dependencies among the instances and their interfaces. Components are connected to other components by dashed dependency arrows (possibly through channels). This indicates that one component uses the services of another component.

12.2.1. Channel communication model

Test component instances can be connected with other component instances.

When a program detects something that has happened, it can notify its clients. This notification process is referred to as firing a message event. Test component instances are connected via their message-based channels i.e., connections among component instances and between a component instance and the test system instance are channel-oriented. Each channel is modeled as an infinite FIFO queue, which stores the incoming messages or procedure calls until they are processed by the component instance owning that channel.

ATDL connectable objects provide outgoing channels to their clients in addition to their incoming channels. As a result, components and their clients can engage in bidirectional communication. Incoming channels are implemented on a server object and receive calls from external clients of an object, while outgoing interfaces are implemented on the client's sink and receive calls from the object. The object defines an interface it would like to use, and the client implements it.

An object defines its incoming interfaces and provides implementations of these interfaces. Incoming interfaces are available to clients through the object's **bind** method. Clients call the methods of an incoming interface on the object, and the object performs desired actions on behalf of the client.

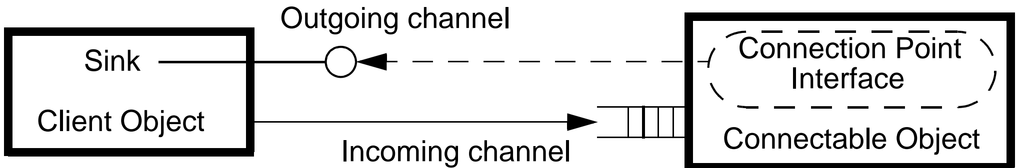


Figure 8. The ATDL channel communication model

Outgoing interfaces are also defined by an object, but the client provides the implementations of the outgoing interfaces on a sink object that the client creates. The object then calls methods of the outgoing interface on the sink object to notify the client of changes in the object, to trigger events in the client, to request something from the client, or, in fact, for any purpose the object creator comes up with.

Connectable signal objects provide a general mechanism for object-to-client communication. Any object that wishes to expose signals or notifications of any kind can use this technology. This technology includes the following elements:

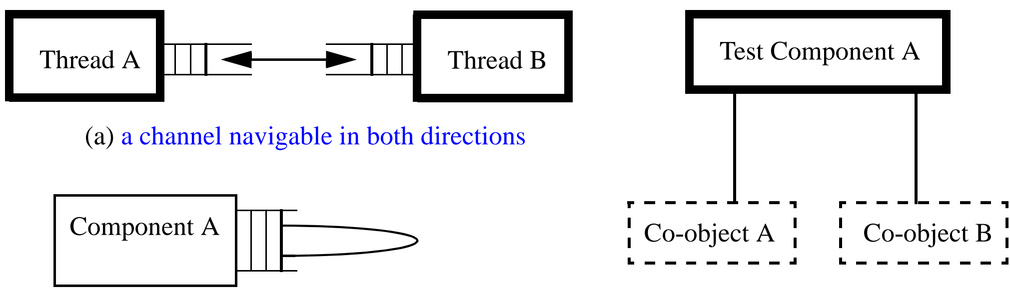
Server object: Implements the signal-based channel and manages connection with the client's sink, defines an outgoing interface for the client.

Client: creates a sink object to implement the outgoing channel defined by the server object.

Sink object: Implements the outgoing channel; used to establish a connection to the server.

12.2.2. Restrictions on connections

ATDL connections are component-to-component connections (see Figure 9). There are no restrictions on the number of connections a component instance may have, but one-to-many connections are NOT allowed at instance level. Meanwhile, connections among the co-class instances within the test system interface are not allowed.



(a) a channel navigable in both directions

(b) self-connection is only allowed for an exception channel

(c) connected with co-objects

Figure 9. Allowed connections

A component instance can only connect to itself through an exception channel.

12.3. Defining interface types

Interfaces facilitate communication between test components and between test components and the test system interface. ATDL supports message-based and operational cp-interfaces. ATDL also supports message-based and operational co-interfaces. Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models. Objects built with ATDL that support interfaces can interact with SUT objects and thread object written in other languages.

An interface declaration introduces a new reference type whose members can be messages or operations. This type has no implementation, but otherwise unrelated classes can implement it by providing signal handlers for its messages and abstract methods.

Unlike other object-oriented languages, ATDL is a ‘pure’ object-oriented language: every coordinating signal in ATDL is an object, whereas in other languages this isn't always the case.

The benefits of this “every signal is an object” philosophy are great, and pure languages such as ATDL are considered to be more productive, and - more importantly - more precise to work with.

Concrete textual grammar

129 InterfaceDef ::= MsgInterfaceDef | CpOpInterfaceDef | CoOpInterfaceDef

143 InterfaceType ::= [ComponentType Dot] InterfaceTypeIdentifier

12.3.1. Interface diagrams

An interface or a co-interface may be shown using the interface symbol with the keyword <<interface>> and <<co interface>> respectively. A list of operations supported by the interface

is placed in the operation compartment. A list of messages supported by the interface is placed in the message compartment. The attribute compartment may be omitted because it is always empty.

An interface may also be displayed as a small circle with the name of the interface placed below the symbol. The circle may be attached to a solid line to the classes that support it. This indicates that the class provides all of the operations in the interface type (and possibly more). A class that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. The dashed arrow implies that the class requires no more than the operations specified in the interface; the client class is not required to actually use *all* of the interface operations.

Concrete graphical grammar

152 <interface area> ::= (<interface area 1> | <interface area 2>)
[**is_connected_to** {<interface extends area>*} **set**]

154 <interface extends symbol> ::= <component extends symbol>

155 <interface area 1> ::= <interface symbol 1> **contains**
(<interface heading> (OperationAttribs | MessageAttribs))

156 <interface heading> ::= (<entity kind symbol> **contains** [**co**] **interface**) InterfaceIdentifier

157 <interface area 2> ::= <interface symbol 2> **is_associated_with** InterfaceIdentifier

158 <interface symbol 2> ::= <connector symbol>

12.3.2. The message-based interface types

A message-based interface is comprised of a set of message types. The term message is used to mean both messages as defined by templates and actual values resulting from expressions. Thus, the list restricting what may be used on a asynchronous channel. Each message type contains the identifier of the message, the information type of the message, and an indication of whether it is a producer or consumer (but not both) with respect to the object which provides the service defined by the interface template.

Concrete textual grammar

130 MsgInterfaceDef ::= MessageInterfaceHeader MessageAttribs

131 MsgInterfaceHeader ::= ["co"] "interface" MsgInterfaceTypeIdentifier [MsgInterfaceHeritage]

132 MsgInterfaceTypeIdentifier ::= Identifier

144 MessageAttribs ::= "{" {MessageList [SemiColon]}+ "}"

It should be noted that the syntax defined here presupposes a directionality with respect to the message-based interface definitions. If two objects are involved in a message binding, then one is designated a service provider, or server, and the other a service consumer, or client. The interface template describing interactions between them is expressed from the viewpoint of the client (defining the server). In many ways, particularly where messages travel in both directions, the choice of client and server may appear rather arbitrary. However, this model is consisted with many familiar service models [25]. Note that the server template includes a declaration that it "supports" the message-based interface, while the client template includes a declaration that it "requires" the message-based interface template. Each message-based interface type definition shall have one or more lists indicating the allowed collection of message types together with the allowed communication direction.

In ATDL, message-based interface templates are defined as the client's view on the server. The directions are specified by the keywords **in** (for the in direction, as a source if information flows from the server to the client), **out** (for the out direction, as a sink if it flows from the client to the server) and **inout** (for both directions). For example,

// Asynchronous interface which allows types MsgType1 and MsgType2 to be received at,

// MsgType3 to be sent via and any integer value to be send and received over the interface

interface MyMessagePortType

```
{
in UpMessage MsgType1;
out DownMessage MsgType2;
inout BiDirectionalMessage MyIntegerType1
}
```

12.3.3. Operational interfaces

A *passive object operational interface* — or simply *operational interface* — defines methods that can be implemented by a class. Operational interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods.

An operational interface is a descriptor for the externally visible operations of a class, or thread class without specification of internal structure. Each operational interface often specifies only a limited part of the behavior of an actual class. A class may support many operational interfaces. Operational interfaces may have inheritance relationships.

A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Concrete textual grammar

```
134 CpOpInterfaceDef ::= CpOpInterfaceHeader OperationAttribs
135 CpOpInterfaceHeader ::= “interface” CpOpInterfaceTypeIdentifier [CpOpInterfaceHeritage]
136 CpOpInterfaceTypeIdentifier ::= Identifier
```

12.3.3.1. Defining operational interface types

Operational interfaces, like classes, can be declared only in the outermost scope of a module or class, not in a testcase or function declaration.

An operational interface is essentially equivalent to a class with no properties and only abstract methods. All the operations in an interface have public visibility. The operational interface can include only operations. Fields are not allowed in operational interfaces.

12.3.3.2. Inheritance and Overriding

If the interface declares an operation, then the declaration of that operation is said to *override* any and all operations with the same signature in the ancestor interfaces of the interface that would otherwise be accessible to code in this interface.

If an operation declaration in an operational interface overrides the declaration of an operation in another operational interface, a compile-time error occurs if the operations have different return types or if one has a return type and the other does not have. Moreover, an operation declaration must not have a **raises** clause that conflicts (clause 7.2.6) with that of any method that it overrides; otherwise, a compile-time error occurs.

12.3.3.3. Implementing operational interfaces

Once an operational interface has been declared, it can be implemented in a thread or a class before it can be used. The operational interfaces implemented by a thread or a class are specified in the *ImplementedInterfaceList*. By default, each interface method is mapped to a method of the same name in the implementing component.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same number of parameters, and identically typed parameters in each position. By default, each cp-interface method is mapped to a method of the same name in the implementing class.

A compile-time error occurs if the same operational interface is mentioned two or more times in a single implements clause, even if the interface is named in different ways.

Concrete textual grammar

103 ImplementedInterfaceList ::= “implements” InterfaceType {“,” InterfaceType}* “;”

Concrete graphical grammar

149 <supported interface area> ::= <channel symbol>
is_connected_to (<interface area> <component area>)

The realization relationship is shown by a <channel symbol> or by a dashed line with a solid triangular arrowhead from a class to an interface it supports. This is the same notation used to indicate realization of a co-class by an implementation class.

12.3.3.4. Changing inherited implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual.

12.3.4. Interface inheritance

If an *extends* clause is provided, then the interface being declared extends each of the other named interfaces and therefore inherits the operations and messages of each of the other named interfaces. These other named interfaces are the *ancestor interfaces* of the interface being declared. Any class that **implements** the declared interface is also considered to implement all the interfaces that this interface *extends* and that are accessible to the class.

An operational interface, like a class, inherits all of its ancestors’ methods. But interfaces, unlike classes, do not *implement* methods. What an interface inherits is the *obligation* to implement methods — an obligation that devolves onto any class supporting the interface.

It should be noted that a message-based interface or a message-based co-interface should not inherit from an operational interface or an operational co-interface and vice versa.

Concrete textual grammar

133 MsgInterfaceHeritage ::= “extends“ MsgInterfaceTypeIdentifier {“,” MsgInterfaceTypeIdentifier}*

137 CpOpInterfaceHeritage ::= “extends“ CpOpInterfaceTypeIdentifier {“,” CpOpInterfaceTypeIdentifier}*

141 CoOpInterfaceHeritage ::= “extends“ CoOpInterfaceTypeIdentifier {“,” CoOpInterfaceTypeIdentifier}*

Concrete graphical grammar

153 <interface extends area> ::= <interface extends symbol>
is_connected_to (<interface area> <interface area>)

12.3.5. Declaring exception types

Exception types are a special kind of message-based interface.

All exceptions in ATDL occur synchronously as a result of an action by the thread in which they occur, and at a point in the ATDL program that is specified to possibly result in such an exception. An ordinary message is, by contrast, an asynchronous message that can potentially occur at any point in the execution of an ATDL program.

It is illegal to define generic type directly or indirectly within an exception type declaration. Consequently, no parameterized types appear anywhere in exception handling.

Concrete textual grammar

187 ExceptionDef ::= “exception” ExceptionIdentifier “{“ {ExceptionMember}* “}”

188 ExceptionIdentifier ::= Identifier

189 ExceptionMember ::= ExceptionTypeIdentifier Type [SemiColon]

190 ExceptionTypeIdentifier ::= Identifier

191 ExceptionName ::= [ModuleName Dot] ExceptionIdentifier

Concrete graphical grammar

192 <exception area> ::= <interface symbol 1> **contains** (<exception heading> {ExceptionMember}*)

193 <exception heading> ::= (<entity kind symbol> **contains exception**) ExceptionIdentifier

13. Defining classes

To incorporate concurrent constructs in a conventional object-oriented system, the notion of execution threads was introduced in ATDL. A model in which concurrency is introduced in this way is called an *object / thread model*. This model has the advantage of familiarity, because a conventional programming language is used as the base language.

In this model, objects are passive entities, and each thread can execute object method code at any time. Good response times and good performance are expected. However, since threads execute independently, a mutual exclusion problem might occur when more than two threads simultaneously execute the same method. To avoid this problem, a locking mechanism must be used to control access to synchronized fields in a method.

A *class* (or *class type*) defines a structure consisting of methods, and *properties*. Instances of a class type are called *objects*. The fields and methods of a class are called its *members*. The members of a class include both declared and inherited members. Field, method, and constructor declarations may include the access modifiers (§13.5) public, protected, or private. Newly declared fields can hide fields declared in an ancestor class. Newly declared methods can hide, implement, or override methods declared in an ancestor class or operational interface.

Method declarations describe code that may be invoked by method invocation expressions. A synchronized method (§13.10.2) automatically locks an object before executing its body and automatically unlocks the object on return, thus allowing its activities to be synchronized with those of other threads (§21.2). Method names may be overloaded (§13.7.4).

Constructors (§13.8) are similar to methods, but cannot be invoked directly by a method call; they are used to initialize new class instances. Like methods, they may be overloaded.

13.1. Defining class types

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a module, not in a test case or function declaration.

A class may participate in association contract, inheritance, and dependency relationships. A class type declaration can also include a list of interfaces implemented by the class. The *class* type defines which interfaces are associated with a class. These definitions shall be made in the module definitions part. The interface names in a class definition are local to that class i.e., another class may have interfaces with the same names.

Methods appear in a class declaration as function or test case headings, with no body. Defining declarations for each method occur elsewhere in the module definition part. Thus testcase and function declarations in the class declaration work like forward declarations, although the **forward** directive isn't used.

Concrete textual grammar

99 ClassDef ::= [(“virtual” | “final”)] “class” ClassIdentifier [ClassHeritage]

100 ClassDefBody ::= “{“ [ClassPropertiesList]
[InterfaceDefSpec]

[ImplementedInterfaceList]
[RequiredInterfaceList]
[ClassMethodList] "}"

104 ClassMethodList ::= {ClassMethodDef SemiColon}*

Concrete graphical grammar

115 <class diagram> ::= <class symbol> **contains**
(ClassIdentifier <class properties area> <class methods area>)
[**is_connected_to** <component extends area>]
[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

116 <class methods area> ::= ({ ClassMethodDef <end> }*) **set**

A class is shown as a solid-outline rectangle with three compartments separated by horizontal lines. The top compartment holds the class name. The middle compartment holds a list of properties. The bottom compartment contains a list of operations or methods. The middle and bottom compartment can be suppressed in a class symbol.

Table 18 shows a basic class declaration with properties and methods.

Table 18: Detailed class declaration with visibilities of features

Window // name compartment

public size Area := (100,100); // property compartment
protected visibility **boolean** := invisible;

public function display (location Point); // method compartment
public function hide ();

Presentation options

Suppressing compartments. Either or both of the property and method compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, on inference can be drawn about the presence or absence of elements in it. Note that an empty compartment implies that there are no elements in the corresponding list.

Additional compartments. Additional compartments may be supplied to show signals handled, exceptions raised, and so on.

13.1.1. Scope of a class type name

The ClassIdentifier in a class declaration specifies the name of the class. This class name has as its scope the entire group or the entire current module, in which the class is declared. Forward declarations allow mutually dependent classes. As an example, the group:

```
group points {
class Point {   x Integer; y  Integer;            // coordinates
                color    PointColor;              // color of this point
                next     Point;                   // next point with this color
                var      nPoints Integer;       }
class PointColor {
                first     Point;                  // first point with this color
                //...
                private   color  Integer;       // color components
                }
}
```

defines two classes that use each other in the declarations of their class members. Because the class type names Point and PointColor have the entire group points, as their scope, this example compiles correctly — that is, forward reference is not a problem.

Concrete textual grammar

101 ClassIdentifier ::= Identifier

13.1.2. Passive object

An object that does not have it own thread of control. Its operations execute under a control thread anchored in a thread object. A thread object is one that owns a thread of control and may initiate control activity. A passive object is one that has a value but does not initiate control. However, a method on a passive object may send messages while processing a request that it has received on an existing thread.

Passive objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Passive objects are created and destroyed by special methods called constructors and destructors.

13.1.3. Final classes

A class can be declared final if its definition is complete and no descendants are desired or required. A compile-time error occurs if the name of a final class appears in the extends clause of another class declaration; this implies that a final class cannot have any descendants. A compile-time error occurs if a class is declared both final and virtual, because the implementation of such a class could never be completed

13.1.4. Class inheritances

Generalization is a transitive, anti-symmetric relationship. One direction of traversal leads to the parent; the other direction leads to the child. An element related in the parent direction is called an ancestor; an element related in the child direction is called a descendant.

The optional **extends** clause in a class declaration specifies the *direct* ancestor of the current class. A class is said to be a *direct* descendant of the class it extends. The direct superclass is the class from whose implementation the implementation of the current class is derived. If the class declaration for any other class has no **extends** clause, then the class has the class TObject as its implicit direct ancestor class.

A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type.

Concrete textual grammar

93 CoclassHeritage ::= “extends“ CoclassIdentifier

102 ClassHeritage ::= “extends“ (ClassIdentifier | CoclassIdentifer)

122 ThreadClassHeritage ::= “extends“ (ThreadClassIdentifer | CoclassIdentifer)

Concrete graphical grammar

97 <component extends area> ::= <component extends symbol>
is_connected_to (<component area> <component area>)

Inheritance between components is shown as a solid-line path from the child component to the parent component, with a large hollow triangle at the end of the path where it meets the ancestor component.

13.1.4.1. The *TObject* class

The *TObject* class is the ultimate ancestor of all other classes. *TObject* defines only a handful of methods, including a basic constructor and destructor.

13.1.5. Ancestor interfaces

The optional implements clause in a class declaration lists the names of interfaces that are *direct ancestor interfaces* of the class being declared. The declarations of the operations defined in each ancestor operational interface must be implemented either by a declaration in this class or by an existing method declaration inherited from the direct ancestor class.

13.1.5.1. Implementation inheritance

The inheritance of the implementation of a parent element. In this relationship, a class inherits both interface and implementation from another component. ATDL does not permit multiple inheritances of class. ATDL will let you use single implementation inheritance within a process.

Certain kinds of elements, such as interfaces and co-classes, are intended for specifying behavior, and they contain no implementation information. Other kinds of elements, such as classes, are intended for implementing behavior. They contain implementation information. Usually, realization relates a specification element, such as a co-class or an interface, to an implementation element, such as a MSC diagram or a class.

A cp-interface or a co-interface is a collection of procedure names, without an implementation. A class can support any number of cp-interfaces and/or co-interfaces and must provide implementations for every function described in the cp-interface and/or co-interfaces. Both artifacts provide a greater level of polymorphism. Interfaces and co-interfaces provide signature inheritance; standard ATDL classes provide implementation inheritance.

13.2. Class members

The members of a class type include the following: a) Members inherited from its direct ancestor class, except in class TObject, which has no direct ancestor class b) Members declared in the body of the class.

Members of a class that are declared private are not inherited by descendant classes of that class. Only members of a class that are declared protected or public are inherited by descendant classes declared in a module other than the one in which the class is declared.

A class type may have two or more methods with the same simple name if the methods have different signatures, that is, a method can be redeclared using the **overload** directive, if they have different numbers of parameters or different parameter types in at least one parameter position.

A class type may contain a declaration for a method with the same name and the same signature as a method that would otherwise be inherited from a ancestor class or ancestor interface. In this case, the method of the ancestor class or ancestor interface is not inherited. If the method not inherited is an operation, then the new declaration is said to *implement* it; if the method not inherited is **virtual**, then the new declaration is said to *override* it. To override a method, redeclare it with the **override** directive.

13.3. Declaring properties

The *field*s, *data types*, *exceptions* and *constants* of a component are called its *properties*. It is possible to declare variables and constants local to a particular component. These declarations are visible to all functions that run on the component.

Concrete textual grammar

110 ClassPropertiesList ::= { (SupportingDef | ClassFieldDef | DefaultAltstepDef) SemiColon}*

Concrete graphical grammar

98 <class properties area> ::= ({ ClassProperty <end> }*) **set**

The first *ClassProperty* in a <class properties area> must be placed uppermost in the middle compartment of the containing component symbol. Each subsequent *ClassProperty* must be placed below the previous one.

13.3.1. Signal handlers

Signal handlers are altsteps that implement responses to dynamically dispatched signals. ATDL uses signal handlers to respond to run-time signals. Default Handler is called by the ATDL underlying system when it cannot find an altstep for a particular signal. Default Handler provides signal handling for all signals for which an object does not have specific handlers.

A signal is a mechanism that links an occurrence to some code. More specifically, a signal is an altstep pointer that points to an altstep in a specific class instance.

The ATDL underlying system uses altstep pointers to implement signals. An altstep pointer is a special pointer type that points to a specific altstep in an instance object. As a component writer, you can treat the altstep pointer as a placeholder: When your code detects that a signal occurs, you call the altstep (if any) specified by the user for that signal.

Altstep pointers maintain a hidden pointer to an object. When the application developer assigns a signal handler to a component's signal, the assignment is not just to an altstep declaration with a particular name, but rather to an altstep instance in a specific instance object.

Components use fields to implement their signal handlers. But while a field is merely a storage location whose contents can be examined and changed, a signal handler is associated with a specific altstep. The declaration of a signal handler specifies a name and a type, and includes one **default** specifier.

Concrete textual grammar

114 DefaultAltstepDef ::= ClassFieldIdentifier (MessageType | Operation) "default" AltstepInstance

13.4. Declaring fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private. To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any method declarations. For example:

```
class MyMTCType
{
        public MyLocalInteger integer[2];
        timer MyLocalTimer;
        implements MyMessagePortPCO1;
        :
}
```

It is a compile-time error for the body of a class declaration to contain declarations of two fields with the same name. Methods and fields may have the same name, since they are used in different contexts. If the class declares a field with a certain name, then the declaration of that field is said to *hide* any and all accessible declarations of fields with the same name in the ancestor classes of the class. If a field declaration hides the declaration of another field, the two fields need not have the same type.

Concrete textual grammar

111 ClassFieldDef ::= ClassVisibility ["synchronized"] ClassFieldIdentifier VarInitializer

13.4.1. Static fields

A static field can be a static variable or a constant field. There exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created.

A static variable, sometimes called a *class variable*, is incarnated when the class is initialized. A field that is not a class variable (sometimes called a non-static field) is called an *instance variable*. Whenever a new instance of a class is created, a new variable associated with that instance is created for every instance variable declared in that class or any of its ancestors.

13.4.1.1. Constant fields

Every constant declaration in the body of a class is called a constant field. Every constant field in the body of a class must have an initialization expression, which must be a constant expression.

13.4.2. Initialization of fields

If a field declarator contains a *variable initializer*, then it has the semantics of an assignment to the declared variable, and:

- If the declarator is for a class variable (that is, a static field), then the variable initializer is evaluated and the assignment performed exactly once, when the class is initialized.

If the keyword self or the keyword inherited occurs in an initialization expression for a class variable, then a compile-time error occurs.

- If the declarator is for an instance variable (i.e. a field), then the variable initializer is evaluated and the assignment performed each time an instance of the class is created.

Variable initializers are also used in local variable declaration statements (§18.1), where the initializer is evaluated and the assignment performed each time the local variable declaration statement is executed.

13.5. Visibility of class members

Every member of a class has an attribute called *visibilit*y, which is indicated by one of the reserved words **private**, **protected**, or **public**. Visibility determines where and how a member can be accessed, with private representing the least accessibility, protected representing an intermediate level of accessibility, and public representing the greatest accessibility.

There are three predefined visibilities:

a) **private**: Class members declared as private can be used only by member functions and test cases of the class.

b) **protected**: Class members declared as **protected** can be used by member functions and test cases of the class. Additionally, they can be used by classes derived from the class.

c) **public**: A *public* member is visible wherever its class can be referenced.

If none of the access modifiers public, protected, or private are specified, a class member or constructor is accessible throughout the group that contains the declaration of the class in which the class member is declared, but the class member or constructor is not accessible in any other group. The default access has no keyword, but it is commonly referred to as "friendly."

Friendly access allows you to group related classes together in a group so that they can easily interact with each other. When you put classes together in a group, you "own" the code in that group. It makes sense that only code that you own should have friendly access to other code that you own. You could say that friendly access gives a meaning or a reason for grouping classes together in a group.

Members of a class that are declared private are not inherited by descendant classes of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

You can increase the visibility of a member in a descendant class by re-declaring it, but you cannot decrease its visibility. For example, a protected attribute can be made public in a descendant, but not private.

Concrete textual grammar

112 ClassVisibility ::= [“public” | “protected” | “private”]

13.6. Virtual classes

The virtual test case or virtual function declarations within a virtual class include only the method’s heading. The implementation of the virtual test case or virtual function follows in the descendant implementation class. Thus virtual test case and virtual function declarations within the virtual class work like abstract declarations, although the **abstract** directive isn’t used.

A virtual class is a class that is incomplete, or to be considered incomplete. Only virtual classes may have abstract methods, that is, methods that are declared but not yet implemented. If a class that is not virtual contains a method template, then a compile-time error occurs.

A compile-time error occurs if an attempt is made to create an instance of a virtual class using a class instance creation expression. It is a compile-time error to declare a virtual class type such that it is not possible to create a descendant class that implements all of its abstract methods. This situation can occur if the class would have as members two abstract methods that have the same method signature but different return types.

A virtual class may also have method template declarations. Method templates are the efficient way to implement polymorphic behavior. Just like type declarations, method declarations can be generic, that is, parameterized by one or more generic type parameters.

Method templates provide a mechanism by which we can preserve the semantics of method definitions and method calls (encapsulate a section of code in one program location and ensure that the arguments are evaluated only once prior to the invocation of the method) without having to bypass ATDL's strong type-checking as is done with the macro solution.

If you have written an arithmetic expression in a programming language, you have used a predefined incarnated function. For example, the expression

1 + 3

invokes the addition operation for integer operands, whereas the expression

1.0 + 3.0

invokes a different addition operation that handles floating point operands. The operation that is actually used is transparent to the user. The addition operation is incarnated to handle the different operand types. It is the responsibility of the compiler, and not of the programmer, to distinguish between the different operations and to apply the appropriate operation depending on the operands' types. Method *incarnating* (§16.5.2) allows multiple methods that provide a common operation on different parameter types to share a common name.

As is the case with the built-in addition operation, we may want to define a set of functions that perform the same general action but that apply to different parameter types. The implementation details of how that is accomplished are of little interest to the users of the function. This lexical complexity reflects a limitation of the programming environment in which each name occurring at the same scope must refer to a unique entity. Such complexity presents a practical problem to the programmer, who must remember or look up each name. Method template incarnating relieves the programmer of this lexical complexity.

13.6.1. Method template

ATDL provides many facilities to ease the use of functions in ATDL programs. The first such facility is different incarnated methods. Methods that provide a common operation but that operate on different data types and require differing implementations may share a common name. This capability eases the use of methods, because programmers do not have to remember different method names for a same operation.

A second facility supported in ATDL to ease the use of methods is method templates. A method template is a generic method definition that is used to automatically generate an infinite set of method definitions that vary by type but whose implementations remain invariant.

The declaration of a method template *m* must appear within a virtual class (call it *A*); otherwise a compile-time error results. The descendant classes of *A* that is not virtual must provide an incarnation for *m*, or a compile-time error occurs.

13.6.2. Incarnating

Method templates can be *incarnated* in descendant classes. A method template specifies how individual methods can be constructed given a set of one or more actual types. If the descendant class is not virtual, this process of construction is referred to as *method template incarnation*. To incarnate a method template, redeclare it in the descendant class. When the method template is incarnated, an actual built-in or user-defined type is substituted for the template type parameter.

In this case, the template argument list explicitly specifies the type of the template argument.

13.6.3. Method template instantiation

The method templates in the base virtual class are always "dummy" methods. That's because the intent of a method template is to create a *common interface* for all the classes derived from it. The only reason to establish this common interface is so it can be expressed differently for each different incarnated type. It establishes a basic form, so you can say what's in common with all the derived classes. All derived-class methods that incarnate the signature of the base-class declaration will be called using the dynamic binding mechanism.

If a virtual class contains one or more method templates, objects of that class almost always have no meaning. A method template specifies how individual functions can be invoked given a set of one or more actual types or values. This process of invocation is referred to as *method template instantiation*. It occurs implicitly as a side effect of invoking a method template. To determine the actual type and value to use as template arguments, the type of the method argument provided on the method call is examined. The process of determining the types and values of the template arguments from the type of the method arguments is called *template argument deduction*.

13.7. Declaring methods

A method is a test case or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example,

SomeObject.Free;

calls the *Free* method in *SomeObject*.

Concrete textual grammar

105 ClassMethodDef ::= ClassVisibility [Virtuality] MethodHeading [RaisesExpr]

107 MethodHeading ::= ["synchronized"] ["class"] RoutineHeading
| ConstructorHeading | DestructorHeading

13.7.1. Method implementations

Within a class declaration, methods appear as test case, altstep and function headings, which work like forward declarations. Somewhere after the class declaration, but within the same scope unit (module or enclosing group), each method must be implemented by a defining declaration.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does so, the order, type, and names of the parameters must match exactly, and, if the method is a function, so must the return value.

When you use an identifier that has been declared in more than one place, it is sometimes necessary to *qualify* the identifier.

Qualifiers can be iterated.

If you don't qualify an identifier, its interpretation is determined by the scope rules [1].

Concrete textual grammar

170 QualifierId ::= (ComponentType | InterfaceIdentifier) Dot

13.7.1.1. Inherited

The reserved word **inherited** plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If **inherited** is followed by a method identifier, it represents a normal method call, except that the search for the method begins with the immediate ancestor of the enclosing method's class. For example, when

inherited Create(...);

occurs in the definition of a method, it calls the inherited *Create*.

When **inherited** has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, **inherited** can appear with or without parameters; if no parameters are specified, it passes to the inherited method the same parameters with which the enclosing method was called.

13.7.2. Method binding

Methods can be *static* (the default), or *virtual*. Virtual methods can be *overridden*, and they can be *abstract*. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Concrete textual grammar

106 Virtuality ::= “final” [“virtual”] | “override” | “overload” | “external” | “template”

13.7.2.1. Class methods

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word **class**.

The defining declaration of a class method must also begin with **class**. In the defining declaration of a class method, the identifier *Self* represents the class where the method is called.

A class method can be called through an object reference. When it is called through an object reference, the class of the object becomes the value of *Self*.

A method that is not declared with the **class** directive is called an *instance method*. An instance method is always invoked with respect to an object, which becomes the current object to which the keywords self and inherited refer during execution of the method body.

13.7.2.2. Static instance methods

Instance methods are by default static. When a static instance method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the *Execute* methods are static.

```
class TTestcase { testcase Execute; }
class TIpTest extends TTestcase { testcase Execute; }
```

Given these declarations, the following code illustrates the effect of calling a static instance method. In the second call to *Testcase.Execute*, the *Testcase* variable references an object of class *TIpTest*, but the call invokes the implementation of *Execute* in *TTestcase*, because the declared type of the *Testcase* variable is *TTestcase*.

EXAMPLE 1:

```
var Testcase TTestcase;
var IpTest TIpTest;
Testcase := TTestcase.Create;
Testcase.Execute;    // calls TTestcase.Execute
Testcase.Destroy;
Testcase := TIpTest.Create;
Testcase.Execute;    // calls TTestcase.Execute
Testcase.Destroy;
IpTest := TIpTest.Create;
IpTest.Execute;      // calls TIpTest.Execute
IpTest.Destroy;
```

13.7.2.3. Abstract methods

An abstract method declaration introduces the method as a member, providing its signature (name and number and type of parameters), return type, and raises clause (if any), but does not provide an implementation. All methods declared in a virtual class (§13.6) are implicitly abstract, it is not required for the declarations of such methods to redundantly include the abstract keyword.

13.7.2.4. Virtual instance methods

It is a compile-time error for a private method to be declared virtual. It would be impossible for a descendant class to implement a private virtual method, because private methods are not visible to descendant classes; therefore such a method could never be used.

It is a compile-time error for a class method to be declared virtual.

To make an instance method virtual, include the **virtual** directive in its declaration. Virtual methods, unlike static methods, can be *overridden* in descendant classes. When an overridden method is called, the actual (runtime) type of the object used in the method call — not the declared type of the variable — determines which implementation to activate.

Virtual methods are the most efficient way to implement polymorphic behavior. An **override** declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any). To override an instance method, re-declare it with the **override** directive. Moreover, an instance method declaration must not have a **raises** clause that conflicts with that of any method that it overrides.

In the following example, the *Execute* method declared in *TTestcase* is overridden in two descendant classes.

class TTestcase {**virtual testcase** Execute}
class TIpTest **extends** TTestcase {**override testcase** Execute}
class TSipTest **extends** TTestcase {**override testcase** Execute}

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime. For example,

```
var Testcase TTestcase;
Testcase := TIpTest.Create;
Testcase.Execute;     // calls TIpTest.Execute
Testcase.Destroy;
Testcase := TSipTest.Create;
Testcase.Execute;     // calls TSipTest.Execute
Testcase.Destroy;
```

Only virtual methods can be overridden. All methods, however, can be *overloaded.*

13.7.2.5. Final methods

A virtual method can be declared final to prevent descendant classes from overriding or hiding it. A static method can be declared final to prevent descendant classes from hiding it. It is a compile-time error to attempt to hide a final static method.

It is a compile-time error to attempt to override or hide a final virtual method. A private method and all methods declared in a final class are implicitly final, because it is impossible to hide them. All virtual methods declared in a final class are implicitly final, because it is impossible to override or to hide them. It is permitted but not required for the declarations of such methods to redundantly include the final keyword.

It is a compile-time error for a final method to be declared abstract.

13.7.2.6. External methods

The **external** directive, which replaces the block in a test case or function declaration, allows you to call test cases and functions that are compiled separately from your program. A method that is external is implemented in platform-dependent code, typically written in another programming language such as assembly language.

13.7.3. Inheritance, overriding, and hiding

A class *inherits* from its ancestors all the methods (whether **virtual** or not) of the ancestors.

13.7.3.1. Overriding versus implementing

If a class declares an instance method, then the declaration of that method is said to *override* the virtual method with the same signature in the ancestor classes. To override an instance method, re-declare it with the **override** directive. Moreover, if the descendant class is not virtual, then the declaration of that method is said to *implement* any and all declarations of abstract methods with the same signature in the ancestor classes or to *implement* the operation with the same signature in the ancestor operational interfaces of the class.

A compile-time error occurs if an instance method overrides a class method.

13.7.3.2. Hiding

If a class declares a virtual method, then the declaration of that virtual method is said to *hide* an inherited virtual method with a new one. The virtual directive suppresses compiler warnings about hiding previously declared virtual methods.

If a class declares a class method, then the declaration of that method is said to *hide* any and all class methods with the same signature in the ancestor classes. A compile-time error occurs if a class method hides an instance method.

If an instance method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn't include **override**, the new declaration merely *hides* the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

class Test1 **extends** TObject {**virtual testcase** Execute;}
class Test2 **extends** Test1 {**testcase** Execute;} // Execute is redeclared, but not overridden

Given these declarations, the following code illustrates the effect of hiding a virtual method.

var SomeObject Test1;
SomeObject := Test2.Create;
SomeObject.Execute; // calls Test1.Execute

If a class declares a class method, then the declaration of that method is said to *hide* any and all methods with the same signature in the ancestor classes of the class.

A hidden method can be accessed by using a qualified name or by using a method invocation expression that contains the keyword inherited or a cast to an ancestor class type. In this respect, hiding of methods is similar to hiding of fields.

13.7.3.3. Requirements in Overriding and Hiding

If a method declaration overrides or hides the declaration of another method, then a compile-time error occurs if they have different return types or if one has a return type and the other does not have. Moreover, a method declaration must not have a **raises** clause that conflicts with that of any method that it overrides or hides; otherwise, a compile-time error occurs.

The access modifier of an overriding or hiding method must provide at least as much access as the overridden or hidden method, or a compile-time error occurs. In more detail:

- If the overridden or hidden method is public, then the overriding or hiding method must be public; otherwise, a compile-time error occurs.
- If the overridden or hidden method is protected, then the overriding or hiding method must be protected or public; otherwise, a compile-time error occurs.

Note that a private method is never accessible to descendant classes and so cannot be hidden or overridden in the technical sense of those terms. This means that a descendant class can declare a method with the same signature as a private method in one of its ancestor classes, and there is no requirement that the return type or raises clause of such a method bear any relationship to those of the private method in the ancestor class.

13.7.4. Overloading methods

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but different signatures, then the method name is said to be *overloaded*. This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the **raises** clauses of two methods with the same name but different signatures.

If you overload a virtual method, use both the virtual and overload directives when you redeclare it in descendant classes.

A method can be redeclared using the **overload** directive. In this case, if the re-declared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. For example,

```
class T1 extends TObject {overload testcase Test(I Smallint); }
class T2 extends T1 {overload testcase Test(S charstring); }
…
SomeObject := T2.Create;
SomeObject.Test('Hello!');     // calls T2.Test
SomeObject.Test(7);            //  calls T1.Test
```

When a method is invoked (§20.6), the number of actual arguments and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked. Calling the method in a descendant class activates whichever implementation matches the parameters in the call. .

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see clause “Overloading test cases and functions” on page 111.

13.7.5. Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory.The declaration of a destructor looks like a testcase declaration, but it begins with the word **destructor**. Examples:

```
destructor Destroy;
override destructor Destroy;
```

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

The last action in a destructor’s implementation is typically to call the inherited destructor to destroy the object’s inherited fields.

When a self-exception is raised during creation of an object, default destructor is automatically called to dispose of the unfinished object. This means that the destructor must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and dynamic-array fields in a partially constructed object are always **null**. A destructor should therefore check for **null** values before operating on class-type or dynamic-array type fields.

Concrete textual grammar

161 DestructorHeading ::= “destructor” [QualifierId] DestructorIdentifier [FormalParList]

162 DestructorIdentifier ::= Identifier

13.7.6. Raises expressions

A **raises** expression specifies which checked exceptions may be raised as a result of an invocation of a method or constructor. The syntax for its specification is as follows:

ATDL concrete textual grammar

186 RaisesExpr ::= “raises” “(“ ExceptionName {“,” ExceptionName}* “)”

The *ExceptionName*s in the raises expression must be previously defined exceptions.

For each checked exception that can result from execution of the body of a method or constructor, a compile-time error occurs unless that exception type or an ancestor class of that exception type is mentioned in a raises clause in the declaration of the method or constructor. The requirement to declare checked exceptions allows the compiler to ensure that code for handling such error

conditions has been included. Methods or constructors that fail to handle exceptional conditions raised as checked exceptions will normally result in a compile-time error because of the lack of a proper exception type in a **raises** clause. ATDL thus encourages a programming style where rare and otherwise truly exceptional conditions are documented in this way.

A method that overrides or hides another method (§13.7.2), including methods that implement operations defined in interfaces, may not be declared to raise more checked exceptions than the overridden or hidden method.

13.8. Declaring constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a testcase declaration, but it begins with the word **constructor**. Examples:

```
constructor Create;
constructor Create(AOwner TComponent);
```

A class can have more than one constructor, but most have only one. It is conventional to call the constructor *create*.

Although the declaration specifies no return value, when a constructor is called using a class reference, it returns a reference to the object it creates. To create an object, call the constructor method in a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

When a constructor is called using an object reference, it does not create an object or return a value. Instead, the constructor operates on the specified object, executing only the statements in the constructor's implementation. A constructor is always invoked on an object reference in conjunction with the reserved word **inherited** to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
class TShape extends TGraphicControl {
        FPenMessage TPenMessage default PenChanged;
        FBrushMessage TBrushMessge default BrushChanged;
        altstep PenChanged(Sender TObject);
        altstep BrushChanged(Sender TObject);
        public override constructor Create(Owner TComponent);
        public override destructor Destroy;
        :
        };

constructor TShape.Create(Owner TComponent) {
        inherited Create(Owner);   // Initialize inherited parts
        Width := 65;               // Change inherited properties
        Height := 65;
        FPen := null;              // Initialize new fields
        FBrush := null;
        } ;
```

There is no practical need for a constructor to be **synchronized**, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their

Concrete textual grammar

159 ConstructorHeading ::= “constructor” [QualifierId] ConstructorIdentifier [FormalParList]

160 ConstructorIdentifier ::= Identifier

In a configuration diagram at specification level, a constructor operation declaration is included as one of the methods in the method list of the class. It may have a parameter list, but the return value is implicitly an instance of the class and may be omitted.

13.8.1. Constructor body

The first statement of a constructor body may be an explicit invocation of another constructor of the same class, written as *self* followed by the constructor name and a parenthesized argument list, or an explicit invocation of a constructor of the ancestor class, written as *inherited* followed by the constructor name and a parenthesized argument list.

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving *self*.

13.8.2. Constructor overloading

Overloading of constructors is identical in behavior to overloading of methods. The overloading is resolved at compile time by each component instance creation expression (§20.4).

13.8.3. Default constructor

If a class contains no constructor declarations, then a *default constructor* (**create**) that takes no parameters is automatically provided. The default constructor takes no parameters and simply invokes the ancestor class constructor with no arguments.

13.8.4. Raises expressions

The **raises** expression for a constructor is identical in structure and behavior to the **raises** expression for a method.

13.9. Class references

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, passive objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*. Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Concrete textual grammar

323 ClassRefType ::= “class” “of” ClassIdentifer

13.10. Coordinating threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful

not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

13.10.1. Synchronized fields

Synchronized fields work like gates that allow only a single thread to enter at a time. Each synchronized field is associated with the global memory you want to protect.

There is a lock associated with every synchronized field. The ATDL language does not provide a way to perform separate *lock* and *unlock* actions; instead, they are implicitly performed by high-level constructs that arrange always to pair such actions correctly. Every thread that accesses that global memory should first implicitly use the *Lock* method to ensure that no other thread is using it. When finished, threads call the *Unlock* method implicitly so that other threads can access the global memory by implicitly calling *Lock*.

13.10.2. Synchronized methods

When you use synchronized fields to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. Best practice is that if a variable is ever to be assigned by one thread and used or assigned by another, then all accesses to that variable should be enclosed in **synchronized** methods.

A synchronized method acquires a lock before it executes. Acquiring the lock associated with an object does not of itself prevent other threads from accessing fields of the object or invoking unsynchronized methods on the object. For example,

```
class TBox {
        private boxContents TObject;
        public synchronized function retrieve () return TObject;
        public synchronized function put (content TObject) return boolean;
}
function TBox.retrive() return TObject {
        var contents TObject := boxContents;
        boxContents := null;
        return contents;
        }
function TBox.put(content TObject) return boolean {
        if [ boxContents != null ] return false;
        boxContents := contents;
        return true;
        }
```

defines a class which is designed for concurrent use. Each instance of the class TBox has an instance variable contents that can hold a reference to any object. You can put an object in a TBox by invoking put, which returns false if the box is already full. You can get something out of a TBox by invoking retrieve, which returns a null reference if the tbox is empty.

If put and retrieve were not synchronized, and two threads were executing methods for the same instance of TBox at the same time, then the code could misbehave. It might, for example, lose track of an object because two invocations to put occurred at the same time.

A **synchronized** method automatically performs a *lock* action when it is invoked; its body is not executed until the *lock* action has successfully completed. If the method is an instance method, it locks the lock associated with the instance for which it was invoked (that is, the object that will be known as this during execution of the body of the method). If execution of the method's body is

ever completed, either normally or abruptly, an *unlock* action is automatically performed on that same lock.

13.11. Exceptions

You will probably have noticed many similarities between exception handling and function calls. A **raise** statement behaves somewhat like a function call, and the **catch** clause behaves somewhat like a function definition. The main difference between these two mechanisms is that all the information necessary to set up a function call is available at compile-time, and that is not true for the exception handling mechanisms in other object-oriented languages. The exception handling mechanisms commonly found in other object-oriented languages, require run-time support.

In ATDL, the exception declaration of a **catch** clause can be changed to a reference declaration. The **catch** clause then directly refers to the exception object created by the **raise** statement instead of creating a local copy.

13.11.1. Self-exceptions

In the case of a self-connection, when an ATDL program violates the semantic constraints of the ATDL language, the underlying test system signals this error to the program as a *self-exception*. An example of such a violation is an attempt to index outside the boundaries of an array. Some programming languages and their implementations react to such errors by peremptorily terminating the program, this approach is not compatible with the design goals of ATDL. Instead, ATDL specifies that an exception will be raised when semantic constraints are violated and will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be *raised* from the point where it occurred and is said to be *caught* at the point to which control is transferred.

13.11.2. Compile-Time Checking of Exceptions

An ATDL compiler may check, at compile time, that an ATDL program contains exception handlers for *checked exceptions*, by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the raises clause for the method (clause 13.7.6) or constructor (§13.8.4) must mention the class of that exception or one of the ancestor classes of the class of that exception. This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled.

The checked exception classes named in the **raises** clause are part of the association contract between the implementor and user of the method or constructor. The **raises** clause of an overriding method may not specify that this method will result in throwing any checked exception which the overridden method is not permitted, by its **raises** clause, to throw. When operational interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a **raises** clause that is compatible with *all* the overridden declarations (§12.3.3).

13.11.3. Unchecked exceptions

In addition to any user-defined exceptions specified in the **raises** expression, there are a standard set of exceptions that may be signalled by the ATDL underlying system. The absence of a **raises** expression on an operation implies that there are no user-defined exceptions. Invocations of such an operation are still liable to receive one of the standard exceptions.

The *standard exceptions* are exempted from compile-time checking because, in the judgment of the designers of ATDL, having to declare such exceptions would not aid significantly in establishing the correctness of ATDL programs. Many of the operations and constructs of the ATDL language can result in runtime exceptions. Requiring such exceptions to be declared would simply be an irritation to ATDL programmers.

13.11.4. The exceptions handling

An exception instance denotes that an exceptional situation (typically an error situation) has occurred while interpreting a system. An exception instance is created implicitly by the underlying system or explicitly by a **raise** operation and the exception instance ceases to exist if it is caught by a **catch** operation or **timeout** exception.

ATDL exception handling uses the **try**, **catch**, and **raise** statements to implement exception handling. With ATDL exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control.

A statement or expression is *dynamically enclosed* by a **catch** clause if it appears within the **try** block of the **try** statement of which the **catch** clause is a part, or if the caller of the statement or expression is dynamically enclosed by the **catch** clause.

The caller of a statement or expression depends on where it occurs. If within a method, then the caller is the method invocation expression that was executed to cause the method to be invoked. If within a constructor, then the caller is the component instance creation expression that was executed to cause an object to be created.

13.11.4.1. Handling of a self-exception

An exception instance transfers control to an exception handler. When a self-exception is raised, control is transferred from the code that caused the exception to the nearest dynamical-enclosing **catch** clause of a **try** statement (§19.5.1) that handles the exception.

The control transfer that occurs when a self-exception is raised causes abrupt completion of expressions and statements until a **catch** clause is encountered that can handle the exception; execution then continues by executing the block of that **catch** clause. The self-exception handling mechanism is said to be *nonresumptive*; once the exception has been handled, the execution of the program does not resume where the exception was originally raised.

14. Declaring variables

Variables are denoted by the keyword **var**. Variables can be declared and used in the module control part, test cases, functions and altsteps. Additionally, variables can be declared in component type definitions. Variables declared within a test case or function are sometimes called *local variable*, while variables declared within a class are called *class variable*. Variables shall not be declared or used in a module definitions part. A variable declaration may have an optional initial value assigned to it. For example:

var MyVar1 Smallint := 1;
var MyVar2 **boolean**:= **true**, MyVar3 **boolean**:= **false**;

Variables should be bound before use, unless they appear on the left-hand side of an assignment. Use of un-initialized variables at runtime shall cause a test case error.

Concrete textual grammar

237 VarInstance ::= "var" SingleVarInstance {"," SingleVarInstance}*
238 SingleVarInstance ::= VarIdentifier {Colon VarIdentifier}* VarInitializer
239 VarInitializer ::= Type [AssignmentChar VarInitialValue]
240 VarInitialValue ::= Expression
241 VarIdentifier ::= Identifier

14.1. Kinds of variables

There are six kinds of variables:

1. A *class variable* is a variable declared within a class declaration. A class variable is created when its class is initialized and may be initialized to a default value. The class variable effectively ceases to exist when its class is destroyed.

2. An *instance variable* is a field declared within a class declaration. If a class *T* has a field *a* that is an instance variable, then a new instance variable *a* is created and initialized to a default value as part of each newly created object of class *T* or of any class that is a descendants of *T*. The instance variable effectively ceases to exist when the object of which it is a field is no longer referenced, after the destructor has been called to destroy the unfinished object.

3. *Thread-local* (or *thread*) variables are used in multithreaded applications. A thread-local variable is also called a thread-local field, each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared within thread class definitions.

4. *Method parameters* name argument values passed to a method. For every parameter declared in a method declaration, a new parameter variable is created each time that method is invoked. The new variable is initialized with the corresponding argument value from the method invocation. The method parameter effectively ceases to exist when the execution of the body of the method is complete.

5. *Constructor parameters* name argument values passed to a constructor. For every parameter declared in a constructor declaration, a new parameter variable is created each time a class instance creation expression or inherited constructor invocation invokes that constructor. The new variable is initialized with the corresponding argument value from the creation expression or constructor invocation. The constructor parameter effectively ceases to exist when the execution of the body of the constructor is complete.

6. *Local variables* are declared by local variable declaration statements. Whenever the flow of control enters a statement block or **for** statement, a new variable is created for each local variable declared in a local variable declaration statement immediately contained within that block or **for** statement. A local variable declaration statement may contain an expression which initializes the variable. The local variable with an initializing expression is not initialized, however, until the local variable declaration statement that declares it is executed. The local variable effectively ceases to exist when the execution of the block or **for** statement is complete.

15. Declaring templates

Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. Templates provide the following possibilities:

a) they are a way to organize and to re-use test data, including a simple form of inheritance;
b) they can be parameterized;
c) they allow matching mechanisms;
d) they can be used with either asynchronous or operational communications.

Within a template values, ranges and matching attributes can be specified and then used in both asynchronous and operational communications. Templates may be specified for any ATDL type or operation signature. The type-based templates are used for asynchronous communications and the operation templates are used in operational communications.

Concrete textual grammar

```
50  TemplateDef ::= “template” BaseTemplate [DerivedDef] AssignmentChar TemplateBody
51  BaseTemplate ::= TemplateIdentifier [FormalCrefParList]
                    (MessageIdentifier | Operation | ExceptionTypeIdentifier)
52  TemplateIdentifier ::= Identifier
56  TemplateBody ::= TemplateValue&Attributes | FieldSpecList
57  TemplateValue&Attributes ::= TemplateValue [ ValueAttributes ]
58  FieldSpecList ::= “{“ [FieldSpec {“,” FieldSpec}*] “}”
```

59 FieldSpec ::= FieldReference AssignmentChar TemplateBody

60 FieldReference ::= StructFieldIdentifier | ArrayOrBitRef | OperationParIdentifier

61 OperationParIdentifier ::= ValueParIdentifier

64 TemplateValue ::= SingleConstExpression | MatchingSymbol | TemplateRefWithPara

72 TemplateInstance ::= InLineTemplate

74 TemplateRef ::= [ModuleName Dot] TemplateIdentifier | TemplateParIdentifier

79 TemplateOps ::= "value" "of" "(" TemplateInstance ")" | "value"

15.1. Declaring message templates

Instances of messages with actual values may be specified using templates. A template can be thought of as being a set of instructions to build a message for sending or to match a received message. Templates may be specified for any ATDL type. However, it is anticipated that the most common use will be with sequences, as shown by the examples in the following clauses.

Templates for communication operations can have an explicit list of formal parameters. The formal parameters of a template can include templates, functions and the special matching symbols.

To enable matching attributes to be passed as parameters the extra keyword **template** shall be added before the type field. This makes the parameter a template and in effect extends the allowed parameters for the associated type to include the appropriate set of matching attributes as well as the normal set of values. Template parameter fields shall not be called by reference.

15.1.1. Templates for receiving messages

A template used in a receiving operation defines a data template against which an incoming message is to be matched. Matching mechanisms, may be used in receive templates. No binding of the incoming values to the template shall occur.

15.2. Parameterization of templates

Templates for both sending and receiving operations can be parameterized. The actual parameters of a template can include values and templates, functions and special matching symbols.

15.2.1. Parameterization with matching attributes

To enable templates or matching symbols to be passed as parameters the extra keyword **template** shall be added before the type field. This makes the parameter a template-type and in effect extends the allowed parameters for the associated type to include the appropriate set of matching attributes (see §15.7) as well as the normal set of values. Template parameter fields shall not be called by reference.

15.2.2. Templates reference

The templates reference shall be present in conjunction with sending and receiving methods. It shall not be present with any other kind of ATDL statement.

Concrete textual grammar

73 TemplateRefWithPara ::= [ModuleName Dot] TemplateIdentifier [ActualCrefParList] |
TemplateParIdentifier

77 ActualCrefParList ::= "(" ActualCrefPar {"," ActualCrefPar}* ")"

78 ActualCrefPar ::= [VarIdentifier AssignmentChar] TemplateInstance | Type

15.3. Template matching mechanisms

Generally, matching mechanisms are used to replace values of single template fields or to replace even the entire contents of a template. Some of the mechanisms may be used in combination.

Matching mechanisms and wildcards may also be used in-line in received events only (i.e. **receive**, **synchronize**, **trigger** and **catch** operations). They may appear in explicit values.

Table 19: ATDL Matching Mechanisms

Used with values of	Value		Constructed Value			Instead Of Value		Inside Value			Attributes	
Type	**Specific Value**	**Omit (-)**	**Complement**	**ValueList**	**Range**	**AnyValue (?)**	**AnyOrOmit (*)**	**AnyOne (?)**	**AnyOrNone (*)**	**Permutation**	**Length**	**IfPresent**
boolean	Y	Y	Y	Y		Y	Y					Y
integer	Y	Y	Y	Y	Y	Y	Y				Y	Y
cardinal	Y	Y	Y	Y	Y	Y	Y				Y	Y
char	Y	Y	Y	Y	Y	Y	Y					Y
wide char	Y	Y	Y	Y	Y	Y	Y					Y
float	Y	Y	Y	Y		Y	Y					Y
real	Y	Y	Y	Y		Y	Y				Y	Y
enumerated	Y	Y	Y	Y		Y	Y					Y
bitstring	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
octetstring	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
hexstring	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
character strings	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
sequence	Y	Y	Y	Y		Y	Y					Y
sequence of	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
set	Y	Y	Y	Y		Y	Y					Y
set of	Y	Y	Y	Y		Y	Y	Y	Y		Y	Y
objid	Y	Y	Y	Y		Y	Y					Y
choice	Y	Y	Y	Y		Y	Y					Y

For example:

MyPCO.**receive** (MyCharstring : "abcxyz");

MyPCO.**receive** (MyInteger: **complement**(1, 2, 3));

The message identifier may be omitted when the value unambiguously identifies the message.

However, the message identifier of the in-line template shall be in the interface definition over which the template is received. In the case where there is an ambiguity between the listed message and the type of the value provided (e.g. through sub-typing) then the message name shall be included in the receive statement.

Matching mechanisms are arranged in five groups:

a) specific values (i.e., an expression that evaluates to a specific value);

b) specific symbols that can be used *constructed* values;

c) special symbols that can be used *instead* of values;

d) special symbols that can be used *inside* values;

e) special symbols which describe *attributes* of values.

An overview of the supported matching mechanisms is defined in Table 19, including the special symbols and the scope of their application. The left-hand column of this table lists all the ATDL and ASN.1 equivalent types as defined in the ITU-T Recommendation X.680 series [8], [9], [10] and [11] to which these matching mechanisms apply.

A full description of each matching mechanism can be found in clause 15.7. Additional supported matching mechanisms and their associated symbols (if any) and the scope of their application are shown in Table 20.

15.4. Modified templates

15.4.1. General

Normally a template specifies a set of base or default values or matching symbols for each and every field defined in the appropriate definition. In cases where small changes are needed to specify a new template it is possible to specify a modified template. A modified template specifies modifications to particular fields of the original template, either directly or indirectly.

The **modifies** keyword denotes the parent template from which the new, or modified template shall be derived. This parent template may be either the original template or a modified template.

The modifications occur in a linked fashion eventually tracing back to the original template. If a template field and its corresponding value or matching symbol is specified in the modified template, then the specified value or matching symbol replaces the one specified in the parent template. If a template field and its corresponding value or matching symbol is not specified in the modified template, then the value or matching symbol in the parent template shall be used. When the field to be modified is nested within a template field which is a structured field itself, no other field of the structured field is changed apart from the explicitly denoted one(s).

A modified template shall not refer to itself, either directly or indirectly i.e. recursive derivation is not allowed.

Concrete textual grammar

53 DerivedDef ::= “modifies” TemplateRef

15.4.2. Parameterization of modified templates

If a base template has a formal parameter list, the following rules apply to all modified templates derived from that base template, whether or not they are derived in one or several modification steps: a) the derived template shall not omit parameters defined at any of the modification steps between the base template and the actual modified template; b) a derived template can have additional (appended) parameters if wished; c) the formal parameter list shall follow the template name for every modified template; d) base template fields containing parameterized templates shall not be modified or explicitly omitted in a modified template.

15.4.3. In-line modified templates

As well as creating explicitly named modified templates ATDL allows the definition of in-line modified templates.

template Setup MyMessageType :=

{ field1 := 75, field2 := "abc", field3 := **true** }

// Could be used to define an in-line modified template of Setup

Pco1.**send** (**modifies** Setup := {field1 76});

Concrete textual grammar

75 InLineTemplate ::= [(MessageIdentifier | ExceptionTypeIdentifier) InLineMatchingSymbol]
[DereivedDef ":="] TemplateBody

15.5. Changing template fields

All changes to template fields shall only be done via parameterization or by in-line derived templates at the time of performing a communication operation (e.g., **send**, **receive**, **call**, **synchronize** etc.). The effects of these changes on the value of the template field do not persist in the template subsequent to the corresponding communication event.

The notation of the kind *MyTemplateId.Fieldid* shall not be used to set or retrieve values in templates in communication events. The keyword "->" shall be used for this purpose.

15.6. Value of Operation

The **`valueof`** operation allows the value specified within a template to be assigned to the fields of a variable. The variable and template shall be type compatible (see 17.6) and each field of the template shall resolve to a single value. For example,

```
type ExampleType ::= sequence
{
        field1 Smallint,
        field2 boolean
}
template SetupTemplate ExampleType :=
{
        field1 := 1,
        field2 := true
}
:
var RxValue ExampleType := value of (SetupTemplate);
```

15.7. Matching incoming values

This clause specifies the matching mechanisms that may be used in ATDL templates (and only incoming templates).

15.7.1. In-line matching operators

The following operators may also be used in-line in communication operations.

Table 20: ATDL in-line matching operators

Operator	Operation	Operand types	Used in
:=	assignment	**any type**	The sending operations
in	value set membership	**any type**	The receiving operations
==	equality	**any type**	The receiving operations
<=	sub value set	**set of**	The receiving operations
>=	super value set	**set of**	The receiving operations

The value set matching operater allows the value of a variable to be compared with a template. The operation returns a boolean value. If the types of the template and variable are not compatible (see §17.6) the operation returns false. If the types are compatible the return value of the operation indicates whether the value of the variable conforms to the specified template.

```
template LessThan10 MyInteger := (-infinity..9);
testcase MyMTCType.TC001()
{
        var RxValue MyInteger;
        :
        PCO1.receive(MyInteger in ?) -> (RxValue := value;)
        if [ RxValue in LessThan10 ] { … }
        // true if the actual value of Rxvalue is less than 10 and false otherwise
        :
}
```

Concrete textual grammar

76 InLineMatchingSymbol ::= AssignmentChar | Colon | ">=" | "<=" | "=="

15.7.1.1. SuperSet

Super Set is an operator for matching that shall be used only on values of **set of** types. Super Set is denoted by the symbol ">=". A message that uses Super Set matches the corresponding incoming message if, and only if, the incoming message contains at least all of the elements defined within the Super Set, and may contain more. For example,

```
interface MyInterface { out MySetOfType set of Smallint; }
var MyChannel := MyInterface(TSenderObject);
MyChannel.receive (MySetOfType >= (1, 2, 3));
// any sequence of integers matches which contains at least one occurrences
// of the numbers 1, 2 and 3 in any order and positions
```

15.7.1.2. SubSet

Sub Set is an operation for matching that can be used only on values of **set of** types. Sub Set is denoted by the symbol "<=". A message that uses Sub Set matches the corresponding incoming message if, and only if, the incoming message contains only elements defined within the Sub Set, and may contain less. To use our previous examples,

```
MyChannel.receive (MySetOfType <= (1, 2, 3));

// any sequence of integers matches which contains zero or one occurrences

// of the numbers 1, 2 and 3 in any order and positions
```

15.7.2. Matching specific values

Specific values are the basic matching mechanism of ATDL templates. Specific values in templates are expressions which do not contain any matching mechanisms or wildcards. Unless otherwise specified, a template field matches the corresponding incoming field value if, and only if, the incoming field value has exactly the same value as the value to which the expression in the template evaluates. For example,

```
// Given the message type definition

interface MyMessageInterface {...;

        MyMessageType   sequence
        {
```

```
                              field1    Smallint,
                              field2    charstring ,
                              field3    boolean  optional,
                              field4    sequence [3] of Smallint
                    }
...}
```

// A message template using specific values could be

```
template MyTemplate MyMessageType :=
{
        field1 := 3+2,              // specific value of integer type
        field2 := "My string",      // specific value of charstring type
        field3 := true,             // specific value of boolean type
        field4 := {1,2,3}           // specific value of integer array
}
```

15.7.2.1. Omit

Omit is a special symbol for matching that can be used on values of all types, provided that the struct field is optional.

Concrete textual grammar

68 Omit ::= “omit”

15.7.3. Constructed value

15.7.3.1. Value List

Value lists specify lists of acceptable incoming values. It can be used on values of all types. A template field that uses a value list matches the corresponding incoming field if, and only if, the incoming field value matches any one of the values in the value list. Each value in the value list shall be of the type declared for the template field in which this mechanism is used. ValueLists are denoted by a parenthesized list of values separated by commas.

Concrete textual grammar

71 ValueList ::= “(“ TemplateBody {“,” TemplateBody}* “)”

15.7.3.2. Complement

The keyword **complement** denotes a list of values that will not be accepted as incoming values (i.e. it is the complement of a value list). It can be used on all values of all types.

Each value in the list shall be of the type declared for the template field in which the complement is used. A template field that uses complement matches the corresponding incoming field if and only if the incoming field does not match any of the values listed in the value list. The value list may be a single value, of course.

15.7.3.3. Value ranges

Ranges indicate a bounded range of acceptable values, when used for values of **integer, cardinal, float,** or **real** types (and integer, cardinal, float, or real sub-types). A boundary value shall be either: a) infinity or -infinity; b) an expression that evaluates to a specific integer, cardinal, real or float value.

The lower boundary shall be put on the left side of the range, the upper boundary at the right side. The lower boundary shall be less than the upper boundary. A template field that uses a range

matches the corresponding incoming field if, and only if, the incoming field value is equal to one of the values in the range.

When used in templates or template fields of **char, wide char, charstring** or **wide charstring** types, the boundaries shall evaluate to valid character positions according to the coded character set table(s) of the type (e.g. the given position shall not be empty). Empty positions between the lower and the upper boundaries are not considered to be valid values of the specified range.

Concrete textual grammar

43 IntegerRange ::= "(" LowerBound ".." UpperBound ")"

44 LowerBound ::= SingleConstExpression | Minus "infinity"

45 UpperBound ::= SingleConstExpression | "infinity"

15.7.4. Instead of Value

The following matching mechanisms may be used in place of explicit values.

Concrete textual grammar

65 MatchingSymbol ::= CharStringMatch | Omit | AnyValue | AnyOrOmit | ValueList | IntegerRange

15.7.4.1. Any value

The matching symbol "?" (*AnyValue*) is used to indicate that any valid incoming value is acceptable. It can be used on values of all types. A template field that uses the any value mechanism matches the corresponding incoming field if, and only if, the incoming field evaluates to a single element of the specified type. For example,

```
template  MyTemplate MyMessageType :=
{
    field1 := ?,        // will match any integer
    field2 := ?,        // will match any non-empty charstring value
    field3 := ?,        // will match true or false
    field4 := ?         // will match any sequence of integers
}
```

Concrete textual grammar

69 AnyValue ::= "?"

15.7.4.2. Any value or none

The matching symbol "*" (*AnyValueOrNone*) is used to indicate that any valid incoming value, including omission of that value, is acceptable. It can be used on values of all types, provided that the template field is declared as optional.

A template field that uses this symbol matches the corresponding incoming field if, and only if, either the incoming field evaluates to any element of the specified type, or if the incoming field is absent. For example,

```
template MyTemplate MyMessageType :=
{ :
    field3 := *,         // will match true or false or omitted field
    :
}
```

Concrete textual grammar

70 AnyOrOmit ::= “*”

15.7.5. Inside Values

Concrete textual grammar

364 Wildcard ::= AnyOne | AnyOrNone

15.7.5.1. Any One

AnyOne is a special symbol for matching that can be used within values of string types and **sequence of**. In both tabular and ATDL templates AnyOne is denoted by "?".

Concrete textual grammar

365 AnyOne ::= “?”

15.7.5.2. Any Or None

AnyOrNone is a special symbol for matching that can be used within values of string types and **sequence of**. In both tabular and ATDL constraints AnyOrNone is denoted by "*".

Concrete textual grammar

366 AnyOrNone ::= “*”

15.7.6. Attributes of values

The following attributes may be associated with matching mechanisms.

66 ValueAttributes ::= LengthRestriction | “ifpresent” | LengthRestriction “ifpresent”

15.7.6.1. Length restriction

The length restriction attribute is used to restrict the length of integer of string values and the number of elements in a **set of,** and **sequence of** structure. It shall be used only as an attribute of the following mechanisms: Complement, *AnyValue*, *AnyValueOrNone*, *Any Element* and *AnyElementsOrNone*. It can also be used in conjunction with the **ifpresent** attribute. The syntax for *length* can be found in clauses 10.3.2 and 10.5.

The units of length are to be interpreted according to Table 16. For **set of**, and **sequence of** types the unit of length is the replicated type. The boundaries shall be denoted by expressions which resolve to specific **cardinal** values. Alternatively, the keyword **infinity** can be used as a value for the upper boundary in order to indicate that there is no upper limit of length.

The length specifications for the template shall not conflict with the length for restrictions (if any) of the corresponding type. A template field that uses Length as an attribute of a symbol matches the corresponding incoming field if, and only if, the incoming field matches both the symbol and its associated attribute. The length attribute matches if the length of the incoming field is greater than or equal to the specified lower bound and less than or equal to the upper bound. In the case of a single length value the length attribute matches only if the length of the received field is exactly the specified value.

Concrete textual grammar

46 LengthRestriction ::= “[“ SingleConstExpression [“..” UpperBound] “]”

15.7.6.2. The IfPresent indicator

The **ifpresent** indicates that a match may be made if an optional field is present (i.e. not omitted). This attribute may be used with all the matching mechanisms, provided the type is declared as optional.

A template field that uses **ifpresent** matches the corresponding incoming field if, and only if, the incoming field matches according to the associated matching mechanism, or if the incoming field is absent.

15.7.7. Matching character pattern

Character patterns can be used in templates to define the format of a required character string to be received. Character patterns can be used to match **charstring** and **wide charstring** values. In addition to literal characters, character patterns allow the use of meta characters ? and * to mean any character and any number of any character respectively.

```
template MyTemplate charstring := pattern "ab??xyz*";
```

This template would match any character string that consists of the characters 'ab', followed by any two characters, followed by the characters 'xyz', followed by any number of any characters.

If it is required to interpret any metacharacter literally it should be preceded with the metacharacter '\'. For example,

```
template MyTemplate charstring := pattern "ab?\?xyz*";
```

This template would match any character string which consists of the characters 'ab', followed by any character, followed by the characters '?xyz', followed by any number of any characters.

Concrete textual grammar

67 CharStringMatch ::= “pattern” Cstring

16. Routines and method templates

In ATDL, functions and altsteps are used to specify and structure test behaviour, define default behaviour and to structure computation in a module etc. as described in the following clauses.

16.1. Functions

Functions are used in ATDL to express test behavior or to structure computation in a module. Functions may return a value. This is denoted by the **return** keyword followed by a type identifier. If no **return** is specified then the function is void. An explicit keyword for void does not exist in ATDL. The keyword **return**, when used in the body of the function, causes the function to terminate and to return a value compatible with the return type. A **raises** expression is used to declare any checked exceptions that can result from the execution of a function.

In a module, the behavior of a function can be defined by using the program statements and operations. If a function includes communication operations the associated component type shall be referenced using the qualifier in the function header to define the number, type and identifiers of the available interfaces. For example:

```
function MyComponent.MyFunction() return Smallint
{
    :
}
```

Instances of different component types may use the same function if they fulfill the method resolution rule. Functions may be parameterized. The rules for formal parameter lists shall be followed as defined in parameter lists clause.

Concrete textual grammar

163 FunctionDef ::= MethodModifier FunctionHeading
| ConstructorHeading | DestructorHeading) StatementBlock

164 FunctionHeading ::= “function” [QualifierId] FunctionIdentifier [FormalParList] [ReturnType]

169 ReturnType ::= “return” Type

166 MethodModifier ::= “overload” | “template | “class”

Concrete graphical grammar

176 <function diagram> ::= <frame symbol>
contains ([“overload”] FunctionHeading
{{<function text area>}* <function graph area> } **set**)

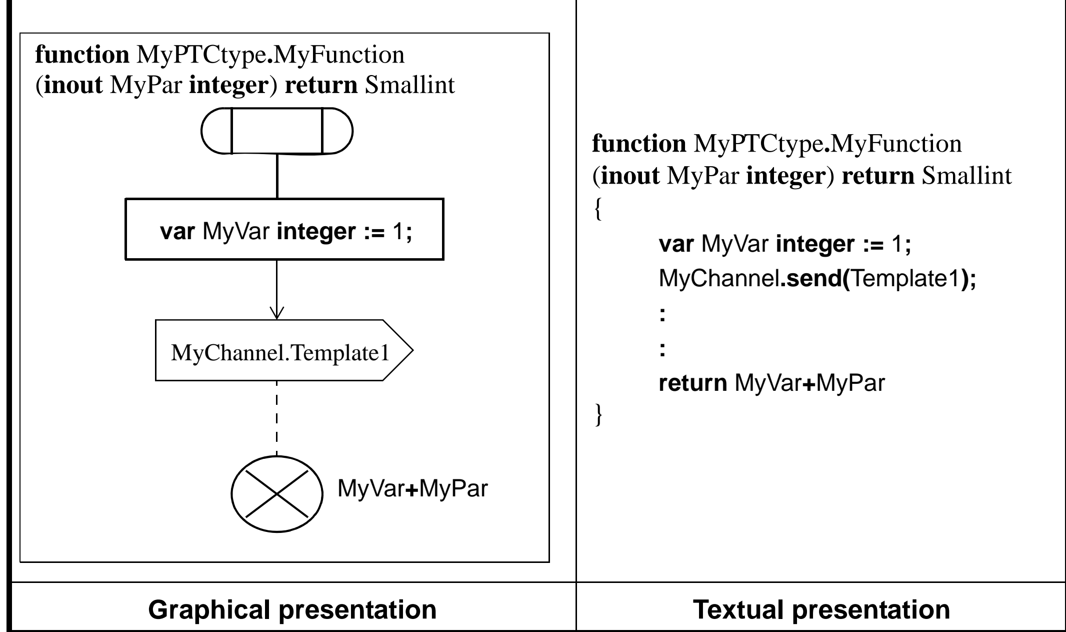
177 <function text area> ::= <control text area>

178 <function graph area> ::= <function start area> **is_followed_by** <function block area>

179 <function block area> ::= <statement block area>
is_followed_by (<stop symbol> | <statement end symbol>)

180 <function start area> ::= <function start symbol> **contains** ([Virtuality])

Figure 10. Principle shape of an ATDL function diagram and corresponding textual language



ATDL/gr presents functions by means of function diagrams. The heading of a function diagram shall be the keyword **function** followed by the complete signature of the function. Complete means that at least function name and parameter list shall be present. The **return** clause and the qualifier identifier are optional in the textual language. If these clauses are specified in the corresponding textual language, they shall also be present in the header of the function diagram.

Attributes associated to the function presented in ATDL/gr shall be specified within a <text symbol> in the <function diagram>. The principle shape of an ATDL function diagram and the corresponding textual description are sketched in Figure 10.

A function is used to specify and structure test behaviour, define default behaviour or to structure computation in a module. A function may contain declarations, statements, communication and timer operations and invocation of function or altsteps and an optional return statement.

16.2. Test cases

Test cases are a special kind of function. The result of an executed test case is always a value of type **verdicttype**. Its purpose is to stimulate the tested classes by creating objects and calling their methods. It provides different ways to check that the objects behavior is the one that was expected. A **raises** expression is used to declare any checked exceptions that can result from the execution of a test case.

Concrete textual grammar

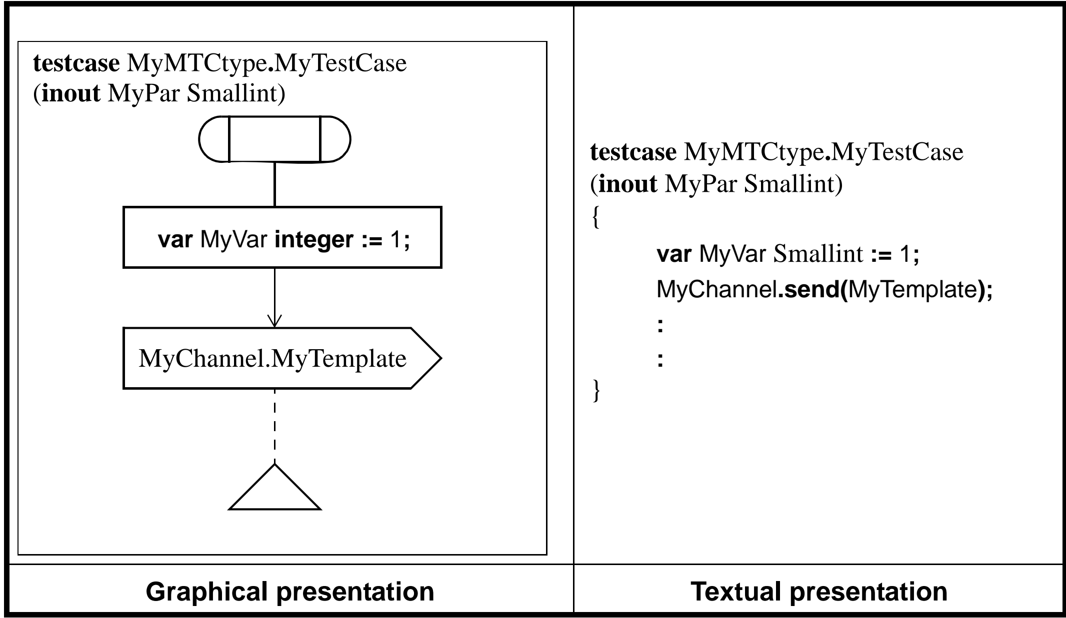
194 TestcaseDef ::= MethodModifier TestcaseHeading StatementBlock

195 TestcaseHeading ::= “testcase” QualifierId TestcaseIdentifier [FormalCrefParList]

16.2.1. Test case diagram

A test case diagram provides a graphical presentation of an ATDL test case. The heading of a test case diagram shall be the keyword **testcase** followed by the complete signature of the test case. Complete means that at least test case name and parameter list shall be present. The qualifier identifier is mandatory in the textual language, it shall also be present in the heading of the test case diagram.

Figure 11. Principle shape of an ATDL test case diagram and corresponding textual language



Attributes associated to the test case presented in ATDL/gr shall be specified within a text symbol in the test case diagram. The principle shape of an ATDL test case diagram and the corresponding textual description are sketched in Figure 11.

A test case represents the dynamic test behaviour and can create test components. A test case may contain declarations, statements, communication and timer class methods and invocation of functions or altsteps.

Concrete graphical grammar

199 <testcase diagram> ::= <frame symbol>
contains ([“overload”] TestcaseHeading
{{<function text area>}* <testcase graph area>} **set**)

200 <testcase graph area> ::= <function start area> is_followed_by <testcase block area>

201 <testcase block area> ::= <statement block area> **is_followed_by** <statement end symbol>

16.2.2. Parameterization of test cases

Test cases may be parameterized. The rules for formal parameter lists shall be followed as defined in clause 5.2.

16.3. Overloading test cases and functions

You can declare more than one method in the same scope with the same name. This is called *overloading*. Overloaded methods must be declared with the **overload** modifier and must have distinguishing parameter lists. For example, consider the declarations

overload function Divide(X **float**, Y **float**) **return float**
{ Result := X/Y; }

overload function Divide(X Smallint, Y Smallint) **return** Smallint
{ Result := X **mod** Y; }

These declarations create two functions, both called *Divide*, that take parameters of different types. When you call *Divide*, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, Divide(6.0, 3.0) calls the first *Divide* function, because its arguments are real-valued.

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

overload function Capital(S **charstring**) return **charstring**;
:
overload testcase Capital(**inout** Str **charstring**);
:

But the declarations

overload function Func(X **float**, Y Smallint) **return float**;
:
overload function Func(X Smallint, Y **float**) **return float**;
:

are legal.

You can limit the potential effects of overloading by qualifying a method's name when you call it. For example, Module1.MyProcedure(X, Y) can call only methods declared in *Modole1*; if no routine in *Module1* matches the name and parameter list in the call, an error results.

16.4. Altsteps

ATDL uses altsteps to specify default behaviour and default signal handler or to structure the alternatives of an **alt** statement. Altsteps are scope units similar to functions. The altstep body defines an optional set of local definitions and a set of alternatives, the so-called *top alternatives*, that forms the altstep body. The syntax rules of the top alternatives are identical to the syntax rules of the alternatives of **alt** statements.

Concrete textual grammar

203 AltstepDef ::= [“overload”] AltstepHeading “{“ AltGuardList “}”
204 AltstepHeading ::= “altstep” [QualifierId] AltstepIdentifier [FormalParList]
205 AltstepIdentifier ::= Identifier
207 AltstepRef ::= [ModuleName Dot] AltstepIdentifier

16.4.1. Parameterization of altsteps

Altsteps may be parameterized. An altstep that is activated as default signal handler shall only have value parameters, i.e. **in** parameters. An altstep that only is invoked as an alternative in an **alt** statement or as stand-alone statement in an ATDL behaviour description may have **in**, **out** and **inout** parameters. The rules for formal parameter lists shall be followed as defined in clause 5.2.

16.4.2. Altstep diagram

ATDL/gr presents altsteps by means of altstep diagrams. The heading of an altstep diagram shall be the keyword **altstep** followed by the complete signature of the altstep. Complete means that at least altstep name and parameter list shall be present. The qualifier identifier is optional in the textual language. If the qualifier identifier is specified in the corresponding textual language, it shall also be present in the header of the altstep diagram.

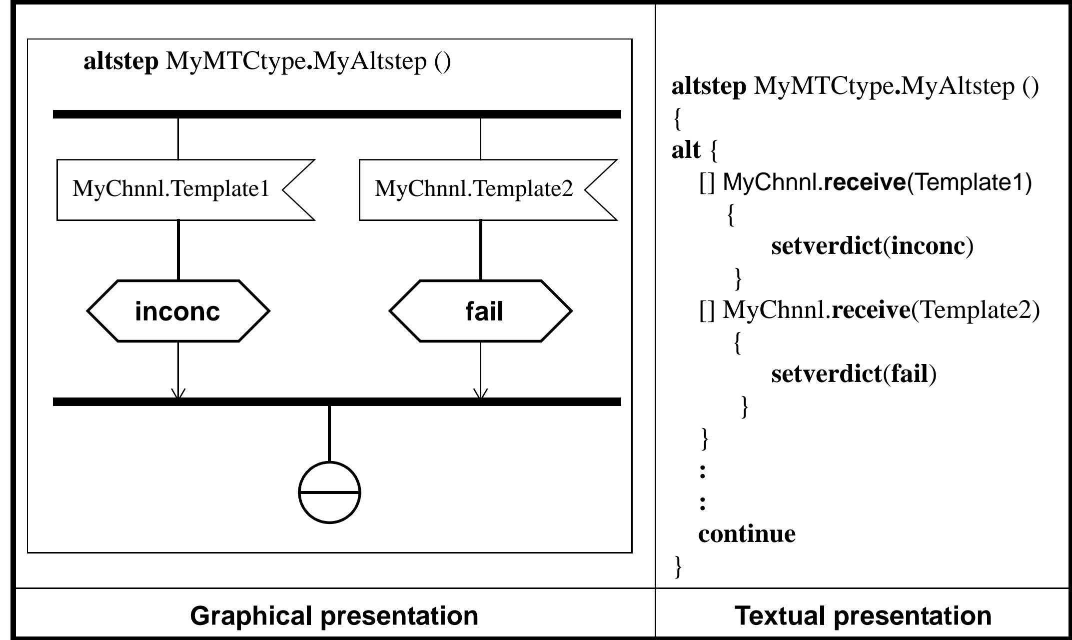
Attributes associated to the altstep shall be specified within a <text symbol> in the ATDL/gr altstep diagram. The principle shape of an ATDL altstep diagram and the corresponding ATDL textual language are sketched in Figure 12.

Concrete graphical grammar

208 <altstep diagram> ::= <frame symbol>
contains ([“overload”] AltstepHeading <altstep body area>)

209 <altstep body area> ::= <fgr alt area>

Figure 12. Principle shape of an ATDL altstep diagram and corresponding textual language



An altstep is used to specify default behaviour or to specify default signal handler or to structure the alternatives of an **alt** statement. An altstep may contain statements, communication and timer operations and invocation of function or altsteps.

16.4.3. Invocation of altsteps

The invocation of an altstep is related to an **alt** statement or a default signal handler. The invocation may be done either implicitly by the default mechanism (see §19.7) or explicitly by a direct call within an **alt** statement (see §19.1.8). The invocation of an altstep causes no new snapshot and the evaluation of the top alternatives of an altstep is done by using the actual snapshot of the **alt** statement from which the altstep was called.

An explicit call of an altstep within an **alt** statement looks like a function call as alternative.

An **altstep** can also be called as stand-alone statement in an ATDL behaviour description. In this case the call of the **altstep** can be interpreted as shorthand for an **alt** statement with only one alternative describing the explicit call of the **altstep**.

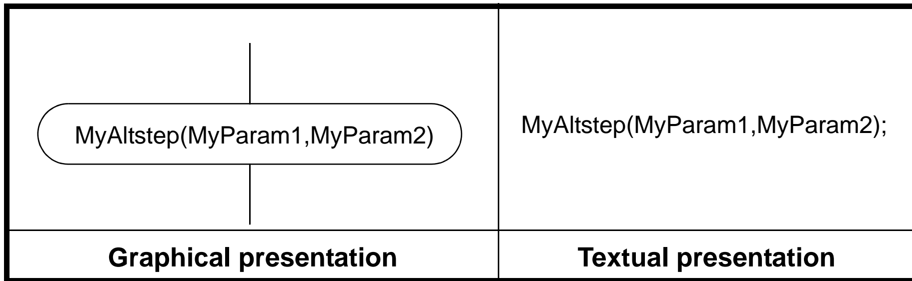
Concrete textual grammar

206 AltstepInstance ::= AltstepRef [ActualParList]

Concrete graphical grammar

210 <altstep instance area> ::= <reference symbol> **contains** AltstepInstance

Figure 13. Altstep invocation



The invocation of altsteps is represented by use of the reference symbol (Figure 13). The syntax of the altstep invocation is placed within that symbol. The symbol may contain the invocation of an altstep with optional parameters. It shall be used within alternative behaviour only, where the altstep invocation shall be one of the branches of the alternative statements.

16.5. Method templates

This section describes what a method template is and discusses how to define and use a method template. A strongly typed language can sometimes seem an obstacle to implementing what are otherwise straightforward functions. A method template provides an algorithm that is used to automatically generate particular instances of a method varying by type. The programmer *parameterizes* all or a subset of the types in the interface (the parameter and return types) of a method whose body otherwise remains invariant. A method is a candidate to be made a template when its implementation remains invariant over a set of instances.

16.5.1. Method template definition

The keyword template begins a definition of a method template. The keyword is followed by a comma-separated list of template parameters enclosed in parentheses. This list is the *template parameter list*. It cannot be empty. A method template parameter can be a *template generic type parameter* representing a generic type or a *template non-generic type parameter* representing an ordinary built-in or user-defined type.

A template non-generic type parameter consists of an ordinary parameter declaration. A template non-generic type parameter indicates that the parameter name represents a potential value. This value represents a constant in the method template definition.

When the method template is instantiated, an actual built-in or user-defined type is substituted for the template generic type parameter. Generic type will be substituted with various built-in and user-defined types and non-generic type will be substituted with various constant values determined by the actual uses of method template. This process of type and value substitution is referred to as *method template instantiation*.

The name of a method template parameter can be used after it has been declared as a template parameter and until the end of the template declaration or definition. A template generic type parameter serves as a type specifier for the remainder of the template definition; it can be used in exactly the same way as a built-in or user-defined type specifier, such as for variable declarations and casts. A template non-generic type parameter serves as a constant value for the remainder of the template definition; it can be used when constant values are required. The name of a template generic type parameter can be used to specify the return type of the method template.

16.5.2. Method template explicit incarnation

It is not always possible to write a single function template that is best suited for every possible type with which the template may be instantiated. In some cases, we may be able to take advantage of some specific knowledge about a type to write a more efficient function than the one that is instantiated from the template. At other times, the general template definition is simply wrong for a type. For example, suppose we have this definition of the function template max():

```
virtual class MyVirtualClass { //...
        // Only virtual classes may have abstract type declarations
        type MyType ::= variant;
        template function max (t1 MyType, t2 MyType) return MyType;
        //...
}

// generic function template definition

template function MyVirtualClass.max (t1 MyType, t2 MyType) return MyType {

        return (t1>t2 ? t1:t2)

}
```

If the function template is instantiated with a template argument of type charstring, the generic template definition does not give the right semantics if we intend each argument to be interpreted as an ATDL character string. To get the right semantics, we must provide an incarnated definition for the function template instantiation.

It is possible to declare a method template explicit incarnation without defining it within a class declaration. When we are declaring or defining a method template explicit incarnation, we must not omit the template keyword from the incarnation declaration. Similarly, the function parameter list cannot be omitted from the incarnation declaration. For example,

```
class MyCstringClass extends MyVirtualClass { //...
        type MyType charstring;         // type incarnation
        template function max(s1 MyType, s2 MyType) return MyType;
        // ...
}
```

An *explicit incarnation definition* is a definition in which the function keyword is followed by the definition of the function template incarnation. This definition indicates the method template name, the method template arguments for which the method template is incarnated, the function parameter list, and the function body. In the following example, an explicit incarnation is defined for max(charstring, charstring):

```
template function MyCstringClass.max(s1 MyType, s2 MyType) return MyType {

        return ( strcmp (s1, s2) >0 ? s1 : s2);

}
```

An explicit incarnation for a method template can be defined only after the general method template has been declared in the ancestor virtual class. The declaration of a method template explicit incarnation must be seen before it is used in the source file.

16.5.3. Name resolution in method templates

Inside a method template definition, some constructs have a meaning that differs from one instantiation of the method template to another, whereas other constructs have the same meaning for all instantiations of the method template. This depends on whether or not the construct involves a method template parameter. For example:

```
virtual class MyVClass { //...
        // Only virtual classes may have abstract type declarations
        type MyType ::= variant;
        template function min(MyArray sequence of MyType, size Byte) return MyType;
        // ...
}
template function MyVClass.min(MyArray sequence of MyType, size Byte) return MyType
{
    var min_val MyType := MyArray[0];
    for ( var i Smallint :=1; i < size; i := i+1)
    { if [ MyArray[i] < min_val ]
              { min_val := MyArray[i] }
    }
    print ("Mimimum value found: " );
    print( min_val );
    return min_val;
}
```

In min(), the types of MyArray and of min_val depend on the actual type with which MyType will be replaced when the template is instantiated, whereas the type of size, for example, remains Smallint regardless of the type of the template parameter. The types of MyArray and of min_val vary from one instantiation of the function template to another. Because of this, we say that the types of these variables *depend on a template parameter*, whereas the type of size does not depend on a function template parameter.

Because the type of min_val is unknown, it is also unknown which operation is used when min_val appears in an expression. For example, which print() function should be called by the function call print(min_val)? Should it be the print() function used for Smallint types? Or should it be the function for float types? Is the call in error because there is no function print() that can be called with an argument of min_val's type? It is impossible to answer these questions until the template is instantiated and until we know what the type of min_val is. Because of this, we also say that the call print (min_val) depends on a function template parameter.

There are no such questions with the constructs within min() that do not depend on a template parameter. For example, it is always known which function should be called for the call print ("Minimum value found: "). It is the function used to print character strings. The print() function called does not vary from one instantiation of the function template to another. Thus, we say that this call does not depend on a function template parameter.

Because the call print ("Minimum value found: ") is not a call that depends on a function template parameter, the function print() for character strings must be declared before it can be used in the template definition. On the other hand, the declaration for the print() function used by the call print (min_val) is not yet needed because we do not yet know which print() function to look for. It is not possible to know which print() function is called by print(min_val) until the type of min_val is known. For example:

```
const arrayi sequence [4] of Smallint = { 12, 8, 73, 45 };

MyResult := min( arrayi, 4 );
```

The function invocation expression calls the function template instantiation. In this instantiation of min(), MyType is replaced by Smallint and the type of the variable min_val is Smallint. The function call print (min_val) therefore calls a function that can be invoked with an argument of type Smallint. It is when min () is instantiated that we know that the second call to the print() function has an argument of type int. It is at this time that a function print() that can be called with an argument of type Smallint needs to be visible.

17. Overview of program statements and operations

There are many kinds of statements in the ATDL language. Most correspond to statements in the TTCN-2 and TTCN-3 languages, but some are unique to ATDL.

Table 21: Overview of ATDL statements and symbols

Statement	Associated keyword or symbol	Associated graphical symbol
Basic program statements		
Expressions	**(…)**	yes
Function invocation expression		<procedure call symbol>
Assignments	**:=**	<task symbol>
Logging	**write**	<task symbol>
Label	**label**	<connector symbol>
Break	**break**	<break symbol>
Continue	**continue**	<continue symbol>
If statement without else	**if […]**	<inline expression symbol>
If statement with else branch	**if […] {…} else {…}**	<decision symbol>
For loop	**for (…) {…}**	<inline expression symbol>
While loop	**while […]**	<inline expression symbol>
Do while loop	**do {…} while […]**	<inline expression symbol>
Assignment list	**:=**	<save symbol>
Behavioral program statements		
Start of an alternative behavior	**alt {…}**	<alt symbol>
End of an alternative behavior	**alt {…}**	<alt outlet symbo>
Start of a try statement	**try { … }**	<try symbol>
End of a try statement	**try { … }**	<connector symbol>
Raise exception (to a procedure)	**raise**	<exception out symbol>
Catch exception (from callee)	**catch**	<exception in symbol>
Returning control	**return**	<return symbol>
Statements for default handling		
Activate a default	**activate**	<default symbol>
Deactivate a default	**deactivate**	<default symbol>
Thread operations		
Create parallel test thread	**create**	<create request symbol>
Get MTC address	**mtc**	<task symbol>
Get test system interface address	**system**	<task symbol>
Get own address	**self**	<task symbol>
Start execution of test thread	**start**	<procedure call symbol>
Stop execution of test thread	**stop**	<procedure call symbol>
Check termination of a PTC	**running**	<condition symbol>
Wait for termination of a PTC	**done**	<internal input symbol>

Table 21: Overview of ATDL statements and symbols

Communication operations		
Send message	**send**	<message out symbol>
Invoke procedure call	**call**	<procedure call symbol>
Receive message	**receive**	<message in symbol>
Trigger on message	**trigger**	<message in symbol>
Synchronize a procedure call	**synchronize**	<procedure in symbol>
Server-side channel controlling operations		
Give access to channel	**start**	<condition symbol>
Clear channel	**clear**	<condition symbol>
Stop access at channel	**stop**	<condition symbol>
Client-side channel controlling operations		
Bind channel to server component	**bind**	<task symbol>
Release a channel	**release**	<task symbol>
Timer class methods		
Start timer	**start**	<internal output symbol>
Stop timer	**stop**	<internal output symbol>
Read elapsed time	**read**	<task symbol>
Check if timer running	**running**	<condition symbol>
Timeout event	**timeout**	<internal input symbol>
Verdict class methods		
Set local verdict	**setverdict**	<condition symbol>
Get local verdict	**getverdict**	<task symbol>
External actions		
Stimulate an (SUT) action	**sutaction**	<task symbol>

The fundamental program elements of the control part of ATDL modules and functions are basic program statements such as expressions, assignments, loop constructs etc., behavioral statements such as sequential behavior, alternative behavior, interleaving, defaults etc., and operations such as **send**, **receive**, **create,** etc.

Statements can be either single statements (which do not include other program statements) or compound statements (which may include other statements).

Statements shall be executed in the order of their appearance, i.e. sequentially.

The individual statements in the sequence shall be separated by the delimiter ";".

17.1. Statement block

A statement block is a list of statements. The statements are executed sequentially.

Statement blocks are a mechanism to group statements. Statement blocks may be used in different scope units i.e., module control, functions and test behaviors. The kind of statements that may be used in a block will depend on the scope unit in which the block is used. For example, a statement block appearing in a function shall only use those program statements which may be used in functions.

A statement block is syntactically equivalent to a single statement; thus, wherever a statement is allowed in a function a statement block may appear. This implies that blocks may be nested. Declarations, if any, shall be made at the beginning of the statement block. These declarations are only visible inside the statement block and to nested sub-blocks.

The statements in the statement block shall be executed in the order of their appearance. The specification of an empty statement block i.e., { }, is allowed. An empty statement block implies that no actions are taken.

A statement block is executed by executing each of the local declaration statements and other statements in order from first to last (left to right). If all of these block statements complete normally, then the block completes normally. If any of these block statements complete abruptly for any reason, then the block completes abruptly for the same reason.

Concrete textual grammar

394 StatementBlock ::= "{" BlockStatement [TerminatorStatement] | TerminatorStatement "}"

395 BlockStatement ::= { ActionStatement [SemiColon]}+

17.1.1. Statement diagrams

A statement graph models the possible life histories of an instance of a component (Figure 54).

The operational semantics represents ATDL behavior in form of flow graphs. The construction algorithm for the flow graphs representing behavior is described. It is based on templates for flow graphs and flow graph segments that have to be used for the construction of concrete flow graphs for module control, test cases, and functions defined in an ATDL module.

A statement graph is like a traditional flow graph. A flow graph is a directed graph that consists of labeled nodes and labeled edges. Walking through a flow graph describes the flow of control during the execution of a represented behavior description.

Concrete graphical grammar

398 <statement block area> ::= [<action statement area> **is_followed_by**]
{ <return area> | <stop symbol> }

17.1.2. Statements

There are many kinds of statements in the ATDL language. Most correspond to statements in the TTCN-3 language, but some are unique to ATDL.

Concrete textual grammar

396 ActionStatement ::= ControlStatement | ConfigurationStatement |
CommunicationStatement | SetLocalVerdict

397 TerminatorStatement ::= ReturnStatement | "stop" | RaiseStatement

Concrete graphical grammar

399 <action statement area> ::= <control statement area> | <configuration statement area> |
<communication statement area> | <set verdict area>

17.1.3. Unreachable Statements

It is a compile-time error if a statement cannot be executed because it is unreachable. Every ATDL compiler must carry out the conservative flow analysis specified here to make sure all statements are reachable.

17.2. Kinds of conversion

Specific type conversions in ATDL are divided into five categories.

17.2.1. Identity conversions

A conversion from a type to that same type is permitted for any type. This may seem trivial, but it has two practical consequences. First, it is always permitted for an expression to have the desired

type to begin with, thus allowing the simply stated rule that every expression is subject to conversion, if only a trivial identity conversion. Second, it implies that it is permitted for a program to include redundant cast operators for the sake of clarity.

The only permitted conversion that involves the type boolean is the identity conversion from boolean to boolean.

17.2.2. Widening primitive conversions

The following conversions on primitive types (or derived primitive types) are called the *widening primitive conversions*. For example,

1. Byte or Shortint to Word, Smallint, Longword, Longint, Longlongword, Longlongint, float, IEEE754extfloat, IEEE754double or IEEE754extdouble.
2. Word or Smallint to Longword, Longint, Longlongword, Longlongint, float, IEEE754extfloat, IEEE754double or IEEE754extdouble.
3. char to Word, Smallint, Longword, Longint, Longlongword, Longlongint, float, IEEE754extfloat, IEEE754double or IEEE754extdouble.
4. wide char to Longword, Longint, Longlongword, Longlongint, float, IEEE754extfloat, IEEE754double or IEEE754extdouble.
5. Longword or Longint to Longlongword, Longlongint, float, IEEE754extfloat, IEEE754double or IEEE754extdouble.
6. Longlongword or Longlongint to float, IEEE754extfloat, IEEE754double or IEEE754extdouble.

Widening primitive conversions do not lose information about the overall magnitude of a numeric value. Indeed, conversions widening from an integral type to another integral type and from **float** to **real** do not lose any information at all; the numeric value is preserved exactly. Conversion of an **integer** or a **cardinal** value to **float** may result in *loss of precision* — that is, the result may lose some of the least significant bits of the value. In this case, the resulting floating-point value will be a correctly rounded version of the integer value, using IEEE 754 round-to-nearest mode.

A widening conversion of a signed integer value to a wider integer type simply sign-extends the two's-complement representation of the integer value to fill the wider format.

Despite the fact that loss of precision may occur, widening conversions among primitive types(or derived primitive types) never result in a run-time exception

17.2.3. Narrowing primitive conversions

Narrowing conversions may lose information about the overall magnitude of a numeric value and may also lose precision.

A narrowing conversion from Double to float behaves in accordance with IEEE 754. The result is correctly rounded using IEEE 754 round-to-nearest mode. A value too small to be represented as a float is converted to positive or negative zero; a value too large to be represented as a float is converted to a (positive or negative) infinity. A real NaN is always converted to a float NaN.

17.2.4. Widening reference conversions

The following conversions are called the *widening reference conversions*:

a) From any class type *S* to any class type *T*, provided that *S* is a descendant of *T*.

b) From any class type *S* to any interface type *K*, provided that *S* implements *K*.

c) From the null type to any class type, interface type, or array type.

d) An interface type is assignment-compatible with any ancestor interface type.

Such conversions never require a special action at run time and therefore never raise an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time.

17.2.5. Narrowing reference conversions

The *narrowing reference conversions* require a test at run time to find out whether the actual reference value is a legitimate value of the new type.

17.2.6. Charstring conversions

There is a string conversion to type charstring from every other type, including the null type.

Charstring conversion applies only to the operands of the binary charstring operator (+) when one of the arguments is a charstring. In this single special case, the other argument to the + is converted to a charstring, and a new charstring which is the concatenation of the two charstrings is the result of the +. Charstring conversion is specified in detail within the description of the string concatenation + operator (§9.2).

17.2.7. Forbidden Conversions

a) There is no permitted conversion from any reference type to any primitive type.

b) Except for the string conversions, there is no permitted conversion from any primitive type (or derived primitive types) to any reference type.

c) There is no permitted conversion from the null type to any primitive type.

d) There is no permitted conversion to the type boolean other than the identity conversion.

e) There is no permitted conversion from the type boolean other than the identity conversion and string conversion.

f) There is no permitted conversion from class type *S* to interface type *K* if *S* is final and does not implement *K*.

17.3. Assignment conversion

Assignment conversion occurs when the value of an expression is assigned to a variable: the type of the expression must be converted to the type of the variable. Assignment contexts allow the use of an identity conversion, a widening primitive conversion, or a widening reference conversion. If the type of the expression cannot be converted to the type of the variable by a conversion permitted in an assignment context, then a compile-time error occurs.

If the type of an expression can be converted to the type a variable by assignment conversion, we say the expression (or its value) is *assignable to* the variable or, equivalently, that the type of the expression is *assignment compatible with* the type of the variable.

A value of primitive type must not be assigned to a variable of reference type; an attempt to do so will result in a compile-time error. A value of type boolean can be assigned only to a variable of type boolean.

A value of reference type must not be assigned to a variable of primitive type; an attempt to do so will result in a compile-time error.

A value of the null type (the null reference is the only such value) may be assigned to any reference type, resulting in a null reference of that type.

17.4. Method invocation conversion

Method invocation conversion is applied to each argument value in a method or constructor invocation: the type of the actual parameter expression must be converted to the type of the corresponding parameter. Method invocation contexts allow the use of an identity conversion (§17.2.1), a widening primitive conversion(§17.2.2), or a widening reference conversion (17.2.4).

17.5. Casting conversion

Casting conversion is applied to the operand of a cast operator: the type of the operand expression must be converted to the type explicitly named by the cast operator. Casting contexts allow the use of an identity conversion (§17.2.1), a widening primitive conversion (§17.2.2), a narrowing primitive conversion (clause 17.2.3), a widening reference conversion (§17.2.4), or a narrowing reference conversion (§17.2.5). Thus casting conversions are more inclusive than assignment or method invocation conversions: a cast can do any permitted conversion other than a charstring conversion.

A value of a primitive type (or a derived primitive type) can be cast to another primitive type (or a derived primitive type) by identity conversion, if the types are the same, or by a widening primitive conversion or a narrowing primitive conversion.

A value of a primitive type (or a derived primitive type) cannot be cast to a reference type by casting conversion, nor can a value of a reference type be cast to a primitive type.

17.6. Type compatibility and identity

Generally, ATDL requires type compatibility of values at assignments, instantiations and comparison. To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include *type identity*, *type compatibility*, and *assignment-compatibility.*

17.6.1. Type identity

Type identity is almost straightforward. When one type identifier is declared using another type identifier, they denote the same type.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

type TS1 ::= **set of char**;
type TS2 ::= **set of char**;

create two distinct types, *TS1* and *TS2*. Similarly, the variable declarations

var S1 **charstring**[10];
var S2 **charstring**[10];

create two variables of distinct types. To create variables of the same type, use

var S1:S2 **charstring**[10];

or

type MyString ::= **charstring**[10];

var S1 MyString, S2 MyString;

17.6.2. Type compatibility

Every type is compatible with itself.

For the purpose of this clause value "b" is called the actual value to be assigned, passed as parameter etc. , type "B" is called the type of value "b" and type "A" is called the type definition of the value, which is to obtain the actual value of value "b".

17.6.2.1. Type compatibility of non-structured types

For non-structured variables, constants, templates etc. the value "b" is compatible to type "A"if type "B" resolves to the same root type as type "A" (i.e. **integer**) and it does not violate subtyping (e.g. ranges, length restrictions) of type "A".

17.6.2.2. Type compatibility of structured types

In the case of structured types (except the **enumerated** type) a value "b" of type "B" is compatible with type "A", if the effective value structures of type "B" and type "A" are compatible, in which case assignments, instantiations and comparisons are allowed.

17.6.2.2.1. Type compatibility of enumerated types

Enumerated types are never compatible with other basic or structured types (i.e. for enumerated types strong typing is required).

17.6.2.2.2. Type compatibility of sequence types

For **sequence** types the effective value structures are compatible if the number, type, and optionality of the fields at the textual order of definition are identical and values of each existing field of the value "b" is compatible with the type of its corresponding field in type "A". Values of each field in the value "b" are assigned to the corresponding field in the value of type "A".

17.6.2.2.3. Type compatibility of sequence of types

For **sequence of** types the effective value structures are compatible if their component types are compatible and value "b" of type "B" does not violate any length subtyping of the **sequence of** type or dimension of the sequence of type "A". Values of elements of the value "b" shall be assigned sequentially to the instance of type "A", including undefined elements.

17.6.2.2.4. Compatibility between slices

The rules defined in this clause for structured types compatibility are also valid for the sub-structure of such types i.e. equivalence between slices.

18. Basic program statements

Basic program statements are expressions, assignments, operations, loop constructs etc. All basic program statements can be used in the control part of a module and in ATDL functions.

Table 22: Overview of ATDL basic program statements

Basic program statements		
Statement	**Associated keyword or symbol**	**Associated graphical symbol**
Expressions	**(…)**	<task symbol>
Assignments	**:=**	<task symbol>
Logging	**write**	<task symbol>
Label	**label**	<inline expression symbol>
Break	**break**	<break symbol>
Continue	**continue**	<continue symbol>
If statement without else	**if […]**	<inline expression symbol>
If statement with else branch	**if […] {…} else {…}**	<decision symbol>
For loop	**for (…) {…}**	<inline expression symbol>
While loop	**while […]**	<inline expression symbol>
Do while loop	**do {…} while […]**	<inline expression symbol>

18.1. Local variable declaration statements

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a **for** statement. In this case it is executed in the same manner as if it were part of a local variable declaration statement.

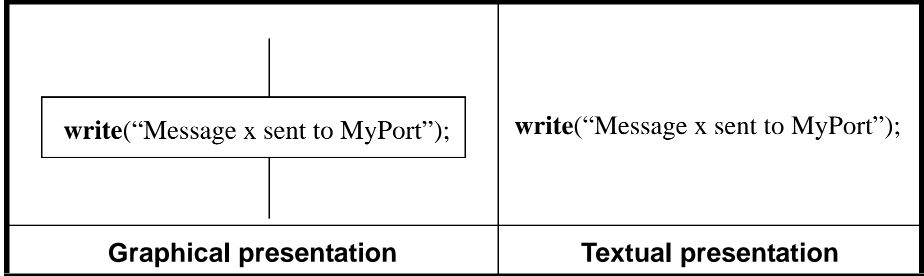
18.2. The task statements

Apart from assignments [17], task boxes may contain ATDL statements like the bind, release, SUT operation, interface typecast statement or write operation.

Concrete textual grammar

402 TaskStatement ::= Assignment | ConstDef | VarInstance | WriteStatement | BindStatement
| ReleaseStatement | SUTStatement

Figure 14. Write Statement



Concrete graphical grammar

405 <task area> ::= <task symbol> **contains** ({TaskStatement [SemiColon]}+)

18.2.1. The Write statement

The **write** statement provides the means to write a character string to some logging device associated with test control or the test component in which the statement is used.

ATDL concrete textual grammar

403 WriteStatement ::= “write” “(“ [Cstring] “)”

The **write** statement shall be represented within a <task symbol> (Figure 14).

18.2.2. External actions

In some testing situations some electrical interface(s) to the SUT may be missing or unknown a priori (e.g. management interface) but it may be necessary that the SUT is stimulated to carry out certain actions (e.g. send a message to the test system). Also certain actions may be required from the test executing personnel (e.g. to change the environmental conditions of testing like the temperature, voltage of the power feeding etc.).

The required action may be defined as a string.

EXAMPLE 2:

sutaction("Send MyTemplate on lower PCO"); // Informal description of the SUT action

or as a reference to a template which specifies the structure of the message to be sent by the SUT.

In both cases there is no specification of what is done to or by the SUT to trigger this action, only an informal specification of the required action itself.

SUT actions can be specified in test cases, functions, altsteps and module control.

ATDL concrete textual grammar

404 SUTStatement ::= “sutaction” “(“ (FreeText | TemplateRefWithPara) “)”

Figure 15. External actions

sutaction("Send MyTemplate on lower PCO ");	**sutaction**("Send MyTemplate on lower PCO ");
Graphical presentation	**Textual presentation**

External actions are represented within <task symbol>s (Figure 15). The syntax of the external action is placed within that symbol.

18.2.3. Expression statements

Certain kinds of expressions may be used as statements by following them with semicolons.

18.3. The If-else statement

The **if-else** statement, also known as the conditional statement, is used to denote branching in the control flow due to **boolean** expressions. Schematically the conditional looks as follows:

if [expression$_1$]

statementblock$_1$

[**else**]

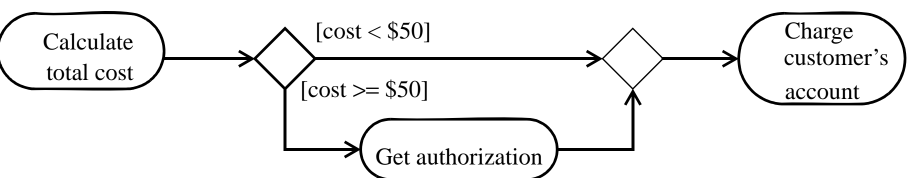
statementblock$_2$

Where `statementblockx` refers to a block of statements.

18.3.1. The if statement with else branch

In statement graphs, all of the paths leaving a branch must eventually merge in a subsequent join in the model. Furthermore, multiple layers of branches and merges must be well nested. A statement diagram expresses a **decision** when guard conditions are used to indicate different possible statement blocks that depend on guard conditions of the owning object. ATDL provides a shorthand notation for showing decisions and merging their separate paths back together.

Figure 16. Decision and merge



Decision branch. A decision branch is a set of transitions leaving a single node such that exactly one guard condition on one of the transitions must always be satisfied. The guard conditions essentially represent a branch of control. A decision branch is notated as a diamond with one input arrow and two or more output arrows. Each output is labeled with a guard condition.

Decision merge. A decision merge is a place at which two or more alternate paths of control come together. It is the inverse of a decision branch. A diamond is the symbol for either a decision branch or decision merge.

The meaning of a branching pseudo-events is as follows: if only a guard condition is specified: the guard condition is evaluated and execution continues with subsequent behavior, if the guard condition evaluates to **true**; if it evaluates to **false** the next alternative is attempted. If no alternative exists, then this is a test case error.

Concrete textual grammar

447 DecisionConstruct ::= “if” GuardCondition StatementBlock {ElseIfClause}* [ElseClause]

449 ElseIfClause ::= “else” “if” GuardCondition StatementBlock

Concrete graphical grammar

453 <decision area> ::= <decision symbol> **is_followed_by**
(<decision if part> {<decision if part>}* [<decision else part>]) **set**

454 <decision if part> ::= <flow line symbol> **is_associated_with** GuardCondition
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

455 <decision outlet symbol> ::= <decision symbol>

456 <decision else part> ::= <flow line symbol>
is_associated_with (“[“ **else** “]”)
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

The icon provided for a decision is the traditional diamond shape, with one incoming flow line arrow and with two or more outgoing flow line arrows, each labeled by a distinct guard condition with no event trigger. The reserved word **else** can be used as a guard condition.

The same icon can be used to merge decision branches back together, in which case it is called a merge. In the case of a merge, there are two or more incoming flow line symbol and one outgoing flow line symbol. No guard conditions are necessary.

18.3.2. Control icons

Control icons are used to define decision branch and alternative fork behavior. A pseudo-event is an abstraction that encompasses different types of transient vertices in the statement diagram. Pseudo-events include decision, merge, alternative fork and join. They are used, typically, to connect multiple statement blocks into more complex statement blocks paths. Pseudo-events are intended for use in statement diagrams, but they can also be used in ETSCs, if desired.

18.4. The Choice statement

The choice statement transfers control to one of several statements depending on the value of an expression. The choice statement provides an alternative to complex nested **if** conditionals.

The body of a choice statement must be a block. Any statement immediately contained by the block may be labeled with one or more ChoiceSelector or else clauses. Each value represented by a ChoiceSelector must be unique in the choice statement; subranges and lists cannot overlap. A choice statement can have a final else clause.

When a choice statement is executed, at most one of its constituent *statement blocks* is executed. Whichever ChoiceSelector has a value equal to that of *SingleExpression* determines the *statement block* to be used. If none of the guard conditions has the same value as *SingleExpression*, then the statement in the else clause (if there is one) is executed.

The choice statement

```
choice I of {
        [1..5]  { Caption := "Low"; }
        [6..9]  { Caption := "High"; }
        [0, 10..99] { Caption := "Out of range"; }
        [ else ]  { Caption := ""; }
}
```

is equivalent to the nested conditional

```
if [I in (1..5)] { Caption := "Low" }

else if [I in (6..10)] { Caption := "High" }

else if [(I == 0) | (I in (10..99)] { Caption := "Out of range" }

else { Caption := "" }
```

Concrete textual grammar

450 ChoiceConstruct ::= "choice" SingleExpression "of" "{" ChoiceList [ElseClause] "}"

451 ChoiceList ::= {ChoiceSelector StatementBlock}*

452 ChoiceSelector ::= "[" SingleConstExpression "]"

Concrete graphical grammar

454 <choice area> ::= <decision symbol> **contains** SingleExpression **is_followed_by**
({<choice selector part>}* [<decision else part>]) **set**

454 <choice selector part> ::= <flow line symbol> **is_associated_with** ChoiceSelector
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

18.5. The in-line expressions

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly. Structured inline statements include **if** statement without **else** branch, **while**, **for**, **do-while**, and **label**.

The **for**, **while**, **do while** statement of allows the specification of cyclic behavior. These statements are represented in ATDL/gr with an inline loop expression having in the upper left side box the associated keyword. An inline loop expression has exactly one statement block area.

Concrete textual grammar

427 LoopConstruct ::= ForStatement | WhileStatement | DoWhileStatement | LabeledStatement

Concrete graphical grammar

438 <fgr inline expression area> ::= <fgr for area> | <fgr while area>
| <fgr do-while area> | <fgr labeled area> | <fgr opt area>

The <inline expression symbol> may contain only one operand.

18.5.1. The labeled statement

The **label** statement allows the specification of labels in test cases, functions, altstep and the control part of a module. It can be used before or after an ATDL statement but shall not as first statement of an alternative in an **alt** statement.

Unlike TTCN-3, ATDL has no goto statement; identifier statement labels are used with **break** or **continue** statements appearing anywhere within the labeled statement.

A statement labeled by an identifier must not appear anywhere within another statement labeled by the same identifier, or a compile-time error will occur. Two statements can be labeled with the same identifier only if neither statement contains the other.

Concrete textual grammar

434 LabeledStatement ::= “label” LabelIdentifier StatementBlock

435 LabelIdentifier ::= Identifier

Concrete graphical grammar

442 <labeled area> ::= <inline expression symbol> **contains**
(**label** LabelIdentifier <statement block area>)

The labeled statement shall be represented by an <inline expression symbol> labelled with the *LabelIdentifier*. A <flow line symbol> is drawn from the previous node to the <inline expression symbol>, and another <flow line symbol> is drawn from the <inline expression symbol> to the next node. Figure 21 illustrates a labeled statement in which the continue statement causes the labeled statement to be repeated.

18.5.1.1. The page continuation

When any ATDL statement graph is too long to fit on a single page the following mechanism shall be used:

a) A **label** statement is used with a **continue** statement, the **continue** statement is used to redirect the interpretation to the next page;

b) There must be one **label** statement with the same name as the **continue** statement indicates the entry point on the next page.

18.5.2. The if statement without else branch

The **if** statement without **else** branch shall be represented by an <inline expression symbol> labelled with the **if** keyword and a Boolean expression. A <flow line symbol> is drawn from the previous node to the <inline expression symbol>, and another <flow line symbol> is drawn from the <inline expression symbol> to the next node.

Figure 17. If Statement without else branch

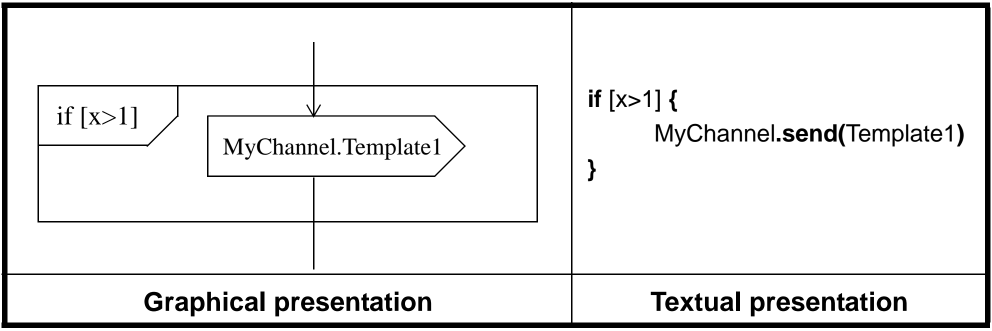


Figure 17 illustrates an **if** statement with a single operand, which is executed when the Boolean expression `x>1` evaluates to true.

Concrete graphical grammar

459 <fgr opt area> ::= <inline expression symbol> **contains**
(**if** GuardCondition <statement block area>)

18.5.3. The For statement

The **for** statement defines a counter loop. The value of the index variable is increased, decreased or manipulated in such a manner that after a certain number of execution loops a termination criteria is reached.

The **for** statement contains two assignments and a **boolean** expression. The first assignment is necessary to initialize the index (or counter) variable of the loop. The **boolean** expression terminates the loop and the second assignment is used to manipulate the index variable.

Concrete textual grammar

428 ForStatement ::= “for” LoopCondition StatementBlock

429 LoopCondition ::= “(“ ForInit SemiColon BooleanExpression SemiColon ForUpdate “)”

430 ForInit ::= SingleVarInstance | Assignment

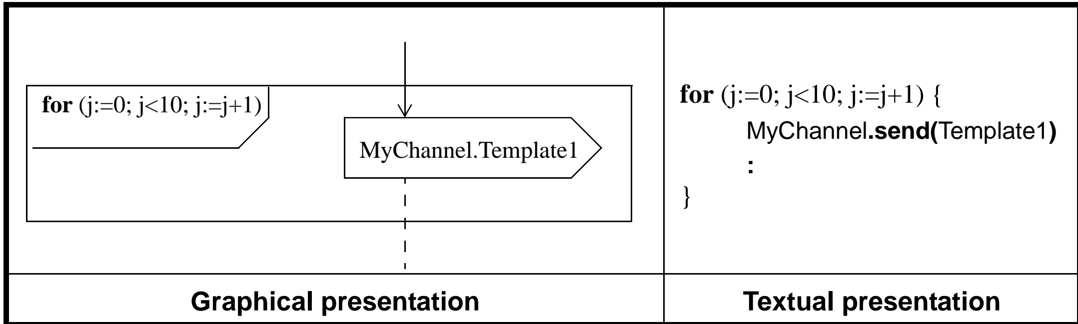
431 ForUpdate ::= Assignment

Concrete graphical grammar

439 <fgr for area> ::= <inline expression symbol> **contains**
(**for** LoopCondition <statement block area>)

The **for** statement shall be represented by an <inline expression symbol> labelled with a **for** loop definition. The **for** body shall be represented as the operand of the for <inline expression symbol>. A <flow line symbol> is drawn from the previous node to the <inline expression symbol>, and another <flow line symbol> is drawn from the <inline expression symbol> to the next node. Figure 18 represents a simple **for** statement.

Figure 18. For Statement



18.5.3.1. Initialization of for statement

If the *ForInit* code is a local variable declaration, it is executed as if it were a local variable declaration statement appearing in a statement block. In this case, the scope of a declared local variable is its own initializer and any further declarators in the *ForInit* part, plus the *BooleanExpression*, *ForUpdate*, and contained *Statement Block* of the for statement.

18.5.3.2. Iteration of for statement

The termination criterion of the loop shall be expressed by the **boolean** expression. It is checked at the beginning of each new loop iteration. If it evaluates to **true**, the execution continues with the statement which immediately follows the **for** loop.

If execution of the *Statement Block* completes abruptly because of a **break** with no label, no further action is taken and the **for** statement completes normally.

If execution of the *Statement Block* completes abruptly because of a **continue** with no label, then the entire **for** statement is executed again.

If execution of the *Statement Block* completes abruptly because of a **continue** with label *L*, then there is a choice:

- If the **for** statement has label *L*, then the entire **for** statement is executed again.
- If the **for** statement does not have label *L*, the **for** statement completes abruptly because of a **continue** with label *L*.

18.5.4. The While statement

A **while** loop is executed as long as the loop condition holds. The loop condition shall be checked at the beginning of each new loop iteration. If the loop condition does not hold, then the loop is exited and execution shall continue with the statement, which immediately follows the **while** loop. For example,

while [j<10] {... };

If the value of the *Guard Condition* is **false** the first time it is evaluated, then the *Statement Block* is not executed.

If execution of the *Statement Block* completes abruptly because of a **break** with no label, no further action is taken and the **while** statement completes normally.

If execution of the *Statement Block* completes abruptly because of a **continue** with no label, then the entire **while** statement is executed again.

If execution of the *Statement Block* completes abruptly because of a **continue** with label *L*, then there is a choice:

- If the **while** statement has label *L*, then the entire **while** statement is executed again.

- If the **while** statement does not have label *L*, the **while** statement completes abruptly because of a **continue** with label *L*.

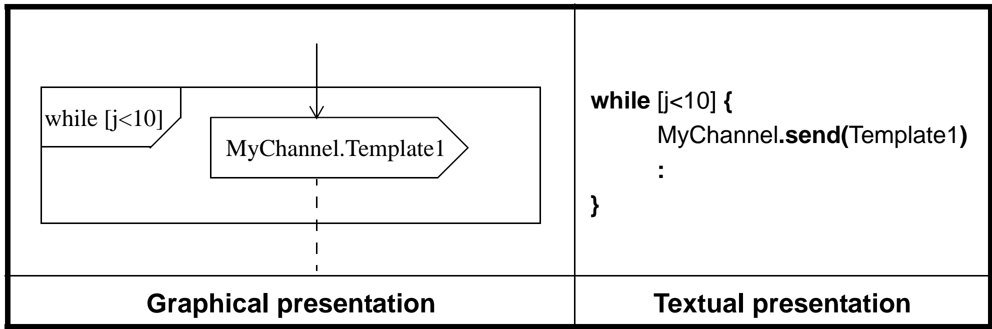
Concrete textual grammar

432 WhileStatement ::= “while” GuardCondition StatementBlock

Concrete graphical grammar

440 <fgr while area> ::= <inline expression symbol> **contains**
(**while** GuardCondition <statement block area>)

Figure 19. While Statement



The **while** symbol shall be represented by an <inline expression symbol> labelled with a **while** definition. The **while** body shall be represented as the operand of the while <inline expression symbol>. A <flow line symbol> is drawn from the previous node to the <inline expression symbol>, and another <flow line symbol> is drawn from the <inline expression symbol> to the next node. Figure 19 represents an example of a **while** statement.

18.5.5. The Do-while statement

The **do-while** loop is identical to a **while** loop with the exception that the loop condition shall be checked at the *end* of each loop iteration. This means when using a **do-while** loop the behaviour is executed at least once before the loop condition is evaluated for the first time.

EXAMPLE 3: **do** { … } **while** [j<10];

If execution of the *Statement Block* completes abruptly because of a **break** with no label, then no further action is taken and the do statement completes normally.

If execution of the *Statement Block* completes abruptly because of a **continue** with no label, then the *Guard Condition* is evaluated. Then there is a choice based on the resulting value:

- If the value is **true**, then the entire **do while** statement is executed again.
- If the value is **false**, no further action is taken and the **do** statement completes normally.

If execution of the *Statement Block* completes abruptly because of a **continue** with label *L*, then there is a choice:

a) If the do statement has label *L*, then the *Guard Condition* is evaluated. Then there is a choice:

- If the value of the *Guard Condition* is **true**, then the entire **do** statement is executed again.
- If the value of the *Guard Condition* is **false**, no further action is taken and the **do while** statement completes normally.

b) If the **do while** statement does not have label *L*, the **do** statement completes abruptly because of a **continue** with label *L*.

Concrete textual grammar

433 DoWhileStatement ::= “do” StatementBlock “while” GuardCondition

Concrete graphical grammar

441 <fgr do-while area> ::= <inline expression symbol> **contains**
(**do while** GuardCondition <statement block area>)

The **do-while** statement shall be represented by an <inline expression symbol> labelled with a **do-while** definition. The **do-while** body shall be represented as the operand of the do-while <inline expression symbol>. A <flow line symbol> is drawn from the previous node to the <inline expression symbol>, and another <flow line symbol> is drawn from the <inline expression symbol> to the next node. Figure 20 represents an example of a **do-while** statement.

Figure 20. Do-while Statement

do while [j<10] MyChannel.Template1	**do {** MyChannel.**send(**Template1**)** **:** **} while** [j<10]
Graphical presentation	**Textual presentation**

18.6. The Break statement

A **break** statement transfers control out of an enclosing statement.

A **break** statement with no label attempts to transfer control to the innermost enclosing **alt**, **do while**, **while**, or **for** statement; this statement, which is called the *break target*, then immediately completes normally. To be precise, a **break** statement with no label always completes abruptly, the reason being a **break** with no label. If no **alt**, **while**, **do while**, or **for** statement encloses the **break** statement, a compile-time error occurs.

A **break** statement with *Label Identifier* attempts to transfer control to the enclosing labeled statement that has the same *Label Identifier* as its label; this statement, which is called the *break target*, then immediately completes normally. In this case, the break target need not be a **while**, **do while**, **for**, or **alt** statement. To be precise, a **break** statement with *Label Identifier* always completes abruptly, the reason being a **break** with *Label Identifier*. If no labeled statement with *Label Identifier* as its label encloses the **break** statement, a compile-time error occurs.

It can be seen, then, that a **break** statement always completes abruptly.

The preceding descriptions say “attempts to transfer control” rather than just “transfers control” because if there are any **try** statements within the break target whose **try** blocks contain the **break** statement, then any catch-any clauses of those **try** statements are executed, in order, innermost to outermost, before control is transferred to the break target.

If necessary, it is possible to enable/disable a **break** statement or a **continue** statement by means of a Boolean expression placed between the '[ ]' brackets of the statement.

Concrete textual grammar

436 BreakStatement ::= [GuardCondition] “break” [LabelIdentifier]

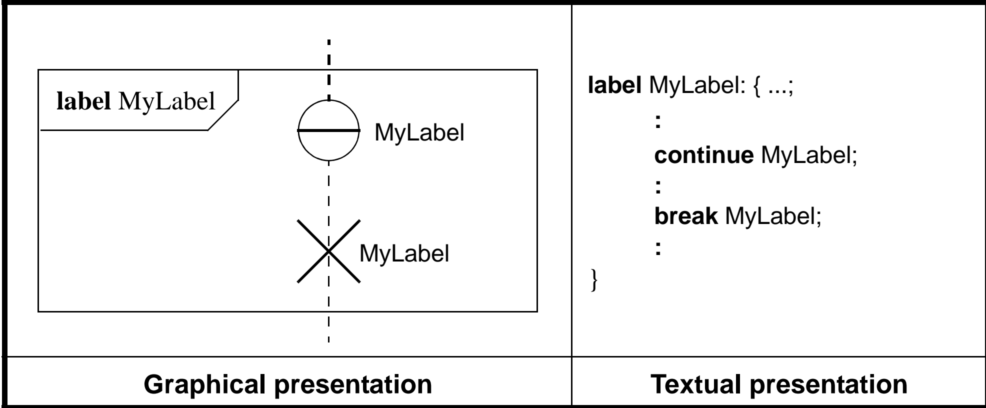
Concrete graphical grammar

444 <flow line symbol> ::= <flow line symbol1> | <flow line symbol2>

445 <break area> ::= <break symbol> [**is_associated_with** LabelIdentifier]
[**is_associated_with** GuardCondition]

446 <break symbol> ::= <stop symbol>

Figure 21. The Label, Break and Continue Statement



Graphical presentation	Textual presentation

The **break** statement shall be represented by a <stop symbol>, which is attached to the <flow line symbol>.

18.7. The Continue statement

A continue statement may occur only in a **while**, **do while**, or **for** statement; statements of these three kinds are called *iteration statements*. Control passes to the loop-continuation point of an iteration statement.

A **continue** statement with no label attempts to transfer control to the innermost enclosing **while**, **do while**, or **for** statement; this statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one. To be precise, such a **continue** statement always completes abruptly, the reason being a **continue** with no label. If no **while**, **do while**, or **for** statement encloses the continue statement, a compile-time error occurs.

A **continue** statement with *Label Identifier* attempts to transfer control to the enclosing labeled statement that has the same *Label Identifier* as its label, that statement, which is called the *continue target*, then immediately ends the current iteration and begins a new one. More precisely, a **continue** statement with *Label Identifier* always completes abruptly, the reason being a **continue** statement with *Label Identifier*. If no labeled statement with *Label Identifier* as its label contains the **continue** statement, a compile-time error occurs.

It can be seen, then, that a **continue** statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any **try** statements within the continue target whose **try** blocks contain the **continue** statement, then any catch-any clauses of those **try** statements are executed, in order, innermost to outermost, before control is transferred to the continue target.

Concrete textual grammar

437 ContinueStatement ::= [GuardCondition] "continue" [(LabelIdentifier | "alt")]

Concrete graphical grammar

443 <continue area> ::= <repeat symbol> [**is_associated_with** LabelIdentifier]
[**is_associated_with** GuardCondition]

The **continue** statement shall be represented by a <repeat symbol>.

18.8. The Stop execution statement

The **stop** statement terminates execution in different ways depending on the context in which it is used. When used in the control part of a module or in a function used by the control part of a

module, it terminates the test execution. When used in a test case, altstep or function that are executed on a test thread, it terminates the relevant test thread.

NOTE 1: The semantics of a **stop** statement that terminates a test thread is identical to the stop thread operation **self.stop** (see §21.2.5) .

The **stop** execution operation shall be represented by a <stop symbol>, which is attached to the <flow line symbol>, which performs the **stop** execution operation (Figure 22). It shall only be used as last event of a component instance or as last event of an operand in an <inline expression symbol>.

Figure 22. Stop execution operation

	stop;
Graphical presentation	**Textual presentation**

19. Behavioural program statements

Behavioural program statements may be used in test cases, functions and altsteps, except for:

(a) the **return** statement which shall only be used in functions; and

(b) the **alt** statement and the **continue** statement which may also be used in module control.

Table 23: Overview of ATDL behavioural program statements

Behavioural program statements	Associated keyword or symbol	Associated graphical symbol
Alternative behavior	**alt {…}**	<alt symbol>
Start of a try statement	**try { … }**	<try symbol>
End of a try statement	**try { … }**	<connector symbol>
Raise exception (to a procedure)	**raise**	<exception out symbol>
Catch exception (from callee)	**catch**	<exception in symbol>
Returning control	**return**	<return symbol>

Behavioural program statements specify the dynamic behaviour of the test components over the communication channels. Test behaviour can be expressed sequentially, as a set of alternatives or combinations of both.

19.1. Alternative behaviour

A more complex form of behavior is where sequences of statements are expressed as sets of possible alternatives to form a tree of execution paths. The **alt** statement denotes branching of test behavior due to the reception and handling of communication and/or timer events and/or the termination of parallel test components, i.e., it is related to the use of the ATDL operations **receive**, **trigger**, **synchronize**, **catch**, **timeout** and **done**. The **alt** statement denotes a set of possible events that are to be matched against a particular snapshot.

Concrete textual grammar

```
411   AltConstruct ::= "alt" "{" AltGuardList "}"
412   AltGuardList ::= {GuardStatement [SemiColon]}+ [ElseClause [SemiColon]]
```

413 GuardStatement ::= AltGuardChar (AltstepInstance | GuardOp StatementBlock)

414 ElseClause ::= "[" "else" "]" StatementBlock

19.1.1. Graphical notation

Fork vertices serve to split an incoming statement block into two or more statement blocks terminating on orthogonal target vertices. **Join** vertices serve to merge several statement blocks emanating from source vertices in different orthogonal regions. Antonym: join.

Execution starts at the <alt symbol>. Next, it continues with a node that follows one of the outgoing edges of this symbol. These nodes are considered to be branches of an **alt** operator. After execution of the selected node, the process of selection and execution is repeated for the outgoing edges of the selected node. Each branch can be labeled with a guard condition. The reserved word **else** can be used as a guard condition. Its value is true if all the other (explicit) guard conditions are false.

A guarding condition means that the execution may not continue beyond the condition if it evaluates to **false**. If all available branches are blocked by false conditions, and no <alt outlet symbol> has been reached, the whole statement graph has no legal traces.

Note that the <alt symbol> can also be used for a join (the inverse of a fork), in which two alternate paths come together, as shown in Figure 23. In the case of a join, there are two or more input <flow line symbol>s and a single output <flow line symbol>.

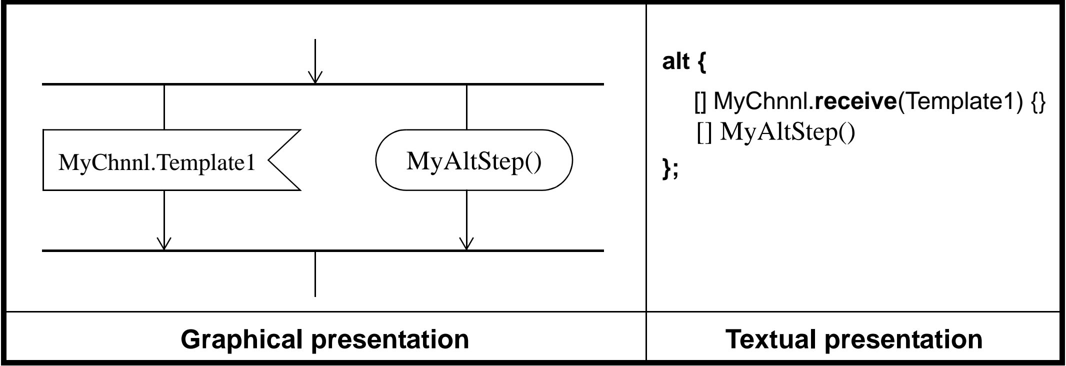
Figure 23. Alternative behaviour statement

MyChnnl.Template1 MyChnnl.Template2	**alt** {
	[] MyChnnl.**receive**(Template1) {}
	[] MyChnnl.**receive**(Template2) {}
	};

Graphical presentation	**Textual presentation**

Figure 23 illustrates an alternative behaviour in which either a message event is received with the value defined by Template1, or a message event is received with the value defined by Template2.

Figure 24. Alternative behaviour with altstep invocation



In addition, it is possible to call an altstep as the only event within an alternative operand. This shall be drawn using a reference symbol (see §16.4.3).

Concrete graphical grammar

417 <fgr alt area> ::= <alt symbol> **is_followed_by**
((<graphical guard part>}+ [<alt else part>]) **set**)

418 <alt outlet symbol> ::= <alt symbol>

420 <graphical guard part> ::= <flow line symbol>
[**is_associated_with** GuardCondition]
is_followed_by <fgr guard area>
is_followed_by <statement block area>
is_connected_to <alt outlet symbol>

421 <fgr guard area> ::= <fgr receive area> | <fgr trigger area> |
<fgr synchronize area> | <fgr catch area> |
<fgr timeout area> | <fgr done area> | <altstep instance area>

19.1.2. Execution of alternative behaviour

The alternative statements are processed in their order of appearance. ATDL operational semantics assume that the status of any of the events cannot change during the process of trying to match one of a set of alternatives. This implies that snapshot semantics are used for received events and **timeout**s *i.e.*, each time around a set of alternatives a snapshot is taken of which events have been received and which **timeout**s have fired. Only those identified in the snapshot can match on the next cycle through the alternatives.

The alternative branches in the **alt** statement and the top alternatives of invoked altsteps and altsteps that are activated as defaults are processed in the order of their appearance. If several defaults are active, the activation order determines the evaluation order of the top alternatives in the defaults. The alternative branches in active defaults are reached by the default mechanism described in clause 19.7.

19.1.3. Selecting/deselecting an alternative

If necessary, it is possible to enable/disable an alternative by means of a Boolean expression placed between the '[ ]' brackets of the alternative.

The evaluation of a Boolean expression guarding an alternative may have side-effects. To avoid side effects that cause an inconsistency between the actual snapshot and the state of the component the same restrictions as the restrictions for the initialization of local definitions within altsteps shall apply.

Concrete textual grammar

415 AltGuardChar ::= “[“ [BooleanExpression] “]”

Figure 25. Selecting/deselecting an alternative

[x>1] [x<=1] MyChnnl.Template1 MyChnnl.Template2	**alt** { [x>1] MyChnnl.**receive**(Template1) {} [x<=1] MyChnnl.**receive**(Template2) {} **}**;
Graphical presentation	**Textual presentation**

It is possible to disable/enable an alternative operand by means of a guard condition. Figure 25 illustrates a simple alternative statement in which the first branch is guarded with the expression x>1, and the second with the expression x <= 1.

19.1.4. Guard condition

A guard condition is a Boolean expression that is part of the specification of a guarded statement. A guard condition must be a query i.e., it may not modify the value of the system or its state; it may not have side effects.

Concrete textual grammar

448 GuardCondition ::= “[“ BooleanExpression “]”

Graphical notation

A guard condition is part of the string for a guarded statement. It has the form of a Boolean expression enclosed in square brackets.

19.1.5. Else branch in alternatives

The last branch in an **alt** statement can be defined as an else branch by including the **else** keyword between the open and close brackets at the beginning of the alternative. The else branch shall not contain any of the actions allowed in branches guarded by a boolean expression (i.e. an **altstep** call or a done, a timeout or a receiving operation). The statement block of the else branch is always executed if no other alternative textually preceding the else branch has proceeded.

Figure 26. Else within an alternative

Graphical presentation	Textual presentation
[x>1] MyChnnl.Template1 / [else] MyErrorHandler()	**alt** { [x>1] MyChnnl.**receive**(Template1) {} [**else**] MyErrorHandler() };

Concrete graphical grammar

419 <alt else part> ::= <flow line symbol>
is_associated_with (“[“ **else** “]”)
is_followed_by <statement block area>
is_connected_to <alt outlet symbol>

The **else** branch shall be denoted using a guard condition labelled with the **else** keyword. Figure 26 illustrates a simple alternative statement where the second branch represents an **else** branch. The invocation of altsteps within alternatives is represented using the reference symbol (see § 16.4.3).

19.1.6. ATDL test events

An occurrence of an event has an actual parameter corresponding to each event parameter.

There are five kinds of events: catch event, synchronize event, message event, done event and timeout event.

synchronize event: The receipt of a request to invoke an operation or method.

message event: The receipt of a message, which is an explicit named entity intended for explicit communication between objects. A message has an explicit list of parameters, it is explicitly sent by an object to another object or set of objects.

Concrete textual grammar

416 GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement |
SynchronizeStatement | CatchStatement | DoneStatement
/* STATIC SEMANTICS - GuardOp used within the module control part. Shall only contain the TimeoutStatement */

19.1.7. Re-evaluation of alt statements

The re-evaluation of an **alt** statement can be specified by using a **continue** statement (see clause 19.2).

19.1.8. Invocation of altsteps as alternatives

ATDL allows the invocation of altsteps as alternatives in **alt** statements.

Concrete graphical grammar

210 <altstep instance area> ::= <reference symbol> **contains** AltstepInstance

19.2. The Continue statement

The **continue** statement causes the re-evaluation of an **alt** statement, i.e. a new snapshot is taken and the alternatives of the **alt** statement are evaluated in the order of their specification. The **continue** statement shall only be used as last statement of an alternative in an **alt** statement or as last statement of a top alternative in an altstep definition.

If a **continue** statement is used as last statement of an alternative in an **alt** statement, it causes a new snapshot and the re-evaluation of the **alt** statement.

If a **continue** statement is used as last statement of a top alternative in an altstep definition, it causes a new snapshot and the re-evaluation of the **alt** statement from which the altstep has been called. The call of the altstep may either be done implicitly by the default mechanism or explicitly in the **alt** statement

Graphical notation

The **continue** statement shall be represented by a <repeat symbol>. This symbol shall only be used as last event of an alternative branch in an alt statement or as last event of an operand of the top alternative in an altstep definition.

19.3. The Return statement

A **return** action causes a transfer of control to the caller of a function or constructor. The action has an optional list of return values that are made available to the caller when it receives control.

A **return** statement with no *Expression* must be contained in the body of a function that is declared, not to return any value, or in the body of a constructor. A **return** statement with no *Expression* attempts to transfer control to the invoker of the function or constructor that contains it. To be precise, a **return** statement with no *Expression* always completes abruptly, the reason being a **return** with no value.

A **return** statement with an *Expression* must be contained in a function declaration that is declared to return a value or a compile-time error occurs. The *Expression* must denote a variable or value of some type *T*, or a compile-time error occurs. The type *T* must be assignable to the declared result type of the method, or a compile-time error occurs.

A **return** statement with an *Expression* attempts to transfer control to the invoker of the function that contains it; the value of the *Expression* becomes the value of the method invocation.

It can be seen, then, that a **return** statement always completes abruptly.

The preceding descriptions say "attempts to transfer control" rather than just "transfers control" because if there are any **try** statements within the method or constructor whose **try** blocks contain the **return** statement, then any catch-any clauses of those **try** statements will be executed, in order, innermost to outermost, before control is transferred to the invoker of the method or constructor.

Concrete textual grammar

409 ReturnStatement ::= "return" [Expression]

Concrete graphical grammar

410 <return area> ::= <return symbol> [**is_associated_with** Expression]

The **return** statement shall be represented by a <return symbol>. This may be optionally associated with a return value. A <return symbol> shall only be used in a function diagram. It shall only be used as last event of a component instance or as last event of an operand in an <inline expression symbol>. Figure 27 illustrates a simple function that returns a value.

Figure 27. Return symbol with a return value

ReturnValue	**return** ReturnValue;
Graphical presentation	**Textual presentation**

19.4. The Raise statement

A **raise** statement causes a remote exception or a self-exception to be raised. A remote-exception is a reaction to an accepted procedure call the result of which leads to an exceptional event.

Exceptions are specified as types. Therefore the exception value may either be derived from a template or be the value resulting from an expression (which of course can be an explicit value). The optional *exception field identifier* in the value specification to the raise operation shall be used in cases where it is necessary to avoid any ambiguity of the type of the value being sent.

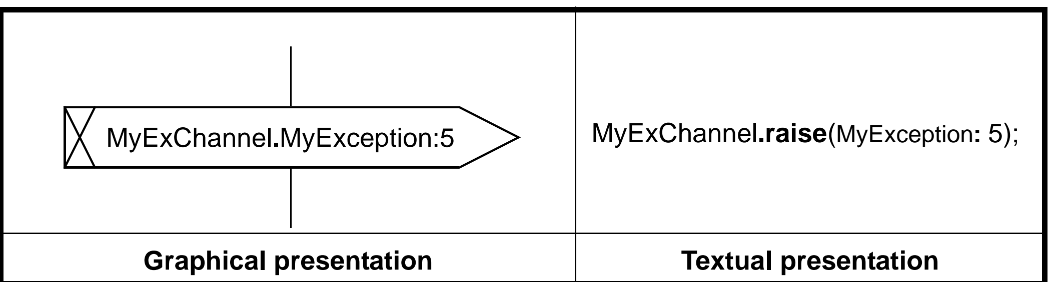
Concrete textual grammar

270 RaiseStatement ::= Channel Dot "raise" "(" TemplateInstance ")"

Concrete graphical grammar

282 <fgr raise area> ::= <exception out symbol> **contains** ([Channel Dot] TemplateInstance)

Figure 28. The Raise statement



Exception sending. The sending of an exception may be shown as an <exception out symbol>. The template instance of the exception is shown inside the symbol. A <flow line symbol> is

drawn from the previous node to the <exception out symbol>, and another <flow line symbol> is drawn from the <exception out symbol>.

19.4.1. Raise a self-exception

The result of raising a self-exception is an immediate transfer of control that may exit multiple statements evaluations and multiple constructor, and method invocations until a **try** statement is found that catches the raised value. If no such **try** statement is found, then execution of the thread that executed the **raise** statement is terminated.

If a self-exception **raise** statement is contained in a method declaration, but its value is not caught by some **try** statement that contains it, then the invocation of the method completes abruptly because of the **raise** statement.

If a self-exception **raise** statement is contained in a constructor declaration, but its value is not caught by some **try** statement that contains it, then the class instance creation expression that invoked the constructor will complete abruptly because of the **raise** statement.

19.4.2. Re-raising exceptions

When the reserved word **raise** occurs in an exception handling block without an object reference following it, it raises whatever exception is handled by the try block. This allows a self-exception handler to respond to an error in a limited way and then *re-raise* the exception. Re-raising is useful when a test case or function has to clean up after an exception occurs but cannot fully handle the exception.

19.5. Exception handling

Exception handling is a mechanism that allows two separately developed program components to communicate when an *exception* is encountered during the execution of the program. In this section we first look at how to associate handlers, or catch clauses, with a set of program statements using a try block, and we look at how exceptions are handled by catch clauses.

19.5.1. The Try statement

The **try** statment is introduced in order to facilitate the specification of the alternatives of the possible responses to the blocking call operation or method invocation. The **try** operation may be followed by alternatives of **catch** and timeout.

A **try** block must enclose the statements that can throw exceptions. A **try** statement executes a statement block. A try block begins with the **try** keyword followed by a sequence of program statements enclosed in braces. Following the try block is a list of handlers called *catch clauses*. The try block groups a set of statements and associates with these statements a set of handlers to handle the exceptions that the statements can throw.

The handling of exceptions to a **try** statement is done by means of the **catch** operation. This operation defines the alternative behavior depending on the exception (if any) that has been generated as a result of the **try** statement. If a value is thrown and the **try** statement has one or more **catch** clauses that can catch it, then control will be transferred to the first such **catch** clause.

If no exception occurs, the code within the **try** block is executed and the handlers associated with the try block are ignored. When an exception is thrown, the statements following the statement that throws the exception are skipped. Program execution resumes in the **catch** clause handling the exception.

Concrete textual grammar

460 TryStatement ::= “try” [“(“ TimerValue “)”] (Statement | StatementBlock) CatchesStatementList

/* STATIC SEMANTICS - TimerValue must be of type float */

461 CatchesStatementList ::= { [AltGuardChar] CatchStatement StatementBlock [SemiColon] }+

Concrete graphical grammar

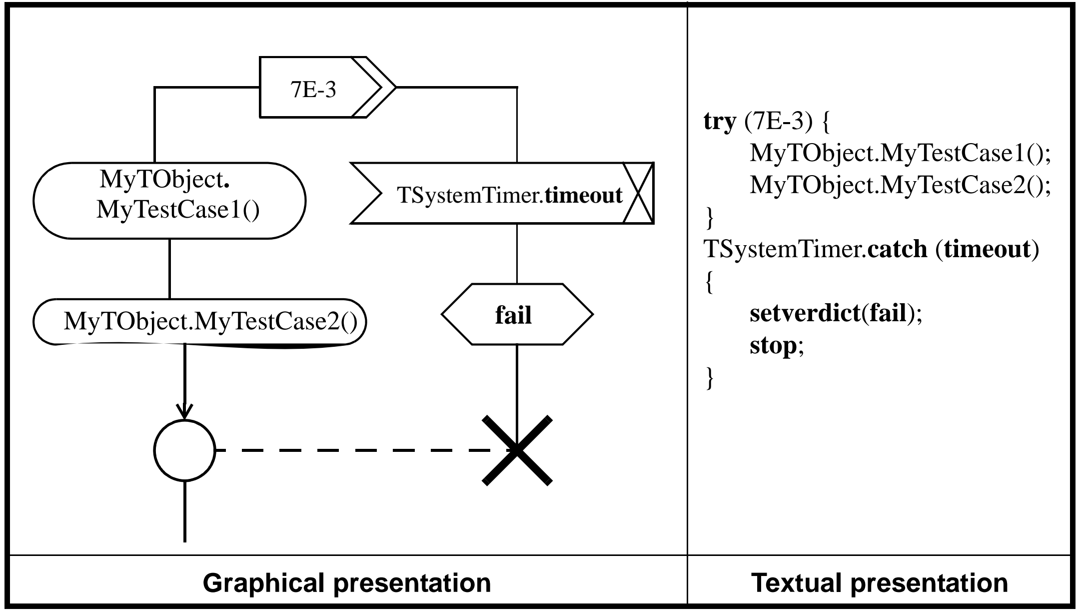
462 <try area> ::= <try symbol> **contains** [°(° TimerValue °)°]
is_associated_with ({<catch association area> }*
<try statement block area>) **set**

463 <try statement block area> ::= <statement block area>
is_connected_to <try out-connector symbol>

464 <catch association area> ::= <solid association symbol>
[**is_associated_with** GuardCondition]
is_connected_to <fgr catch area>
is_followed_by <statement block area>
is_connected_to <try out-connector symbol>

465 <try out-connector symbol> ::= <connector symbol>

Figure 29. The Try statement



The **try** statement is represented by use of the <try symbol>. An optional time supervision may be placed within that symbol (Figure 29). In a <try area>, all of the paths leaving a <try symbol> must eventually merge in a subsequent <try out-connector symbol>. If the last event of a catch clause is the **stop** execution operation, a <dashed association symbol> shall be drawn from the <stop symbol> to the <try out-connector symbol>.

19.5.2. The Catch clause

An ATDL exception handler is a *catch clause*. When an exception is thrown from statements within a try block, the list of catch clauses that follows the try block is searched to find a catch clause that can handle the exception.

The **catch** operation may be part of the exception handling part of a **try** statement or be used to determine an alternative in an **alt** statement.

A catch clause consists of a catch statement with the keyword **catch**, and a set of statements within a compound statement. If the **catch** clause is selected to handle an exception, the compound statement is executed.

Exceptions are specified as types and thus, can be treated like messages e.g. templates can be used to distinguish between different values of the same exception type. The (optional) assignment part of the **catch** operation comprises the assignment of the exception value. The keyword **value** is used to retrieve the value of an exception. For example,

MyExChannel.**catch**(MyExType:?) -> (MyVar := **value**;)

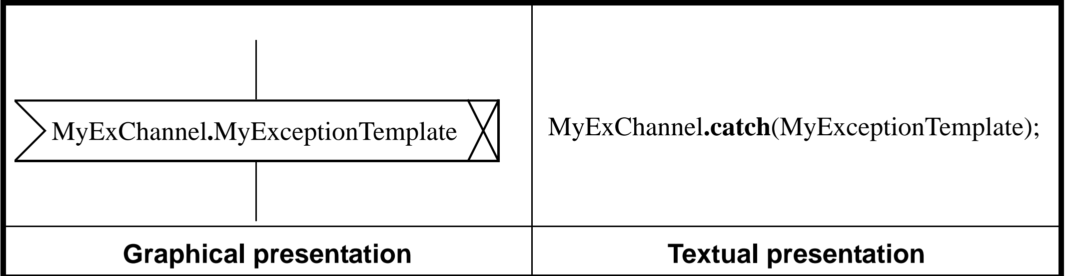
// Catches an exception and assigns its value to MyVar.

Concrete textual grammar

277 CatchStatement ::= ChannelOrAny Dot "catch" ["(" CatchOpParameter ")"] [AssignmentList]

278 CatchOpParameter ::= TemplateInstance | "timeout" | "all"

Figure 30. The Catch statement



Concrete graphical grammar

287 <fgr catch area> ::= <exception in symbol>
contains [[ChannelOrAny Dot]"(" CatchOpParameter ")"]
[**is_associated_with** <save area>]

Exception receipt. The receipt of an exception may be shown as an <exception in symbol>. The template instance of the exception is shown inside the symbol. A <flow line symbol> is drawn from the previous node to the <exception in symbol>, and another <flow line symbol> is drawn from the <exception in symbol>.

The (optional) assignment part (denoted by the **'->'**) shall be placed within a <save symbol>.

19.5.3. Catch a remote-exception

The **catch** remote-exception operation is used to catch exceptions raised by a peer entity as a reaction to an operation call. The type of the caught exception shall be specified in the signature of the operation that raised the exception.

The **catch** remote-exception operation removes the top exception from the associated incoming channel queue if, and only if, that top exception satisfies all the matching criteria associated with the **catch** operation. No binding of the incoming values to the terms of the expression or to the template shall occur.

19.5.4. The Timeout exception

The **try** statement may optionally include a timeout. This is defined as an explicit value or constant of **float** type and defines the length of time after the **try** statement has started that a **timeout** exception shall be generated by the test system. If no timeout value part is present in the **try** statement no **timeout** exception shall be generated.

The **timeout** exception is an emergency exit for cases where a called procedure neither replies nor raises an exception within a predetermined time. For example:

MyChannel.**catch**(**timeou**t); // Catches a timeout exception.

Catching **timeout** exceptions shall be restricted to the exception handling part of a **try** statement.

19.5.5. The catch all handler

A **catch all** clause allows any valid exception to be caught. For example,

MyExceptionChannel.**catch**(**all**);

19.5.6. The catch any clause

A method may want to perform some action before it exits with a raised self-exception even though it cannot handle the exception that is thrown. For example, a method may acquire some resource, such as opening a file or allocating memory on the heap, and it may want to release this resource (close the file or release the memory) before it exits with the raised self-exception.

The release of a resource may be bypassed if a self-exception is thrown. To guarantee that the resource is released, rather than provide a specific catch clause for every possible exception and because we can't know all the exceptions that might be thrown, we can use a catch-any clause.

A `catch any` is used in combination with a re-raise expression. The resource that has been locked is released within the compound statement of the catch clause before the self-exception is propagated further up the list of function calls with a re-raise expression.

To ensure that the resource is appropriately released if a self-exception is raised and a method exits with an exception a catch-any clause is used to release the resource before the exception is propagated to functions further up the list of method calls. We can also manage the acquisition and release of a resource by encapsulating the resource in a class, and having the class constructor acquire the resource and the class destructor release the resource automatically.

To **catch** an exception on any channel use the **any** keyword. This statement will also catch the **timeout** exception. For example,

any ExceptionChannel.**catch;**

A **catch any** clause can be used by itself or in combination with other catch clauses. If it is used in combination with other catch clauses, we must take some care when organizing the set of catch clauses associated with the **try** block.

Catch clauses are examined in turn, in the order in which they appear following the **try** block. Once a match is found, subsequent catch clauses are not examined. This implies that if a `catch any` is used in combination with other catch clauses, it must always be placed last in a list of exception handlers; otherwise, a compiler error is issued.

19.6. Test verdict operations

Verdict operations allow to set and retrieve verdicts using the **getverdict** and **setverdict** operations respectively. These operations shall only be used in test cases, altsteps and functions.

Each test component of the active configuration shall maintain its own local verdict. The local verdict is an object which is created for each test component at the time of its instantiation. It is used to track the individual verdict in each test component (i.e. in the MTC and in each and every PTC).

Table 24: Overview of ATDL test verdict operations

Test verdict operation	Associated keyword	Associated graphical symbol
Set local verdict	**setverdict**	<condition symbol>
Get local verdict	**getverdict**	<task symbol>

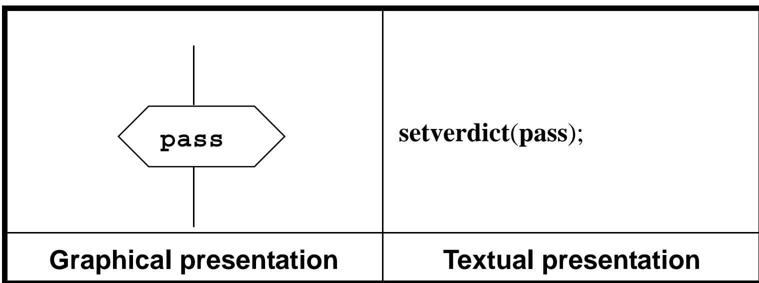
Concrete textual grammar

406 SetLocalVerdict ::= “setverdict” “(“ SingleExpression “)”

Concrete graphical grammar

408 <set verdict area> ::= <condition symbol> **contains** VerdictValue

Figure 31. Set local verdict



The verdict set operation **setverdict** is represented with a <condition symbol> within which the values **pass**, **fail**, **inconc** or **none** are denoted (Figure 31).

ATDL does not provide any graphical representation for the **getverdict** operation (being an expression). It is textually denoted at the places of its use.

19.6.1. Test case verdict

Additionally there is a global verdict that is updated when each test component (i.e. the MTC and each and every PTC) terminates execution. This verdict is not accessible to the **getverdict** and **setverdict** operations. The value of this verdict shall be returned by the test case when it terminates execution. If the returned verdict is not explicitly saved in the control part (e.g. assigned to a variable) then it is lost.

19.6.2. Verdict values and overwriting rules

The verdict can have five different values: **pass**, **fail**, **inconc**, **none** and **error** i.e. the distinguished values of the **verdicttype**.

NOTE: **inconc** means an inconclusive verdict.

The **setverdict** operation shall only be used with the values **pass**, **fail**, **inconc** and **none**. For example,

```
setverdict(pass);
setverdict(inconc);
```

The value of the local verdict may be retrieved using the **getverdict** operation. For example,

```
MyResult := getverdict;     // Where MyResult is a variable of type verdicttype
```

Current Value of Verdict	New verdict assignment value			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

Table 25: Overwriting rules for the verdict

When a test component is instantiated, its local verdict object is created and set to the value **none**.

When changing the value of the local verdict (i.e. using the **setverdict** operation) the effect of this change shall follow the overwriting rules listed in Table 25. The test case verdict is implicitly

updated on the termination of a thread or a passive object. The effect of this implicit operation shall also follow the overwriting rules listed in Table 25. For example,

```
:
setverdict(pass);          // the local verdict is set to pass
:
```

19.6.2.1. Error verdict

The **error** verdict is special in that it is set by the test system to indicate that a test case (i.e. run-time) error has occurred. It shall not be set by the **setverdict** operation and will not be returned by the getverdict operation. No other verdict value can override an **error** verdict. This means that an **error** verdict can only be a result of an execution of a test case.

19.7. Default Handling

ATDL allows the activation of altsteps (see clause 16.4) as defaults. For each test component the defaults, i.e. activated altsteps, are stored in form of a list. The defaults are listed in the order of their activation. The ATDL operations **activate** (see §19.7.3) and **deactivate** (see §19.7.4) operate on the list of defaults. An **activate** appends a new default to the end of the list and a **deactivate** removes a default from the list. A default in the default list can be identified by means of default reference that is generated as a result of the corresponding **activate** operation.

A Default Signal Handler is called by ATDL underlying system when it cannot find a method for a particular signal. Default Signal Handler provides signal handling for all signals for which an object does not have specific handlers. Descendant classes that process signals **activate** different Default Handler according to the types of signals they handle.

Concrete graphical grammar

424 <default area> ::= <default symbol> contains (ActivateStatement | DeactivateStatement)

Variables of type **default** can either be declared within a <task symbol> or within a <default symbol> as part of an activate statement.

19.7.1. The default mechanism

The default mechanism is evoked at the end of each **alt** statement, if due to the actual snapshot none of the specified alternatives could be executed. An evoked default mechanism invokes the first altstep in the list of defaults and waits for the result of its termination. The termination can be successful or unsuccessful. Unsuccessful means that none of the top alternatives of the **altstep** (see clause 16.4) defining the default behaviour could be selected, successful means that one of the top alternatives has been selected and executed.

In case of an unsuccessful termination, the default mechanism invokes the next default in the list. If the last default in the list has terminated unsuccessfully, the default mechanism will return to the place in the **alt** statement in which it has been invoked, i.e. at the end of the **alt** statement, and indicate an unsuccessful default execution. An unsuccessful default execution will also be indicated if the list of defaults is empty.

In case of a successful termination, the default may either stop the test component by means of a **stop** statement, or the main control flow of the test component will continue immediately after the **alt** statement from which the default mechanism was called or the test component will take new snapshot and re-evaluate the **alt** statement. The latter has to be specified by means of a **continue** statement (see §19.2). If the selected top alternative of the default ends without a **continue** statement the control flow of the test component will continue immediately after the **alt** statement.

19.7.2. Default references

Default references are unique references to activated defaults. Such a unique default reference is generated by a test component when an altstep is activated as a default, i.e. a default reference is the result of an **activate** operation (see clause 19.7.3).

Default references have the special and predefined type **default**. Variables of type **default** can be used to handle activated defaults in test components. The special value **null** is available to indicate an undefined default reference, e.g. for the initialization of variables to handle default references.

19.7.3. The activate operation

The **activate** operation is used to activate altsteps as defaults. An **activate** operation will append the referenced altstep to the list of defaults and return a default reference. The default reference is a unique identifier for the default and may be used in a **deactivate** operation for the deactivation of the default.

Concrete textual grammar

422 ActivateStatement ::= “activate” ”(“ AltstepInstance “)”

The activation of defaults shall be represented by the placement of the **activate** statement within a <default symbol> (Figure 32). A <flow line symbol> is drawn from the previous node to the <default symbol>, and another <flow line symbol> is drawn from the <default symbol> to the next node.

Figure 32. Default activation

MyDefaultVar:=**activate**(MyAltStep());	MyDefaultVar:=**activate**(MyAltStep());
Graphical presentation	**Textual presentation**

19.7.3.1. Activation of parameterized altsteps

The actual parameters of a parameterized altstep (see §16.4.1) that should be activated as a default, shall be provided in the corresponding **activate** statement. This means the actual parameters are bound to the default at the time of its activation (and not e.g. at the time of its invocation by the default mechanism).

19.7.4. The deactivate operation

The **deactivate** operation is used to deactivate defaults, i.e. previously activated altsteps. A **deactivate** operation will remove the referenced default from the list of defaults.

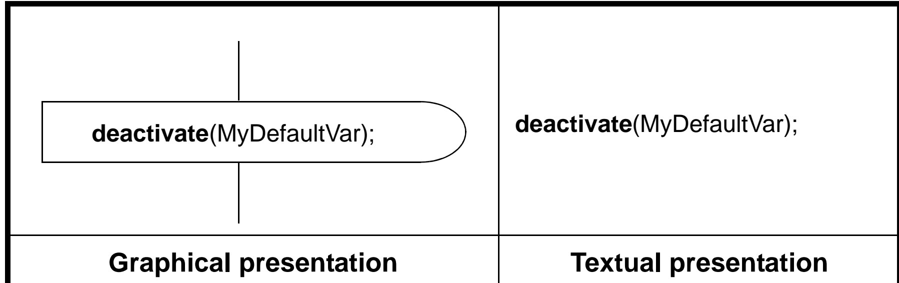
The effect of a **deactivate** operation is local to the test component in which it is called. This means, a test component cannot deactivate a default in another test component.

Concrete textual grammar

423 DeactivateStatement ::= “deactivate” [“(“ Expression “)”]

The deactivation of defaults shall be represented by the placement of the **deactivate** statement within a <default symbol> (Figure 33).

Figure 33. Deactivation of defaults



20. Expressions

Much of the work in an ATDL program is done by evaluating expressions, either for their side effects, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

ATDL allows the specification of expressions using the operators defined in Table 7. Expressions are built from other (simple) expressions. Expressions may contain functions. The result of an expression shall be the value of a specific type and the operators used shall be compatible with the type of the operands.

Concrete textual grammar

470 Expression ::= SingleExpression | CompoundExpression

471 CompoundExpression ::= FieldExpressionList | ArrayExpression

20.1. Boolean expressions

An expression that evaluates to a Boolean value.

A *boolean expression* shall only contain **boolean** values and/or **boolean** operators and/or relational operators and shall evaluate to a **boolean** value of either **true** or **false**.

Concrete textual grammar

478 BooleanExpression ::= SingleExpression

20.1.1. Conditional ? operator

The conditional operator ? : uses the boolean value of one expression to decide which of two other expressions should be evaluated.

The conditional operator has three operand expressions; ? appears between the first and second expressions, and : appears between the second and third expressions.

The first expression must be of type boolean, or a compile-time error occurs.

The conditional ? operator may be used to choose between second and third operands of numeric type, or second and third operands of type boolean, or second and third operands that are each of either reference type or the null type. All other cases result in a compile-time error.

At run time, the first operand expression of the conditional expression is evaluated first; its boolean value is then used to choose either the second or the third operand expression:

- If the value of the first operand is true, then the second operand expression is chosen.
- If the value of the first operand is false, then the third operand expression is chosen.

Concrete textual grammar

482 SingleExpression ::= ConditionalExpression [? SimpleExpression Colon ConditionalExpression]

20.2. Primary expressions

Primary expressions include most of the simplest kinds of expressions, from which all others are constructed: literals, field accesses, method invocations, and data object accesses. A parenthesized expression is also treated syntactically as a primary expression.

ATDL concrete textual grammar

```
494   Primary ::= OpCall | DataObjectReference | Value | "(" Expression ")"
```

20.2.1. Self

The keyword **self** may be used only in the body of an instance method or constructor, or in the initializer of an instance variable of a class. If it appears anywhere else, a compile-time error occurs. When used as an object reference, the keyword **self** denotes a value, that is a reference to the object for which the instance method was invoked, or to the object being constructed. **Self** is useful for a variety of reasons. For example, a member identifier declared in a class type might be re-declared in the block of one of the class's methods. In this case, you can access the original member identifier as **self**.*Identifie*r.

The keyword **self** is also used in a special explicit constructor invocation statement, which can appear at the beginning of a constructor body.

20.2.2. Parenthesized expressions

A parenthesized expression is a primary expression whose type is the type of the contained expression and whose value at run time is the value of the contained expression.

20.3. Typecast expressions

The type of a typecast expression is the type whose name appears before the parentheses. (The parentheses and the type are sometimes called the *cast operator*.) The result of a typecast expression is not a variable, but a value, even if the result of the operand expression is a variable. If the expression is a variable, the result is called a *variable typecast*; otherwise, the result is a *value typecast*. While their syntax is the same, different rules apply to the two kinds of typecast.

At run time, the operand value is converted by casting conversion (see clause 9.10) to the type specified by the cast operator. Not all typecasts are permitted by ATDL. Some typecasts result in an error at compile time. For example, a primitive value may not be cast to a restricted type. Some typecasts can be proven, at compile time, always to be correct at run time. For example, it is always correct to convert a value of a class type to the type of its ancestor class; such a typecast should require no special action at run time. Finally, some typecasts cannot be proven to be either always correct or always incorrect at compile time. Such casts require a test at run time.

20.3.1. Value typecasts

In a value typecast, the type identifier and the cast expression must both be ATDL *predefined* basic types. The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

A value typecast cannot appear on the left side of an assignment statement.

20.3.2. Variable typecasts

You can cast any variable to any type, provided their sizes are the same.

20.4. Component instance creation expressions

A component instance creation expression is used to create new objects that are instances of components. Creation of an component is the result of a signal that instantiates the object. A creation operation may have actual parameters that are used for initialization of the new instance. At the conclusion of the creation operation, the new object obeys the constraints of its class and may receive signals.

To create an object, call the constructor method in a class type. For example,

```
MyNewComponent := MyComponentType.create;
MyNewComponent. MyComponentBehaviour1(...);
```

The arguments in the argument list, if any, are used to select a constructor declared in the body of the named class type, using the same matching rules as for method invocations (§20.6). As in method invocations, a compile-time method matching error results if there is no unique constructor that is both applicable to the provided arguments and the most specific of all the applicable constructors.

Concrete textual grammar

250 CreateOp ::= (ComponentType Dot | “inherited”) (“create” | ConstructorIdentifier [ActualParList])

Concrete graphical grammar

260 <create request area> ::= <create request symbol> **contains** [VarPrefix] CreateOp

Instance creation. The creation of an instance may be shown as a <create request symbol>. A flow line symbol is drawn from the previous node to the <create request symbol>, and another flow line symbol is drawn from the <create request symbol> (Figure 34). The <create request symbol> contains the instance creation statement.

Figure 34. Instance creation operation

MyComp := MyCType.**create**	MyComp := MyCType.**create;**
Graphical presentation	**Textual presentation**

20.4.1. Initializing the test component

The MTC is the only thread which is automatically created when a test case starts. All other test components (the PTCs) shall be created explicitly during test execution by Instance Creation operations. A component is created with its full set of interfaces of which the incoming queues are empty. Furthermore, its incoming interfaces shall be in a listening state ready to receive traffic over the connection.

All component variables and timers are reset to their initial value (if any) and all component constants are reset to their assigned values when the component is explicitly or implicitly created.

The Instance Creation operation shall return the unique component reference of the newly created instance. The unique reference to the component will typically be stored in a variable and can be used for connecting instances and for field access expressions.

Components can be created at any point in a behaviour definition providing full flexibility with regard to dynamic configurations (i.e. any component can create any other PTC). The visibility of component references shall follow the same scope rules as that of variables and in order to

reference components outside their scope of creation the component reference shall be passed as a parameter or as a field in a message.

20.4.2. Component instance

An object is an instance that originates from a class. A co-object is an instance that originates from a co-class. A thread object is an instance that originates from a thread class.

An object is an instance of a component, which describes the set of possible objects that can exist. An object can be viewed from two related perspectives: as an entity at a particular point in time with a specific value and as a holder of identity that has different values over time.

A variable of a component type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value **null**. But you don't have to explicitly de-reference a class-type variable to access the object it points to. For example, SomeObject.Size := 100 assigns the value 100 to the *Size* property of the object referenced by *SomeObject*.

Concrete textual grammar

251 ComponentIdentifier ::= VariableRef | FunctionInstance | CastExpression

Graphical notation

The canonical notation for an object is a rectangle with two compartments. The top compartment contains the object name and class, and the bottom compartment contains a list of property names and values. There is no need to show methods because they are the same for all objects of a class.

The top compartment shows the name of the object and its class, all underlined, using the syntax:

objectname: classname

The property value compartment as a whole may be suppressed.

20.5. Field access expressions

A field access expression may access a field of an object, a reference to which is the value of either an expression or the special keyword inherited. It is also possible to refer to a field of the current instance or current class by using a simple name, if it is a simple name, then the *field name* is treated exactly as if it had been the field access expression self.field identifier.

Fields are statically bound; that is, references to them are fixed at compile time. This lack of dynamic lookup for field accesses allows ATDL to run efficiently with straightforward implementations. The power of late binding and overriding is available in ATDL, but only when instance methods are used. To see what this means, consider the following code.

```
class TAncestor { Value Smallint; }
class TDescendant extends TAncestor {
        Value charstring;        // hides the inherited Value field
}

{ var MyObject TAncestor;

        MyObject := TDescendant.Create;
        MyObject.Value := "Hello!";                  // error
        TDescendant(MyObject).Value := "Hello!";   // works!
}
```

Although *MyObject* holds an instance of *TDescendant*, it is declared as *TAncestor*. The compiler therefore interprets *MyObject.Value* as referring to the (integer) field declared in *TAncestor*. Both fields, however, exist in the *TDescendant* object; the inherited *Value* is hidden by the new one, and can be accessed through a typecast.

20.5.1. Field access using an object reference

The type of the *variable reference* must be a reference type *T*, or a compile-time error occurs. The meaning of the field access expression is determined as follows:

a) If the field identifier names several accessible member fields of type *T*, then the field access is ambiguous and a compile-time error occurs.

b) If the field identifier does not name an accessible member field of type *T*, then the field access is undefined and a compile-time error occurs.

c) Otherwise, the field identifier names a single accessible member field of type *T* and the type of the field access expression is the declared type of the field. At run time, the result of the field access expression is computed as follows:

- If the field is static: a) If the field is a constant, then the result is the value of the specified class constant in the class that is the type of the *object reference*. b) If the field is a class variable, then the result is the value of the specified class variable in the class that is the type of the *variable reference*.
- If the field is not static, then the result is a variable, namely, the specified instance variable in the object referenced by the value of the *variable reference*.

20.5.2. Accessing inherited members

The special form using the keyword inherited is valid only in an instance method or constructor, or in the initializer of an instance variable of a class; these are exactly the same situations in which the keyword self may be used.

20.6. Method invocation expressions

A method invocation expression is used to invoke a class or instance method.

When you call a test case or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers). In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list.

When calling a routine, remember that

- expressions used to pass typed **template** and **in** parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass **inout** and **out** parameters must be identically typed with the corresponding formal parameters.
- only assignable expressions can be used to pass **inout** and **out** parameters.

The signature arguments of the method invocation expression can be used to retrieve variable values for **out** and **inout** parameters. The actual assignment of the **out** and **inout** parameter values to variables shall implicitly be made in the method invocation.

A class or instance method can be called through an interface reference or an object reference.

If a method's declaration specifies an **inout** parameter, you must pass an assignable expression i.e., a variable to the method when you call it.

Concrete textual grammar

```
496  OpCall ::= ConfigurationOps | GetLocalVerdict | TimerOps | TestcaseInstance | FunctionInstance
              | CallStatement | InterfaceOps | TemplateOps | ActivateStatement
```

20.6.1. Invocation of functions

A function is invoked by referring to its name and providing the actual list of parameter. Functions that do not return values shall be invoked directly. Functions that return values may be invoked directly or inside expressions. The rules for actual parameter lists shall be followed as defined in clause 5.2.

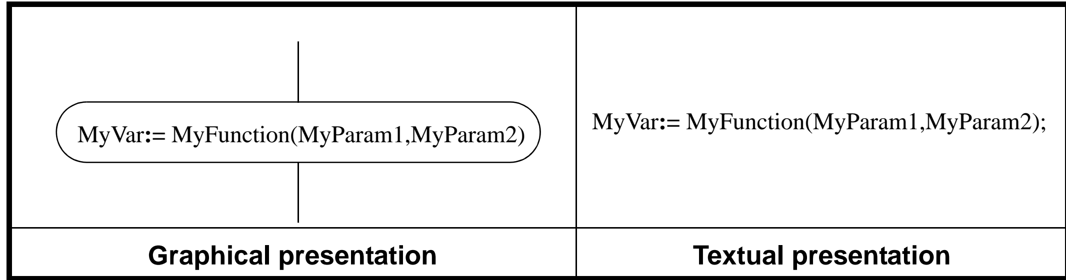
Special restrictions apply to functions bound to test threads using the **start** thread operation. These restrictions are described in clause 21.2.4.

Concrete textual grammar

171 FunctionInstance ::= FunctionRef [ActualParList]

172 FunctionRef ::= [DataObjectReference Dot]
(FunctionIdentifier | DestructorIdentifier) | PreDefFunctionIdentifier

Figure 35. Invocation of user defined function



Concrete graphical grammar

181 <function instance area> ::= <reference symbol> **contains** [VarPrefix] FunctionInstance

The invocation of functions is represented by the reference symbol (Figure 35), except of external and predefined functions and except where a function is called inside an ATDL language element that has a graphical representation.

The syntax of the function invocation is placed within the reference symbol. The symbol may contain: a) the invocation of a function with optional parameters; and b) an optional assignment of the returned value to a variable.

The reference symbol is only used for user defined functions defined within the current module. It shall not be used for external functions or predefined ATDL functions, which shall be represented in their text form within a task form or other graphical symbols.

20.6.2. Execution of test cases

A test case is called using a test case invocation expression or a try statement. As the result of the execution of a test case a test case verdict of either **none**, **pass**, **inconclusive**, **fail** or **error** shall be returned and may be assigned to a variable for further processing.

Optionally, the **try** statement allows supervision of a test case by means of a timer duration (see clause 19.5.1).

Concrete textual grammar

197 TestcaseInstance ::= TestcaseRef [ActualCrefParList]

198 TestcaseRef ::= [DataObjectReference Dot] TestcaseIdentifier

Concrete graphical grammar

202 <testcase instance area> ::= <reference symbol> **contains** [VarPrefix] TestcaseInstance

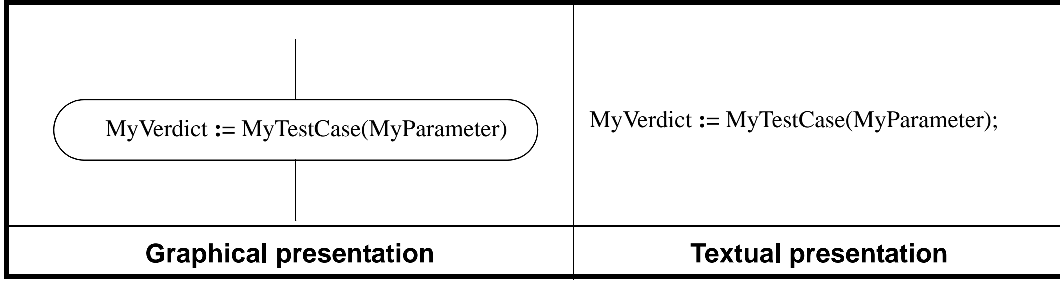
The invocation of test cases is represented by the reference symbol. The syntax of the test case invocation is placed within the <reference symbol>. The symbol may contain:

a) the invocation of a test case with optional parameters;

b) optionally, the assignment of the returned verdict to a **verdicttype** variable; and

c) optionally, the inline declaration of the **verdicttype** variable.

Figure 36. Test case execution



20.6.3. Determining the method

Resolving a method name at compile time is more complicated than resolving a field name because of the possibility of method overloading. Invoking a method at run time is also more complicated than accessing a field because of the possibility of instance method overriding.

Determining the method that will be invoked by a method invocation expression involves several steps. The first step in processing a method invocation at compile time is to figure out the name of the method to be invoked and which class to check for definitions of methods of that name. There are several cases to consider, depending on the form that precedes the left parenthesis, as follows:

1) If the form is ComponentRef . Method Identifier, then the name of the method is the Method Identifier and the class to be searched is the type of the ComponentRef expression.

2) If the form is inherited . Method Identifier, then the name of the method is the Method Identifier and the class to be searched is the ancestor class of the class whose declaration contains the method invocation. It follows that a method invocation of this form may appear only in a class other than TObject, and only in the body of an instance method, the body of a constructor, the body of a destructor, or an initializer for an instance variable.

The second step searches the class determined in the previous step for method declarations.This step uses the name of the method and the types of the actual parameter expressions to locate method declarations that are both *applicable* and *accessible*, that is, declarations that can be correctly invoked on the given actual parameters. There may be more than one such method declaration, in which case the *most specific* one is chosen. The descriptor (signature plus return type) of the most specific method declaration is one used at run time to do the method dispatch.

A method declaration is *applicable* to a method invocation if and only if both of the following are true: a) The number of formal parameters in the method declaration equals the number of actual parameter expressions in the method invocation. b) The type of each actual parameter can be converted by method invocation conversion to the type of the corresponding formal parameter. Method invocation conversion is the same as assignment conversion.

The class determined by the first step is searched for all method declarations applicable to this method invocation; method definitions inherited from ancestor classes are included in this search.

Whether a method declaration is *accessible* to a method invocation depends on the access modifier (public, none, protected, or private) in the method declaration and on where the method invocation appears.

If the class has no method declaration that is both applicable and accessible, then a compile-time error occurs.

20.6.4. Choose the most specific method

If more than one method is both accessible and applicable to a method invocation or a remote procedure call, it is necessary to choose one to provide the descriptor for the run-time method dispatch. ATDL uses the rule that the *most specific* method is chosen.

The informal intuition is that one method declaration is more specific than another if any procedure invocation handled by the first method could be passed on to the other one without a compile-time type error.

A method is said to be *maximally specific* for a method invocation if it is applicable and accessible and there is no other applicable and accessible method that is more specific.

If there is exactly one maximally specific method, then it is in fact *the most specific* method; it is necessarily more specific than any other method that is applicable and accessible. It is then subjected to some further compile-time checks as described in the first step.

It is possible that no method is the most specific, because there are two or more maximally specific method declarations. In this case, we say that the procedure invocation is *ambiguous*, and a compile-time error occurs.

20.7. References for data objects

In order to permit references to fields of structured data objects or data objects defined using ASN1, ATDL provides three access mechanisms: record references, array references and bit references.

Concrete textual grammar

468 DataObjectReference ::= [ModuleName Dot] ComponentRef [ExtendedFieldReference]

469 ExtendedFieldReference ::= { ArrayOrBitRef | (Dot (StructFieldIdentifier | ClassFieldIdentifier) }+

62 ArrayOrBitRef ::= “[“ FieldOrBitNumber “]”

63 FieldOrBitNumber ::= SingleConstExpression

20.7.1. Array references

An array reference may be used to reference a field of a data object of the type **sequence of** or ASN.1 **set of**. An array reference shall be constructed using an index notation, appending the index of the desired component to the data object identifier. The index, giving the position of the field within the data object (when the object is viewed as a linear array), is enclosed within square brackets. By definition within ASN.1, the indexing of fields starts with zero. The index may be an expression, in which case it shall evaluate to a **cardinal**.

20.7.2. Record references

A record reference may be used to reference a struct field of a data object of the type **sequence**, **set** or **choice**. A record reference is constructed using a dot notation, appending a dot and the name (struct field identifier) of the desired component to the data object identifier. The struct field identifier should be used in preference to the struct field position. You can access the fields of a record by qualifying the field designators with the record’s name.

Recommendations X.680 defines **set** types having unordered components. This is relevant only if values of that type are encoded and sent over the underlying service-provider. ATDL therefore treats data objects of **set** type in the same way as objects of **sequence** type.

When a record field is chained to another record, a record reference may be used to identify a component of the latter record type. The record reference shall identify the relevant complete sequence of field or element names separated by dots, starting with a data object identifier which resolves to the relevant record’s name. Beyond this initial data object identifier the sequence shall not contain any record identifiers, but rather just the identifiers of the relevant fields.

EXAMPLE 4: Record references with chaining

```
type ASP1_type ::= sequence {
                        par1    octetstring,
                        par2    octetstring,
                        Pdu1    PDU1_type
}

type PDU1_type ::= sequence {
                        Field1  octetstring,
                        Field2  octetstring,
                        F       F_type
}

type F_type ::= sequence {
                        Data1   charstring,
                        Data2   charstring
}
```

When using variables of type ASP1_type, PDU1_type and F_type, the values of Data1 and Data2 may be referenced as follows:

```
ASP1_Type.pdu1.F.Data1
ASP1_Type.pdu1.F.Data2
```

Similarly the whole PDU field F may be referenced as:

```
ASP1_Type.pdu1.F
```

20.7.3. String references

Individual elements in a string type may be accessed using an array-like syntax. Only single elements of the string may be accessed. For this purpose, data objects of *bitstring* type are assumed to be defined as **sequence of** {*boolean*}, data objects of *octetstring* type are assumed to be defined as **sequence of** {*octet*}, data objects of *hexstring* type are assumed to be defined as **sequence of** {*hexdecimal*}, data objects of *charstring* type are assumed to be defined as **sequence of** {*char*}, data objects of *wide charstring* type are assumed to be defined as **sequence of** {*wide char*}. Thus, a string reference may be constructed using the index notation as for array references. The leftmost position has the index zero (0). Indexing shall begin with the value zero. An expression used as an index in a string reference shall evaluate to a non-negative cardinal.

Units of length of different string type elements are indicated in Table 16.

20.8. Assignments

Values may be assigned to variables. This is indicated by the symbol ":=". The **:=** symbol is sometimes called the *assignment operator*. During execution of an assignment the righthand side of the assignment shall evaluate to an element of the same type of the left-hand side.

The effect of an assignment is to bind the variable to the value of the expression. The expression shall contain no unbound variables. All assignments occur in the order in which they appear, that is left to right processing. The result of the first operand of an assignment operator must be a variable, or a compile-time error occurs. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or a variable typecast, or a data object reference. The type of the assignment expression is the type of the variable.

At run time, the result of the assignment expression is the value of the variable after the assignment has occurred. The result of an assignment expression is not itself a variable.

A compile-time error occurs if the type of the right-hand operand cannot be converted to the type of the variable by assignment conversion.

Concrete textual grammar

467 Assignment ::= DataObjectReference ":=" Expression

20.8.1. Assignment rules for array types

Arrays are assignment-compatible only if they are of the same type. Because ATDL uses name-equivalence for types, the following code will not compile.

```
var Int1 sequence [10] of Smallint;
var Int2 sequence[10] of Smallint;
:
Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1: Int2 sequence[10] of Smallint;
```

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var A:B sequence of Smallint;
SetLength(A, 1);
SetLength(B, 1);
A[0] := 2;
B[0] := 2;
```

A == B returns *False* but A[0] == B[0] returns *True*.

Indexed value notations can be used on both the right-hand side and left-hand side of assignments. The index of the first element shall be zero and the index value shall not exceed the limitation placed by length subtyping. If the value of the element notated by the index at the right-hand of an assignment is undefined, this shall cause a semantical or run-time error. If an indexing operator at the left-hand side of an assignment refers to a non-existent element, the value at the right-hand side is assigned to the element and all elements with an index smaller than the actual index and without assigned value are created with an undefined value. Undefined elements are permitted only in transient states (while the value remains invisible). Sending a **sequence of** value with undefined elements shall cause a dynamic testcase error. For example,

```
var MyRecordVar sequence of Smallint;
SetLength(MyRecordVar, 6);
// The following two assignments
MyRecordVar := { 0, 1, -, 2, omit };
MyRecordVar[6] := 6;
// will result in { 0, 1, <unchanged>, 2, <undefined>, <undefined>, 6 };
// Note also, that the 3rd element would remain undefined if had no assigned value before.
// and the 6th element (with index 5) had no assigned value before this assignment.
```

This makes possible to copy **sequence of** values element by element in a for loop.

Concrete textual grammar

474 ArrayExpression ::= “{“ [NotUsedOrExpression {“,” NotUsedOrExpression}*] “}”

475 NotUsedOrExpression ::= Expression | “-”

20.8.2. Assignment rules for string types

If length-restricted string types are used within an assignment, the following rules apply:

a) if the destination string type is defined to be shorter than the source string, the source string is truncated on the right to the maximum length of the destination string type;

b) if the source string is shorter than that allowed by the destination string type, then the source string is left-aligned and padded with fill characters up to the maximum size of the destination string type.

Fill characters are:

- " " (blank) for all CharacterStrings;
- "0" (zero) for **bitstring**s, **hexstring**s and **octetstring**s.

When an unbounded (i.e., arbitrary length) string type variable is used on the left-hand side of an assignment, it shall become bound to the value of the right-hand side without padding. Padding is only necessary when the variable is of a fixed length string type.

21. Object-based programming

Objects provide a way to divide a program up into independent sections. Often, you also need to turn a program into separate, independentlyrunning subtasks. Each of these independent subtasks is called a *thread*, and you program as if each thread runs by itself and has the CPU to itself.

A class template is a prescription for creating a class in which one or more types or values are parameterized. A vector class, for example, may parameterize the generic type of the elements it contains. A buffer class may parameterize not only the generic type of the elements it holds, but the size of its buffer as well. This chapter discusses how to define a class template and how to create specific instances of a class template.

21.1. Class templates

This section describes class templates, and how to define and use them. A class template is a prescription for creating a class in which one or more types or values are parameterized. Let's assume that we want to define a class to support the mechanism of a queue. A queue is a data structure for a collection of objects in which objects are added at one end, the back, and removed at the other end, the front. The behavior of a queue is spoken of as first in, first out, or *FIFO*.

A definition for our class Queue might look like this:

```
virtual class Queue {
        public constructor Queue();
        public destructor Destroy();
        public function remove() return variant;
        public function append (MyType variant);
        public function is_empty() return boolean;
        :
        }
```

The question is, what type should we use for variant? Let's assume we choose to incarnate our virtual class Queue, replacing variant with Shortint. The class Queue is then defined to handle collections of objects of type Shortint. Because each object in the collection is an object of type Shortint, the ATDL type system guarantees that only values of type Shortint can be assigned to an object of type Queue. This is good when the programmer wishes to use a queue of objects of type Shortint, of course. This is not as good, however, when the programmer wishes to use the class Queue to represent a collection of floats, chars, or hexstrings.

One method of coping is simply to use brute force. The programmer copies the entire Queue class implementation, modifying it to work with floats, then with chars, then with hexstrings, and so on. And, since class names cannot be overloaded, each implementation must be given a unique name: ShortintQueue, FloatQueue, CharQueue, HexstringQueue. As each new class type is needed, the code is copied, changed, and renamed.

What are the problems with this method of class type duplication? Not only there is the lexical complexity of each uniquely named Queue class. And also, there is the administrative complexity — imagine having to propagate a modification in the general implementation of the class ShortintQueue to each specific instance. In general, providing manually generated copies for individual types is a never-ending process and is endlessly complicated to maintain.

The ATDL class template facility provides for the automatic generation of class types. One can use a class template to generate a class Queue automatically for a queue of any particular type. The class template definition for the Queue class might look like this:

```
template class Queue(MyType variant) {
        public constructor Create(inout MyType);
        public destructor Destroy();
        public function remove() return MyType;
        public function append (MyType);
        public function is_empty() return boolean;
        :
        }
```

The programmer writes

```
    var qshortint    Queue(Shortint);
    var qchar        Queue(char);
    var qhexstring   Queue(hexstring);
```

to generate, in turn, a Queue class of Shortints, chars, and hexstrings.

The implementation of our Queue class is presented in the following sections to illustrate the definition and use of class templates. The incarnation uses a pair of class template abstractions:

1. The class template Queue itself provides the public interface described earlier, and a pair of data fields: front and back. The class template Queue is implemented as a linked list.

2. The class template QueueItem represents one node of a Queue's linked list. Each item entered into the queue is stored in a QueueItem object. A QueueItem object contains a pair of data fields: value and next. The actual type of value varies with each instance of Queue. next is a link to the next QueueItem object in the queue.

Here is the definition of the class template Queue.

```
template class Queue(MyType variant) {
        public constructor Create(inout MyType);
        public destructor Destroy();
        public function remove() return MyType;
        public function append (MyType);
        public function is_empty() return boolean;
        :
        private front QueueItem(MyType);
        private back QueueItem(MyType);
}
```

21.1.1. Class template definition

The template keyword always begins both a definition and a declaration of a class template. This keyword is followed by a comma-separated list of template parameters enclosed in parentheses. This list is referred to as the *template parameter list* of the class template. It cannot be empty. A template parameter may be a generic type parameter or a non-generic type parameter.

A template *generic type parameter* consists of the type dentifiers followed by a generic type keyword, such as integer or cardinal. The followed keyword indicates that the parameter name represents a built-in or a user-defined type.

A class template can have multiple generic type parameters. Once declared, the generic type parameter serves as a generic type specifier for the remainder of the class template definition. It can be used in the class template definition in exactly the same way as a built-in or user-defined type is used in a nontemplate class definition. For example, a generic type parameter can be used to declare data members, member functions, and so forth.

A template *non-generic type parameter* consists of an ordinary parameter declaration. A non-generic type parameter indicates that the parameter name represents a potential value. This value represents a constant in the class template definition. For example, a Buffer class template may have a non-generic type parameter to indicate the type of the elements it holds and a non-generic type parameter that is a constant value representing its size.

A class definition follows the template parameter list. Except for the presence of the template parameters, the definition of a class template looks the same as that of a non-template class:

```
template class QueueItem (MyType variant) {
            public ...    // ...
            // MyType represents the type of a data field
            private item  MyType;
            private next  QueueItem;        // next pointer with this class
};
```

In the example, MyType is used to indicate the type of the data field item. In the course of the program, MyType will be substituted with various built-in and user-defined types. This process of type substitution is called *template instantiation*.

The name of a template parameter can be used after it has been declared as a template parameter and until the end of the template declaration or definition. If a variable with the same name as the template parameter is declared in global scope, that name is hidden.

The name of a template parameter cannot be used as the name for a class member declared within the class template definition. The name of a template parameter can be introduced only once within the template parameter list.

Inside the class template definition, the name of the class template can be used as a type specifier whenever a non-template class name can be used.

A class template is implicitly final, because its definition is complete and no descendant class templates are desired or required. This implies that a class template cannot have any descendant class templates.

Concrete textual grammar

```
117  ClassTemplateDef ::= "template" "class" ClassTemplateIdentifier
                          [TypeDefFormalParList] [ClassHeritage] ClassDefBody
118  ClassTemplateIdentifier ::= Identifier
```

21.1.2. Class template instantiation

A class template definition specifies how individual classes can be constructed given a set of one or more actual types or values. The class template definition for Queue serves as a template for

the automatic generation of type-specific instances of Queue classes. For example, a Queue class for objects of type Shortint is created automatically from the generic class template definition when the programmer writes

var qshortint := Queue(Byte);

This generation of a class from the generic class template definition is called *class template instantiation*. When a Queue class for objects of type Byte is instantiated, each occurrence of the template parameter MyType within the class template definition is replaced with type Byte. Similarly, to create a Queue class for objects of type hexstring, the programmer writes

var qhexstring Queue(hexstring);

In this case, each occurrence of the generic type template parameter within the class template definition is replaced by the actual type parameter hexstring.

There is no special relationship between the instantiations of a class template for different types. Rather, each instantiation of a class template constitutes an independent class type. The template actual parameters must be specified in a comma-separated list and enclosed in parentheses. The name of a class template instantiation must always specify the template arguments explicitly.

Therefore, declaring variables and references to a class template instantiation do not cause the class template to be instantiated. For example, the following function foo() declares a variable and a reference to the class template instantiation Queue(Byte). However, these declarations do not cause the template Queue to be instantiated:

```
// Queue(Byte) is not instantiated for its uses in foo()
function foo (QByte Queue(Byte))
{           var rQByte Queue(Byte);
            // ...
}
```

A class template is therefore instantiated when an object is initialized with a type that is a class template instantiation. In the following example, the definition of the object QByte causes the template Queue(Byte) to be instantiated:

var QByte Queue(Byte) := Queue(Byte).Create; // Queue(Byte) is instantiated

The definition of the class Queue(Byte) becomes known to the compiler before its member method is called.

Depending on the types with which a class template is instantiated, some design considerations must be taken into account when defining a class template. For example,

```
template class QueueItem (MyType variant) {
            public constructor QueueItem(MyType);    // bad design choice
            // ...
}
```

This definition of the QueueItem constructor implements the pass-by-value argument semantics. This performs adequately when QueueItem is instantiated with a built-in type (as in the instantiation of QueueItem(Byte), for example). However, when QueueItem is instantiated with a large object type, the run-time impact of this choice is no longer acceptable. This is why the argument to the constructor is declared as a reference to an inout type:

public constructor QueueItem(inout MyType);

Concrete textual grammar

119 ClassTemplateInstance ::= ClassTemplateIdentifier ActualParList

21.1.3. Template arguments for non-generic type parameters

A class template parameter can be a non-generic type template parameter. There are some restrictions on the kind of template argument that can be used with such a non-generic type template parameter. The expression to which a non-generic type parameter is bound must be a constant expression. That is, it must be possible to evaluate it at compile-time. Constant expressions that evaluate to the same value are considered equivalent template arguments for a template non-generic type parameter.

21.1.4. Member methods of class templates

As with non-template classes, a member method of a class template can either be defined within the class template definition, or the member function can be inherited from an ancestor virtual class. All member methods defined within a class template are implicitly final, because it is impossible to override or to hide them. It is not required for the declarations of such methods to redundantly include the final keyword.

If a member class method of a class template is itself a method template (§13.6.1). ATDL requires that such a member method be instantiated only when the class template is itself instantiated.

A member instance method of a class template can be a method template or an ordinary instance method. A member instance method of a class template is not instantiated automatically when the class template is itself instantiated. The member instance method is instantiated only if it is used by the program, ATDL requires that such a member method be instantiated only when it is called.

Exactly when the member instance method of a class template is instantiated impacts how names are resolved in the definition of a class template member method (§21.1.8) and when a member method incarnation can be declared (§21.1.6).

21.1.5. Static members of class templates

Class templates can have static members, including static fields and class methods. Each of these class members exists once per enclosing class type, that is, independently of the number of objects of the enclosing class type and regardless of the number of instantiations of the generic type that may be used somewhere in the program. The name of the static member consists - as is usual for static members - of the scope (groups and enclosing class type) and the member's name. If the enclosing class type of a static member is generic, then the type in the scope qualification must be an ordinary or an instantiated generic-type, not a raw type or a parameterized type.

A class template can declare static fields. Each instantiation of the class template has its own set of static fields. A static field is instantiated from the class template definition only if it is used in a program. If a static field of a class template is itself a template. The template definition for the static field does not cause any memory to be allocated. Memory is only allocated for particular instantiations of the static field. Each static field instantiation corresponds to a class template instantiation. An instantiation of a static data member, then, is always referred to through a particular class template instantiation. For example,

```
// error: QueueItem is not an actual instantiation
var ival0 Smallint := QueueItem.QueueItem_chunk;
var ival1 char := QueueItem(char).QueueItem_chunk;             // ok
var ival2 Smallint := QueueItem(Smallint).QueueItem_chunk;   // ok
```

21.1.5.1. True constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue cardinal := 237;
```

declares a constant called *MaxValue* that returns the cardinal 237.

21.1.6. Class template incarnations

To see why our programs might need to define class template incarnations, let's add two new member functions to the class template Queue. The member functions min() and max () iterate through the items in the Queue to find the minimum value and the maximum value respectively.

```
template class Queue(MyType variant) {
        // ...
        public function min() return MyType;
        public function max() return MyType;
        //...
        }
```

```
// find minimum value in the Queue
```

```
template function Queue(MyType).min() return MyType;
{       if [!is_empty()]
        {
                var min_val MyType := front.item;
                for ( var pq QueueItem := front.next; pq != null; pq := pq.next )
                { if [ pq.item < min_val ] { min_val := pq.item }}
        }
        return min_val;
}
```

```
// find maximum value in the Queue
```

```
template function Queue(MyType).max() return MyType;
{       if [!is_empty()]
        {
                var max_val MyType := front.item;
                for ( var pq QueueItem := front.next; pq != null; pq := pq.next )
                { if [pq.item > max_val] { max_val := pq.item }}
        }
        return max_val;
}
```

Let's assume that we have the IEEE754double type with which we would like to instantiate the class template Queue. The following statement in the member function min() compares two items in the Queue:

```
pq.item < min_val
```

If operator<() is not defined for IEEE754double type, and an attempt is made to call min() on a Queue of items of this type, a compile-time error is issued at the point where the invalid comparison operator is used in min(). However, no operator (<) exists to compare two values of type IEEE754double, and the member functions min() and max() cannot be used with a Queue of type Queue(IEEE754double). (A similar problem exists with the member function max() and its use of operator(>).)

One solution to this problem is to define global operators (<) and (>) compare two values of type Queue(IEEE754double). However, to introduce class template incarnations, we consider another solution. We do not want the generic member function definitions for the class template Queue to be used to instantiate the member functions min() and max() if the template argument is the class type IEEE754double. Instead, we want to define instances for Queue(IEEE754double).min() and Queue(IEEE754double).max() that use the IEEE754double member function compareLess().

```
class IEEE754double {
        // ...
        public compareLess (min_val IEEE754double) return boolean;
        private value IEEE754double;
        }
```

We can do this by providing a specialized definition for a member of a class template instantiation using an *explicit incarnation definition*. An explicit incarnation definition is a definition in which the keyword template is followed by the definition of the incarnation for the class member. In the following example, explicit incarnations are defined for the member functions min() and max() of the class template instantiation Queue(IEEE754double):

```
// explicit incarnation definitions

template function Queue(IEEE754double).min() return IEEE754double;
{     if [!is_empty()]
        {
               var min_val IEEE754double := front.item;
               for ( var pq QueueItem := front.next; pq != null; pq := pq.next )
              { if [ pq.item.compareLess( min_val) ] { min_val := pq.item }}
        }
        return min_val;
}

template function Queue(IEEE754double).max() return IEEE754double;
{     if [!is_empty()]
        {
               var max_val IEEE754double := front.item;
               for ( var pq QueueItem := front.next; pq != null; pq := pq.next )
              { if [ max_val.compareLess(pq.item) ] { max_val := pq.item }}
        }
        return max_val;
}
```

Even though the class type Queue(IEEE754double) is instantiated from the generic class template definition, each object of type Queue(IEEE754double) uses the specializations for the member functions min() and max() — these member functions are not instantiated from the generic member function definitions for the class template Queue.

Because the explicit incarnation definitions for the member functions min() and max() are function definitions and not method template definitions, their headings must be placed within a scope unit (groups or enclosing class type), it is just to declare a function template explicit incarnation without defining it.

In some cases the entire class template definition may be inappropriate for use with a particular type. In this case the programmer can provide an incarnation to specialize the entire class template. For example, the programmer may choose to provide a complete definition of Queue(IEEE754double):

```
template class Queue(IEEE754double) {
        public constructor Queue(inout IEEE754double);
        public destructor Destroy();
        public function remove() return IEEE754double;
        public function append (inout IEEE754double);
        public function is_empty() return boolean;
        public min() return IEEE754double;
```

```
public max() return IEEE754double;
// Some particular implementation
// private ...
}
```

An explicit incarnation for a class template can be defined only after the general class template has been declared. That is, the name must be known to be a class template name before the template can be specialized. If we define a class template incarnation, we must also define each member method or static data member associated with this incarnation. The generic member definitions of the class template are never used to create the definitions for the members of an explicit incarnation. This is because the class template incarnation may have a completely different set of class members from the generic template. If we decide to provide an explicit incarnation definition for the class type Queue(IEEE754double), not only must we provide the definitions for the member functions min() and max(), but we must also provide the definitions for all of the other member functions as well.

21.1.7. Class template partial incarnations

If a class template has more than one template parameter, one might want to specialize the class template for a particular template argument or a set of template arguments without specializing the template for every template parameter. That is, one might want to provide a template that matches a generic template except that some of the template parameters have been replaced by actual types or values. This is possible using a class template *partial incarnation*. A class template partial incarnation might be needed to define a more appropriate or efficient implementation than the generic template definition for a particular set of template arguments.

A class template partial incarnation is a template, and the definition of a partial incarnation looks like a template definition. Such a definition begins with the keyword template followed by a template parameter list enclosed in parentheses. The parameter list of a class template partial incarnation differs from the parameter list of the corresponding generic class template definition.

21.1.8. Name resolution in class templates

In the discussion on name resolution in function templates in Section 16.5.3, we mention that this resolution proceeds in two steps. The same two steps apply to the resolution of names used in class template definitions and in the definition of their members. Each step applies to different kinds of names: the first step applies to names that have the same meaning in all instantiations of the class template, and the second step applies to names that have potentially different meanings from one template instantiation to another.

21.1.9. Groups and class templates

As with any other global scope definitions, a class template definition can be placed in a group. The meaning of such a class template definition is the same as when the class template is defined in global scope, except that the name of the class template is hidden within the group. The class template name must either be qualified by the group name when the class template is used outside its group, or an import declaration must be provided.

21.2. Threads and operations

Thread operations are used to set up and control test threads. These operations shall only be used in ATDL test cases and functions (i.e., not in the module control part).

Concrete textual grammar

247 ConfigurationStatement ::= DoneStatement | StartThreadStatement | StopThreadStatement

248 ConfigurationOps ::= CreateOp | ComponentIdExpression | ThreadRunningOp

254 ThreadId ::= ThreadIdentifier | (“any” | “all”) “thread”

258 ComponentRef ::= ComponentIdentifier | ComponentIdExpression | ClassInstance

Concrete graphical grammar

259 <configuration statement area> ::= <create request area> | <fgr done area> |
<start thread area> | <stop thread area> | <stop symbol>

Table 26: Overview of ATDL thread operations

Thread operations	Operation Name	Associated graphical symbol
Create parallel thread object	**create**	<create request symbol>
Get MTC address	**mtc**	<task symbol>
Get test system interface address	**system**	<task symbol>
Get own address	**self**	<task symbol>
Start execution of test thread	**start**	<procedure call symbol>
Stop execution of test thread	**stop**	<stop symbol>
Check termination of a PTC	**running**	<condition symbol>
Wait for termination of a PTC	**done**	<condition symbol>

21.2.1. Defining thread classes

ATDL provides several objects that make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Concrete textual grammar

120 ThreadClassDef ::= ThreadClassHeading “{“
[SupportingDefSpec]
[{ClassFieldSpec [SemiColon]}*]
[InterfaceDefSpec]
[ImplementedInterfaceList]
[RequiredInterfaceList]
[ClassMethodList] “}”

121 ThreadClassHeading ::= “thread” ThreadClassIdentifier [ThreadClassHeritage]

Concrete graphical grammar

128 <thread class diagram> ::= <thread class symbol> **contains**
(ThreadClassIdentifier <class properties area> <class methods area>)
[**is_connected_to** <component extends area>]
[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

An thread class is shown with a heavy border.

21.2.1.1. Declaring thread-local variables and timers

It is possible to declare constants, variables and timers local to a particular component.

Thread-local variables and timers are associated with the thread instance and follow the scope rules defined in clause 5.4. Each new instance of a thread will thus have its own set of variables and timers as specified in the thread definition (including any initial values, if stated).

You may declare variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Thread-local variables are associated with the thread instance and follow the scope rules defined

in clause 5.4. Each new instance of a thread will thus have its own set of variables as specified in the thread type definition (including any initial values, if stated). For example,

thread MyPTCType { **var** MyLocalInteger Smallint;
timer MyLocalTimer;
...; }

declares an integer type variable that is private to each thread in the application, but global within each thread.

21.2.1.2. Initializing the thread

If you want to write initialization code for your new thread class, you must override the **create** method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.

21.2.1.3. Active object model

The active object model views a thread as an active entity which communicates with other threads by sending signals. In this model, a thread is composed from instance data, a communication module, an incoming signal queue and its own execution context. Each signal arriving at the thread is, at first, spooled in the signal queue by the communication module. Each thread also has the ability to select a signal from the queue to be executed next. If the thread decides not to handle a signal at this moment, the signal can be pushed back into the signal queue. Since each active object communicates with others only by sending signals, they can run concurrently. ATDL has been implemented using this model.

21.2.1.4. Thread object

A thread object does not run within another thread, or stack frame. It has an independent locus of control within the overall execution of a system.

Concrete textual grammar

123 ThreadClassIdentifier ::= Identifier

21.2.1.5. Thread references

Thread references are unique references to the test threads created during the execution of a test case. This unique thread reference is generated by the test system at the time when a thread is created, i.e., a thread reference is the result of a **create** operation. In addition thread references are returned by the predefined functions **system** (returns the thread reference to identify the test system interface), **mtc** (returns the thread reference of the MTC), **self** (returns the thread reference of the component in which **self** is called), and **sender** (returns the component reference from which the last input signal has been consumed).

In addition, the special value **null** is available to indicate an undefined thread reference, e.g., for the initialization of variables to handle thread references.

The only operations allowed on thread references are assignment and equivalence.

Concrete textual grammar

249 ComponentIdExpression ::= “system” | “self” | “mtc” | “sender” | “inherited”

Example:

Reference: TTCN3 [1] clause 8.6

// A thread type definition

thread MyCompType {**requires** MyMessagePort1, MyMessagePort2; ...}

```
// Declaring two variables for the handling of references to threads of type MyCompType
// and creating a thread of this type
var MyPCO1 MyMessagePort1;
var MyPCO2 MyMessagePort2;
var MyCompInst MyCompType := MyCompType.create;

// Usage of thread references in configuration operations
// always referring to the thread created above
MyPCO1 := bind(MyMessagePort1, MyCompInst);      // Obtaining the interface reference
MyPCO2 := bind(MyMessagePort2, system);
MyCompInst.start(MyBehavior(self));         // self is passed as a parameter to MyBehavior
MyPCO1.receive;        // receive from MyCompInst
:
MyPCO2.receive(MyIntegerMessage1:*);   // receive from SUT
:
MyPCO2.send(MyIntegerMessage1:5);      // send to SUT
:
var MyMessage M1, MyResult M1;
var MyInst1 MyCompType1 := null;
var MyInst2 MyCompType2 := null;
var MyInst3 MyCompType3:= null;
:
alt {
        [] MyMessagePort1(MyCompType1).receive(M1:*)      // interface typecast
                        -> (MyMessage:=value; MyInst1:=sender) { }
        [] MyMessagePort1(MyCompType2).receive(M1:*)      // interface typecast
                        -> (MyMessage:=value; MyInst2:=sender) { }
        [] MyMessagePort1(MyCompType3).receive(M1:*)
                        -> (MyMessage:=value; MyInst3:=sender) { }
}
:
MyResult := MyMessageHandling(MyMessage); // some result is retrieved from a function
:
if (MyInst1 != null) {MyPCO1(MyInst1).send(MyResult)};
if (MyInst2 != null) {MyPCO1(MyInst2).send(MyResult)};
if (MyInst3 != null) {MyPCO1(MyInst3).send(MyResult)};
:
```

21.2.2. The Priority field (informative)

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the *Priority* field.

For example, if writing a Windows application, *Priority* values fall along a seven-point scale.

Value	Priority
tpIdle	The thread executes only when the system is idle.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.

tpHighest The thread's priority is two points above normal.

tpTimeCritical The thread gets highest priority.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application’s performance:

```
constructor TMyThread.Create(CreateSuspended boolean);
{
        inherited Create(CreateSuspended);
        Priority := tpLower;        // set the priority to lower than normal
        :                           // The thread's priority is one point below normal.
}
```

21.2.3. The Running operation

The **running** operation allows behaviour executing on a test component to ascertain whether behaviour running on a different thread has completed. The **running** operation can be used for PTCs only. The **running** operation is considered to be a **boolean** expression and, thus, returns a **boolean** value to indicate whether the specified thread (or all threads) has terminated. In contrast to the **done** operation, the **running** operation can be used freely in boolean expressions.

Concrete textual grammar

255 ThreadRunningOp ::= ThreadId Dot “running”

Concrete graphical grammar

264 <fgr thread running area> ::= <condition symbol> **contains** ThreadRunningOp

21.2.4. The Start thread method

Once a thread has been created and connected behaviour has to be bound to this thread and the execution of its behaviour has to be started. This is done by using the **start** operation (Thread creation does not start execution of the thread behaviour). The reason for the distinction between Instance Creation and **start** is to allow connection operations to be done before actually running the thread component.

The **start** operation shall bind the required behaviour to the thread component. This behaviour is defined by reference to an already defined function.

The following restrictions apply to a function invoked in a **start** thread operation:

1) If this function has parameters they shall only be **in** parameters, i.e., value parameters.

2) Channels and timers can only be passed into this function if they refer to channels and timers in the thread class definition of the newly created thread object, i.e., channels and timers are local to component instances and shall not be passed to other components.

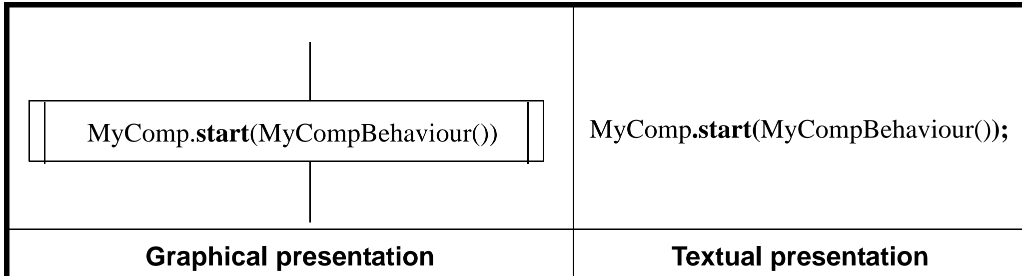
Concrete textual grammar

256 StartThreadStatement ::= ThreadIdentifier Dot “start” “(“ FunctionInstance “)”

Concrete graphical grammar

262 <start thread area> ::= <procedure call symbol> **contains** StartThreadStatement

Figure 37. **Start thread class method**



Start thread message sending. The sending of a start thread message may be shown as a <procedure call symbol>. The <procedure call symbol> contains the **start** statement (Figure 37). A <flow line symbol> is drawn from the previous node to the <procedure call symbol>, and another <flow line symbol> is drawn from the <procedure call symbol> to the next node.

21.2.5. The Stop thread method

You can request that a thread end execution prematurely by calling the **stop** method. **Stop** method sets the **done** property of the thread to **true**. The operation has no arguments. For example:

if [date == "1.1.2000"] { MyObject.**stop**; } // execution stops on the 1.1.2000

If the test component that is stopped is the MTC all remaining PTCs that are still running shall also be stopped and the test case terminates.

All resources shall be released when a thread object terminates, either explicitly using the **stop** operation or through reaching a **return** statement in the function that originally started the thread object or implicitly when the thread object reaches the end of its behavior tree. Any variables storing a stopped thread object reference shall refer to nothing.

Concrete textual grammar

257 StopThreadStatement ::= ThreadIdentifier Dot “stop” | “all” “thread” Dot “stop”

Figure 38. **Stop test component operation**

ComponentId	ComponentId**.stop;**	**all**
(a) Graphical presentation	(b) Textual presentation	(c) Stopping all PTCs

Concrete graphical grammar

263 <stop thread area> ::= <stop symbol> [**is_associated_with** (ComponentRef | “all”)]

The stop thread operation shall be represented by a <stop symbol>, which is attached to the <flow line symbol>, which performs the stop thread operation. A <flow line symbol> is drawn from the previous node to the <stop symbol>, and another <flow line symbol> is drawn from the <stop symbol> to the next node. It shall have an associated expression that identifies the component to be stopped (see Figure 38 (a)). The MTC may stop all PTCs in one step by using the stop component operation with the keyword **all** (see Figure 38 (c)).

21.2.5.1. The FreeOnTerminate field (informative)

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the *FreeOnTerminate* property to **true**.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to **false** and then explicitly freeing the first thread from the second.

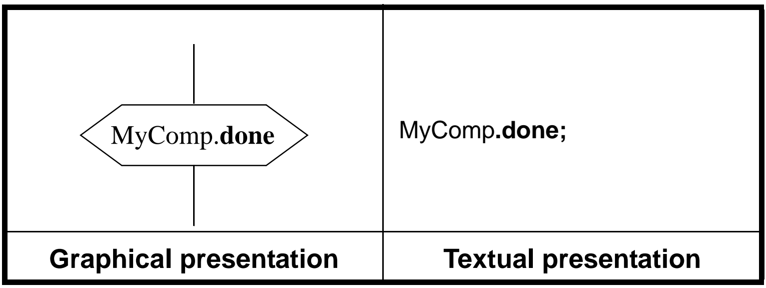
21.2.6. The Done operation

The **done** operation allows behaviour executing on a test component to ascertain whether the behaviour running on a different test thread has completed. The **done** operation can be used for PTCs only.

Concrete textual grammar

253 DoneStatement ::= ThreadId Dot "done"

Figure 39. Done operation



Concrete graphical grammar

261 <fgr done area> ::= <condition symbol> **contains** DoneStatement

The **done** operation shall be represented within a <condition symbol>. The <condition symbol> contains the **done** statement (Figure 39).

21.2.7. The MTC, System , Sender and Self operations

The thread reference (see clause 21.2.1.5) has three operations, **mtc, sender** and **system** which return the reference of the master thread, the sender component and the test system interface respectively. In addition, the operation **self** can be used to return the reference of the thread in which it is called.

The **sender** method is supported for backward compatibility only. Its use is not recommended.

22. Communication operations

ATDL supports message-based (asynchronous) and operational (synchronous) communication.

Concrete textual grammar

266 CommunicationStatement ::= SendStatement | CallStatement | RaiseStatement |
ReceiveStatement | TriggerStatement |
SynchronizeStatement | CatchStatement

Concrete graphical grammar

279 <communication statement area> ::= <fgr call area> | <fgr send area> | <fgr raise area> |
<fgr receive area> | <fgr trigger area> |
<fgr synchronize area> | <fgr catch area>

Table 27: Overview of ATDL communication operations

Communication operation	keyword	Associated graphical symbol
Message-based communication		
Send message	**send**	<message out symbol>
Receive message	**receive**	<message in symbol>
Trigger on message	**trigger**	<message in symbol>
Procedure-based communication		
Invoke procedure call	**call**	<procedure call symbol>
Synchronize a procedure call	**synchronize**	<procedure in symbol>
Channel controlling operations		
Bind client-side channel to component	**bind**	<task symbol>
Obtain an interface reference	typecast	<task symbol>
Release channel to component	**release**	<task symbol>
Give access to channel	**start**	<condition symbol>
Clear channel	**clear**	<condition symbol>
Stop access at channel	**stop**	<condition symbol>

22.1. Connection Points

A client uses the **bind** method to discover the functionality supported by an object. The more interfaces that are common to the client and the object, the more intertwined their relationship is. But regardless of the number of interfaces the object supports, the basic model remains the same: the client calls the methods implemented by the object, the object performs the desired service, and then the object returns the results to the client. This type of relationship is rather one-sided the client always makes requests of the object. ATDL supports connection points, a technology that enables an object to "talk back" to its client.

Objects that support connection points are often called connectable objects. Connection points involve some rather peculiar terminology, which deserves a moment of attention. A *source interface*, also called an outgoing interface, is an interface defined in the object but implemented by the client. A *sink object*, which resides within the client, is the object that implements the object's source interface. Knowing these definitions will help you understand connection points.

22.1.1. Simple connectable object

A connection point is an object managed by the server-side connectable object that implements the Connection Point interface. The ATDL underlying system requires two important methods of the Connection Point interface: *Advise* and *Unadvise*. The main purpose of this interface is to let a client provide a connectable object with a pointer to the client's sink. Each connection point supports exactly one *source interface*.

When a server side connectable object calls the interface typecast, the client calls the *Advise* method to provide the server side connectable object with a pointer to the client's sink object. In a way, you can consider this call a **bind** method in reverse. A client uses the **bind** method to discover the interfaces exposed by a server object. It calls the *Advise* method to provide a server object with a pointer to its sink.

Although the *Advise* method provides the server side connectable object with a pointer to the client sink's interface, this pointer alone is not sufficient. To call the methods of the client's sink object, we must obtain a pointer to one of the more interesting interfaces implemented by the sink. Thus, the **bind** method is the first call made implicitly from the connectable object to the client's sink! Then the *Advise* method returns a unique number to the client that identifies the

advisory relationship that has been established. The client retains this number, called a *cookie*, for later use in terminating the connection.

Since this code is prepared to handle only a single connection, the *cookie* returned to the client is a dummy placeholder. The connectable object then calls the sink object's **bind** method implicitly to obtain a pointer to its Outgoing interface. This pointer is stored in a global variable for later use. The implicit **bind** call might seem redundant. After all, since each connection point supports one source interface, why doesn't the client simply provide that interface pointer in the *Advise* call in place of the implicit **bind** method? The answer is that connection points are designed to be a general-purpose mechanism for configuring bi-directional communication in ATDL. Because the designers of the connection point interfaces had no way of knowing what custom interfaces a sink object might implement, the implicit **bind** method was the only option.

The **release** method on the server-side calls the client side *Unadvise* method to terminate an advisory relationship that was previously established using the *Advise* method. The cookie argument passed to release identifies which connection should be terminated.

22.2. Interface references

A channel is an individual connection among two or more component instances. It is a tuple (ordered list) of component references. The component instances must be direct or indirect instances of the components at corresponding positions in the association contract.

Concrete textual grammar

265 Channel ::= ChannelIdentifier | VarIdentifier | InterfaceParIdentifier | CastExpression

273 ChannelOrAny ::= Channel | “any” “interface”

274 ChannelOrAll ::= Channel | “all” “interface”

Graphical notation

A binary channel is shown as a path between two component instances i.e., one or more connected line segments or arcs (see component instance diagram).

As a convenience, the arrowheads may be omitted on channels that are navigable in both directions. In theory, this can be confused with a channel that is not navigable in either direction, but such a channel is unlikely in practice and can be explicitly noted if it occurs.

22.2.1. Interface typecast

An interface typecast shall return the unique interface reference. The unique reference to the interface will typically be stored in the variable and can be used for communication purposes such as sending and receiving. Interface types follow the same rules as class types in variable and value typecasts. You can declare a variable of an interface type.

Class-type expressions can be cast to interface types — for example, IMyInterface(SomeObject) — provided the class implements the interface.

On the client side, an interface typecast is a shorthand notation for a **bind** operation. On the server side, an interface typecast dynamically queries a given client-side object and returns an interface reference to the object.

22.3. General format of communication operations

22.3.1. General format of the sending operations

Sending operations consist of a *send* part and, in the case of a blocking procedure-based **call** operation, a *response* and *exception handling* part.

The send part:

- specifies the channel at which the specified operation shall take place;
- defines the value of the information to be transmitted.

The channel name, operation name and value shall be present in all sending operations.

Response and exception handling is only needed in cases of procedure-based communication. The response and exception handling part of the **call** operation is optional and is required for cases where the called procedure returns a value or has **out** or **inout** parameters whose values are needed within the calling component and for cases where the called procedure may raise exceptions which need to be handled by the calling component.

The returned value and exception handling part of the call operation makes use of assignment and **catch** operations to provide the required functionality.

22.3.2. General format of the receiving operations

A receiving operation consists of a *receive* part and an (optional) *assignment* part.

The receive part:

a) specifies the channel at which the operation shall take place;

b) defines a matching part which specifies the acceptable input which will match the statement.

The channel name, operation name and value part of all receiving operations shall be present.

The (optional) assignment part in a receiving operation is optional. For message-based channels it is used when it is required to store received messages. In the case of procedure-based channels it is used for storing the **in** and **inout** parameters of an accepted call or for storing exceptions. For the assignment part strong typing is required, e.g. the variable used for storing a message shall have the same type as the incoming message.

22.3.2.1. Value assignment

On a receiving event assignments are performed after the event occurs. If the match is successful, the value removed from the channel queue can be stored in a variable. This is denoted by the keyword **value**. For example,

```
MyPort.receive(MyType:?) -> (MyVar := value);    // The value of the received message is
                                                  // assigned to MyVar.
```

Concrete textual grammar

466 AssignmentList ::= -> "(" Assignment {SemiColon Assignment}* ")"

Concrete graphical grammar

284 <save area> ::= <save symbol> **contains** (Assignment {SemiColon Assignment}*)

22.4. Message-based communication

Message-based communication is communication based on an asynchronous message exchange. Message-based communication is non-blocking on the **send** operation, as illustrated in Figure 40, where processing in the SENDER continues immediately after the **send** operation occurs. The RECEIVER is blocked on the **receive** operation until it processes the received message.

Figure 40. Illustration of the asynchronous send and receive

22.4.1. The Send operation

The **send** operation is used to place a value on an outgoing message channel queue. The value may be specified by referencing a template, a variable, or a constant or can be defined in-line from an expression (which of course can be an explicit value). When defining the value in-line the optional message identifier field shall be used if there is ambiguity of the type of the value being sent.

The **send** operation shall only be used on message-based channels and the type of the value to be sent shall be in the list of outgoing types of the interface type definition.. For example:

MyChannel.**send**(MyTemplate(5,MyVar));
// Sends the template MyTemplate with the actual parameters 5 and MyVar via MyChannel.

Concrete textual grammar

267 SendStatement ::= Channel Dot "send" "(" TemplateInstance ")"

Concrete graphical grammar

280 <fgr send area> ::= <message out symbol> **contains** ([Channel Dot] TemplateInstance)

Message sending. The sending of a message may be shown as a convex pentagon (Figure 41). The template instance of the message is shown inside the symbol. A <flow line symbol> is drawn from the previous node to the pentagon, and another <flow line symbol> is drawn from the pentagon to the next node.

Figure 41. The Send statement

MyChannel.MyTemplate(5, MyVar)	MyChannel.**send**(MyTemplate(5,MyVar));
Graphical presentation	**Textual presentation**

22.4.2. The Receive operation

The **receive** operation is used to receive a value from an incoming message channel queue. The value may be specified by referencing a template, a variable, or a constant or can be defined in-line from an expression (which of course can be an explicit value). When defining the value in-line the optional type field shall be used to avoid any ambiguity of the type of the value being received. The **receive** operation shall only be used on message-based channels and the type of the value to be received shall be included in the list of incoming types of the interface definition.

The **receive** operation removes the top message from the associated incoming channel queue if, and only if, that top message satisfies all the matching criteria associated with the **receive** operation. No binding of the incoming values to the terms of the expression or to the template shall occur.

If the match is not successful, the top message shall not be removed from the port queue i.e. if the **receive** operation is used as an alternative of an **alt** statement and it is not successful the execution of the test case shall continue with the next alternative of the **alt** statement.

An optional message identifier field in the matching criteria to the **receive** operation shall be used to avoid any ambiguity of the type of the value being received.

NOTE 1: Encoding attributes are also participating in matching in an implicit way, by preventing the decoder to produce an abstract value from the received message encoded in a different way than specified by the attributes.

Concrete textual grammar

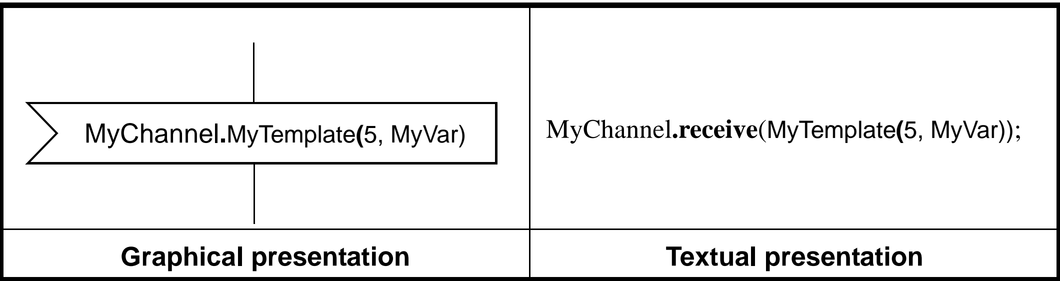
271 ReceiveStatement ::= ChannelOrAny Dot "receive" ReceiveParameter [AssignmentList]

272 ReceiveParameter ::= ["(" TemplateInstance ")"]

Concrete graphical grammar

283 <fgr receive area> ::= <message in symbol> **contains** ([ChannelOrAny Dot] ReceiveParameter)
[**is_associated_with** <save area>]

Figure 42. The Receive statement



Message receipt. The receipt of a message may be shown as a concave pentagon <message in symbol>. The template instance of the message is shown inside the <message in symbol>. A <flow line symbol> is drawn from the previous node to the pentagon, and another <flow line symbol> is drawn from the pentagon to the next node (Figure 42).

The (optional) assignment part (denoted by the **'->'**) shall be placed within a <save symbol>.

22.4.3. The Trigger operation

The **trigger** operation filters messages with certain matching criteria from a stream of received messages on a given incoming channel. The **trigger** operation shall only be used on message-based channels and the type of the value to be received shall be included in the list of incoming types of the interface definition. All messages that do not fulfill the matching criteria shall be removed from the queue without any further action i.e., the trigger operation waits for the next message on that queue. If a message meets the matching criteria, the **trigger** operation behaves in the same manner as a **receive** operation.

The **trigger** operation can be used as a stand-alone statement in a behaviour description. In this latter case the **trigger** operation is considered to be shorthand for an **alt** statement with only one alternative, i.e. it has blocking semantics, and therefore provides the ability of waiting for the next message matching the specified template or value on that queue.

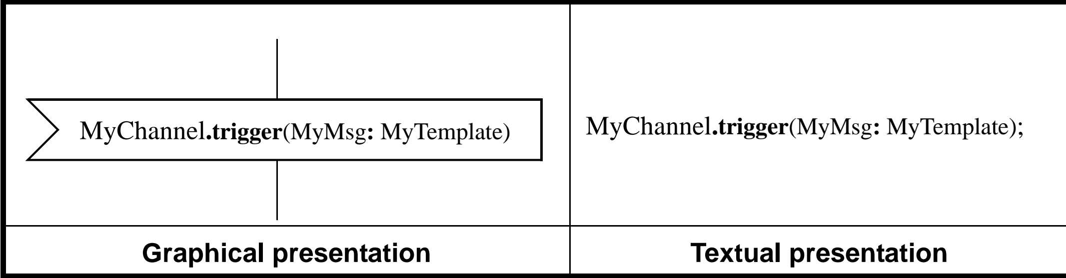
Concrete textual grammar

275 TriggerStatement ::= ChannelOrAny Dot "trigger" ReceiveParameter [AssignmentList]

Concrete graphical grammar

285 <fgr trigger area> ::= <message in symbol> **contains** TriggerStatement
[**is_associated_with** <save area>]

Figure 43. The Trigger statement



The **trigger** operation shall be represented within the <message in symbol>. The <message in symbol> contains the **trigger** statement (Figure 43).

22.5. Operation templates

Instances of operation parameter lists with actual values may be specified using templates. Templates may be defined for any operation by referencing the associated operation definition.

For example,

```
	// signature definition for a remote procedure

operation RemoteProc(in Par1 Word, out Par2 Smallint, inout Par3 Smallint) return Smallint;

// example templates associated to defined procedure signature

	template Template2 RemoteProc:=
	{
			Par1 := 1,
			Par2 := ?,
			Par3 := 3
	}

	template Template3 RemoteProc:=
	{
			Par1 := 1,
			Par2 := ?,
			Par3 := ?
	}
```

22.5.1. Templates for invoking procedures

An operation template used in a **call** operation defines a complete set of default parameter values for all **in** and **inout** parameters. Default parameter values are limited to values that can be specified by a constant expression. At the time of the **call** operation all **in** and **inout** parameters in the template shall resolve to actual values, no matching mechanisms shall be used in these fields, either directly or indirectly. Any template specification for **out** parameters is simply ignored, therefore it is allowed to specify matching mechanisms for these fields, or to omit them.

For example,

```
// Given the examples in the introduction of clause (§22.5)

// Valid invocation since all in and inout parameters have a distinct value

	MyChannel.call Template2(MyVar2, MyVar3);

// Invalid invocation because the inout parameter Par3 has a matching attribute not a value
```

MyChannel.**call** Template3(MyVar2, MyVar3);

// Templates never return values. In the case of Par2 and Par3 the values returned by the

// call operation must be retrieved using assignable expressions

You can make the remote procedure call using the operation template's declared name, in this case you must omit all the default **in** parameters when passing them to a remote procedure.

When you call a remote procedure that uses default parameter values, all actual **in** parameters must also use the default values. Expressions used to pass **inout** and **out** parameters must be identically typed with the corresponding formal parameters, and only assignable expressions can be used to pass **inout** and **out** parameters.

The precise definition is as follows. Suppose that an operation has *m* parameters and *n* **in** parameters, then your call to an operation that uses a template must pass the remaining *m-n* **inout** and **out** parameters that correspond in order and type to the operation's parameter list.

Default values for **inout** parameters specified in a template *override* those specified individually. Thus, given the declarations in the introduction of clause (§22.5)

var MyVar3 Smallint := 5;

:

MyChannel.**call** Template2(MyVar2, MyVar3);

result in the values (1, ?, 3) being passed to *RemoteProc*.

22.5.2. Templates for accepting operation invocations

A template used in a **synchronize** operation defines a data template against which the incoming parameter fields are matched. Matching mechanisms, as defined in clause 15.7, may be used in any templates used by this operation. No binding of incoming values to the template shall occur. Any **out** parameters shall be ignored in the matching process. For example,

// Given the examples in the introduction of clause (§22.5)

// Valid synchronize call, it will match if Par1 == 1 and Par3 == 3
MyChannel.**synchronize** Template2;
// Valid synchronize call, it will match on Par1 == 1 and Any value of Par3
MyChannel.**synchronize** Template3;

22.5.3. In-line assignments for invoking operations

The in-line assignments used in a **call** method define a complete set of field values for all **in** and **inout** parameters. At the time of the **call** operation all **in** and **inout** parameters in the assignments shall resolve to actual values, no matching mechanisms shall be used in these fields, either directly or indirectly. Any in-line assignments specification for **out** parameters is simply ignored, therefore it is allowed to specify matching mechanisms for these fields, or to omit them.

For example,

// Given the examples in the introduction of clause (§22.5)

MyChannel.**call** RemoteProc(1, MyVar1:=?, MyVar2:=3)

// inline assignments for the call of RemoteProc

Default values for **inout** parameters specified in-line *hide* those specified in a template. For example, given the declarations in the introduction of clause (§22.5)

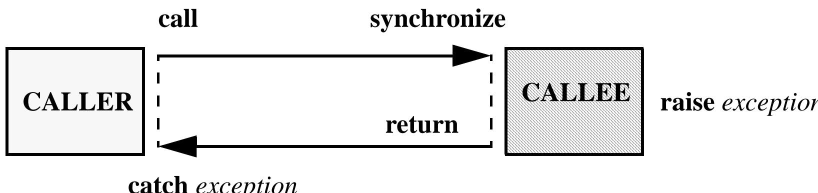
MyChannel.**call** Template2(MyVar2, MyVar3:=5);

result in the values (1,?,5) being passed to *RemoteProc*.

22.6. Procedure-based communication

The principle of procedure-based communication is to call procedures in remote entities. ATDL supports *blocking* and *non-blocking* procedure-based communication. Blocking procedure-based communication is blocking on the calling and the called side, whereas non-blocking procedure-based communication only is blocking on the called side.

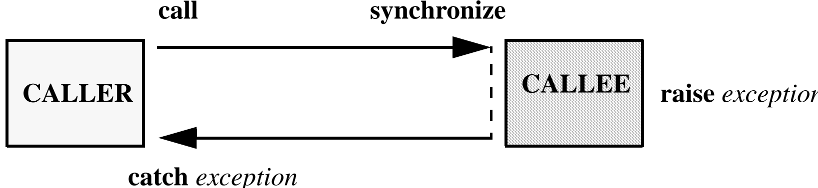
Figure 44. Illustration of blocking procedure-based communication



The communication scheme of blocking procedure-based communication is shown in Figure 44. The CALLER calls a remote procedure in the CALLEE by using the **call** operation. The CALLEE accepts the call by means of a **synchronize** operation and reacts by either using a **return** statement to answer the call or by raising (**raise** operation) an exception. The CALLER handles reply or exception by using assignable expressions or **catch** operations. In Figure 44, the blocking of CALLER and CALLEE is indicated by means of dashed lines.

The communication scheme of non-blocking procedure-based communication is shown in Figure 45. The CALLER calls a remote procedure in the CALLEE by using the **call** operation and continues its execution, i.e. does not wait for a reply or exception. The CALLEE accepts the call by means of a **synchronize** operation and executes the requested procedure. If the execution is not successful, the CALLEE may raise an exception to inform the CALLER. The CALLER may handle the exception by using a **catch** operation in an **alt** statement. In Figure 45, the blocking of the CALLEE until the end of the call handling and possible raise of an exception is indicated by means of a dashed line.

Figure 45. Illustration of non-blocking procedure-based communication



22.6.1. The Call operation

The **call** operation is used to specify that a test component calls a procedure in the SUT or in another test component. The **call** operation shall only be used on procedure-based interfaces. The type definition of the interface at which the call operation takes place shall include the operation name in its signature list i.e. it must be allowed to call this procedure at this interface.

The **in** parameters to be transmitted in the send part of the **call** operation can be defined in the form of an operation template and be passed as actual parameters. The **inout** parameters to be transmitted in the send part of the **call** operation is a signature that may either be defined in the form of an operation template or be assigned in-line. All **in** and **inout** parameters of the signature shall have a specific value i.e. the use of matching mechanisms such as *AnyValue* is not allowed.

The signature arguments of the **call** operation can be used to retrieve variable values for **out** and **inout** parameters. The actual assignment of the **out** and **inout** parameter values to variables shall

implicitly be made in the **call** operation. The actual assignment of the procedure return value shall be made by means of an assignment statement. For example:

```
// Given …

operation MyProc (out MyPar1 Smallint, inout MyPar2 boolean);

// a call of MyProc

MyChannel.call MyProc(MyVar1, MyVar2);

// Assignable expressions are used to pass inout and out parameters.

// Calls the remote procedure MyProc at MyChannel with the out and inout parameters
```

Operations that do not return values can be invoked directly. Operations that return values may be invoked inside expressions. For example:

```
MyVar := MyChannel.call MyProc(MyVar1,MyVar2);
        // The value returned by MyProc is assigned to MyVar.
        // The types of the returned value and MyVar have to be the same
```

In general, a **call** operation is assumed to have blocking-semantics. However, ATDL also supports non-blocking calls. A call, which has no return values, is assumed to be a non-blocking call. Exceptions raised by a call without return values shall be caught within a **try** statement.

Example

The handling of exceptions to a call is done by means of the **catch** operation within a **try** statement. This operation defines the alternative behavior depending on the exception (if any) that has been generated as a result of the **call** operation. For example:

```
exception MyException {Exception1 Type1, Exception2 Type2, Exception3 Type3}
operation MyProc3 (out MyPar1 Smallint, inout MyPar2 Smallint) return MyResultType
        raises (MyException);
:
// The following call operation shows the result return and exception handling mechanism of the
// call operation
try(30E-3) { MyResult := MyChannel.call MyProc3 (MyPar1Var,MyPar2Var:=5)
        }
    MyException.catch(Exception1: *)
    {       // catch an exception
        setverdict(fail);       // set the verdict and
        stop                    // stop as result of the exception
    }
    MyException.catch(Exception2: *)        // catch a second exception
        {setverdict(inconc);                // set the verdict and continue after
          }                                 // the call as result of the second exception
    [MyCondition] MyException.catch(Exception3: *) {...}
                // catch a third exception which
                // may occur if MyCondition evalutates to true
    MyChannel.catch(timeout) {...}          // timeout exception i.e., the called party
                                            // does not react in time, nothing is done
```

Concrete textual grammar

268 CallStatement ::= Channel Dot "call" OperationRefWithPara

/* STATIC SEMANTICS - only *out* parameters may be omitted or specified with a matching attribute */

Concrete graphical grammar

281 <fgr call area> ::= <procedure call symbol> **contains** CallStatement

[**is_associated_with** <save area>]

Call sending. The sending of a call may be shown as a <procedure call symbol>. The signature of the call is shown inside the symbol. A <flow line symbol> is drawn from the previous node to the <procedure call symbol>, and another <flow line symbol> is drawn from the <procedure call symbol> to the next node.

22.6.1.1. Calling non-blocking operations

A non-blocking procedure has no out and inout parameters, no return value and the non-blocking property is indicated in the corresponding signature definition by means of a **noblock** keyword. When a client invokes an operation with the **noblock** attribute, the invocation semantics are best-effort, which does not guarantee delivery of the call; best-effort implies that the operation will be invoked at most once.

Possible exceptions raised by non-blocking procedures have to be removed from the channel queue by using **catch** operations in subsequent **alt** statements.

22.6.2. Determining the method

Determining the method that will be invoked by a remote procedure call involves several steps. The first step in processing a remote procedure call at compile time is to figure out the name of the method to be invoked and which implementation class of the operational interface to check for definitions of methods of that name.

The second step searches the implementation class of the interface determined in the previous step for method declarations. This step uses the name of the method and the types of the actual parameter expressions to locate method declarations that are both *applicable* and *accessible*, that is, declarations that can be correctly invoked on the given actual parameters. There may be more than one such method declaration, in which case the *most specific* one is chosen(§20.6.4).

22.6.3. The Synchronize operation

ATDL may describe control flows of not only via asynchronous messages, but also by means of calls and returns. A control stimulus will cause a procedure to be invoked, raise an exception, or cause an instance to be created or destroyed.

An activation represents the period during which an object performs an operation either directly or through a subordinate operation. It models both the duration of the execution in time and the control relationship between the execution and its callers.

Control flow comprises procedure-based (synchronous) communication mechanisms defined by means of calls and replies.

Semantics

The **synchronize** event is a way to deliver requests from a client to an object implementation. The **synchronize** event is a way of implementing an operation that is an alternative to the execution of a procedure. The **synchronize** event is a shorthand notation for a pair of TTCN-3 **getcall** operation and **reply** operation. The **synchronize** operation is used to specify that a test component accepts a call from the SUT, or another test component.

ATDL allows the specification of test components which may be executed concurrently, test components may reside in a single machine or be distributed over several machines. A client does not need to know where an object resides, it simply makes a call to an object's interface. The **synchronize** operation performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network.

The different types of servers are known as: 1) **In-process server:** A thread running in the *same process space* as the client. 2) **Out-of-process server** (or local server): Another thread running in

a *different process space* but on the *same machine* as the client. 3) **Remote server:** A thread or another application running on a *different machine* from that of the client.

For in-process servers, pointers to the object interfaces are in the same process space as the client, so the **synchronize** operation makes direct calls into the object implementation.

As with other ATDL receiving operations matching mechanisms are allowed in the **synchronize** operation for run-time type checking. For example:

```
MyChannel.synchronize Template3;
```

// Will accept a call of RemoteProc at MyChannel with Par1 == 1 and Any value of Par3

To **synchronize** on any channel is denoted by the any keyword. For example:

```
any interface.synchronize MyProc;
```

Concrete textual grammar

276 SynchronizeStatement ::= ChannelOrAny Dot “synchronize” [“(“ TemplateInstance “)”]

Concrete graphical grammar

286 <fgr synchronize area> ::= <procedure in symbol> **contains** SynchronizeStatement
[**is_associated_with** <save area>]

The event of receiving a call for a procedure, the event may be shown as a <procedure in symbol>. The signature of the call is shown inside the symbol. A <flow line symbol> is drawn from the previous node to the <procedure in symbol>, and another <flow line symbol> is drawn from the <procedure in symbol> to the next node.

22.7. Interceptors

This section defines ATDL operations that allow services such as dynamic type checking to be inserted in the invocation path. Interceptors are not specific to dynamic type checking; they could be used to invoke any ATDL function. Logically, an interceptor is interposed in the invocation (and response) path(s) between a client and a target object. Interceptors are an optional extension to ATDL to allow for backward compatibility with TTCN-3.

Interceptors provide a highly flexible means of adding portable Services to a ATDL-compliant object system. The flexibility derives from the capacity of a binding between client and target object to be extended and specialized to reflect the mutual requirements of client and target.

When remote invocation is required, the ATDL run-time system will transform the request into a message, which can be sent over the network. As functions such as encryption are performed on messages, a second kind on interceptor interface is required.

The ATDL code invokes each message-level interceptor via the **send_message** operation (when sending a message, for example, the request at the client and the reply at the target) or the **receive_message** operation (when receiving a message).

When a client message-level interceptor is activated to perform a **send_message** operation, it transforms the message as required, and calls a **call** operation to pass the message on to the ATDL run-time system and hence to its target. Unlike invoke operations, **call** operations may return to the caller without completing the operation. The interceptor can then perform other operations if required before exiting. The client interceptor may next be called either using **send_message** to process another outgoing message, or using **receive_message** to process an incoming message.

22.8. Channel controlling operations

The channel control operations provide the essential functionality of an interface, i.e., dynamic querying and lifetime management. This functionality is established in the three methods, ATDL operations for controlling message-based, operational channels are:

- **clear**: remove the contents of a server-side channel queue;
- **start**: start listening at and give access to a channel;
- **stop**: stop listening and disallow sending operations at a channel.

Concrete textual grammar

290 ChannelStartStatement ::= ChannelOrAll Dot “start”

291 ClearStatement ::= ChannelOrAll Dot “clear”

292 ChannelStopStatement ::= ChannelOrAll Dot “stop”

Concrete graphical grammar

293 <channel controlling area> ::= <condition symbol> **contains**
(ClearStatement | ChannelStopStatement | ChannelStartStatement)

The server-side channel controlling methods shall be represented within the <condition symbol>. The <condition symbol> contains the channel controlling statement.

22.8.1. The Bind method

The **bind** operation provides a method for dynamically querying a given server component and obtaining *interface reference*s for the interfaces the component supports. The **bind** operation determines if the object supports a particular ATDL interface. If it does, the system increases the component's reference count, and the application can use that interface immediately.

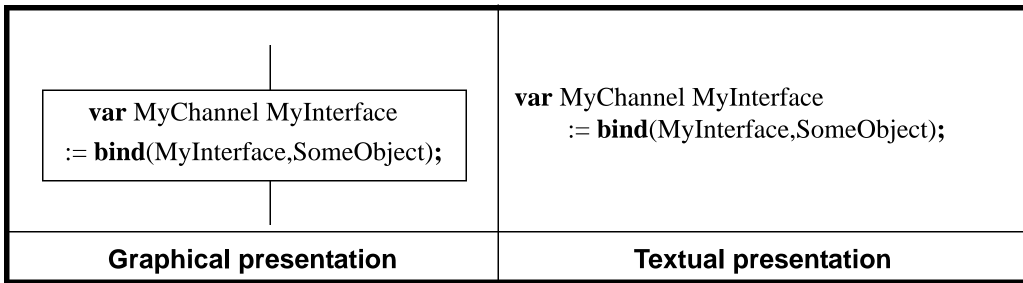
A client program uses a remote co-object by obtaining a co-interface reference to the object. Co-interface references are usually obtained using the **bind** operation in ATDL. The **bind** operation in ATDL returns a co-interface reference to a run-time object to your client program. Your client program can use the co-interface reference to invoke operations on the object that have been defined in the object’s AODL co-interface specification.

If the application does not need to use the interface (co-interface) retrieved by a call to this method, it must call the **release** operation for that interface (co-interface) to free it.

Concrete textual grammar

288 BindOp ::= “bind” “(“ (InterfaceType | ExceptionIdentifier) “,” ComponentRef “)”

Figure 46. The bind statement



Graphical notation

The **bind** statement shall be represented within a <task symbol>. The <task symbol> contains the **bind** statement (Figure 46).

Mapping

The TTCN-3 **connect** and **map** operations are considered to be equivalent to ATDL **bind** operations. For example, TTCN-3 statement:

connect(MyNewComponent:Port1, **mtc**:Port3);

Is equivalent to ATDL statement: Port1 := **bind** (Port3, **mtc**);

map(MyNewComponent:Port2, **system**:PCO1); // TTCN-3 statement

Is equivalent to ATDL statement: Port2 := **bind**(PCO1, **system**);

22.8.2. The Release method

Call **release** operation when you no longer need to use an interface reference, the **release** operation decrements the reference count for the calling interface on a object. If the reference count on the object falls to 0, the object is freed from memory.

Because co-interface references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them.

When a co-interface reference is no longer required by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

Concrete textual grammar

289 ReleaseStatement ::= ChannelIdentifier Dot “release”

Mapping

The TTCN-3 **disconnect** and **unmap** operations are considered to be equivalent to ATDL **release** operations. For example, TTCN-3 statement:

disconnect(MyNewComponent:Port1, **mtc**:Port3);

Is equivalent to ATDL statement: Port1**.release**;

TTCN-3 statement: **unmap**(MyNewComponent:Port2, **system**:PCO1);

Is equivalent to ATDL statement: Port2**.release**;

Graphical notation

The **release** operation shall be represented within a <task symbol>. The <task symbol> contains the **release** statement.

Figure 47. The release operation

MyChannel**.release**	MyChannel**.release;**
Graphical presentation	**Textual presentation**

22.8.3. The Clear channel operation

The **clear** operation removes the contents of the *incoming* queue of the specified channel. If the channel queue is already empty then this operation shall have no action.

Graphical notation

The clear channel operation shall be represented within the <condition symbol>. The <condition symbol> contains the clear channel statement.

22.8.4. The Start channel operation

If a channel is defined as allowing receiving operations then the **start** operation clears the incoming queue of the named channel and starts listening for traffic over the channel. If the channel is defined to allow sending operations then the operations are also allowed to be performed at that channel.

By default, all channels of a component shall be started implicitly when a component is created. The start channel operation will cause unstopped channels to be restarted by removing all messages waiting in the incoming queue.

Graphical notation

The start channel operation shall be represented within the <condition symbol>. The <condition symbol> contains the start channel statement.

22.8.5. The Stop channel operation

The **stop** operation causes listening at the named channel to cease. To cease listening at the channel means that all receiving operations defined before the stop operation shall be completely performed before the working of the channel is suspended.

Figure 48. Stop channel operation

MyChannel.**stop**	MyChannel.**stop;**
Graphical presentation	**Textual presentation**

The stop channel operation shall be represented within the <condition symbol>. The <condition symbol> contains the stop channel statement (Figure 48).

22.8.6. Use of any and all with channels

The keywords **any** and **all** may be used with channel controlling operations. For example:

all interface.stop;
all interface.clear;

23. Timers and operations

Timers can be declared and used in the module control part. Additionally, timers can be declared in component type definitions. These timers can be used in test cases, functions and altsteps which are running on the given component type. A timer declaration may have an optional default duration value assigned to it. The timer shall be started with this value if no other value is specified. This value shall be a non-negative **float** value (i.e. greater or equal 0.0) where the base unit is seconds. For example,

timer MyTimer1 := 5e**-3**f; // declaration of the timer MyTimer1 with the default value of 5ms

Concrete textual grammar

243 TimerInstance ::= “timer” TimerIdentifier [AssignmentChar TimerValue]

244 TimerIdentifier ::= Identifier

245 TimerValue ::= SingleConstExpression

/* STATIC SEMANTICS - SingleConstExpression shall resolve to a value of type float. */

246 TimerRef ::= TimerIdentifier | TimerParIdentifier

23.1. Timers as parameters

Timers can only be passed by reference to functions and altsteps. Timers passed into a function or altstep are known inside the behaviour definition of the function or altstep.

Timers passed in as parameters by reference can be used like any other timer, i.e. it needs not to be declared. A started timer can also be passed into a function or altstep. The timer continues to run, i.e. it is not stopped implicitly. Thereby, possible timeout events can be handled inside the function or altstep to which the timer is passed.

```
// Function definition with a timer in the formal parameter list

function MyBehaviour (timer MyTimer)

{       :

        MyTimer.start;

        :

}
```

23.2. Timer class methods

ATDL supports a number of timer class methods. These class methods may be used in test cases, functions, altsteps and module control.

Table 28: Overview of ATDL timer operations

Timer class method	Associated keyword	Associated graphical symbol
Start timer	**start**	<internal output symbol>
Stop timer	**stop**	<internal output symbol>
Read elapsed time	**read**	<task symbol>
Check if timer running	**running**	<condition symbol>
Timeout event	**timeout**	<internal input symbol>

It is assumed that each ATDL scope unit in which timers are declared, maintains its own *running-timers list* and *timeout-list*, i.e. a list of all timers that is actually running and a list of all timers that has timed out. The timeout-lists are part of the snapshots that are taken when a test case is executed. A timeout-list is updated, if a timer in the scope unit is started, is stopped, times out or a **timeout** operation is executed.

NOTE: The running-timers list and the timeout-list are only a conceptual lists and does not restrict the implementation of timers. Other data structures like a set, where the access to timeout events is not restricted by, e.g. the order in which the timeout events have happened, may also be used.

When a timer expires (conceptually immediately before a snapshot processing of a set of alternative events), a timeout event is placed into the timeout-list of the scope unit in which the

timer has been declared. The timer becomes immediately inactive. Only one entry for any particular timer may appear in the timeout-list of the scope unit in which the timer has been declared at any one time.

All running timers shall automatically be cancelled when the thread is explicitly or implicitly stopped, or when the passive object is destroyed.

ATDL concrete textual grammar

294 TimerStatement ::= StartTimerStatement | StopTimerStatement | TimeoutStatement

295 TimerOps ::= ReadTimerOp | RunningTimerOp

ATDL concrete graphical grammar

303 <timer statement area> ::= <fgr timer start area> | <fgr timer stop area> | <fgr timeout area> | <fgr timer running area>

23.2.1. The Start timer operation

The **start** timer operation is used to indicate that a timer should start running. Timer values shall be non-negative **float** numbers (i.e. greater or equal 0.0). When a timer is started, its name is added to the list of running timers (for the given scope unit).

The optional timer value parameter shall be used if no default duration is given, or if it is desired to override the default value specified in the timer declaration. When a timer duration is overridden, the new value applies only to the current instance of the timer, any later **start** operations for this timer, which do not specify a duration, shall use the default duration.

Starting a timer with the timer value 0.0 means that the timer times out immediately. Starting a timer with a negative timer value, e.g. the timer value is the result of an expression, or without a specified timer value shall cause a runtime error.

The timer clock runs from the float value zero (0.0) up to maximum stated by the duration parameter.

The **start** operation may be applied to a running timer, in which case the timer is stopped and re-started. Any entry in a timeout-list for this timer shall be removed from the timeout-list.

Concrete textual grammar

296 StartTimerStatement ::= TimerRef Dot “start” [“(“ TimerValue “)”]

Figure 49. The start timer operation

MyTimer**.start**(10.0)	MyTimer**.start**(10.0);
Graphical presentation	**Textual presentation**

Concrete graphical grammar

304 <fgr timer start area> ::= <internal output symbol> **contains** StartTimerStatement

The **start** timer operation shall be represented within an <internal output symbol>. The <internal output symbol> contains the **start** timer statement (Figure 49).

23.2.2. The Stop timer method

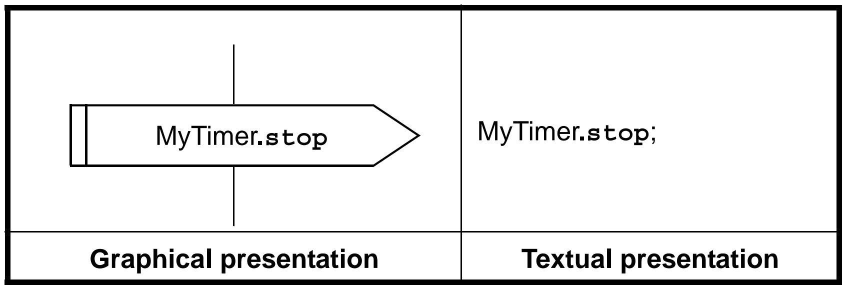
The **stop** operation is used to stop a running timer and to remove it from the list of running timers. A stopped timer becomes inactive and its elapsed time is set to the float value zero (0.0).

Stopping an inactive timer is a valid operation, although it does not have any effect. Any entry in a timeout-list for this timer shall be removed from the timeout-list.

The **all** keyword may be used to stop all timers that are visible in the scope unit in which the **stop** operation has been called. For example,

MyTimer1.**stop**; // stops MyTimer1
all timer.stop; // stops all running timers

Figure 50. The stop timer operation



ATDL concrete textual grammar

297 StopTimerStatement ::= TimerRefOrAll Dot “stop”

ATDL concrete graphical grammar

305 <fgr timer stop area> ::= <internal output symbol> **contains** StopTimerStatement

The **stop** timer operation shall be represented within an <internal output symbol>. The <internal output symbol> contains the **stop** timer statement (Figure 50).

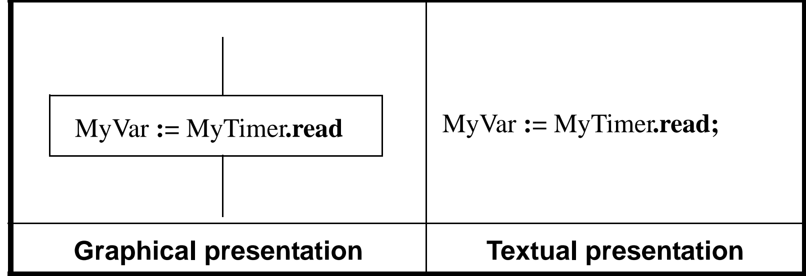
23.2.3. The Read timer method

The **read** operation is used to retrieve the time that has elapsed since the specified timer was started and to store it into the specified variable. This variable shall be of type **float**.

Applying the **read** operation on an inactive timer i.e. on a timer not listed on the running-timer list, will return the value zero. For example,

var MyVar ::= **float**;
MyVar := MyTimer1.**read**; // assign to MyVar the time that has elapsed since MyTimer1 was started

Figure 51. The read timer operation



Concrete textual grammar

299 ReadTimerOp ::= TimerRef Dot “read”

The read timer operation shall be put into a <task symbol> (Figure 51).

23.2.4. The Running timer operation

The **running** timer operation is used to check whether a timer is listed on the running-timer list of the given scope unit or not (i.e. that it has been started and has neither timed out nor been stopped). The operation returns the value **true** if the timer is listed on the list, **false** otherwise. For example,

if [MyTimer1.**running**] { ... }

Concrete textual grammar

300 RunningTimerOp ::= TimerRefOrAny Dot “running”

Concrete graphical grammar

307 <fgr timer running area> ::= <condition symbol> **contains** RunningTimerOp

23.2.5. The Timeout operation

The **timeout** operation allows to check expiration of a timer, or of all timers, in a scope unit of a test component or module control in which the timeout operation has been called.

When a **timeout** operation is processed, if a timer name is indicated, the timeout-lists of the component or module control are searched according to the ATDL scope rules. If there is a timeout event matching the timer name, that event is removed from the timeout-list, and the **timeout** operation succeeds. The **timeout** shall not be used in a **boolean** expression, but it can be used to determine an alternative in an **alt** statement or as stand-alone statement in a behaviour description. In the latter case a **timeout** operation is considered to be shorthand for an **alt** statement with only one alternative, i.e. it has blocking semantics, and therefore provides the ability of passive waiting for the timeout of timer(s).

Concrete textual grammar

301 TimeoutStatement ::= TimerRefOrAny Dot "timeout"

Figure 52. The timeout operation

MyTimer.timeout	MyTimer.timeout;
Graphical presentation	**Textual presentation**

Concrete graphical grammar

306 <fgr timeout area> ::= <internal input symbol> **contains** TimeoutStatement

The **timeout** operation shall be represented within an <internal input symbol>. The <internal input symbol> contains the **timeout** statement (Figure 52).

23.2.6. Summary of use of any and all with timers

The keywords **any** and **all** may be used with timer operations as indicated in Table 29.

Concrete textual grammar

298 TimerRefOrAll ::= TimerRef | “all” “timer”

302 TimerRefOrAny ::= TimerRef | “any” “timer”

Table 29: Any and All with Timers

Operation	Allowed		Example
	any	**all**	
start			
stop		yes	**all timer.stop**
read			
running	yes		**if [any timer.running] {…}**
timeout	yes		**any timer.timeout**

24. Specifying attributes

Language attributes can be associated with ATDL language elements by means of the **with** statement. The syntax for the argument of the **with** statement (i.e., the actual attributes) is simply defined as a free text string.

There are four kinds of language attributes:

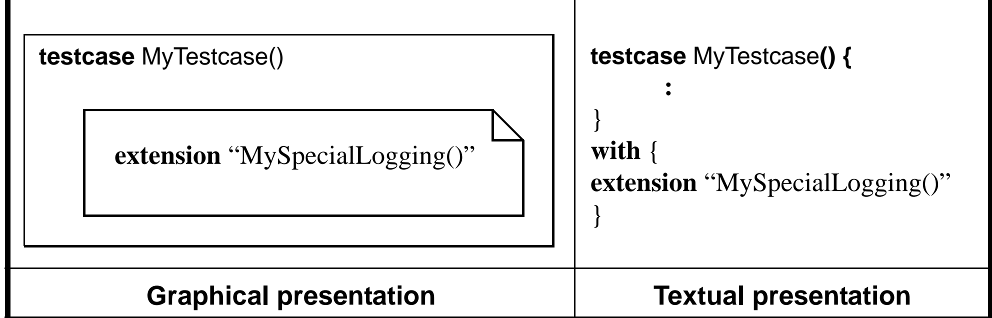
a) **display**: allows the specification of display attributes related to specific presentation formats;

b) **encode**: allows references to specific encoding rules;

c) `variant`: allows references to specific encoding variants;

d) **extension**: allows the specification of user-defined attributes.

Figure 53. Specifying attributes



Concrete textual grammar

385 WithStatement ::= "with" ["tabular"] WithAttribList

386 WithAttribList ::= SingleWithAttrib SemiColon | "{" MultiWithAttrib "}" [SemiColon]

387 MultiWithAttrib ::= SingleWithAttrib {SemiColon SingleWithAttrib} [SemiColon]

388 SingleWithAttrib ::= ["encode" | "display" | "extension"] ["override"] [AttribQualifier] AttribSpec

389 AttribQualifier ::= "(" DefOrFieldRefList ")"

390 DefOrFieldRefList ::= DefOrFieldRef {"," DefOrFieldRef}*

391 DefOrFieldRef ::= DefinitionRef | FieldReference

392 DefinitionRef ::= TypeIdentifier | InterfaceIdentifier | ComponentIdentifier |
ConstIdentifier | TemplateIdentifier | AltstepIdentifier |
TestcaseIdentifier | FunctionIdentifier | OperationIdentifier

393 AttribSpec ::= FreeText

Graphical notation

The attributes defined for the module control part, testcases, functions and altsteps are represented within the text symbol. The syntax of the **with** statement is placed within that symbol. An example is given in Figure 53.

24.1. Display attributes

All ATDL language elements can have **display** attributes to specify how particular language elements should be displayed in, for example, a tabular format.

24.2. Encoding of values

Encoding rules define how a particular value, template etc. shall be encoded and transmitted over a communication **interface** and how received signals shall be decoded. ATDL does not have a default encoding mechanism. This means that encoding rules or encoding directives are defined in some external manner to ATDL.

In ATDL, general or particular encoding rules can be specified by using **encode** and **variant** attributes.

24.2.1. Encode attributes

The **encode** attribute allows the association of some referenced encoding rule or encoding directive to be made to an ATDL definition.

The manner in which the actual encoding rules are defined (e.g. prose, functions etc.) is outside the scope of the present document. If no specific rules are referenced then encoding shall be a matter for individual implementation.

24.2.2. Variant attributes

To specify a refinement of the currently specified encoding scheme instead of its replacement, the **variant** attribute shall be used.

24.2.3. Special strings

The following strings are the predefined (standardized) **variant** attributes for simple basic types (see clause 10.10.1):

a) "8 bit" and "unsigned 8 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 8-bits (single byte) within the system.

b) "16 bit" and "unsigned 16 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 16-bits (two bytes) within the system.

c) "32 bit" and "unsigned 32 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 32-bits (four bytes) within the system.

d) "64 bit" and "unsigned 64 bit" mean, when applied to integer and enumerated types, that the integer value or the integer numbers associated with enumerations shall be handled as it was represented on 64-bits (eight bytes) within the system.

e) "IEEE754 float","IEEE754 double", "IEEE754 extended float" and "IEEE754 extended double" mean, when applied to a float type, that the value shall be encoded and decoded according to the standard IEEE 754.

The following strings are the predefined (standardized) **variant** attributes for **char**, **wide char, charstring** and **wide charstring** (see clause 10.10.2):

a) "UTF-8" means, when applied to wide char and wide charstring types, that each character of the value shall be individually encoded and decoded according to the UCS Transformation Format 8 (UTF-8) as defined in annex R of ISO/IEC 10646 [7].

b) "UCS-2" means, when applied to wide char and wide charstring types, that each character of the value shall be individually encoded and decoded according to the UCS-2 coded representation form (see clause 14.1 of ISO/IEC 10646 [7]).

c) "8 bit" means, when applied to char, wide char, charstring and wide charstring types, that each character of the value shall be individually encoded and decoded according to the coded representation as specified in ISO/IEC 8859 (an 8-bit coding).

These variant attributes can be used in combination with the more general encode attributes specified at a higher level. For example a **wide charstring** specified with the **variant** attribute "UTF-8" within a module which itself has a global encoding attribute "BER:1997" will cause each character of the values within the string to first be encoded following the UTF-8 rules and then this UTF-8 value will be encoded following the more global BER rules.

24.2.4. Invalid encodings

If it is desired to specify invalid encoding rules then these shall be specified in a referenceable source external to the module in the same way that valid encoding rules are referenced.

24.3. Extension attributes

All ATDL language elements can have **extension** attributes specified by the user.

NOTE: Because user-defined attributes are not standardized the interpretation of these attributes between tools supplied by different vendors may differ or even not be supported.

24.4. Scope of attributes

A **with** statement may associate attributes to a single language element. It is also possible to associate attributes to a number of language elements by e.g. listing fields of a structured type in an attribute statement associated with a single type definition or associating a **with** statement to the surrounding scope unit or **group** of language elements.

24.5. Overwriting rules for attributes

An attribute definition in a lower scope unit will override a general attribute definition in a higher scope.

A **with** statement that is placed inside the scope of another **with** statement shall override the outermost **with**. This shall also apply to the use of the **with** statement with groups. Care should be taken when the overwriting scheme is used in combination with references to single definitions. The general rule is that attributes shall be assigned and overwritten according to the order of their occurrence.

An attribute definition in a lower scope can be overwritten in a higher scope by using the **override** directive.

The **override** directive forces all contained types at all lower scopes to be forced to the specified attribute.

24.6. Changing attributes of imported language elements

In general, a language element is imported together with its attributes. In some cases these attributes may have to be changed when importing the language element e.g. a type may be displayed in one module as ASP, then it is imported by another module where it should be displayed as PDU. For such cases it is allowed to change attributes on the **import** statement.

25. The System module

This chapter discusses and summarizes standard library functions. Many of the functions listed here are defined in the *System* module, which is implicitly compiled with every application.

25.1. The Group System.lang

The System.lang group contains classes that are fundamental to the design of the ATDL language. The most important classes are TObject, which is the root of the class hierarchy, and TClass, instances of which represent classes at run time.

25.1.1. The Class TNumber

The virtual class TNumber has descendant classes TInteger, TCardinal, TFloat, and TDouble which wrap primitive types, defining virtual methods to convert the represented numeric value to pre-defined Integer, Cardinal, float, and Double.

virtual class **TNumber** {
    public virtual function **IntegerValue**() return Integer;
    public virtual function **CardinalValue**() return Cardinal;
    public virtual function **floatValue**() return float;
    public virtual function **DoubleValue**() return Double;
    }

25.1.2. The Class TInteger

final class **TInteger** extends TNumber {
    const **MIN_VALUE** Integer := -2147483648;
    const **MAX_VALUE** Integer := 2147483647;
    public constructor TInteger(value Integer);
    public override function **IntegerValue**() return Integer;
    public override function **floatValue**() return float;
    public override function **DoubleValue**() return Double;
    public class function **toCharString**(value Integer) return charstring;
    public class function **toHexString**(value Integer, length Cardinal) return hexstring;
    public class function **toOctetString**(value Integer, length Cardinal) return octetstring;
    public class function **toBitString**(value Integer, length Cardinal) return bitstring;
    public class function **parseInteger**(cs charstring) return Integer;
    public class function **valueOf**(cs charstring) return Integer;
    public class function **valueOf**(bs bitstring) return Integer;
    public class function **valueOf**(hs hexstring) return Integer;
    public class function **valueOf**(os octetstring) return Integer;
    public class function **valueOf**(chr char) return Integer;
    public class function **valueOf**(uchr wide char) return Integer;
    }

25.1.2.1. Constructor Integer

public constructor TInteger(value Integer);

This constructor initializes a newly created TInteger object so that it represents the primitive value that is the argument.

25.1.2.2. Get integer value

public override function **IntegerValue**() return Integer;

The integer value represented by this TInteger object is returned.

25.1.2.3. Integer to float

public override function **floatValue**() return float;

The **integer** value represented by this TInteger object is converted to type float and the result of the conversion is returned.

25.1.2.4. Integer to double

public override function **DoubleValue**() return Double;

The **integer** value represented by this TInteger object is converted to type Double and the result of the conversion is returned.

25.1.2.5. Integral to bitstring

public class function **toBitString**(value Integer, length Cardinal) return bitstring;

This function converts a single integer value to a single fixed length **bitstring** value. The resulting string is length bits long.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **integer** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively. If the conversion yields a value with fewer bits than the **bitstring** length, then the **bitstring** shall be padded on the left with zeros. A test case error shall occur if the value is negative or if the resulting **bitstring** contains more bits than the **bitstring** length.

25.1.2.6. Integer to hexstring

public class function **toHexString**(value Integer, length Cardinal) return hexstring;

This function converts a single integer value to a single fixed length **hexstring** value. The resulting string is length hexadecimal digits long.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 … F represent the decimal values 0 … 15 respectively. If the conversion yields a value with fewer hexadecimal digits than specified in the **hexstring** length, then the **hexstring** shall be padded on the left with zeros. A test case error shall occur if the value is negative or if the resulting **hexstring** contains more hexadecimal digits than the **hexstring** length.

25.1.2.7. Integer to octetstring

public class function **toOctetString**(value Integer, length Cardinal) return octetstring;

This function converts a single integer value to a single fixed length **octetstring** value. The resulting string is length octets long.

For the purposes of this conversion, an **octetstring** shall be interpreted as a positive base 16 **integer** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively. If the conversion yields a value with fewer hexadecimal digits than the **hexstring** length, then the **hexstring** shall be padded on the left with zeros. A

test case error shall occur if the value is negative or if the resulting **hexstring** contains more hexadecimal digits than the **hexstring** length.

25.1.2.8. Integer to charstring

public class function **toCharstring**(value Integer) return charstring;

This function converts the integer value into its string equivalent (the base of the return string is always decimal). The result is a charstring that represents the sign and magnitude (absolute value) of the integer value. If the sign is negative, the first character of the result is “-”; if the sign is positive, no sign character appears in the result. As for the magnitude *m*:

If *m* is infinity, it is represented by the characters “infinity”; thus, positive infinity produces the result “infinity” and negative infinity produces the result “-infinity”.

25.1.2.9. Parse integer charstring

public class function **parseInteger**(cs charstring) return Integer;

The actual parameter is interpreted as representing a signed decimal integer. The components of the charstring must all be decimal digits, except that the first character may be '-' to indicate a negative value.

25.1.2.10. Charstring to integer

public class function **valueOf**(cs charstring) return Integer;

The actual parameter is interpreted as representing a signed decimal integer, exactly as if the actual parameter were given to the parseInt method that takes one actual parameter. If the string does not represent a valid integer value the function returns the value zero (0).

25.1.2.11. Bitstring to integer

public class function **valueOf**(bs bitstring) return Integer;

This function converts a single **bitstring** value to a single **integer** value.

For the purposes of this conversion, a **bitstring** shall be interpreted as a positive base 2 **cardinal** value. The rightmost bit is least significant, the leftmost bit is the most significant. The bits 0 and 1 represent the decimal values 0 and 1 respectively.

25.1.2.12. Hexstring to integral

public class function **valueOf**(hs hexstring) return Integer;

This function converts a single **hexstring** value to a single **integer** value.

For the purposes of this conversion, a **hexstring** shall be interpreted as a positive base 16 **cardinal** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively.

25.1.2.13. Octetstring to integral

public class function **valueOf**(os octetstring) return Integer;

This function converts a single **octetstring** value to a single **integer** value.

For the purposes of this conversion, an **octetstring** shall be interpreted as a positive base 16 **cardinal** value. The rightmost hexadecimal digit is least significant, the leftmost hexadecimal digit is the most significant. The number of hexadecimal digits provided shall be multiples of 2 since one octet is composed of two hexadecimal digits. The hexadecimal digits 0 .. F represent the decimal values 0 .. 15 respectively.

25.1.2.14. Character to integer

public class function **valueOf**(chr char) return Integer;

This function converts a **char** value of ISO/IEC 646 [6] into an integer value in the range of 0 … 127. The integer value describes the 8-bit encoding of the character.

25.1.2.15. Wide character to integral

public class function **valueOf**(uchr wide char) return Integer;

This function converts a **wide char** value of ISO/IEC 10646 [7] into an **integer** value in the range of 0 … 2 147 483 647. The integer value describes the 32-bit encoding of the character.

25.1.2.16. Integer to character

This function converts an **integer** value in the range of 0 … 127 (8-bit encoding) into a character value of ISO/IEC 646 [6]. The integer or cardinal value describes the 8-bit encoding of the character.

25.1.2.17. Integer to wide character

This function converts an **integer** value (32-bit encoding) in the range of 0…2 147 483 647 into a character value of ISO/IEC 10646 [7]. The integer or cardinal value describes the 32-bit encoding of the character.

25.1.3. The Class TFloat

```
final class TFloat extends TNumber {
    const  MIN_VALUE float := 1.4E-45f;
    const MAX_VALUE float := 3.4028235E38f;
    const NaN float := 0.0f/0.0f;
    public override function IntegerValue() return Integer;
    public override function CardinalValue() return Cardinal;
    public override function floatValue() return float;
    public override function DoubleValue() return Double;
    public class function toCharString(value float) return charstring;
    }
```

25.1.3.1. MIN_VALUE float := 1.4E-45f

The constant value of this field is the smallest positive nonzero value of type float.

25.1.3.2. MAX_VALUE float := 3.4028235E38f

The constant value of this field is the largest positive finite value of type float.

25.1.3.3. NaN float := 0.0f/0.0f

The constant value of this field is the Not-a-Number value of type float.

25.1.3.4. Float to integral

public override function **IntegerValue**() return Integer;

The float value represented by this TFloat object is converted to type Integer by removing the fractional part of the argument and returning the resulting Integer.

25.1.3.5. Float to cardinal

public override function **CardinalValue**() return Cardinal;

The float value represented by this TFloat object is converted to type Cardinal by removing the fractional part of the argument and returning the resulting Cardinal.

25.1.3.6. Get float value

public override function **floatValue**() return float;

The float value represented by this TFloat object is returned.

25.1.3.7. Float to double

public override function **DoubleValue**() return Double;

The float value represented by this TFloat object is converted (§17.2.2) to type Double and the result of the conversion is returned.

25.1.3.8. Float to charstring

public class function **toCharString**(value float) return charstring;

The float type value is converted to a readable charstring format as follows. All characters and characters in charstrings mentioned below are ISO/IEC 646 [6] characters.

a) If the argument is Not-a-Number (NaN) value, the result is the charstring "NaN".

b) Otherwise, the result is a charstring that represents the sign and magnitude (absolute value) of the float value. If the sign is negative, the first character of the result is "-"; if the sign is positive, no sign character appears in the result. As for the magnitude *m*:

- If *m* is greater than 10^{-3} or equal to but less than 10^{7}, then it is represented as the integer part of *m*, in decimal form with no leading zeroes, followed by ".", followed by one or more decimal digits representing the fractional part of *m*.
- If *m* is less than 10^{-3} or not less than 10^{7}, then it is represented in so-called "computerized scientific notation."

25.1.4. The Class TDouble

final class **TDouble** extends TNumber {
const **NEGATIVE_INFINITY** := -1.0/0.0;
const **POSITIVE_INFINITY** := 1.0/0.0;
const **NaN** real := 0.0/0.0;
public override function **IntegerValue**() return Integer;
public override function **CardinalValue**() return Cardinal;
public override function **floatValue**() return float;
public override function **DoubleValue**() return Double;
public class function **toCharString**(value Double) return charstring;
}

25.1.4.1. NEGATIVE_INFINITY := -1.0/0.0

The constant value of this field is the negative infinity of type Double.

25.1.4.2. POSITIVE_INFINITY := 1.0/0.0

The constant value of this field is the positive infinity of type Double.

25.1.4.3. Double to integral

public override function **IntegerValue**() return Integer;

The Double value represented by this TDouble object is converted to type Integer by removing the fractional part of the argument and returning the resulting Integer.

25.1.4.4. Double to float

public override function **floatValue**() return float;

The Double value represented by this TDouble object is converted to type float and the result of the conversion is returned.

25.1.4.5. Get double value

public override function **DoubleValue**() return Double;

The Double value represented by this TDouble object is returned.

25.1.4.6. Double to charstring

public class function **toCharString**(value Double) return charstring;

The Double type value is converted to a readable charstring format as follows. All characters and characters in charstrings mentioned below are ISO/IEC 646 [6] characters.

a) If the argument is Not-a-Number (NaN) value, the result is the charstring "NaN".

b) Otherwise, the result is a charstring that represents the sign and magnitude (absolute value) of the real value. If the sign is negative, the first character of the result is "-"; if the sign is positive, no sign character appears in the result. As for the magnitude *m*:

- If *m* is infinity, it is represented by the characters "infinity"; thus, positive infinity produces the result "infinity" and negative infinity produces the result "-infinity".
- If *m* is zero, it is represented by the characters "0.0"; thus, negative zero produces the result "-0.0" and positive zero produces the result "0.0".
- If *m* is greater than 10^{-3} or equal to but less than 10^{7}, then it is represented as the integer part of *m*, in decimal form with no leading zeroes, followed by ".", followed by one or more decimal digits representing the fractional part of *m*.
- If *m* is less than 10^{-3} or not less than 10^{7}, then it is represented in so-called "computerized scientific notation."

25.1.5. The Class TBitString

An object of type TBitString, once created, is immutable. It represents a fixed-length sequence of bits. Compare this to the class TBitStringBuffer (§25.1.6), which represents a modifiable, variable-length sequence of bits.

final class **TBitString** {
public class function **toCharString**(value bitstring) return charstring;
public class function **toHexString**(value bitstring) return hexstring;
public class function **toOctetString**(value bitstring) return octetstring;
public function **getBit**(position Cardinal) return boolean;
public function **setBit**(position Cardinal, value boolean);
public function **getLength**() return Cardinal;
}

25.1.5.1. Bitstring to charstring

public class function **toCharString**(value bitstring) return charstring;

This function converts a single **bitstring** value to a single **charstring**. The resulting **charstring** has the same length as the **bitstring** and contains only the characters '0' and '1'.

For the purpose of this conversion, a **bitstring** should be converted into a **charstring**. Each bit of the **bitstring** is converted into a character '0' or '1' depending on the value 0 or 1 of the bit. The consecutive order of characters in the resulting **charstring** is the same as the order of bits in the **bitstring**.

25.1.5.2. Bitstring to hexstring

public class function **toHexString**(value bitstring) return hexstring;

This function converts a single **bitstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **bitstring**.

For the purpose of this conversion, a bitstring should be converted into a hexstring, where the bitstring is divided into groups of four bits beginning with the rightmost bit. Each group of four bits is converted into a hex digit as follows:

'0000'B -> '0'H, '0001'B -> '1'H, '0010'B -> '2'H, '0011'B -> '3'H, '0100'B -> '4'H, '0101'B -> '5'H, '0110'B -> '6'H, '0111'B -> '7'H, '1000'B -> '8'H, '1001'B -> '9'H, '1010'B -> 'A'H, '1011'B -> 'B'H, '1100'B -> 'C'H, '1101'B -> 'D'H, '1110'B ->'E'H, and '1111'B -> 'F'H.

When the leftmost group of bits does contain less than 4 bits, this group is filled with '0'B from the left until it contains exactly 4 bits and is converted afterwards. The consecutive order of hex digits in the resulting hexstring is the same as the order of groups of 4 bits in the bitstring.

25.1.5.3. Bitstring to octetstring

public class function **toOctetString**(value bitstring) return octetstring;

This function converts a single **bitstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **bitstring**.

25.1.5.4. The getBit function

public function **getBit**(position Cardinal) return boolean;

The result is true if the bit with index position is currently set in this TBitString; otherwise, the result is false. Valid values for position are 0 to length - 1;

25.1.5.5. The setBit function

public function **setBit**(position Cardinal, value boolean);

Set the bit at position to value (0 | 1). position 0 denotes the first bit in this TBitString. Valid values for position are 0 to length - 1;

25.1.5.6. Length of bitstring type

public function **getLength**() return Cardinal;

The length of the sequence of bits represented by this TBitString object is returned.

25.1.6. The Class TBitStringBuffer

A bitstring buffer is like a TBitString (§25.1.5), but can be modified. At any point in time it contains some particular sequence of bits, but the length and content of the sequence can be changed through certain method calls.

25.1.7. The Class TOctetString

An object of type TOctetString, once created, is immutable. It represents a fixed-length sequence of octets. Compare this to the class TOctetStringBuffer (§25.1.8), which represents a modifiable, variable-length sequence of octets.

```
final class TOctetString {
    public class function toCharString(value octetstring) return charstring;
    public class function toHexString(value octetstring) return hexstring;
    public class function toBitString(value octetstring) return bitstring;
    public function getOctet(position Cardinal) return char;
    public function getLength() return Cardinal;
    }
```

25.1.7.1. Octetstring to character string

```
public class function toCharString(value octetstring) return charstring;
```

This function converts an **octetstring** value to a **charstring**. The resulting **charstring** will have the same length as the incoming **octetstring**. The octets are interpreted as ISO/IEC 646 [6] codes (according to the IRV) and the resulting characters are stored in the returned value. Octet values higher than 7F shall cause an error.

25.1.7.2. Octetstring to hexstring

```
public class function toHexString(value octetstring) return hexstring;
```

This function converts a single **octetstring** value to a single **hexstring**. The resulting **hexstring** represents the same value as the **octetstring**.

For the purpose of this conversion, an **octetstring** should be converted into a **hexstring** containing the same sequence of hex digits as the **octetstring**.

25.1.7.3. Octetstring to bitstring

```
public class function toBitString(value octetstring) return bitstring;
```

This function converts a single **octetstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **octetstring**.

25.1.7.4. Length of octetstring type

```
public function getLength() return Cardinal;
```

The length of the sequence of octets represented by this TOctetString object is returned.

25.1.8. The Class TOctetStringBuffer

An octetstring buffer is like a TOctetString (§25.1.7), but can be modified. At any point in time it contains some particular sequence of octets, but the length and content of the sequence can be changed through certain method calls.

```
class TOctetStringBuffer {
    public function getOctet(position Cardinal) return char;
    public function setOctet(position Cardinal, value Cardinal);
    public function getLength() return Cardinal;
    public function setLength(newLength Cardinal);
    }
```

25.1.8.1. Length of hexstring type

public function **getLength**() return cardinal;

This function returns the length of the sequence of octets currently represented by this TOctetStringBuffer object. Returns zero if the value of this TOctetStringBuffer is omit.

25.1.8.2. The setLength function

public function **setLength**(newLength cardinal);

This octetstring buffer is altered to represent a new octet sequence whose length is specified by the actual parameter.

25.1.8.3. The getOctet function

public function **getOctet**(position Cardinal) return char;

This function returns the value (0..255) at position of this ATDL octetstring. Position 0 denotes the first octet of the ATDL octetstring.

25.1.8.4. The setOctet function

public function **setOctet**(position Cardinal, value Cardinal);

Sets the octet at position to value (0..255). Position 0 denotes the first octet in the octetstring.

25.1.9. The Class THexString

An object of type THexString, once created, is immutable. It represents a fixed-length sequence of hexadecimal digits. Compare this to the class THexStringBuffer (§25.1.10), which represents a modifiable, variable-length sequence of hexadecimal digits.

final class **THexString** {
public class function **toCharString**(value hexstring) return charstring;
public class function **toOctetString**(value hexstring) return octetstring;
public class function **toBitString**(value hexstring) return bitstring;
public function **getLength**() return Cardinal;
}

25.1.9.1. Hexstring to charstring

public class function **toCharString**(value hexstring) return charstring;

This function converts a single hexstring value to a single charstring. The resulting character string has the same length as the hexstring and contains only the characters '0' to '9'and 'A' to 'F'.

For the purpose of this conversion, a **hexstring** should be converted into a **charstring**. Each hex digit of the **hexstring** is converted into a character '0' to '9' and 'A' to 'F' depending on the value 0 to 9 or A to F of the hex digit. The consecutive order of characters in the resulting **charstring** is the same as the order of digits in the **hexstring**.

25.1.9.2. Hexstring to octetstring

public class function **toOctetString**(value hexstring) return octetstring;

This function converts a single **hexstring** value to a single **octetstring**. The resulting **octetstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** should be converted into an **octetstring**, where the **octetstring** contains the same sequence of hex digits as the **hexstring** when the length of

the **hexstring** modulo 2 is 0. Otherwise, the resulting **octetstring** contains 0 as leftmost hex digit followed by the same sequence of hex digits as in the **hexstring**.

25.1.9.3. Hexstring to bitstring

This function converts a single **hexstring** value to a single **bitstring**. The resulting **bitstring** represents the same value as the **hexstring**.

For the purpose of this conversion, a **hexstring** should be converted into a **bitstring**, where the hex digits of the **hexstring** are converted in groups of bits as follows:

'0'H -> '0000'B, '1'H -> '0001'B, '2'H -> '0010'B, '3'H -> '0011'B, '4'H -> '0100'B, '5'H -> '0101'B, '6'H -> '0110'B, '7'H -> '0111'B, '8'H-> '1000'B, '9'H -> '1001'B, 'A'H -> '1010'B, 'B'H -> '1011'B, 'C'H -> '1100'B, 'D'H -> '1101'B, 'E'H -> '1110'B, and 'F'H -> '1111'B.

The consecutive order of the groups of 4 bits in the resulting **bitstring** is the same as the order of hex digits in the **hexstring**.

25.1.9.4. Length of hexstring type

public function **getLength**() return Cardinal;

The length of the sequence of hexadecimal digits represented by this THexString object is returned.

25.1.10. The Class THexStringBuffer

A hexstring buffer is like a THexString (§25.1.9), but can be modified. At any point in time it contains some particular sequence of hexadecimal digits, but the length and content of the sequence can be changed through certain method calls.

```
class THexStringBuffer {
    public function getHex(position Cardinal) return char;
    public function setHex(position Cardinal, value Cardinal);
    public function getLength() return Cardinal;
    public function setLength(newLength Cardinal);
    }
```

25.1.10.1. Length of hexstring type

public function **getLength**() return Cardinal;

This function returns the length of the sequence of hexadecimal digits currently represented by this THexStringBuffer object.

25.1.10.2. The setLength function

public function **setLength**(newLength Cardinal);

This hexstring buffer is altered to represent a new hexadecimal digit sequence whose length is specified by the actual parameter.

25.1.10.3. The getHex function

public function **getHex**(position Cardinal) return char;

This function returns the value (0..15) at position of this ATDL hexstring. Position 0 denotes the first hexadecimal digits of the ATDL hexstring. Valid values for position are from 0 to current length - 1.

25.1.10.4. The setHex function

public function **setHex**(position Cardinal, value Cardinal);

This function sets the hex digit at position to value (0..15). Position 0 denotes the first octet in the hexstring. Valid values for position are from 0 to current length - 1.

25.1.11. The Class TCharString

An object of type TCharString, once created, is immutable. It represents a fixed-length sequence of characters. Compare this to the class TCharStringBuffer (§25.1.12), which represents a modifiable, variable-length sequence of characters.

final class **TCharString** {

public class function **toOctetString**(value charstring) return octetstring;

public class function **valueOf**(value integer) return charstring;

public function **getLength**() return Cardinal;

public function **getChar**(position Cardinal) return char;

}

25.1.11.1. Character string to octetstring

public class function **toOctetString**(value charstring) return octetstring;

This function converts a **charstring** value to an **octetstring**. The resulting **octetstring** will have the same length as the incoming **charstring**. Each octet of the **octetstring** will contain the ISO/IEC 646 [6] codes (according to the IRV) of the appropriate characters of the **charstring**.

25.1.11.2. Integer to charstring

public class function **valueOf**(value integer) return charstring;

A charstring is created and returned. The charstring is computed exactly as if by the method TInteger.toCharString of one argument (§25.1.2.8).

25.1.11.3. Length of charstring type

public function **getLength**() return Cardinal;

The length of the sequence of characters represented by this TCharString object is returned.

25.1.11.4. The getChar function

public function **getChar**(position Cardinal) return char;

This function returns the character indicated by the position argument within the sequence of characters represented by this TCharString. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing. If the position argument is negative or not less than the length of this string, then an Exception is raised.

25.1.12. The Class TCharStringBuffer

A charstring buffer is like a TCharString (§25.1.11), but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

class **TCharStringBuffer** {

public function **getLength**() return Cardinal;

public function **setLength**(newLength Cardinal);

public function **getChar**(position Cardinal) return char;
public function **setChar**(position Cardinal, value char);
}

25.1.12.1. Length of charstring type

public function **getLength**() return Cardinal;

This function returns the length of the sequence of characters currently represented by this TCharStringBuffer object.

25.1.12.2. The setLength function

public function **setLength**(newLength Cardinal);

This charstring buffer is altered to represent a new character sequence whose length is specified by the actual parameter.

25.1.12.3. The getChar function

public function **getChar**(position Cardinal) return char;

The specified character of the sequence currently represented by the charstring buffer, as indicated by the position argument, is returned. The first character of the sequence is at position 0, the next at position 1, and so on, as for array indexing. Valid values for position are from 0 to current length - 1. If the position argument is negative or not less than the length of this charstring, then a standard Exception is raised.

25.1.12.4. The setChar function

public function **setChar**(position Cardinal, value char);

The string buffer is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character value at index position.

25.2. The Group System.io

Input and output in ATDL is organized around the concept of bitstring streams. A bitstring stream is a sequence of bits, read or written over the course of time.

25.2.1. The Class DataInputStream

The class DataInputStream provides for reading bits from a bitstring stream and reconstructing from them data in any of the ATDL primitive types.

class **DataInputStream** {
public final function **getInteger** () return integer raises IOException;
public final function **getCardinal** () return cardinal raises IOException;
public final function **getFloat** () return float raises IOException;
public final function **getReal** () return real raises IOException;
public final function **getBoolean** () return boolean raises IOException;
public final function **getObjid** () return objid raises IOException;
public final function **getCharstring** () return charstring raises IOException;
public final function **getWideCharstring** () return wide charstring raises IOException;
public final function **getHexstring** () return hexstring raises IOException;
public final function **getBitstring** () return bitstring raises IOException;
public final function **getOctetstring** () return octetstring raises IOException;
public final function **getVerdict** () return verdicttype raises IOException;
}

25.2.1.1. The getInteger function

The **getInteger** function constructs and returns a basic ATDL integer type.

25.3. Predefined functions

ATDL contains a number of predefined (built-in) functions that need not be declared before use.

25.3.1. Number of elements in a structured type

sizeof(value structured_type) **return integer**;

The (**sizeof**) function returns the actual number of elements of a value that is of type **sequence**, **sequence of**, **template** or ASN.1 equivalent type SEQUENCE OF or SET OF. Its result is fully compatible with that of the equivalent ASN.1 SIZE constraint applied to objects of these types.

25.3.2. The IsPresent function

ispresent(value any_type) **return boolean**

This function returns the value **true** if and only if the value of the referenced field is present in the actual instance of the referenced data object. The argument to **ispresent** shall be a reference to a field within a data object that is defined as being **optional**.

25.3.3. The IsChosen function

ischosen(value any_type) **return boolean**

This function returns the value **true** if and only if the data object reference specifies the variant of the **choice** type that is actually selected for a given data object.

25.3.4. The LowerBoundary function

lowerboundary(value any_ordinal_or_array_type) **return integer**;

This function returns the lowest value that is of type ordinal, or array. For ordinal types, it returns the lowest value in the range of the type. For array types, it returns the lowest value within the length range of the index type of the array.

25.3.5. The UpperBoundary function

This function returns the highest value that is of type ordinal, or array. For ordinal types, it returns the highest value in the range of the type. For array types, it returns the highest value within the length range of the index type of the array. For empty arrays, it returns -1.

26. ATDL BNF and static semantics

This clause defines the syntax of ATDL using extended BNF (henceforth just called BNF).

26.1. ATDL grammars

The following conventions have been used when defining the ATDL/gr and the ATDL/pr grammar.

26.1.1. ATDL terminals

ATDL terminal symbols and reserved words are listed in Table 30.

Table 30: List of ATDL terminals which are reserved words

activate	extends	noblock	self
all	extension	none	send
alt	external	not	sender
altstep		null	sequence
and	fail		set
any	false	objid	setverdict
	final	octetstring	start
bind	float	of	stop
bitstring	for	omit	supports
boolean	from	operation	sutaction
break	function	optional	synchronize
		or	synchronized
call	get	out	system
cardinal	getverdict	overload	
catch	group	override	template
char			testcase
charstring	hexstring	pass	thread
choice		pattern	timeout
class	if	private	timer
clear	ifpresent	protected	trigger
co	implements	public	true
complement	import		try
const	in	raise	type
constructor	inconc	raises	
continue	infinity	read	uses
control	inherited	real	
create	inout	receive	value
	instanceof	recursive	var
deactivate	integer	release	variant
default	interface	rem	verdicttype
destructor		requires	virtual
display	label	return	
do	language	running	while
done			with
	members		wide
else	mod		write
encode	modifies		
enumerated	module		xor
error	mtc		
exception			

The following lists the special identifiers reserved for the predefined functions:

lengthof, **sizeof**, **ischosen**, **ispresent, setlength**

These terminals shall be written in all lowercase letters.

26.1.2. Meta-language for graphical grammar

The graphical syntax for ATDL/GR is defined on the basis of the graphical syntax of SDL-92 [17]. The graphical syntax definition uses a meta-language, which is explained in clause 5.4.3 of [17]. Syntactic categories are the non-terminals indicated by one or more words enclosed between angle brackets. See [17] for more details.

26.1.3. Static and dynamic objects

Making a static description of a system amounts to defining its architecture. A system defined by ATDL, which respects the syntactical rules and verifies the conditions of the static description is a *valid system*.

Table 31: ATDL Graphical Symbols for the Static Objects		
<frame symbol>	<group symbol>	<module symbol>
<thread class symbol>	<class symbol>	<coclass symbol>
<text symbol>	<comment symbol>	<interface symbol 1>
<dashed association symbol>	<dependency symbol>	<entity kind symbol> << >>
<component extends symbol>	<separator symbol>	<realization symbol>
<solid association symbol>	<channel symbol>	

An ATDL description always begins with the module object, which is the object of the highest hierarchical level in the description.

The aim of an ATDL module is to model a consistent set of communicating components grouped as groups. Groups are the main conceptual language elements of the module. Each group must be internally consistent and easy to understand.

The hierarchical decomposition of an ATDL module uses the structural objects listed in Table 31.

An ATDL model is described dynamically by communicating component instances. In ATDL, these component instances are described by functions and test cases. The description of an ATDL function or test case uses the dynamic objects listed in Table 32.

Table 32: ATDL Graphical Symbols for the Dynamic Objects

<internal input symbol>	<message in symbol>	<procedure in symbol>
<procedure call symbol>	<create request symbol>	<condition symbol>
<exception in symbol>	<decision symbol>	<return symbol>
<connector symbol>	<save symbol>	<stop symbol>
<statement start symbol>	<statement end symbol>	<repeat symbol>
<message out symbol>	<try symbol>	<internal output symbol>
<exception out symbol>	<function start symbol>	<inline expression symbol>
<reference symbol>	<default symbol>	<task symbol>
<flow line symbol>	<alt symbol>	

26.2. ATDL syntax BNF productions

This section defines the syntax of ATDL using extended BNF.

26.2.1. ATDL Module

1 ATDL_Module ::=ModuleHeading “{“ [ModuleDefinitionsPart] [ModuleControlPart] “}”

2 ModuleHeading ::= “module” ATDL_ModuleId [ModuleParList]

3 ATDL_ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]

4 DefinitiveIdentifier ::= Dot ObjectIdType “{“ DefinitiveObjIdComponentList “}”

5 DefinitiveObjIdComponentList ::= {DefinitiveObjIdComponent}+

6 DefinitiveObjIdComponent ::= NameForm | Number | NameForm "(" Number ")"

7 ModuleIdentifier ::= Identifier

8 ModuleParList ::= "(" ModulePar {"," ModulePar}* ")"

9 ModulePar ::= ["in"] ModuleParIdentifier Type [":=" ConstantExpression]

/* STATIC SEMANTICS - The Value of the ConstantExpression shall be of the same type as the stated type for the Parameter */

10 ModuleParIdentifier ::= Identifier

26.2.1.1. Concrete graphical grammar

11 <module diagram> ::= <frame symbol> **contains**
(ModuleHeading {{<module text area>}*
{<group diagram>}* {<group reference area>}*
<component interaction area> } **set**)
[**is_followed_by** <control part area>]

12 <module text area> ::= <text symbol> **contains**
{(SupporingDef | ImportDef | ExtFunctionDef) [SemiColon]}*

13 <group reference area> ::= <reference symbol> **contains** GroupHeading

14 <separator area> ::= <separator symbol>

15 <control part area> ::= <reference symbol> **contains** ("control" ATDL_ModuleId)

16 <component interaction area> ::= {<component area> | <component dependency area>
| <interface definition area>}+

17 <component area> ::= <component reference area> | <component diagram>

18 <component diagram> ::= <thread class diagram> | <class diagram> | <coclass diagram>

19 <component dependency area> ::= <dependency symbol>
is_connected_to (<component area> <component area>)

20 <component reference area> ::= <reference symbol> **contains** ComponentType
[**is_connected_to** <component extends area>]
[**is_connected_to** { <required interface area>+ } **set**]
[**is_connected_to** { <supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

26.2.2. Module Definitions Part

21 ModuleDefinitionsPart ::= {ModuleDefinition [SemiColon] }+

22 ModuleDefinition ::= (SupportingDef | TemplateDef | ImportDef | GroupDef | InterfaceDef
| FunctionDef | TestcaseDef | AltstepDef | ExtFunctionDef | ClassDef
| CoclassDef | ThreadClassDef | ClassTemplateDef) [WithStatement]

23 SupportingDef ::= TypeDef | ConstDef | ExceptionDef

26.2.2.1. Typedef Definitions

24 TypeDef ::= "type" TypeIdentifier "::=" Type

25 TypeIdentifier ::= Identifier

26 TypeDefFormalParList ::= "(" FormalValuePar {"," FormalValuePar}* ")"

/* STATIC SEMANTICS - FormalValuePar shall resolve to an in parameter */

27 SequenceType ::= "sequence" [TypeDefFormalParList] "{" [StructFieldDef {"," StructFieldDef}*] "}"

28 StructFieldDef ::= StructFieldIdentifier Type [SubTypeSpec] ["optional"]

29 StructFieldIdentifier ::= Identifier

30 SetType ::= "set" [TypeDefFormalParList] "{" [StructFieldDef {"," StructFieldDef}*] "}"

31 ChoiceType ::= "choice" "(" OrdinalType ")" "{" ChoiceFieldDef {";" ChoiceFieldDef}* "}"

32 ChoiceFieldDef ::= StructFieldIdentifier TaggedType [SubTypeSpec]

33 TaggedType ::= "[" (SingleConstExpression | "else") "]" Type

/* STATIC SEMANTICS - The value of the SingleConstExpression shall be of the same type as the TagTypeSpec. */

34 SequenceOfType ::= “sequence” [{LengthRestriction}+] “of” Type [SubTypeSpec]

35 SetOfType ::= “set” [{LengthRestriction}+] “of” Type [SubTypeSpec]

36 EnumType ::= “enumerated” “{“ NamedValue {“,” NamedValue}* “}”

37 NamedValue ::= NamedValueIdentifier [“(“ Number “)”]

38 NamedValueIdentifier ::= Identifier

39 ConstrainedType ::= BasicType [SubTypeSpec]

40 SubTypeSpec ::= SimpleValueSet | LengthRestriction

/* STATIC SEMANTICS - SimpleValueSet shall be of the same type as the field being subtyped */

41 SimpleValueSet ::= SimpleValueList | IntegerRange

42 SimpleValueList ::= “(“ SingleConstExpression {“,” SingleConstExpression}* “)”

/* STATIC SEMANTICS - SimpleValueList shall be of the same type as the field being subtyped */

43 IntegerRange ::= “(“ LowerBound “..” UpperBound “)”

/* STATIC SEMANTICS - IntegerRange shall only be used with ordinal types */

/* STATIC SEMANTICS - When subtyping charstring or wide charstring range and values shall not be mixed in the same SubTypeSpec */

44 LowerBound ::= SingleConstExpression | Minus “infinity”

45 UpperBound ::= SingleConstExpression | “infinity”

/* STATIC SEMANTICS - LowerBound and UpperBound shall evaluate to types integer, cardinal, char, or wide char. In case LowerBound or UpperBound evaluates to types char or wide char, only SingleConstExpression may be present */

46 LengthRestriction ::= “[“ SingleConstExpression [“..” UpperBound] “]”

/* STATIC SEMANTICS - LengthRestriction will resolve to a value of cardinal type. LengthRestriction shall only be used with String types, Integer types or to limit sequence of type */

26.2.2.2. Constant Definitions

47 ConstDef ::= “const” SingleConstDef {“,” SingleConstDef}*

48 SingleConstDef ::= ConstIdentifier Type “:=” ConstantExpression

/* STATIC SEMANTICS - The value of the ConstantExpression shall be of the same type as the stated type for the constant */

49 ConstIdentifier ::= Identifier

26.2.2.3. Template Definitions

50 TemplateDef ::= “template” BaseTemplate [DerivedDef] AssignmentChar TemplateBody

51 BaseTemplate ::= TemplateIdentifier [FormalCrefParList]
(MessageIdentifier | Operation | ExceptionTypeIdentifier)

52 TemplateIdentifier ::= Identifier

53 DerivedDef ::= “modifies” TemplateRef

54 FormalCrefParList ::= FormalCrefPar {“,” FormalCrefPar}*

55 FormalCrefPar ::= FormalValuePar | FormalTemplatePar | FormalTypePar

/* STATIC SEMANTICS - FormalValuePar shall resolve to an *in* parameter */

56 TemplateBody ::= TemplateValue&Attributes | FieldSpecList

57 TemplateValue&Attributes ::= TemplateValue [ValueAttributes]

/* NOTE - TemplateValue&Attributes can be reached via DefinedValue in the ATDL and the ASN.1 syntax. See the reference on the production 325 for Value. */
/* STATIC SEMANTICS - TemplateValue shall fulfil all restrictions defined for the ASP parameter, PDU field or structure element type, including value ranges, value lists, alphabet restrictions and/or length restrictions */

58 FieldSpecList ::= “{“ [FieldSpec {“,” FieldSpec}*] “}”

59 FieldSpec ::= FieldReference AssignmentChar TemplateBody

60 FieldReference ::= StructFieldIdentifier | ArrayOrBitRef | OperationParIdentifier

/* OPERATIONAL SEMANTICS - OperationParIdentifier shall be a formal parameter Identifier from the associated operation definition */

61 OperationParIdentifier ::= ValueParIdentifier

62 ArrayOrBitRef ::= “[“ FieldOrBitNumber “]”

/* STATIC SEMANTICS - ArrayRef shall be optionally used for array types and ASN.1 SET OF and SEQUENCE OF and ATDL sequence, array, and set type. The same notation can be used for a Bit reference inside an ASN.1 or ATDL bitstring type */

63 FieldOrBitNumber ::= SingleConstExpression

/* STATIC SEMANTICS - SingleConstExpression will resolve to a value of integer type */

64 TemplateValue ::= SingleConstExpression | MatchingSymbol | TemplateRefWithPara

/* STATIC SEMANTICS - VariableIdentifier (accessed via singleExpression) may only be used in inline template definitions to reference variables in the current scope */

65 MatchingSymbol ::= CharStringMatch | Omit | AnyValue | AnyOrOmit | ValueList | IntegerRange

66 ValueAttributes ::= LengthRestriction | “ifpresent” | LengthRestriction “ifpresent”

67 CharStringMatch ::= “pattern” Cstring

68 Omit ::= “omit”

/* STATIC SEMANTICS - In ATDL constraints Omit shall be used only for *sequence* field that are declared *optional* */

69 AnyValue ::= “?”

70 AnyOrOmit ::= “*”

71 ValueList ::= “(“ TemplateBody {“,” TemplateBody}* “)”

72 TemplateInstance ::= InLineTemplate

73 TemplateRefWithPara ::= [ModuleName Dot] TemplateIdentifier [ActualCrefParList] |
TemplateParIdentifier

74 TemplateRef ::= [ModuleName Dot] TemplateIdentifier | TemplateParIdentifier

75 InLineTemplate ::= [(MessageIdentifier | ExceptionTypeIdentifier) InLineMatchingSymbol]
[DereivedDef “:=”] TemplateBody

/* STATIC SEMANTICS - The MessageIdentifier or ExceptionTypeIdentifier field may only be omitted when the type is implicitly unambigous */

76 InLineMatchingSymbol ::= AssignmentChar | Colon | ">=" | "<=" | “==”

77 ActualCrefParList ::= “(“ ActualCrefPar {“,” ActualCrefPar}* “)”

78 ActualCrefPar ::= [VarIdentifier AssignmentChar] TemplateInstance | Type

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type, the TemplateInstance production shall resolve to one or more SingleExpressions */

79 TemplateOps ::= “value” “of” “(“ TemplateInstance “)” | “value”

26.2.2.4. Group Definitions

80 GroupDef ::= GroupHeading “{“
[SupportingDefSpec]
[InterfaceDefSpec]
[ComponentDefSpec]
MemberComponentList “}”

81 GroupHeading ::= “group” GroupIdentifier

82 GroupIdentifier ::= Identifier

83 SupportingDefSpec ::= { (SupportingDef | FunctionDef | TestcaseDef | AltstepDef) SemiColon }*

84 InterfaceDefSpec ::= {InterfaceDef SemiColon }*

85 ComponentDefSpec ::= { (CoclassDef | ClassDef | ThreadClassDef) SemiColon }*

86 MemberComponentList ::= “members” MemberComponentDef {“,” MemberComponentDef}* “;”

87 MemberComponentDef ::= GroupIdentifier | ComponentTypeIdentifier

Concrete graphical grammar

88 <group diagram> ::= <frame symbol>
contains {GroupHeading {{<group text area>}*
[<component interaction area>] {<group reference area>}*}**set**}

89 <group text area> ::= <text symbol> **contains** {(SupportingDef | InterfaceDef) [SemiColon]}*

26.2.2.5. Co-class Definitions

90 CoclassDef ::= CoclassHeading “{“
[ThreadPropertiesList]
[InterfaceDefSpec]
SupportedInterfaceList
[RequiredInterfaceList]
[ConstructorHeading] “}”

91 CoclassHeading ::= “co” “class” CoclassIdentifier [CoclassHeritage]

92 CoclassIdentifier ::= Identifier

93 CoclassHeritage ::= “extends“ CoclassIdentifier

94 SupportedInterfaceList ::= “supports” InterfaceType {“,” InterfaceType}* SemiColon

95 RequiredInterfaceList ::= “requires” InterfaceType {“,” InterfaceType}* SemiColon

Concrete graphical grammar

96 <coclass diagram> ::= <coclass symbol>
(CoclassIdentifier <class properties area> ConstructorHeading)
[**is_connected_to** <component extends area>]
[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

97 <component extends area> ::= <component extends symbol>
is_connected_to (<component area> <component area>)

98 <class properties area> ::= ({ ClassProperty SemiColon }*) **set**

26.2.2.6. Class Definitions

99 ClassDef ::= [(“virtual” | “final”)] “class” ClassIdentifier [ClassHeritage]

100 ClassDefBody ::= “{“ [ClassPropertiesList]
[ClassLocalInstList]
[InterfaceDefSpec]
[ImplementedInterfaceList]
[RequiredInterfaceList]
[ClassMethodList] “}”

101 ClassIdentifier ::= Identifier

102 ClassHeritage ::= “extends“ (ClassIdentifier | CoclassIdentifer)

103 ImplementedInterfaceList ::= “implements” InterfaceType {“,” InterfaceType}* “;”

104 ClassMethodList ::= {ClassMethodDef SemiColon}*

105 ClassMethodDef ::= ClassVisibility [Virtuality] MethodHeading [RaisesExpr]

106 Virtuality ::= “final” [“virtual”] | “override” | “overload” | “external” | “template”

107 MethodHeading ::= [“synchronized”] [“class”] RoutineHeading
| ConstructorHeading | DestructorHeading

108 RoutineHeading ::= FunctionHeading | TestcaseHeading | AltstepHeading

109 ClassLocalInstList ::= {(VarInstance | TimerInstance) SemiColon}*

110 ClassPropertiesList ::= { (SupportingDef | ClassFieldDef | DefaultAltstepDef) SemiColon}*

111 ClassFieldDef ::= ClassVisibility [“synchronized”] ClassFieldIdentifier VarInitializer

112 ClassVisibility ::= [“public” | “protected” | “private”]

113 ClassFieldIdentifier ::= Identifier

114 DefaultAltstepDef ::= ClassFieldIdentifier (MessageType | Operation) “default“ AltstepInstance

Concrete graphical grammar

115 <class diagram> ::= <class symbol> **contains**
(ClassIdentifier <class properties area> <class methods area>)
[**is_connected_to** <component extends area>]

[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

116 <class methods area> ::= ({ ClassMethodDef <end> }*) **set**

26.2.2.7. Class Template Definitions

117 ClassTemplateDef ::= “template” “class” ClassTemplateIdentifier
[FormalParList] [ClassHeritage] ClassDefBody

118 ClassTemplateIdentifier ::= Identifier

119 ClassTemplateInstance ::= ClassTemplateIdentifier ActualParList

26.2.2.8. Thread Class Definitions

120 ThreadClassDef ::= ThreadClassHeading “{“
[SupportingDefSpec]
[{ClassFieldSpec [SemiColon]}*]
[ClassLocalInstList]
[InterfaceDefSpec]
[ImplementedInterfaceList]
[RequiredInterfaceList]
[ThreadMethodList] “}”

121 ThreadClassHeading ::= “thread” ThreadClassIdentifier [ThreadClassHeritage]

122 ThreadClassHeritage ::= “extends“ (ThreadClassIdentifer | CoclassIdentifer)

123 ThreadClassIdentifier ::= Identifier

124 ThreadMethodList ::= { [Virtuality] (RoutineHeading | ConstructorHeading) SemiColon}*

125 ComponentTypeIdentifier ::= ThreadClassIdentifier | ClassInstance | CoclassIdentifier

126 ClassInstance ::= ClassIdentifier | ClassTemplateInstance

127 ComponentType ::= [ModuleName Dot] ComponentTypeIdentifier

Concrete graphical grammar

128 <thread class diagram> ::= <thread class symbol> **contains**
(ThreadClassIdentifier <class properties area> <class methods area>)
[**is_connected_to** <component extends area>]
[**is_connected_to** {<required interface area>+ } **set**]
[**is_connected_to** {<supported interface area>+ } **set**]
[**is_connected_to** { <dependency symbol>+ } **set**]

26.2.2.9. Interface Definitions

129 InterfaceDef ::= MsgInterfaceDef | CpOpInterfaceDef | CoOpInterfaceDef

130 MsgInterfaceDef ::= MessageInterfaceHeader MessageAttribs

131 MsgInterfaceHeader ::= [“co”] “interface” MsgInterfaceTypeIdentifier [MsgInterfaceHeritage]

132 MsgInterfaceTypeIdentifier ::= Identifier

133 MsgInterfaceHeritage ::= “extends“ MsgInterfaceTypeIdentifier {“,” MsgInterfaceTypeIdentifier}*

134 CpOpInterfaceDef ::= CpOpInterfaceHeader OperationAttribs

135 CpOpInterfaceHeader ::= “interface” CpOpInterfaceTypeIdentifier [CpOpInterfaceHeritage]

136 CpOpInterfaceTypeIdentifier ::= Identifier

137 CpOpInterfaceHeritage ::= “extends“ CpOpInterfaceTypeIdentifier {“,” CpOpInterfaceTypeIdentifier}*

138 CoOpInterfaceDef ::= CoOpInterfaceHeader OperationAttribs

139 CoOpInterfaceHeader ::= “co” “interface” CpOpInterfaceTypeIdentifier [CoOpInterfaceHeritage]

140 CoOpInterfaceIdentifier ::= Identifier

141 CoOpInterfaceHeritage ::= “extends“ CoOpInterfaceTypeIdentifier {“,” CoOpInterfaceTypeIdentifier}*

142 InterfaceTypeIdentifier ::= MsgInterfaceTypeIdentifier | CpOpInterfaceTypeIdentifier | CoOpInterfaceTypeIdentifier

143 InterfaceType ::= [ComponentType Dot] InterfaceTypeIdentifier

144 MessageAttribs ::= "{" {MessageList [SemiColon]}+ "}"

145 MessageList ::= [Direction] MessageIdentifier Type

146 MessageIdentifier ::= Identifier

147 OperationAttribs ::= "{" {OperationDef [SemiColon]}+ "}"

Concrete graphical grammar

148 <required interface area> ::= <dependency symbol>
is_connected_to (<component area> <interface area>)

149 <supported interface area> ::= <channel symbol>
is_connected_to (<interface area> <component area>)

150 <channel symbol> ::= <channel symbol 1> | <channel symbol 2> | <channel symbol 3>

151 <channel symbol 1> ::= <solid association symbol>

152 <interface area> ::= (<interface area 1> | <interface area 2>)
is_connected_to (<dependency symbol> <channel symbol>)
[**is_connected_to** {<interface extends area>*} **set**]

153 <interface extends area> ::= <interface extends symbol>
is_connected_to (<interface area> <interface area>)

154 <interface extends symbol> ::= <component extends symbol>

155 <interface area 1> ::= <interface symbol 1> **contains**
(<interface heading> (OperationAttribs | MessageAttribs))

156 <interface heading> ::= (<entity kind symbol> **contains** [**co**] **interface**) InterfaceIdentifier

157 <interface area 2> ::= <interface symbol 2> **is_associated_with** InterfaceIdentifier

158 <interface symbol 2> ::= <connector symbol>

26.2.2.10. Constructors and destructors

159 ConstructorHeading ::= "constructor" [QualifierId] ConstructorIdentifier [FormalParList]

160 ConstructorIdentifier ::= Identifier

161 DestructorHeading ::= "destructor" [QualifierId] DestructorIdentifier [FormalParList]

162 DestructorIdentifier ::= Identifier

26.2.2.11. Function Definitions

163 FunctionDef ::= MethodModifier FunctionHeading
| ConstructorHeading | DestructorHeading) StatementBlock

164 FunctionHeading ::= "function" [QualifierId] FunctionIdentifier [FormalParList] [ReturnType]

165 FunctionIdentifier ::= Identifier

166 MethodModifier ::= "overload" | "template | "class"

167 FormalParList ::= "(" FormalPar&Type {"," FormalPar&Type}* ")"

168 FormalPar&Type ::= FormalValuePar | FormalTimerPar | FormalTypePar
| FormalTemplatePar | FormalInterfacePar

/* STATIC SEMANTICS - In an operation or an encoding operation FormalPar&Type shall not be an Interface type */

169 ReturnType ::= "return" Type

170 QualifierId ::= (ComponentType | InterfaceIdentifier) Dot

171 FunctionInstance ::= FunctionRef [ActualParList]

172 FunctionRef ::= [DataObjectReference Dot]
(FunctionIdentifier | DestructorIdentifier) | PreDefFunctionIdentifier

/* STATIC SEMANTICS - the variable associated with DataObjectReference must be of class type or component instance type */

173 PreDefFunctionIdentifier ::= Identifier

/* STATIC SEMANTICS - The Identifier will be one of the pre-defined ATDL Function Identifiers */

174 ActualParList ::= "(" ActualPar {"," ActualPar}* ")"

175 ActualPar ::= TimerRef | TemplateInstance | Type | Channel | ComponentRef

/* STATIC SEMANTICS - When the corresponding formal parameter is not of template type the TemplateInstance production shall resolve to one or more Expressions. */

Concrete graphical grammar

176 <function diagram> ::= <frame symbol>
contains (["overload"] FunctionHeading
{{<function text area>}* <function graph area> } **set**)

177 <function text area> ::= <control text area>

178 <function graph area> ::= <function start area> **is_followed_by** <function block area>

179 <function block area> ::= <statement block area>
is_followed_by (<stop symbol> | <statement end symbol>)

180 <function start area> ::= <function start symbol> **contains** ([Virtuality])

181 <function instance area> ::= <reference symbol> **contains** [VarPrefix] FunctionInstance

26.2.2.12. Operation Definitions

182 OperationDef ::= [OpAttribute] "operation" OperationIdentifier [FormatParList] [ReturnType] [RaisesExpr]

183 OperationIdentifier ::= Identifier

184 OpAttribute ::= "noblock" | "template"

185 Operation ::= [ModuleName Dot] OperationIdentifier

186 RaisesExpr ::= "raises" "(" ExceptionName {"," ExceptionName}* ")"

26.2.2.13. Exception Definitions

187 ExceptionDef ::= "exception" ExceptionIdentifier "{" {ExceptionMember}* "}"

188 ExceptionIdentifier ::= Identifier

189 ExceptionMember ::= ExceptionTypeIdentifier Type [SemiColon]

190 ExceptionTypeIdentifier ::= Identifier

191 ExceptionName ::= [ModuleName Dot] ExceptionIdentifier

Concrete graphical grammar

192 <exception area> ::= <interface symbol 1> **contains** (<exception heading> {ExceptionMember}*)

193 <exception heading> ::= (<entity kind symbol> **contains exception**) ExceptionIdentifier

26.2.2.14. Testcase Definitions

194 TestcaseDef ::= MethodModifier TestcaseHeading StatementBlock

195 TestcaseHeading ::= "testcase" QualifierId TestcaseIdentifier [FormalCrefParList]

196 TestcaseIdentifier ::= Identifier

197 TestcaseInstance ::= TestcaseRef [ActualCrefParList]

198 TestcaseRef ::= [DataObjectReference Dot] TestcaseIdentifier

/* STATIC SEMANTICS - the variable associated with DataObjectReference must be of class type or component instance type */

Concrete graphical grammar

199 <testcase diagram> ::= <frame symbol>
contains (["overload"] TestcaseHeading
{{<function text area>}* <testcase graph area>} **set**)

200 <testcase graph area> ::= <function start area> **is_followed_by** <testcase block area>

201 <testcase block area> ::= <statement block area> **is_followed_by** <statement end symbol>

202 <testcase instance area> ::= <reference symbol> **contains** [VarPrefix] TestcaseInstance

26.2.2.15.Altstep Definitions

203 AltstepDef ::= [“overload”] AltstepHeading “{“ AltGuardList “}”

204 AltstepHeading ::= “altstep” [QualifierId] AltstepIdentifier [FormalParList]

/* STATIC SEMANTICS - `all formal parameter must be value parameters i.e., in parameters` */

205 AltstepIdentifier ::= Identifier

206 AltstepInstance ::= AltstepRef [ActualParList]

207 AltstepRef ::= [ModuleName Dot] AltstepIdentifier

Concrete graphical grammar

208 <altstep diagram> ::= <frame symbol>
contains ([“overload”] AltstepHeading <altstep body area>)

209 <altstep body area> ::= <fgr alt area>

210 <altstep instance area> ::= <reference symbol> **contains** AltstepInstance

26.2.2.16.Import Definitions

211 ImportDef ::= “import” ModuleId (ImportSpec | “{“ {ImportSpec [SemiColon]}* “}”) [“recursive”]

212 ImportSpec ::= ImportAllSpec | ImportGroupSpec | ImportInterfaceSpec | ImportConstSpec
ImportComponentSpec | ImportTypeDefSpec | ImportTemplateSpec |
| ImportTestcaseSpec | ImportFunctionSpec | ImportAltstepSpec

213 ImportAllSpec ::= [DefKeyword] Dot “*”

214 ModuleId ::= ModuleName [“language” FreeText]

/* STATIC SEMANTICS - LanguageSpec may only be omitted if the referenced module contains ATDL notation */

215 ModuleName ::= GlobalModuleId | LocalModuleId

216 GlobalModuleId ::= ModuleIdentifier [Dot ObjectIdentifierValue]

217 LocalModuleId ::= ModuleIdentifier {Dot GroupIdentifier}*

218 DefKeyword ::= “type” | “const” | “class” | “interface” | “template” | “testcase” | “function” | “altstep”

219 ImportGroupSpec ::= “group” GroupIdentifier {“,” GroupIdentifier}*

220 ImportInterfaceSpec ::= “interface” InterfaceIdentifier {“,” InterfaceIdentifier}*

221 ImportComponentSpec ::= “class” ComponentTypeIdentifier {“,” ComponentTypeIdentifier}*

222 ImportTypeDefSpec ::= “type” TypeIdentifier {“,” TypeIdentifier}*

223 ImportTemplateSpec ::= “template” TemplateIdentifier {“,” TemplateIdentifier}*

224 ImportConstSpec ::= “const” ConstIdentifier {“,” ConstIdentifier}*

225 ImportTestcaseSpec ::= “testcase” TestcaseIdentifier {“,” TestcaseIdentifier}*

226 ImportFunctionSpec ::= “function” FunctionIdentifier {“,” FunctionIdentifier}*

227 ImportAltstepSpec ::= “altstep” AltstepIdentifier {“,” AltstepIdentifier}*

26.2.3. Control Part

228 ModuleControlPart ::= “control” “{“ ModuleControlBody “}” [WithStatement] [SemiColon]

229 ModuleControlBody ::= [ControlStatementOrDefList [“stop”] | “stop”]

230 ControlStatementOrDefList ::= {ControlStatementOrDef [SemiColon]}+

231 ControlStatementOrDef ::= ControlStatement | ClassLocalInst | ConstDef

26.2.3.1. Control Diagram

232 <control diagram> ::= <frame symbol> **contains** ((“control” ATDL_ModuleId)
({<control text area>}* <control graph area>) **set**)

233 <control text area> ::= <text symbol> **contains**
({ATDLComments}* [MultiWithAttrib] { ATDLComments}*)

234 <control graph area> ::= <statement start symbol> **is_followed_by** <control block area>

235 <control block area> ::= [<control statement block area> **is_followed_by**]

(<stop symbol> | <statement end symbol>)

236 <control statement block area> ::= <control statement area>
[**is_followed_by** <control statement block area>]

26.2.3.2. Variable Instantiation

237 VarInstance ::= “var” SingleVarInstance {“,” SingleVarInstance}*

238 SingleVarInstance ::= VarIdentifier {Colon VarIdentifier}* VarInitializer

239 VarInitializer ::= Type [AssignmentChar Expression]

240 VarIdentifier ::= Identifier

241 VariableRef ::= (VarIdentifier | ValueParIdentifier | ClassFieldIdentifier) [ExtendedFieldReference]

242 VarPrefix ::= [“var”] VarIdentifier Type AssignmentChar

26.2.3.3. Timer Instantiation

243 TimerInstance ::= “timer” TimerIdentifier [AssignmentChar TimerValue]

244 TimerIdentifier ::= Identifier

245 TimerValue ::= SingleConstExpression

/* STATIC SEMANTICS - SingleConstExpression must resolve to a value of type float */

246 TimerRef ::= TimerIdentifier | TimerParIdentifier

26.2.3.4. Component Operations

247 ConfigurationStatement ::= DoneStatement | StartThreadStatement | StopThreadStatement

248 ConfigurationOps ::= CreateOp | ComponentIdExpression | ThreadRunningOp

249 ComponentIdExpression ::= “system” | “self” | “mtc” | “sender” | “inherited”

250 CreateOp ::= (ComponentType Dot | “inherited”) (“create” | ConstructorIdentifier [ActualParList])

251 ComponentIdentifier ::= VariableRef | FunctionInstance | CastExpression

/* STATIC SEMANTICS - the variable associated with VariableRef or the Return type associated with FunctionInstance and CastExpression must be of component type */

252 ThreadIdentifier ::= ComponentIdentifier

253 DoneStatement ::= ThreadId Dot “done”

254 ThreadId ::= ThreadIdentifier | (“any” | “all”) “thread”

255 ThreadRunningOp ::= ThreadId Dot “running”

256 StartThreadStatement ::= ThreadIdentifier Dot “start” “(“ FunctionInstance “)”

/* STATIC SEMANTICS - the Function instance may only have in parameters */

/* STATIC SEMANTICS - the Function instance shall not have timer parameters */

257 StopThreadStatement ::= ThreadIdentifier Dot “stop” | “all” “thread” Dot “stop”

258 ComponentRef ::= ComponentIdentifier | ComponentIdExpression | ClassInstance

Concrete graphical grammar

259 <configuration statement area> ::= <create request area> | <fgr done area> |
<start thread area> | <stop thread area> | <stop symbol>

260 <create request area> ::= <create request symbol> **contains** [VarPrefix] CreateOp

261 <fgr done area> ::= <condition symbol> **contains** DoneStatement

262 <start thread area> ::= <procedure call symbol> **contains** StartThreadStatement

263 <stop thread area> ::= <stop symbol> [**is_associated_with** (ComponentRef | “all”)]

264 <fgr thread running area> ::= <condition symbol> **contains** ThreadRunningOp

26.2.3.5. Communication Operations

265 Channel ::= ChannelIdentifier | VarIdentifier | InterfaceParIdentifier | CastExpression

/* STATIC SEMANTICS - the variable associated with VarIdentifier and CastExpression must be of interface type*/

266 CommunicationStatement ::= SendStatement | CallStatement | RaiseStatement |

ReceiveStatement | TriggerStatement |
SynchronizeStatement | CatchStatement

267 SendStatement ::= Channel Dot “send” “(“ TemplateInstance “)”

268 CallStatement ::= Channel Dot “call” OperationRefWithPara

269 OperationRefWithPara ::= (Operation | TemplateIdentifier) [ActualCrefParList]

/* STATIC SEMANTICS - only out parameters may be omitted or specified with a matching attribute */

270 RaiseStatement ::= Channel Dot “raise” “(“ TemplateInstance “)”

271 ReceiveStatement ::= ChannelOrAny Dot “receive” ReceiveParameter [AssignmentList]

/* STATIC SEMANTICS - The AssignmentList option may only be present if the TemplateInstance option is also present */

272 ReceiveParameter ::= [“(“ TemplateInstance “)”]

273 ChannelOrAny ::= Channel | “any” “interface”

274 ChannelOrAll ::= Channel | “all” “interface”

275 TriggerStatement ::= ChannelOrAny Dot “trigger” ReceiveParameter [AssignmentList]

/* STATIC SEMANTICS - The AssignmentList option may only be present if the TemplateInstance option is also present */

276 SynchronizeStatement ::= ChannelOrAny Dot “synchronize” [“(“ TemplateInstance “)”]

277 CatchStatement ::= ChannelOrAny Dot “catch” [“(“ CatchOpParameter “)”] [AssignmentList]

/* STATIC SEMANTICS - The AssignmentList option may only be present if the CatchOpParameter option is also present */

278 CatchOpParameter ::= TemplateInstance | “timeout” | “all”

Concrete graphical grammar

279 <communication statement area> ::= <fgr call area> | <fgr send area> | <fgr raise area> |
<fgr receive area> | <fgr trigger area> |
<fgr synchronize area> | <fgr catch area>

280 <fgr send area> ::= <message out symbol> **contains** ([Channel Dot] TemplateInstance)

281 <fgr call area> ::= <procedure call symbol> **contains** [VarPrefix] CallStatement
[**is_associated_with** <save area>]

282 <fgr raise area> ::= <exception out symbol> **contains** ([Channel Dot] TemplateInstance)

283 <fgr receive area> ::= <message in symbol> **contains** ([ChannelOrAny Dot] ReceiveParameter)
[**is_associated_with** <save area>]

284 <save area> ::= <save symbol> **contains** (Assignment {SemiColon Assignment}*)

285 <fgr trigger area> ::= <message in symbol> **contains** TriggerStatement
[**is_associated_with** <save area>]

286 <fgr synchronize area> ::= <procedure in symbol> **contains** SynchronizeStatement
[**is_associated_with** <save area>]

287 <fgr catch area> ::= <exception in symbol>
contains [[ChannelOrAny Dot]“(“ CatchOpParameter “)”]
[**is_associated_with** <save area>]

26.2.3.6. Interface Operations

288 BindOp ::= “bind” “(“ (InterfaceType | ExceptionIdentifier) “,” ComponentRef “)”

289 ReleaseStatement ::= ChannelIdentifier Dot “release”

290 ChannelStartStatement ::= ChannelOrAll Dot “start”

291 ClearStatement ::= ChannelOrAll Dot “clear”

292 ChannelStopStatement ::= ChannelOrAll Dot “stop”

Concrete graphical grammar

293 <channel controlling area> ::= <condition symbol> **contains**
(ClearStatement | ChannelStopStatement | ChannelStartStatement)

26.2.3.7. Timer Operations

294 TimerStatement ::= StartTimerStatement | StopTimerStatement | TimeoutStatement

295 TimerOps ::= ReadTimerOp | RunningTimerOp

296 StartTimerStatement ::= TimerRef Dot “start” [“(“ TimerValue “)”]

297 StopTimerStatement ::= TimerRefOrAll Dot “stop”

298 TimerRefOrAll ::= TimerRef | “all” “timer”

299 ReadTimerOp ::= TimerRef Dot “read”

300 RunningTimerOp ::= TimerRefOrAny Dot “running”

301 TimeoutStatement ::= TimerRefOrAny Dot "timeout"

/* STATIC SEMANTICS - The TimerRef may only be omitted within the exception handling part of a call. */

302 TimerRefOrAny ::= TimerRef | “any” “timer”

Concrete graphical grammar

303 <timer statement area> ::= <fgr timer start area> | <fgr timer stop area> | <fgr timeout area> | <fgr timer running area>

304 <fgr timer start area> ::= <internal output symbol> **contains** StartTimerStatement

305 <fgr timer stop area> ::= <internal output symbol> **contains** StopTimerStatement

306 <fgr timeout area> ::= <internal input symbol> **contains** TimeoutStatement

307 <fgr timer running area> ::= <condition symbol> **contains** RunningTimerOp

26.2.4. Type

308 Type ::= BasicType | ConstrainedType | StructuredType | ReferencedType | RestrictedType

309 BasicType ::= OrdinalType | “float” | RealType | StringType | “objid” | VerdictType | BooleanType

310 StructuredType ::= SequenceType | SequenceOfType | SetType | SetOfType | ChoiceType

311 OrdinalType ::= IntegerType | CharType | WideChar | EnumType

312 StringType ::= “bitstring” | CharStringType | WideCharString | “octetstring” | “hexstring”

313 IntegerType ::= (“integer” | “cardinal”) [LengthRestriction]

/* STATIC SEMANTICS - The length restriction may only be omitted when used as generic-type template parameter or return type associated with a procedure template. */

314 BooleanType ::= “boolean”

315 RealType ::= “real” [LengthRestriction]

/* STATIC SEMANTICS - The length restriction may only be omitted when used as generic-type template parameter or return type associated with a procedure template. */

316 CharType ::= “char”

317 WideChar ::= “wide” “char”

318 CharStringType ::= “charstring”

319 WideCharString ::= “wide” “charstring”

320 ReferencedType ::= [GlobalModuleId Dot] TypeReference [ExtendedFieldReference]

321 TypeReference ::= TypeIdentifier TypeActualParList | ValueParIdentifier

/* STATIC SEMANTICS - ValueParIdentifier must be previously defined generic type template parameter */

322 RestrictedType ::= ComponentTypeIdentifier | ClassRefType
| InterfaceType | ExceptionIdentifier | “default“ | “variant”

323 ClassRefType ::= “class” “of” ClassIdentifer

324 TypeActualParList ::= “(“ SingleConstExpression {“,” SingleConstExpression}* “)”

26.2.5. Value

325 Value ::= LiteralValue | StringValue | ReferencedValue | TemplateValue&Attributes

/* REFERENCE - For the purposes of ATDL tabular presentation format this production is redefined to be: Value ::= LiteralValue | ReferencedValue | ASN1_Value, Where ASN1_Value is Value as defined in ISO/IEC 8824 */

/* In ISO/IEC 8824 the production DefinedValue is defined as:
DefinedValue ::= Externalvaluereference | valuereference.
For the purposes of ATDL this production is redefined to be:
DefinedValue ::= TemplateValue&Attributes.

Note that this means that external value references are not allowed in ATDL */

326 LiteralValue ::= BooleanValue | ChoiceValue | IntegerValue | FloatingPointLiteral | CharValue
| ObjectIdentifierValue | EnumeratedValue | VerdictValue | NullValue

327 StringValue ::= Bstring | CharStringValue | Ostring | Hstring

328 BooleanValue ::= "true" | "false"

329 ChoiceValue ::= StructFieldIdentifer Colon Value

330 IntegerValue ::= Number

331 ObjectIdentifierValue ::= "{" ObjIdComponentList "}"

/* STATIC SEMANTICS - ReferencedValue shall be of type object Identifier */

332 ObjectIdComponentList ::= {ObjIdComponent}+

333 ObjIdComponent ::= NameForm | NumberForm | NameAndNumberForm

334 NumberForm ::= Number | ReferencedValue

/* STATIC SEMANTICS - ReferencedValue shall be of type cardinal and have a non negative Value */

335 NameAndNumberForm ::= Identifier NumberForm

336 NameForm ::= Identifier

337 VerdictValue ::= "pass" | "fail" | "inconc" | "none" | "error"

338 EnumeratedValue ::= NamedValueIdentifier

339 CharStringValue ::= Cstring | Quadruple

340 UnicodeInputCharacter ::= UnicodeEscape | Char

341 UnicodeEscape ::= \ UnicodeMarker Hex Hex Hex Hex

342 UnicodeMarker ::= {u}+

343 InputCharacter ::= UnicodeInputCharacter

/* STATIC SEMANTICS - The InputCharacter shall not be Carriage Return or Line Feed character */

344 FloatingPointLiteral ::= (FloatDotNotation | FloatENotation) [FloatTypeSuffix]

345 FloatDotNotation ::= Number Dot DecimalNumber

346 FloatENotation ::= Number [Dot DecimalNumber] Exponential [Minus] Number

347 Exponential ::= "e" | "E"

348 FloatTypeSuffix ::= "f" | "F" | "r" | "R"

349 ReferencedValue ::= [GlobalModuleId Dot] ValueReference [ExtendedFieldReference]

350 ValueReference ::= ConstIdentifier | ValueParIdentifier | ModuleParIdentifier | VarIdentifier

351 Number ::= (NonZeroNum {Num}*) | "0"

352 NonZeroNum ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

353 DecimalNumber ::= {Num}*

354 Num ::= "0" | NonZeroNum

355 CharValue ::= "'" Char "'" ["C"]

356 Bstring ::= " ' " {Bin | Wildcard}* " ' " "B"

357 Bin ::= "0" | "1"

358 Hstring ::= " ' " {Hex | Wildcard}* " ' " "H"

359 Hex ::= Num | "A" | "B" | "C" | "D" | "E" | "F"| "a" | "b" | "c" | "d" | "e" | "f"

360 Ostring ::= " ' " {Oct | Wildcard}* " ' " "O"

361 Oct ::= Hex Hex

362 Cstring ::= " " " {InputCharacter | Wildcard | "\"}* " " "

363 Char ::= /* REFERENCE - A character defined by the relevant CharacterString type. For charstring a character from the character set defined in ISO/IEC 646. For wide charstring a character from any character set defined in ISO/IEC 10646 */

364 Wildcard ::= AnyOne | AnyOrNone

365 AnyOne ::= "?"

/* STATIC SEMANTICS - AnyOne shall be used only within values of string types, and *sequence of*. */

366 AnyOrNone ::= “*”

/* STATIC SEMANTICS - AnyOrNone shall be used only within values of string types, and *sequence of*.*/

367 Identifier ::= Alpha {AlphaNum | “_”}*

368 Alpha ::= UpperAlpha | LowerAlpha

369 AlphaNum ::= Alpha | Num

370 UpperAlpha ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |"N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

371 LowerAlpha ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |"n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

372 ExtendedAlphaNum ::= /* REFERENCE - A graphical character from the BASIC LATIN or from the LATIN-1 SUPPLEMENT character sets defined in ISO/IEC 10646 (characters from char (0,0,0,33) to char (0,0,0,126), from char (0,0,0,161) to char (0,0,0,172) and from char (0,0,0,174) to char (0,0,0,255) */

373 FreeText ::= “ “ “ {ExtendedAlphaNum}* “ “ “

374 NullValue ::= “null”

26.2.6. Parameterisation

375 Direction ::= “in” | “out” | “inout”

376 FormalValuePar ::= [Direction] ValueParIdentifier {“:” ValueParIdentifier}* Type

377 ValueParIdentifier ::= Identifier

378 FormalTypePar ::= [Direction] ValueParIdentifier

/* STATIC SEMANTICS - ValueParIdentifier must be previously defined generic type template parameter */

379 FormalInterfacePar ::= [“inout”] InterfaceParIdentifier InterfaceTypeIdentifier

380 InterfaceParIdentifier ::= Identifier

381 FormalTimerPar ::= [“inout”] “timer” TimerParIdentifier

382 TimerParIdentifier ::= Identifier

383 FormalTemplatePar ::= [“in”] “template” TemplateParIdentifier Type

384 TemplateParIdentifier ::= Identifier

26.2.7. With Statement

385 WithStatement ::= “with” [“tabular”] WithAttribList

386 WithAttribList ::= SingleWithAttrib SemiColon | "{" MultiWithAttrib "}" [SemiColon]

387 MultiWithAttrib ::= SingleWithAttrib {SemiColon SingleWithAttrib} [SemiColon]

388 SingleWithAttrib ::= [“encode” | “display” | “extension”] [“override”] [AttribQualifier] AttribSpec

389 AttribQualifier ::= “(“ DefOrFieldRefList “)”

390 DefOrFieldRefList ::= DefOrFieldRef {“,” DefOrFieldRef}*

391 DefOrFieldRef ::= DefinitionRef | FieldReference

392 DefinitionRef ::= TypeIdentifier | InterfaceIdentifier | ComponentTypeIdentifier |
ConstIdentifier | TemplateIdentifier | AltstepIdentifier |
TestcaseIdentifier | FunctionIdentifier | OperationIdentifier

393 AttribSpec ::= FreeText

26.2.8. Statement Blocks

394 StatementBlock ::= “{“ BlockStatement [TerminatorStatement] | TerminatorStatement “}”

395 BlockStatement ::= { ActionStatement [SemiColon]}+

396 ActionStatement ::= ControlStatement | ConfigurationStatement |
CommunicationStatement | SetLocalVerdict

397 TerminatorStatement ::= ReturnStatement | “stop” | RaiseStatement

Concrete graphical grammar

398 <statement block area> ::= [<action statement area> **is_followed_by**]
{ <return area> | <stop symbol> | <fgr raise area>}

399 <action statement area> ::= <control statement area> | <configuration statement area> |
<communication statement area> | <set verdict area>

26.2.9. Behavior Statements

400 ControlStatement ::= TestcaseInstance | FunctionInstance | AltConstruct | LoopConstruct
| DecisionConstruct | ActivateStatement | DeactivateStatement
| ChoiceConstruct | AltstepInstance | BasicStatement | TaskStatement

Concrete graphical grammar

401 <control statement area> ::= <testcase instance area> | <function instance area>
| <fgr inline expression area> | <default area>
| <altstep instance area> | <choice area>
| <decision area> | <task area>

26.2.9.1. Task

402 TaskStatement ::= Assignment | ConstDef | VarInstance | WriteStatement | BindStatement
| ReleaseStatement | SUTStatement

403 WriteStatement ::= “write” “(“ [Cstring] “)”

404 SUTStatement ::= “sutaction” “(“ (FreeText | TemplateRefWithPara) “)”

Concrete graphical grammar

405 <task area> ::= <task symbol> **contains** ({TaskStatement [SemiColon]}+)

26.2.9.2. Verdict Statement

406 SetLocalVerdict ::= “setverdict” “(“ SingleExpression “)”
/* STATIC SEMANTICS - SingleExpression must resolve to a value of type verdict */
/* STATIC SEMANTICS - The SetLocalVerdict shall not be used to assign the value *error*. */

407 GetLocalVerdict ::= “getverdict”

Concrete graphical grammar

408 <set verdict area> ::= <condition symbol> **contains** VerdictValue

26.2.9.3. Return

409 ReturnStatement ::= “return” [Expression]

Concrete graphical grammar

410 <return area> ::= <return symbol> [**is_associated_with** Expression]

26.2.9.4. Alternative behavior

411 AltConstruct ::= “alt” “{“ AltGuardList “}”

412 AltGuardList ::= {GuardStatement [SemiColon]}+ [ElseClause [SemiColon]]

413 GuardStatement ::= AltGuardChar (AltstepInstance | GuardOp StatementBlock)

414 ElseClause ::= “[“ “else” “]” StatementBlock

415 AltGuardChar ::= “[“ [BooleanExpression] “]”

416 GuardOp ::= TimeoutStatement | ReceiveStatement | TriggerStatement |
SynchronizeStatement | CatchStatement | DoneStatement
/* STATIC SEMANTICS - GuardOp used within the module control part. Shall only contain the TimeoutStatement */

Concrete graphical grammar

417 <fgr alt area> ::= <alt symbol> **is_followed_by**
((<graphical guard part>}+ [<alt else part>]) **set**)

418 <alt outlet symbol> ::= <alt symbol>

419 <alt else part> ::= <flow line symbol>
is_associated_with ("[" **else** "]")
is_followed_by <statement block area>
is_connected_to <alt outlet symbol>

420 <graphical guard part> ::= <flow line symbol>
[**is_associated_with** GuardCondition]
is_followed_by <fgr guard area>
is_followed_by <statement block area>
is_connected_to <alt outlet symbol>

421 <fgr guard area> ::= <fgr receive area> | <fgr trigger area> |
<fgr synchronize area> | <fgr catch area> |
<fgr timeout area> | <fgr done area> | <altstep instance area>

26.2.9.5. The Activate and Deactivate statements

422 ActivateStatement ::= "activate" "(" AltstepInstance ")"

423 DeactivateStatement ::= "deactivate" ["(" `Expression` ")"]
/* STATIC SEMANTICS - expression shall evaluate to a value of default type */

Concrete graphical grammar

424 <default area> ::= <default symbol> **contains** (ActivateStatement | DeactivateStatement)

26.2.10. Basic Statements

425 BasicStatement ::= TimerStatement | BreakStatement | ContinueStatement | TryStatement

Concrete graphical grammar

426 <basic statement area> ::= <timer statement area> | <continue area> |
<break area> | <try statement area>

26.2.10.1.Loop Construct

427 LoopConstruct ::= ForStatement | WhileStatement | DoWhileStatement | LabeledStatement

428 ForStatement ::= "for" LoopCondition StatementBlock

429 LoopCondition ::= "(" ForInit SemiColon BooleanExpression SemiColon ForUpdate ")"

430 ForInit ::= SingleVarInstance | Assignment

431 ForUpdate ::= Assignment

432 WhileStatement ::= "while" GuardCondition StatementBlock

433 DoWhileStatement ::= "do" StatementBlock "while" GuardCondition

434 LabeledStatement ::= "label" LabelIdentifier StatementBlock

435 LabelIdentifier ::= Identifier

436 BreakStatement ::= [GuardCondition] "break" [LabelIdentifier]

437 ContinueStatement ::= [GuardCondition] "continue" [(LabelIdentifier | "alt")]
/* STATIC SEMANTICS - The *alt* option may only be used within an *alt* construct. */

Concrete graphical grammar

438 <fgr inline expression area> ::= <fgr for area> | <fgr while area>
| <fgr do-while area> | <fgr labeled area> | <fgr opt area>

439 <fgr for area> ::= <inline expression symbol> **contains**
(**for** LoopCondition <statement block area>)

440 <fgr while area> ::= <inline expression symbol> **contains**
(**while** GuardCondition <statement block area>)

441 <fgr do-while area> ::= <inline expression symbol> **contains**
(**do while** GuardCondition <statement block area>)

442 <fgr labeled area> ::= <inline expression symbol> **contains**
(**label** LabelIdentifier <statement block area>)

443 <continue area> ::= <repeat symbol> [**is_associated_with** LabelIdentifier]
[**is_associated_with** GuardCondition]

444 <flow line symbol> ::= <flow line symbol1> | <flow line symbol2>

445 <break area> ::= <break symbol> [**is_associated_with** LabelIdentifier]
[**is_associated_with** GuardCondition]

446 <break symbol> ::= <stop symbol>

26.2.10.2. Desision

447 DecisionConstruct ::= "if" GuardCondition StatementBlock {ElseIfClause}* [ElseClause]

448 GuardCondition ::= "[" BooleanExpression "]"

449 ElseIfClause ::= "else" "if" GuardCondition StatementBlock

450 ChoiceConstruct ::= "choice" SingleExpression "of" "{" ChoiceList [ElseClause] "}"

451 ChoiceList ::= {ChoiceSelector StatementBlock}*

452 ChoiceSelector ::= "[" SingleConstExpression "]"

Concrete graphical grammar

453 <decision area> ::= <decision symbol> **is_followed_by**
(<decision if part> {<decision if part>}* [<decision else part>]) **set**

454 <decision if part> ::= <flow line symbol> **is_associated_with** GuardCondition
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

455 <decision outlet symbol> ::= <decision symbol>

456 <decision else part> ::= <flow line symbol>
is_associated_with ("[" **else** "]")
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

457 <choice area> ::= <decision symbol> **contains** SingleExpression **is_followed_by**
({<choice selector part>}* [<decision else part>]) **set**

458 <choice selector part> ::= <flow line symbol> **is_associated_with** ChoiceSelector
is_followed_by <statement block area>
is_connected_to <decision outlet symbol>

459 <fgr opt area> ::= <inline expression symbol> **contains**
(**if** GuardCondition <statement block area>)

26.2.10.3. Try Statement

460 TryStatement ::= "try" ["(" TimerValue ")"] (Statement | StatementBlock) CatchesStatementList
/* STATIC SEMANTICS - TimerValue must be of type float */

461 CatchesStatementList ::= { [AltGuardChar] CatchStatement StatementBlock [SemiColon] }+

Concrete graphical grammar

462 <try area> ::= <try symbol> **contains** ["(" TimerValue ")"]
is_associated_with ({<catch association area> }*
| <try statement block area>) **set**

463 <try statement block area> ::= <statement block area>
is_connected_to <try out-connector symbol>

464 <catch association area> ::= <solid association symbol>
[**is_associated_with** GuardCondition]
is_connected_to <fgr catch area>
is_followed_by <statement block area>
is_connected_to <try out-connector symbol>

465 <try out-connector symbol> ::= <connector symbol>

26.2.10.4. Expressions

466 AssignmentList ::= -> "(" Assignment {SemiColon Assignment}* ")"

/* STATIC SEMANTICS - The assignment list must be present in conjunction with a receiving event. */

467 Assignment ::= DataObjectReference ":=" Expression

/* OPERATIONAL SEMANTICS - The Expression on the Right-Hand Side of Assignment shall evaluate to an explicit Value of the type of the LHS(Left-Hand Side). */

468 DataObjectReference ::= [ModuleName Dot] ComponentRef [ExtendedFieldReference]

469 ExtendedFieldReference ::= { ArrayOrBitRef | (Dot (StructFieldIdentifier | ClassFieldIdentifier) }+

470 Expression ::= SingleExpression | CompoundExpression

471 CompoundExpression ::= FieldExpressionList | ArrayExpression

472 FieldExpressionList ::= “{“ FieldExpressionSpec {“,” FieldExpressionSpec}* “}”

473 FieldExpressionSpec ::= FieldReference AssignmentChar Expression

474 ArrayExpression ::= “{“ [NotUsedOrExpression {“,” NotUsedOrExpression}*] “}”

475 NotUsedOrExpression ::= Expression | “-”

476 ConstantExpression ::= SingleConstExpression | CompoundConstExpression

477 SingleConstExpression ::= SingleExpression

/* STATIC SEMANTICS - SingleConstExpression shall not contain Variables or Module parameters and shall resolve to a constant Value at compile time */

478 BooleanExpression ::= SingleExpression

/* STATIC SEMANTICS - BooleanExpression shall resolve to a Value of type Boolean */

479 CompoundConstExpression ::= FieldConstExpressionList | ArrayConstExpression

480 FieldConstExpressionList ::= FieldExpressionList

481 ArrayConstExpression ::= “{“ [ConstantExpression {“,” ConstantExpression}*] “}”

482 SingleExpression ::= ConditionalExpression [? SimpleExpression Colon ConditionalExpression]

483 ConditionalExpression ::= LogicalExpression {(“&&” | “||”) LogicalExpression}*

484 LogicalExpression ::= SimpleExpression {LogicalOp SimpleExpression}*

/* OPERATIONAL SEMANTICS - If both SimpleExpressions and the LogicalOp exist then the SimpleExpressions shall evaluate to specific values of compatible types */

485 SimpleExpression ::= ["not"] EqualityExpression

/* OPERATIONAL SEMANTICS - Operands of the not operator shall be of type boolean (ATDL or ASN.1) or derivatives of type Boolean. */

486 EqualityExpression ::= RelationalExpression [("==" | "!=") RelationalExpression]

/* OPERATIONAL SEMANTICS - the precedence of the operators is defined in Table 8 */

487 RelationalExpression ::= ShiftExpression [("<" | ">" | ">=" | "<=") ShiftExpression]
| ShiftExpression “instanceof” RestrictedType
| ShiftExpression "in" ShiftExpression

/* OPERATIONAL SEMANTICS - If both ShiftExpressions and the RelOp exist then the ShiftExpressions shall evaluate to specific values of compatible types. */

/* OPERATIONAL SEMANTICS - If RelOp is "<" | ">" | ">=" | "<=" then each ShiftExpression shall evaluate to a specific integer, cardinal, Enumerated, real or float Value. */

488 ShiftExpression ::= BitwiseExpression [ShiftOp BitwiseExpression]

/* OPERATIONAL SEMANTICS - Each BitwiseExpression shall resolve to a specific Value. If more than one BitwiseExpression exists the right-hand operand shall be of type integer and if the shift op is '<<' or '>>' then the left-hand operand shall resolve to either bitstring, hexstring or octetstring type. */

489 BitwiseExpression ::= SubResult {BitOp SubResult}*

/* OPERATIONAL SEMANTICS - If both SubResults and the BitOp exist then the SubResults shall evaluate to specific values of compatible types. */

490 SubResult ::= ["not"] AdditiveExpression | “complement” ValueList

/* OPERATIONAL SEMANTICS - If the not operator exists, the operand shall be of type bitstring, octetstring or hexstring. */

/* OPERATIONAL SEMANTICS - Operands of the complement operator shall be of type ValueList */

491 AdditiveExpression ::= MultiplicativeExpression { (“+” | “-”) MultiplicativeExpression}*

/* OPERATIONAL SEMANTICS - Each MultiplicativeExpression shall resolve to a specific Value. If more than one MultiplicativeExpression exists then the MultiplicativeExpressions shall resolve to type integer, cardinal, real or float. */

/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer, cardinal, float or real or derivations of integer, cardinal, float or real (i.e., sub-range) */

/* OPERATIONAL SEMANTICS - Operands of the string operator "+" shall be bitstring, hexstring, octetstring or character string */

492 MultiplicativeExpression ::= UnaryExpression {MultiplyOp UnaryExpression}*

/* OPERATIONAL SEMANTICS - Each UnaryExpression shall resolve to a specific Value. If more than one UnaryExpression exists then the UnaryExpressions shall resolve to type integer, cardinal, real or float. */

493 UnaryExpression ::= [("+" | "-")] Primary | CastExpression

/* OPERATIONAL SEMANTICS - The Primary shall resolve to a specific Value. If UnaryOp exists and is "not" then Primary shall resolve to type BOOLEAN. If the UnaryOp resolves to "not4b" then the Primary shall resolve to the type bitstring, hexstring or octetstring. */

/* OPERATIONAL SEMANTICS - Operands of the "+" or "-" operators shall be of type integer, cardinal, float or real or derivations of integer, cardinal, float or real (i.e., sub-range). */

494 Primary ::= OpCall | DataObjectReference | Value | "(" Expression ")"

495 CastExpression ::= Type "(" SingleExpression ")"

496 OpCall ::= ConfigurationOps | GetLocalVerdict | TimerOps | TestcaseInstance | FunctionInstance
| CallStatement | BindOp | TemplateOps | ActivateStatement

497 MultiplyOp ::= "*" | "/" | "mod" | "rem"

498 BitOp ::= "and" | "xor" | "or"

/* OPERATIONAL SEMANTICS - Operands of the and4b, or4b or xor4b operator shall be of type bitstring, hexstring or octetstring or derivatives of these types. */

/* OPERATIONAL SEMANTICS - the precedence of the operators is defined in Table 8 */

499 LogicOp ::= "&" | "^" | "|"

/* OPERATIONAL SEMANTICS - Operands of the "&, "|" or "^" operators shall be of type boolean or derivatives of type Boolean. */

500 ShiftOp ::= "<<" | ">>" | "<@" | "@>"

26.2.11. Miscellaneous productions

501 Dot ::= "."

502 Dash ::= "-"

503 Minus ::= Dash

504 SemiColon ::= ";"

505 Colon ::= ":"

506 BeginChar ::= "{"

507 EndChar ::= "}"

508 AssignmentChar ::= ":="

509 ATDLComments ::= "//" FreeText

Concrete graphical grammar

510 <comment area> ::= <comment symbol> **contains** FreeText
is_connected_to <dashed association symbol>

27. An INRES example

The example provided here uses the ATDL specification given in this document which provides sequential and concurrent test cases for the INRES protocol [INRES].

Table 33: INRES example of an ATDL test case

```
testcase MTCType.mi_synch1() {
   activate (OtherwiseFail);  //Default activation
   ISAP1.send ({}: ICONreq);  // In-line template definition
   alt {
      [ ] ISAP1.receive ( Disconnection_Indication )
        {setverdict(inconc);}  // connection failure
     [ ] MSAP2.receive ( Medium_Connection_Request ) {
        MSAP2.send (Medium_Connection_Confirmation );  // use of a template
        alt {
           [ ] MSAP2.receive (Medium_Connection_Request )
              {setverdict(inconc);}   // medium connection request repetition
           [ ] ISAP1.receive (Disconnection_Indication )
              {setverdict(inconc); }   // connection failure
           [ ] ISAP1.receive (Connection_Confirmation ) {
              ISAP1.send (Data_Request(TestSuitePar) );
              alt {
                 [ ] ISAP1.receive (Disconnection_Indication )
                    {setverdict(inconc);}   // connection failure
                 [ ] MSAP2.receive (Medium_Data_Transfer ) {
                    MSAP2.send (cmi_synch1 );
                    ISAP1.send (Disconnection_Request );
                    alt {
                      [ ] ISAP1.receive (Disconnection_Indication ) {
                         MSAP2.receive (Medium_Disconnection_Request )
                         {setverdict(pass);}
                           }
                      [ ] MSAP2.receive (Medium_Disconnection_Request ) {
                          ISAP1.receive (Disconnection_Indication )
                            {setverdict(pass);}
                             }
                      [ ] MSAP2.receive (Medium_Data_Transfer )
                            setverdict(inconc); // medium data transfer
                         } // repetition
                     }
                  }
               }
            }
         }
     }
    stop
} /* End of test case mi_synch1 */
```

An ATDL test case for the connection establishment procedure is given in Table 33. The test purpose could be stated as whether the Responder SUT is capable to accept a connect request within a given time limit. An ATDL test case graph for the connection establishment procedure is shown in Figure 54. The sequential test cases use the test component MTC and two channels ISAP1 and MSAP2.

Figure 54. The INRES connection establishment procedure