
ADL 2.0 for IDL Language Reference Manual, Version 1.2

ADL for IDL

SunTest
The Open Group Research Institute

The language definition for ADL annotations
for the IDL interface definition language
defined by Object Management Group.

ISSUE NUMBER	REASON FOR ISSUE
1.0 Alpha	Document Launch For Review
1.0 Beta	First Revision
1.0 Gamma	Second Revision
1.0 Delta	Third Revision
1.0	Final Revision: delivered to IPA
1.1	Second delivery to IPA
1.2	Updated in accordance with version 2.0.2 of the ADL Translation System.

COPYRIGHT AND LICENSE NOTICE

Copyright © 1997-1998 The Open Group

Copyright © 1994-1997 Sun Microsystems Inc.

Copyright © 1994-1998 Information-technology Promotion Agency, Japan

This technology has been developed as part of a collaborative project among the Information-technology Promotion Agency, Japan (IPA), X/Open Company Ltd. and Sun Microsystems Laboratories.

Permission to use, copy, modify and distribute this software and documentation for any purpose and without fee is hereby granted in perpetuity, provided that this **COPYRIGHT AND LICENSE NOTICE** appears in its entirety in all copies of the software and supporting documentation. Certain ideas and concepts contained in the software are protected by pending patents of Sun Microsystems,. Sun hereby grants a limited license to use these patents, if any issued, only in this implementation of the software and documentation and in derivatives thereof prepared in accordance with the permission granted herein.

The names X/Open, Sun Microsystems. and Information-technology Promotion Agency, Japan (IPA) shall not be used in advertising or publicity pertaining to distribution of the software and documentation without specific, written prior permission.

ANY USE OF THE SOFTWARE AND DOCUMENTATION SHALL BE GOVERNED BY CALIFORNIA LAW. X/OPEN, SUN MICROSYSTEMS, INC. AND IPA MAKE NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE OR DOCUMENTATION FOR ANY PURPOSE. THEY ARE PROVIDED “AS IS” WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND. X/OPEN SUN MICROSYSTEMS, INC. AND IPA SEVERALLY AND INDIVIDUALLY DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE AND DOCUMENTATION, INCLUDING THE WARRANTIES OF MERCHANTABILITY, DESIGN, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL X/OPEN, SUN MICROSYSTEMS, INC. OR IPA BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, OR PROFITS, WHETHER IN ACTION ARISING OUT OF CONTRACT, NEGLIGENCE, PRODUCT LIABILITY, OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE OR DOCUMENTATION.

Trademarks

Sun™, Sun Microsystems™, Sun Microsystems Laboratories™, the Sun logo, Solaris™, SunOS™, and Java™ are trademarks or registered trademarks of Sun Microsystems, Inc.

Postscript™ is a trademark of Adobe Systems Inc.

UNIX® is a registered trademark in the USA and other countries licensed exclusively through X/Open™.

X/Open™ is a trademark of the X/Open Company Limited.

Change Log

Release 1.2

Example 4.13: Updated example.

Example 4.15: Removed public modifier to tddclass declaration.

Modifications occur only in chapters 1 to 4 and chapter 6, and are exactly the same than in the ADL 2.0 Language Reference Manual For Java Release 1.2.

Release 1.1

Example 4.15: Changed bankFactory type from BankFactory to BankFactoryHolder.

Release 1.0

Example 4.15: Improved tdd factory code.

6.7: New section dedicated to the portable ORB wrapper (ORBWrapper class).

Release 1.0 Delta

A new chapter has been added that deals with specific issues the test designers may encounter when testing IDL interfaces with the ADL 2.0 for Java. The other chapters are references to the ADL 2.0 for Java Language Reference Manual.

Release 1.0 Gamma

Complete rewriting of the document. ADL for IDL syntax is likely to be replaced for a large part by the ADL syntax for an implementation language (Java for instance). Hence, this document is a revisited version of the ADL for Java document.

Release 1.0 Beta

1. Introduction

1.1: New.

1.2: New.

2. Semantics Annotations

2.1: TBD resolved: ADL annotations are external to the interface.

2.4: Inline operation implementation as procedure identified as stretch feature.

3. Test Annotations

Title changed from “TDD Annotations”.

3.1.2: Terminology change: “bounded” dataset changed to “feasible”.

3.1.3: Added TBD for factory representation.

3.1.4: TBD resolved: explicit invocation of checked version by ADL(...). This affects all the examples.

3.1.5: Terminology change: “Test expression” changed to “Test directive”. This affects the explanation of some of the examples and the grammar.

3.1.5: TBD resolved: TDD/IDL test statements are restricted to operation or test procedure invocation, and test procedures are defined externally.

3.1.6: Assert expressed in support library.

3.2: TBD resolved: factories are not implicit datasets.

Example 3.5: TBD resolved: dataset members are evaluated each time.

Example 3.9: TBD resolved: no special syntax for multiple data values.

Example 3.10: TBD resolved: test directive syntax clarified.

Example 3.11: TBD resolved: multiple reference interpretation changed.

3.3 (TDDDatasetDeclaration): External dataset reference clarified.

3.3 (TDDTestDirective): Syntax cleaned up.

3.3.3 (TDDTestDirective): Cleaned up.

3.3.3 (TDDDatasetMember): Evaluation rule corrected.

3.3.4: Changed to eliminate proposed statement syntax extension.

4. NLD Annotations

4.1: TBD resolved: inheritance clarified.

Example 4.4: TBD resolved: rules on formal argument name clarified.

4.3.1: TBD resolved: SGML entity definition referred to DTD.

4.5 (nld_entity_text): TBD resolved: Rule on markup (DocBook 3.0 Para entity) clarified.

5. Complete Grammar

Replaced entire with revised version generated from source code for parser.

Release 1.0 Alpha

Initial release.

1 Introduction.....	12
2 Semantic Annotations.....	13
2.1 Describing Semantics Of Interface Operations	13
2.2 ADL Syntax.....	15
2.2.1 Assertion Groups	15
2.2.2 ADL Specific Expressions	17
2.2.3 Quantified Assertions	17
2.2.4 ADL If Statement	18
2.3 Behavior Specification.....	19
2.3.1 The Call State Operator	20
2.3.2 Specification of a Constructor	20
2.3.3 Specification Of An Inherited Method	21
2.3.4 Bindings.....	23
2.3.5 Try/Catch Specifications.....	24
2.3.6 Thrown Expressions	27
2.3.7 Behavior Classification.....	28
2.3.8 The Exception Operator	30
2.4 Inline Procedure Declarations	32
2.5 Prologues and Epilogues	33
3 Test Annotations.....	35
3.1 Concepts	35
3.1.1 Re-write	35
3.1.2 Dataset	35
3.1.3 Factory	36
3.1.4 Checked function.....	36
3.1.5 Test Directive.....	37
3.1.6 Assertion.....	38
3.1.7 Importation	38
3.2 Annotated TDD/Java Syntax	38
3.2.1 TDD declarations.....	38
3.2.2 Dataset Expression Syntax	39
3.3 General Syntax & Examples	40
3.3.1 Simple Datasets and Data Construction	40
3.3.2 Compound Datasets: Factories, Concatenation.....	41
3.3.3 Void Datasets	42
3.3.4 Dataset Elements Evaluation	43
3.3.5 Dataset Constants	44
3.3.6 Test Directives	45
3.3.7 Advanced Examples	46
4 NLD Annotations	51
4.1 Concepts	51
4.2 Syntax and Semantics.....	52
4.2.1 Simple Data Member Translation.....	52
4.2.2 Simple Function Member Translation	52
4.2.3 Out Of Line Translations	52
4.2.4 Translations For Overloaded Methods	53

4.2.5	Priorities.....	53
4.2.6	Using semantics And nld Blocks.....	53
4.2.7	Shadowing or Overriding A Translation.....	54
4.2.8	Overriding A Non-Local Translation.....	55
4.2.9	Invocation Translation	55
4.3	NLD Predicates.....	56
4.3.1	Pre-defined Predicates	56
4.3.2	User-defined Predicates	57
4.4	NLD and SGML	57
4.4.1	Reference Manual Document	58
4.5	NLD for TDD	59
4.6	NLD and Localization	59
4.7	NLD Syntax	60
5	Testing CORBA Objects.....	63
5.1	Mapped built-in types	63
5.2	Bindings for objects.....	63
5.3	Inout parameters passing	63
5.3.1	Isolated variables	63
5.3.2	Member attribute of interface/structs/unions and inout/out parameters	64
5.4	Mapping for certain nested types.....	65
5.5	Interface inheritance	65
5.6	Writing CORBA specific code in TDD/Java.	66
5.7	ORBWrapper	67
5.7.1	ORB Wrapper Design.....	68
5.7.2	Adl environment property to use for the ORB wrapper.....	69
5.7.3	Using the wrapper.....	69
5.8	References.....	71
6	Complete Grammar.....	73

EXAMPLE 2.1 StockBroker interface.....	13
EXAMPLE 2.2 StockBroker specification.....	13
EXAMPLE 2.3 StockBroker constructor specification	20
EXAMPLE 2.4 Bank and MyBank classes.....	21
EXAMPLE 2.5 MyBank specification	22
EXAMPLE 2.6 StockBroker2 interface.....	24
EXAMPLE 2.7 StockBroker2 specification.....	25
EXAMPLE 2.8 StockBroker2 specification (corrected).....	26
EXAMPLE 2.9 StockBroker specification with behavior classification	28
EXAMPLE 2.10 StockBroker specification with exceptions	31
EXAMPLE 3.1 The Simplest Test	40
EXAMPLE 3.2 A Simple Dataset.....	41
EXAMPLE 3.3 Compound Data Construction.....	41
EXAMPLE 3.4 Void Datasets	42
EXAMPLE 3.5 Runtime Initializers	43
EXAMPLE 3.6 Provide Test Variables	44
EXAMPLE 3.7 Better Test Variables.....	44
EXAMPLE 3.8 Test Directives and Procedures	45
EXAMPLE 3.9 Void Dataset Use.....	46
EXAMPLE 3.10 Chaining Factories	46
EXAMPLE 3.11 Multiple Data Values	47
EXAMPLE 3.12 Test By Example.....	48
EXAMPLE 3.13 Multiple Dataset References	49
EXAMPLE 4.1 Using includes and description	58
EXAMPLE 4.2 NLD Annotations in a TDD class	59
EXAMPLE 4.3 Using Fully Scope Names	61
EXAMPLE 4.4 Isolated variable used as inout parameter.....	64
EXAMPLE 4.5 Out/inout parameters and member attributes	64
EXAMPLE 4.6 Bank factory TDD example.....	66

For Your Information.

Since tensions in between IDL language independence and requirements for ADL to let the test engineer provide real and complex code constructs were too strong, a decision was made to use ADL/Java instead of pure ADL/IDL to implement test suites for IDL interfaces.

As a consequence test engineers should implement test suites for CORBA objects by using ADL 2.0 Translation System. The tests are performed against Java stubs. These stubs are generated using some third party vendor Java ORBs (e.g. Sun's Joe, Iona's OrbixWeb or Visigenic's VisiBroker for Java).

The remaining of this document is organized as follows:

- chapters 1 to 4 are chapters from the ADL 2.0 Language Reference Manual for Java
- chapter 5 contains information and guidelines for using ADL/Java to test CORBA objects (notably the ORB wrapper provided by the ADL environment),
- chapter 6 is the complete ADL 2.0 for Java grammar.

1 Introduction

ADL is an interface definition and testing system, which adds to a target programming language a notation for describing behavior, for defining tests, and for generating documentation. This document describes ADL for the Java programming language.

ADL provides capabilities to describe the semantics of interfaces, and also the capability to design and implement test drivers.

This document is a concise language reference, intended to define the syntax of the ADL annotation language. There are three kinds of annotation possible in the ADL system: semantic annotations, that define the behavior required of an interface method; test annotations, that define a set of data and a procedure for testing an interface; and natural language annotations, that define how to represent the interface in natural language. These three kinds of annotation are described in the next three sections.

This document does not yet describe how to use the ADL system, nor define the exact interpretation of the annotations; familiarity with the previous version of the ADL system, ADLT 1, will be of great assistance.

The syntax used to describe the language grammar in this document is BNF, and follows these conventions:

- The vertical bar “|” represents a choice between different expansions. Hence “**A** | **B** | **C**” represents either **A**, **B**, or **C**.
- Square brackets “[...]” indicate optional constructs. Hence “**A** [**B**] **C**” is the same as “**ABC** | **AC**”.
- Parentheses “(...)” are used for grouping constructs. Hence “**A** (**B**) **C**” is the same as “**ABC**” and “**A** (**B** | **C**) **D**” is the same as “**ABD** | **ACD**”.
- “(...)*” is used to represent zero or more occurrences of the group, and “(...)+” is used to represent one or more occurrences of the group. Hence “**A** (**B**)* **C**” is the same as “**AC** | **ABC** | **ABBC** | **ABBBBC** | *etc.*” and “**A** (**B**)+ **C**” is the same as “**ABC** | **ABBC** | **ABBBBC** | *etc.*”.
- Non-terminals from the Java language definition are represented in a sans-serif font (like *literal*), and the non-terminals that define the ADL augmentation of Java appear in **boldface**.
- Lexical tokens and reserved words may appear literally within quotations, or the name of the lexical token may appear in angle brackets like <STRING>.
- The left hand and the right hand sides of productions are separated by the symbol “: =”. For presentation purposes, the entire right hand side of a production may not be introduced at the same time. The symbol “+ : =” is used to indicate that the current production is an augmentation of another production with the same left hand side that has been introduced earlier. For example, “**A** : = **B**” followed by “**A** + : = **C**” is the same as “**A** : = **B** | **C**”.

2 Semantic Annotations

The ADL extensions to Java that allow the definition of the semantics of a method are discussed in the sections below.

2.1 Describing Semantics Of Interface Operations

ADL provides syntactic constructs to describe the semantic behavior of methods. To do this, it provides an extended syntax for describing operations — the *annotated declaration* — as shown in the syntax below:

```

ADL_CompilationUnit ::= ( ImportDeclaration )* ADL_ClassDeclaration

ADL_ClassDeclaration ::= [ “public” ] “adlclass” <IDENTIFIER> “{”
                           ( ADL_InlineDeclaration | ADL_BehaviorDeclaration )* “}”

ADL_BehaviorDeclaration ::= [ “static” ] [ ResultType ] Name FormalParameters
                           “{” ADL_BehaviorSpecification “}”

```

These rules are not complete: they will be refined (notation $++::=$) throughout this document as we present new properties. The complete grammar is given in Chapter 5.

The full definitions of the extra declarations added by ADL are given in later sections; a simple example is given here. Suppose that we wish to define the behavior of objects conforming to this interface:

EXAMPLE 2.1 StockBroker interface

```

package stock;

interface StockBroker {

    public long Cash_Balance(long account);

    public long Stock_Balance(long account, String symbol);

    public void Buy(long account,
                    String symbol,
                    long no_of_shares);
}

```

Then the ADL specification of this interface can be written in a separate file, “Stock-Broker.adl” as shown below:

EXAMPLE 2.2 StockBroker specification

```

import stock.*; // import the package of the Java class
                // StockBroker
import StockBrokerAux;

adlclass StockBroker { // define the adl class StockBroker

    inline long cost(String s, long nsh) {
        nsh * StockBrokerAux.price(s);
    }

    void Buy(long account, String symbol, long no_of_shares) {
        semantics {
            assertion1 : Cash_Balance(account) ==
                @ Cash_Balance(account) - @cost(symbol, no_of_shares);
            assertion2 : Stock_Balance(account, symbol) ==
                @ Stock_Balance(account, symbol) + no_of_shares;
        }
    }

    long Cash_Balance(long account) {
        semantics {
            // (...)
        }
    }
}

```

In this example, an interface with three operations `Cash_Balance`, `Stock_Balance`, and `Buy` is augmented with a description of the behaviors of `Buy` and `Cash_Balance`. The behavior description of `Buy` uses a static method “price” from an additional class, `StockBrokerAux`, which is defined only for purposes of testing. The behavior description is written in the annotated declaration syntax. The two boolean expressions appearing within “`semantics { ... }`” describe legitimate behavior of the `Buy` method. In these expressions, “@” is an unary operator (referred to as the *call state operator* — see Section 2.3.1) whose sole function is to evaluate its argument prior to the execution of the method — by default all expressions are evaluated after the execution of the method.

It is important to note that the reference `Cash_Balance` inside the behavior description of `Buy` is a reference to the method `Cash_Balance` of the Java class under test, *not* a reference to the behavior description of `Cash_Balance` that is given below. *Behavior descriptions are not method declarations.*

The first boolean expression, labeled as “`assertion1`”, makes use of the notion of “cost” of a stock purchase. This is implemented as an inline procedure declaration in the semantic annotation file. An inline procedure declaration defines a block that may be used like a macro within a semantic annotation (see Section 2.4).

The syntax described above is external to the interface definition, because ADL assertions can only be evaluated from the perspective of a client of the interface. Such an annotation can be applied to a class or an abstract class equally as well as to an inter-

face. The behavior description for a given class cannot be split into several ADL files; however, the behavior description for a class can inherit the description of the superclass of this class (see Section 2.3.3).

The semantics of “import” clauses are the same as for a Java compilation unit. However, it is not possible to define a Java class or a Java interface inside an ADL file: auxiliary classes must be defined in different, regular Java files.

Note that because semantic annotations are external to the class and method being annotated, the annotations do not have access to private, protected or package members: only **public** fields and methods of the class under test are accessible. If a method is not public, then it cannot be described nor called in the description of another method.

Lastly, it is not possible to define more than one *adl class* in an ADL compilation unit, and the name of the file must be the name of the adl class: to annotate the Java class `foo.class`, one must define a single “`adlclass foo`” in a file named `foo.adl` (or `foo.adljava`).

2.2 ADL Syntax

ADL provides a syntax that is a minor extension to that of Java 1.0.2. Any Java expression *except assignments* may appear in a semantic annotation, as well as expressions using the extensions described in this section.

The new syntactic constructs of Java 1.1 — that deal with “inner classes” and the “`class`” feature for the reflection API — are not supported within the ADL for Java syntax. However, as the ADLT runtime is written in Java 1.1, ADL 2 supports the testing of Java 1.1 interfaces and the use of Java 1.1 auxiliary classes.

2.2.1 Assertion Groups

```

ADL_AssertionGroup ::= “{” ( ADL_Binding ) * ( ADL_Statement ) * “}”

ADL_Statement ::= ADL_Assertion
                | ADL_IfStatement
                | ADL_TryStatement

ADL_Assertion ::= [ ADL_Label ] [ ADL_Tags ]
                 ( ADL_Expression | ADL_QuantifiedAssertion ) “,”

ADL_Label ::= <IDENTIFIER> “:”

ADL_Tags ::= “[” <IDENTIFIER> ( “,” <IDENTIFIER> ) * “]”

```

The basic block construct of ADL is the *assertion group*, which is a list of *statements*. ADL statements have a type (usually boolean) and a value, but can not be mixed directly inside expressions. If there is more than one statement within the assertion group, then all of these statements must be boolean valued. The value of the assertion group in this case is the conjunction (logical AND) of all the statements in the assertion group. If the assertion group contains only one statement, then this statement may be of any type, and

the assertion group is also of this type and has the same value as the statement within it; this can occur either with the inline/define constructs or with the try/catch statement (see Section 2.3.5).

The optional label of an assertion is for documentation purposes only: it will be reported as information when running the generated test. It does not modify the behavior at runtime in any other way.

The optional tags of an assertion are indications for the test runtime environment. The only currently supported tag is “[U]” (for *untestable*) that means that the assertion must not be evaluated.

The assertion group itself is an expression. Its use as an expression is given by the following syntax:

ADL_BasicExpression ::= ADL_AssertionGroup

Assertions are boolean expressions whose evaluation must generate a test report: they do not produce any other side effect (hence assignments or increments/decrements are forbidden inside assertions).

The following fragment is an example of an assertion group:

```
{
  Cash_Balance(account) == @Cash_Balance(account) -
    @cost(symbol, no_of_shares);
  Stock_Balance(account, symbol) ==
    @Stock_Balance(account, symbol) + no_of_shares;
}
```

Since assertion groups are also expressions, they may appear anywhere an expression is expected, and they may be nested within each other. Assertions within nested assertion groups do *not* generate a test report: they are evaluated only so that their return value is used in the computation of the value of the enclosing assertion group.

```
semantics {
  <boolean expression> ==> { <assertion1>; <assertion2>; }
}
```

In this example, there is only one generated test report for the whole assertion, not for “sub-assertions” `assertion1` and `assertion2`.

The list of expressions in an assertion group may be preceded by *bindings*: see Section 2.3.4.

While most ADL specific expressions and statements are described in the two forthcoming sections, some are described later in sections where they are more appropriate. The following is the complete list of all cross references to later sections where ADL features are described:

- The call state operator — Section 2.3.1
- Bindings — Section 2.3.4

- The try/catch statement — Section 2.3.5
- thrown expressions — Section 2.3.6
- The exception operator — Section 2.3.8
- Inline methods — Section 2.4
- Prologues and Epilogues — Section 2.5

2.2.2 ADL Specific Expressions

ADL_Expression ::= **ADL_ImplExpression**

ADL_ImplExpression ::= ConditionalExpression [**ADL_ImplOp** ConditionalExpression]

ADL_ImplOp ::= “==>” | “<==” | “<=>” | “<:>”

The three implication operators are *implication* (==>), *reverse implication* (<==), and *equivalence* (<=>). All these operations operate on boolean parameters and return boolean results. The implication operator evaluates to `false` only when its left operand is `true` and right operand is `false` (otherwise, it evaluates to `true`). The reverse implication operator works like the implication operator with its arguments swapped. The equivalence operator evaluates to `true` if both its operands are the same, otherwise it evaluates to `false`. For the exception operator <:>, see Section 2.3.8.

ADL_BasicExpression ::= PrimaryExpression | “return”

Primaries are extended in ADL with the reserved word **return** (to refer to the return value of a method). The primary return may be used only in behavior specifications (Section 2.3) of methods with non-void return types and may not appear within the scope of a call state operator (Section 2.3.1).

2.2.3 Quantified Assertions

ADL_QuantifiedAssertion ::= **ADL_Quantifier** (“**ADL_DomainList**”) **ADL_AssertionGroup**

ADL_Quantifier ::= “forall” | “exists”

ADL_DomainList ::= **ADL_Domain** (“,” **ADL_Domain**)*

ADL_Domain ::= FormalParameter “:” ConditionalExpression

ADL offers a constrained form of quantified expression in order to iterate over ADL sequence values. These sequences are specified as domains, and a quantified assertions may contain any number of domains. Each domain is specified with the type of the sequence element, a new variable that can take on the values of the sequence one by one, and finally the sequence itself. An example of a domain that iterates over the integers 1 through 10 is:

```
long i : ADL_long_range(1,10)
```

The ADL language supports 3 methods `ADL_long_range(long i, long j)` (resp. `ADL_short_range` and `ADL_int_range`) that return the enumeration of `long` (resp `short` and `int`) starting from `i` and ending at `j`.

In ADL for Java, the `ConditionalExpression` in the domain must be of the type `java.util.Enumeration`.

In the case of the universal quantifier (**forall**), the enclosed assertion group (which must have a boolean value) must be `true` for all value assignments for free variables from their domains. In the case of the existential quantifier (**exists**), the expressions must be `true` for *at least one* set of value assignments for the free variables.

The assertion group within a quantified assertion is nested: its assertions will not generate individual test reports.

The following is an example of the use of an universal quantifier that says that all numbers in the range 1 to 10 are smaller than 100 (obviously):

```
forall (long i : ADL_long_range(1,10)) { i < 100; };
```

The following is an example of the use of an existential quantifier:

```
semantics {
  exists (long i : ADL_long_range(1,10)) {
    i%3 == 0;
    i%7 == 0;
  };
}
```

Because of the nested principle, the assertions within a quantified assertion are not distributed: this example will generate *one* test report, with the value `false`.

A quantified assertion may not appear inside the scope of a call-state/unchanged expression; free variables of a quantified assertion may not be used inside the scope of a call-state/unchanged expression.

2.2.4 ADL If Statement

ADL_IfStatement ::= “if” (“ **ADL_Expression** ”) **ADL_AssertionGroup**
[“else” (**ADL_AssertionGroup** | **ADL_IfStatement**)]

“if statements” provide a way to conditionally evaluate expressions and its meaning is quite similar to the “?:” operator. The types of all the assertion groups of the `if` statement must be the same and this is the type of the `if` statement. If the type of the first “then” branch is boolean, then the else branch may be omitted and is assumed to be “else `true`”. The conditions (the expressions within parentheses) must be boolean valued and are evaluated from top to bottom until the first one that evaluates to `true`. The assertion group of this `true` expression is then evaluated. This is the value of the `if` statement.

The assertion groups in the branches of an if statement are considered to be at the same nested level as the enclosing assertion group. If this enclosing assertion group is the outermost one (i.e. just following the “semantics” keyword), assertions within the if statement will therefore generate test reports.

2.3 Behavior Specification

The specification of the behavior of an interface method in its simplest form is the method declaration as in Java followed by the reserved word “semantics” followed by an assertion group.

The assertions can refer to the visible state of the system both before and after the execution of the method.

ADL_BehaviorDeclaration ::= [ResultType] Name FormalParameters [“throws” NameList]
 { “ **ADL_BehaviorSpecification** ” }

ADL_BehaviorSpecification ::= “semantics” **ADL_AssertionGroup**

Note that a behavior declaration cannot be empty in ADL for Java, as opposed to ADL for C/C++. This is because the Java language does not need forward declarations.

The “ResultType” is optional in the grammar because it is possible to annotate a class constructor.

Every time an interface method with a behavior description is invoked, all arguments to call state operators are evaluated before the method is invoked (call state operators are described below). The remainder of the behavior description is evaluated after the method is invoked. If any expression evaluates to `false`, the method did not behave as specified.

The behavior description of `Buy` from the example of Section 2 is reproduced below:

```
void Buy(long account, String symbol, long no_of_shares) {
  semantics {
    Cash_Balance(account) == @Cash_Balance(account) -
      @cost(symbol, no_of_shares);
    Stock_Balance(account, symbol) ==
      @Stock_Balance(account, symbol) + no_of_shares;
  }
}
```

The evaluation of the behavior description whenever `Buy` is invoked is outlined below:

Step 1: Evaluation of arguments to call state operators:

```
tmp1 = Cash_Balance(account);
tmp2 = cost(symbol, no_of_shares);
tmp3 = Stock_Balance(account, symbol);
```

Step 2: The implementation of `Buy` is invoked.

Step 3: Evaluation of the remainder of the behavior description:

```
assertion_1 = (Cash_Balance(account) == tmp1 - tmp2);
report(assertion1);
assertion_2 = (Stock_Balance(account, symbol) == tmp3 +
  no_of_shares);
report(assertion2);
```

Step 4: Determination of consistent behavior:

```
if (!assertion_1 || !assertion_2) { report_error; }
```

Behavior descriptions can refer to inline method declarations (as illustrated by the above example). Other specifics of behavior descriptions are discussed below.

2.3.1 The Call State Operator

The call state operator is an unary operator. It has the effect of evaluating its argument before the call to the specified method.

UnaryExpression ::= “@” UnaryExpression

ADL_BasicExpression ::= “unchanged” Arguments

Call state operators may nest within each other, in which case the inner operator is overridden by the outer operator. For example, `@(@a + b)` is equivalent to `@(a + b)`.

Care must be taken to decide exactly where to place a call state operator. For example, there is a subtle difference between `@f(a, b)` and `f(@a, @b)`. The first expression is the value of `f(a, b)` before the call to the specified method, while the second is the value returned by `f` when called after the call to the specified method, but passed parameters whose values are saved from the state before the call to the specified method.

Different call-state expressions are independent from each other. For instance in an assertion “`@time() == @time()`”, both expressions are not considered as equal.

The “unchanged” operator of ADL1 is maintained:

```
unchanged(<expr1>, <expr2>)
```

is a syntactic sugar for:

```
<expr1> == @<expr1> && <expr2> == @<expr2>
```

2.3.2 Specification of a Constructor

Specifying a constructor is very similar to specifying a method, except that some constraints are added.

EXAMPLE 2.3 StockBroker constructor specification

```

import stock.*; // import the package of the Java class
                // StockBroker
import StockBrokerAux;

adlclass StockBroker { // define the adl class StockBroker

StockBroker(int amt) {
    semantics {
        Cash_balance() == amt;
    }
}

}

```

In this example, the call to `Cash_balance` is executed on the `StockBroker` object built by an implicit call to the real constructor of the class `StockBroker`.

Note that in a constructor behavior specification:

- it is not possible to use a call-state or an unchanged expression (this would not make sense: there is no “real object” before the call to the tested method (the constructor), because this is this very call that builds the “real object”).
- it is not possible to use the “return” expression.
- the “this” expression refers to the built real object.

2.3.3 Specification Of An Inherited Method

ADL_ClassDeclaration `+:=[“public”] “adlclass” <IDENTIFIER> [“extends” Name]
“{” (ADL_BehaviorDeclaration) * “}”`

ADL_AssertionGroup `+:=[“{” (ADL_Binding) * [“super” “.” “semantics” “,”] (ADL_Statement) * “}”`

To describe the behavior of a method `m`, it is possible to use the behavior description of the method `m'` that `m` overrides.

EXAMPLE 2.4 Bank and MyBank classes

```

/* Bank.java */
package bank;

class Bank {
    public void openAccount() { (...) }
    public void closeAccount() { (...) }
}

/* MyBank.java */
package bank;

```

```
class MyBank extends Bank {  
    public void openAccount() { (...) }  
    // overrides Bank.openAccount  
    public void changeAccount() { (...) } // new method  
}
```

The behavior of the methods `openAccount` and `closeAccount` of class `Bank` is specified in a file `Bank.adl`, and we want now to describe the behavior of the methods of the class `MyBank`:

EXAMPLE 2.5 `MyBank` specification

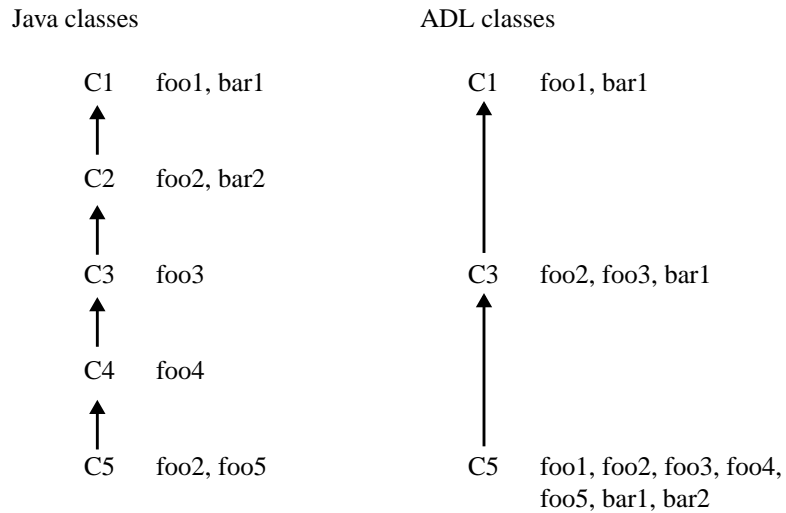
```
import bankAux; // for auxiliary functions  
import bank.*; // package of the annotated class  
  
adlclass MyBank extends Bank {  
  
    void openAccount() {  
        semantics {  
            super.semantics; // assertions specific to the method  
                           // Bank.openAccount  
            <assertion>; // assertion specific to the method  
                           // MyBank.openAccount  
        }  
    }  
}
```

The “extends” clause is quite different from its usual meaning in Java: it is used here to refer to ADL files (`Bank.adl` in the example). The compiler checks the presence and correctness of the *source* adl file; it is left to the responsibility of the user to ensure that at runtime the *class* file obtained by transformation of `Bank.adl` will be accessible from the classpath, along with the class file generated by transformation of `MyBank.adl`. This is close to the C semantics, with the distinction between header files at compilation time and libraries at runtime.

The assertion “`super.semantics;`” in `openAccount` is an explicit invocation of the annotation of `openAccount` in *one* superclass (direct super class or any “ancestor” class) of `MyBank` (here the semantics of the method `openAccount` of the class `Bank` — which `MyBank` inherits from — as defined in `Bank.adl`). It may be called only as the *first* assertion of the main assertion group. Its (boolean) result is the value of the behavior description of method `openAccount` in `Bank.adl`: *all* the assertions defined in this specification are evaluated. The only side-effects are the generation of test reports for the assertions of this behavior description.

The generated Java Assertion Checking Object (ACO) file for `MyBank` will in fact inherit from the generated Java ACO file for `Bank`. See the ADL 2.0 Translation System design document for more details.

To summarize:



This picture represents a Java hierarchy (C5 extends C4 that extends C3...) and a corresponding adl classes hierarchy. The *foos* and *bars* are methods defined in Java classes, some of them being annotated by adl classes. We can notice that:

- the adl classes graph is a subgraph of the java classes graph (it is not compulsory to annotate all java classes).
- an adl class can annotate a method that is *defined* in the corresponding java class (e.g. foo5 in C5) or *overridden* (e.g. foo2 in C5) or just *inherited* (e.g. foo4 in C5).
- the `super.semantics` feature can be used only in methods that have already been annotated in an inherited adl class. For instance in adl class C5, it would be an error to use this feature in methods foo4, foo5 and bar2, and correct in the other methods.

Note that this inherited adl class is not necessarily the direct superclass: for instance in adl class C5, `super.semantics` in foo1 would refer to foo1 of adl class C1.

Last, this process is recursive: if bar1 in C5 and bar1 in C3 both use `super.semantics`, then calling bar1 of C5 will first evaluate the assertions of bar1 in C1, then the assertions of bar1 in C3, and then the assertions of bar1 in C5.

2.3.4 Bindings

Bindings are used to declare local variables and initialize them with useful values. Their main goal is to be used in conjunction with NLD annotations.

ADL_Binding ::= “define” FormalParameter “with” [<IDENTIFIER> “=”] **ADL_Expression** “;”

The earlier `Stock_Balance` example may also be modified to use bindings. The following is equivalent to the earlier behavior description:

```

{
  define long pre_cash_bal with
    pre_cash_bal = @Cash_Balance(account);
  define long post_cash_bal with
    post_cash_bal = Cash_Balance(account);
  define long pre_stock_bal with
    pre_stock_bal = @Stock_Balance(account, symbol);
  define long post_stock_bal with
    post_stock_bal = Stock_Balance(account, symbol);

  post_cash_bal ==
    pre_cash_bal - @cost(symbol, no_of_shares);
  post_stock_bal == pre_stock_bal + no_of_shares;
}

```

In ADL for Java, a binding declaration shall not introduce simultaneously multiple variables (as opposed to ADL for C/C++ where side-effects are possible).

Important note:

It is not possible to reference a binding inside the scope of a call-state operator.

Indeed, in the generated code, the code that implements bindings appear after the code that implements call-state expressions.

It is possible to declare a binding using variables introduced in earlier bindings, but not to rebind a variable: a binding is a variable declared and initialized once for all. The scope of a binding is its innermost enclosing assertion group.

2.3.5 Try/Catch Specifications

During the evaluation of the assertions of an assertion group, it is possible for exceptions to be thrown. Try/catch specifications may be used to catch these exceptions and provide an alternate assertion group whose value is used for that of the parent (“try”) assertion group. The assertion group(s) in the catch specification(s) must therefore be of the same type as the parent assertion group.

ADL_TryStatement ::= “try” **ADL_AssertionGroup**
 (“catch” (“ FormalParameter ”) **ADL_AssertionGroup**)+

A catch specification has to name the particular exception it catches and bind it to a local identifier; this identifier may be used in the following assertion group to select values returned by the exception.

Suppose we modify the original stock broker interface to include some exceptions:

EXAMPLE 2.6 StockBroker2 interface

```
package stock;
```



```

class BadCall extends Exception {
    public boolean bad_account;
    public boolean bad_stock_symbol;
}

interface StockBroker2 {

    public long Cash_Balance(long account) throws BadCall;

    public long Stock_Balance(long account, String symbol)
        throws BadCall;

    public void Buy(long account, String symbol,
        long no_of_shares) throws BadCall;
}

```

Then we can modify the specification of this interface as follows:

EXAMPLE 2.7 StockBroker2 specification

```

import stock.*;
import StockBrokerAux;

adlclass StockBroker2 {

    inline long cost(String s, long nsh) {
        nsh * StockBrokerAux.price(s);
    }

    void Buy(long account, String symbol, long no_of_shares)
        throws BadCall {
        semantics {
            try {
                Cash_Balance(account) == @Cash_Balance(account) -
                    @cost(symbol, no_of_shares); // assertion 1
                Stock_Balance(account, symbol) ==
                    @Stock_Balance(account, symbol) + no_of_shares;
                // assertion 2
            }
            catch (BadCall exc) {
                thrown(BadCall) && {
                    ((BadCall)adl_thrownException).bad_account ==
                        exc.bad_account;
                    ((BadCall)adl_thrownException).bad_stock_symbol ==
                        exc.bad_stock_symbol;
                };
            }
            catch (Throwable t) {
                false;
            }
        }
    }
}

```

```
}

```

In this modified specification, the assertion group of the behavior description of `Buy` is modified to include two catch specifications.

The first catch specification catches the exception `BadCall`. The assertion group of this catch specification states that the exception `BadCall` must be thrown by `Buy` (thrown expressions are described below). Furthermore, this exception must have the same component values as that of the exception that was thrown during the evaluation of the assertion group. Note the mechanism to refer to components of exceptions thrown during the evaluation of a behavior description: the left hand sides of the comparisons use the ADL variable `adl_thrownException` (which is of type `Throwable` and must therefore be casted), that is the exception thrown by the specified method `Buy`, and the right hand sides of the comparisons (`exc.bad_account` and `exc.bad_stock_symbol`) refer to the values of components of the exception caught by the catch specification (i.e. the one that was thrown during the evaluation of the “try” assertion group). Also note the nested assertion group that contains the two comparisons. This prevents these comparisons from being evaluated if `thrown(BadCall)` is not `true` (in this case, the selection of components of `BadCall` will have unexpected results).

The second catch specification catches all other exceptions and its assertion group is simply “`{ false; }`”. This is simply stating that this situation is unexpected and if it does happen for whatever reason, a failure needs to be reported.

The assertion groups in the try and in the catch specifications are considered to be at the same nested level as the enclosing assertion group. In the previous example, a test report is generated for each assertion 1 and 2, or for the whole assertion of the catch clause “`thrown(BadCall) && { ((BadCall)adl_thrownException).bad_account ... }`” in case a `BadCall` exception is thrown.

The above example has a serious flaw. Exceptions may be thrown during the evaluation of an expression in the scope of one of the call state operators. These exceptions cannot be caught by the above catch specifications since they are evaluated after the specified method has been called. The solution to this problem is to catch the exceptions in the call state itself and replace them with harmless values. In this particular example, the same exceptions will be thrown by evaluation in the state after the call to the specified method, and hence the semantics specified by the catch specification will still take effect. The corrected version of the above example follows:

EXAMPLE 2.8 StockBroker2 specification (corrected)

```
import stock.*;
import StockBrokerAux;

adlclass StockBroker2 {

    inline long cost(String s, long nsh) {
        nsh * StockBrokerAux.price(s);
    }
}
```

```

void Buy(long account, String symbol, long no_of_shares)
  throws BadCall {
  semantics {
    try {
      Cash_Balance(account) ==
        @ { try { Cash_Balance(account) -
              cost(symbol, no_of_shares); }
            catch(Throwable t) { 0; }
          };
      Stock_Balance(account, symbol) ==
        @ { try { Stock_Balance(account, symbol); }
            catch(Throwable t) { 0; }
          }
      + no_of_shares;
    }
    catch (BadCall exc) {
      thrown(BadCall) && {
        ((BadCall)adl_thrownException).bad_account ==
          exc.bad_account;
        ((BadCall)adl_thrownException).bad_stock_symbol ==
          exc.bad_stock_symbol;
      };
    }
    catch (Throwable t) {
      false;
    }
  }
}

```

In this version, all exceptions caught in the call state are replaced by the value 0. The nature of this example is such that any exception thrown in the call state will also be thrown after the call to the specified method, hence the 0's passed from the call state are never really used.

Note that this is one of the two possible cases where an assertion and an assertion group may have non-boolean values.

The above example looks messy, but in the presence of exceptions, a lot of catch specifications are necessary. This is true of normal programs too. However, the above example is further cleaned up in Section 2.3.7 where the catch specifications of the call state are moved up into inline method declarations.

2.3.6 Thrown Expressions

Thrown expressions are boolean expressions used to specify whether exceptions have been thrown or not.

ADL_ThrownExpression ::= "thrown" "(" Name ("," Name)* ")"

`Thrown(e1, e2, etc.)` is true if any of the exceptions `e1`, `e2`, etc. is thrown and is false otherwise. `Thrown(Throwable)` is true if any exception or other `Throwable` is thrown. It is false if no `Throwable` is thrown.

“`thrown(e)`” is in fact just syntactic sugar for “`adl_thrownException instanceof e`”.

Note: thrown expressions may not be placed within the scope of a call state operator.

2.3.7 Behavior Classification

It is often very useful to broadly categorize the behavior of a method into its “normal behavior” and “abnormal behavior”. One may then specify more details of the behavior in each of these cases. ADL provides the behavior classification construct for this purpose. The behavior classification is used to associate a boolean expression to the reserved words **normal** and **abnormal**.

ADL_BehaviorSpecification $::=$ “semantics” [**ADL_BehaviorClassification**] **ADL_AssertionGroup**

ADL_BehaviorClassification $::=$ “[(**ADL_NormalBehavior** | **ADL_AbNormalBehavior**)+ ”]

ADL_NormalBehavior $::=$ “normal” “=” **ADL_Expression** “,”

ADL_AbNormalBehavior $::=$ “abnormal” “=” **ADL_Expression** “,”

The default meanings of normal and abnormal are as follows:

- If neither **normal** nor **abnormal** has been defined in a behavior classification, then **normal** defaults to `!thrown(Throwable)` and **abnormal** defaults to `thrown(Throwable)`.
- If only one of **normal** and **abnormal** is defined, the other defaults to the negation of the one defined. For example, if **normal** is defined, then **abnormal** defaults to `!normal`.

When both **normal** and **abnormal** are defined, their definitions need not be negations of each other. They may overlap or exclude portions of the possible output domain.

In a behavior classification, there may be at most one definition for **normal** and one for **abnormal**.

The reserved words **normal** and **abnormal** may then be used in the behavior description of the method as short forms for the expressions associated with them, as per the following syntax:

ADL_BasicExpression $::=$ “normal” | “abnormal”

The following example modifies the earlier example to make use of behavior classifications.

EXAMPLE 2.9 StockBroker specification with behavior classification

```

import stock.*;
import StockBroker2;

adlclass StockBroker2 {

    inline long cost(String s, long nsh) {
        try { nsh * StockBrokerAux.price(s); }
        catch(Throwable t) { 0; }
    }

    inline long _Cash_Balance(long act) {
        try { Cash_Balance(act); }
        catch(Throwable t) { 0; }
    }

    inline long _Stock_Balance(long act, String s) {
        try { Stock_Balance(act, s); }
        catch(Throwable t) { 0; }
    }

    void Buy(long account, String symbol, long no_of_shares)
        throws BadCall {
        semantics
        [ normal = !thrown(Throwable);
          abnormal = thrown(RuntimeException, BadCall); ]
        {
            if (normal) {
                try {
                    Cash_Balance(account) == @Cash_Balance(account)
                    - @cost(symbol, no_of_shares);
                    Stock_Balance(account, symbol) ==
                    @Stock_Balance(account, symbol) + no_of_shares;
                } // end try
                catch (Throwable t) { false; }
            } // end if (normal)
        } // end semantics
    } // end Buy

} // end adl class

```

This version of the stock broker specification is weaker than the previous one in that it talks only about the normal behavior of Buy. It will be extended to describe the abnormal behavior of Buy in Section 2.3.8. Interesting aspects of the above example include:

- The catch specifications to catch exceptions in the call state are moved into inline procedure declarations so as to reduce the clutter in the behavior description of Buy.
- The normal behavior of Buy is defined as any behavior that does not throw any exception. The abnormal behavior of Buy is defined as any that throws either `RuntimeException` or `BadCall`. If a checked invocation of Buy throws an Error or other Throwable, that will be reported as outside the specification.
- The main part of the behavior description of Buy is guarded by the “if” statement (Section 2.2) “if (normal) ...”. In this case, no exception is expected to be thrown

during the evaluation of the assertions, and hence a try/catch specification is used simply to report an error if any exception is thrown.

- Previous versions of this example mixed the description of normal and abnormal behavior. This version provides the beginnings of a clear separation which will be more apparent when the abnormal behavior is also completed in the next section.

2.3.8 The Exception Operator

ADL provides the exception operator $\langle : \rangle$ whose meaning is based on behavior classifications. In usual usage of this operator, the left operand is the enabler of an exception, while the right operand is a “thrown” expression.

Informally, $A \langle : \rangle B$ means that if A is true, then an abnormal condition should be detected but B is not necessarily true. However, if an abnormal condition is detected and B is also true, then A must be true as well. The first part of this rule allows the specification of abnormal conditions for functions that can raise several different abnormal status in a possibly non-deterministic way, e.g., several error conditions are met initially but we don’t care which one is raised as long as at least one of them is actually raised.

More formally, the exception operator is defined as:

$A \langle : \rangle B$ is the same as $((A ==> \text{abnormal}) \text{ AND } ((\text{abnormal} \ \&\& \ B) ==> A))$

As an example of the use of the exception operator, consider the following assertion group (we detour from the stock broker a bit here):

```
{
  !file_exists(f) <:> thrown(not_found);
  disk_full() <:> thrown(disk_error);
}
```

If we assume the default definition of **abnormal**, this assertion group could probably be used to specify a file open method. It reads: If the file f does not exist, then an exception must be thrown. Similarly, if the disk is full, an exception must be thrown. However, it does not restrict exceptions to be thrown for other reasons. But it does say that if the exception `not_found` was thrown, it must be the case that the file f does not exist, and if the exception `disk_error` was thrown, it must be the case that the disk was full. An interesting consequence is that if both the file does not exist and the disk is full, either exception may be thrown. The following assertion group strengthens the above assertion group to require that only these two exceptions or runtime exceptions may be thrown:

```
{
  !file_exists(f) <:> thrown(not_found);
  disk_full() <:> thrown(disk_error);
  abnormal ==> thrown(not_found, disk_error,
    RuntimeException);
}
```

Now the earlier stock broker example is completed with specification of abnormal behavior. A helper interface with two method declarations — `bad_acct` and `bad_sym` — are added:

```

interface StockAuxiliary {
    public boolean bad_acct(StockBroker sb, long account);
    public boolean bad_sym(String symbol);
}

```

EXAMPLE 2.10 StockBroker specification with exceptions

```

import stock.*;
import StockBrokerAux;

adlclass StockBroker2 {

    inline long cost(String s, long nsh) {
        try { nsh * StockBrokerAux.price(s); }
        catch(Throwable t) { 0; }
    }

    inline long _Cash_Balance(long act) {
        try { Cash_Balance(act); }
        catch(Throwable t) { 0; }
    }

    inline long _Stock_Balance(long act, String s) {
        try { Stock_Balance(act, s); }
        catch(Throwable t) { 0; }
    }

    void Buy(long account, String symbol, long no_of_shares)
        throws BadCall {
        semantics
        [ normal = !thrown(Throwable);
          abnormal = thrown(RuntimeException, BadCall); ]
        {
            StockAuxiliary.bad_acct(self, account) <:>
                (thrown(BadCall) &&
                 ((BadCall)adl_thrownException).bad_account);
            StockAuxiliary.bad_sym(symbol) <:> (thrown(BadCall) &&
                ((BadCall)adl_thrownException).bad_stock_symbol);
            if (thrown(BadCall)) {
                ((BadCall)adl_thrownException).bad_account ||
                ((BadCall)adl_thrownException).bad_stock_symbol;
            }
            if (normal) {
                try {
                    Cash_Balance(account) == @_Cash_Balance(account) -
                        @cost(symbol, no_of_shares);
                    Stock_Balance(account, symbol) ==
                        @_Stock_Balance(account, symbol) + no_of_shares;
                } // end try
                catch (Throwable t) { false; }
            }
        }
    }
}

```

```

        } // end if
      } // end semantics
    }
  }

```

Note the right hand side of the two exception operators refer to the same exception, but different additional conditions associated with the raising of the exception. This new behavior description is different from the earlier behavior description in Section 2.3.7 in a few interesting ways, some of which are:

- It makes clear the abnormal behavior. In the earlier example, the abnormal behavior was described through catch specifications in the normal behavior.
- This leaves the particular exception condition (right operand of the exception operator) that occurs non-deterministic. If the left operands of the exception operators are both true, then the behavior description allows either (or even both) the exception conditions to hold.
- It uses an additional interface, `StockAuxiliary`, to give additional information about accounts and symbols. It is often the case that an interface sufficient for normal use is not sufficient for testing; typically, it is useful to add operations to inspect the state of an object or to encapsulate complex actions. Within the constraints of Java, these additional operations can only be defined in a new interface. In some cases, the new interface can be implemented entirely separately from the interface under test; in other cases, it will require some collaboration between the objects.

2.4 Inline Procedure Declarations

Inline macro declarations is another way to define concepts used in behavior descriptions (along with auxiliary interface declarations). Their syntax is:

ADL_InlineDeclaration ::= “inline” ResultType MethodDeclarator **ADL_AssertionGroup**

Inline declarations are macros, in the usual C pre-processor meaning. The call to an inline is replaced by the text of the corresponding assertion group, with ad hoc substitution of the parameters.

In the Stockbroker example, where “cost” is defined as:

```

inline long cost(String s, long nsh) {
    try { nsh * StockBrokerAux.price(s); }
    catch(Throwable t) { 0; }
}

```

any expression `cost(symbol, no_of_shares)` will be replaced by:

```

{ try { no_of_shares * StockBrokerAux.price(symbol); }
  catch(Throwable t) { 0; } }

```

This is the second case (after try/catch specifications) of the two cases where an assertion group may have a non-boolean value.

It is not possible to define bindings or NLD annotations inside an inline declaration.

2.5 Prologues and Epilogues

Before being able to test a specified method, it is sometimes necessary to perform preliminary initializations that require imperative features: this cannot be made inside semantic assertions, which should remain declarative constructs with no side-effect.

For this purpose, the user can use the “**prolog**” and “**epilog**” features, which provide blocks of “pure” Java that will be transmitted without any transformation to the generated code.

There are two kinds of prologues/epilogues: either global (in **ADL_CompilationUnit**) or local (in **ADL_ClassDeclaration**).

ADL_ClassDeclaration ::= [“public”] “adlclass” <IDENTIFIER> [“extends” Name]
 “{” [**ADL_Prologue**] [**ADL_Epilogue**]
 (**ADL_InlineDeclaration** | **ADL_BehaviorDeclaration**) * “}”

ADL_BehaviorDeclaration ::= [“static”] [ResultType] Name FormalParameters [“throws” NameList]
 “{” [**ADL_Prologue**] **ADL_BehaviorSpecification** [**ADL_Epilogue**] “}”

ADL_Prologue ::= “prolog” Block

ADL_Epilogue ::= “epilog” Block

```
adlclass bankAccount {

    prolog {
        String url = "jdbc:odbc:wombat";
        DbConnection dbcon = DriverManager.getConnection(url);
        DbTable dbtbl = dbcon.createTable();
    }

    long deposit(long amt) {
        prolog {
            String sel = "SELECT p.* (...)";
            dbtbl.checkAssertion(sel);
            dbtbl.setInt(1, get_account());
        }
        semantics {
            dbtbl.execute(); // boolean-valued function
        }
        epilog {
            dbcon.close();
        }
    }

}
```

In the generated Java code for this example, the global and local prologue blocks are concatenated (the global before the local) and copied “as is” at the beginning of the

“deposit” generated method, before the code that deals with the semantic assertions. The epilog code is copied at the end of this method (a global epilog would be copied right after the local one).

The overall execution scheme is as follows:

- Step 1: Execution of the global prologue (except in constructors)
- Step 2: Execution of the local prologue
- Step 3: Evaluation and saving of call-state expressions
- Step 4: Call the method on the real object
- Step 5: Evaluation of the assertions and test reporting
- Step 6: Execution of the local epilogue
- Step 7: Execution of the global epilogue (except in constructors)

Note that the global prologue is a purely syntactic construct: variables declared therein are *not* global variables, but variables local to all the specified method — exactly like the variables declared in the local prologue. Its sole purpose is to factorize the statements that need to be executed at the beginning of *all* the methods whose behavior is specified in the adl class.

There is a special case for constructors: it is possible to define a local prologue in the behavior specification of a constructor, but the global prologue/epilogue are not imported in the generated code.

Call-state expressions and inlines cannot be used in prologues and epilogues. Bindings can be used in the local epilogue of the behavior where they are defined, but not in prologues and global epilogue. The global epilogue has only access to variables defined in itself and in the global prologue. It is possible, inside call-state expressions, to reference the variables declared in prologues.

3 Test Annotations

Test data annotations allow the test engineer to define how an interface should be tested, what data and what procedures should be used to exercise the functions in the interface.

3.1 Concepts

The test data description (TDD) language provides a notation in which the user can write descriptions of test sets, which will be processed into test driver programs. TDD is organized by a few concepts; these are presented in the first section, with syntactic details in later sections.

This document focuses on the TDD *language*. The integration of the TDD concepts with relation to the ADLT runtime is presented in more details in the ADL 2.0 Translation System document, chapter 7.

3.1.1 Re-write

The principle behind TDD2 (TDD for ADLT2) is that it is processed by re-writing the input to create a test program. The re-write does not remove any information.

The concepts of TDD2 are applied to a variety of programming languages, called target languages. The concepts of TDD2 are common to all target languages, and the syntax is in large measure common; the parts of the language that get re-written are common to our four target languages (C, C++, IDL, and Java).

3.1.2 Dataset

A dataset is a set of data values. It may be used in place of an expression in the target language syntax. The result of such an expression over a dataset is another dataset. An expression involving more than one dataset is treated as an expression over the Cartesian product of the datasets:

$$A \otimes B \equiv f_0(A, B) \equiv F_0(A \times B) \quad (\text{EQ 3})$$

Dataset Size. A dataset has a definite size, by construction. However, that size may not be feasible to use as a test. Examples of feasible datasets are enum types, array indices, array contents, and datasets created by literal expressions. The concept of feasibility is not precise; there is not an axiomatic way to decide if a dataset is small enough. In practice, a dataset with more than 2^{32} elements is certainly infeasible.

A dataset may be created by a literal expression or by a factory. A dataset may also be created by the combination of a representation type and a constraint. A single value, that is, an expression in the target language, is a trivial dataset.

Dataset size is determined by calculation rather than by construction. It is easy to combine a finite number of feasible datasets and create an infeasible dataset; 32 copies of a Boolean dataset, for example.

3.1.3 Factory

A factory is a data creator. It encapsulates the notions of a constructor, a destructor, and reporting.

A factory is, formally, a function from a dataset to a dataset. A function $f_n(A,B,C\dots)$ of more than one argument is formally treated as function f_1 of a single argument, $A \times B \times C \dots$ – the crossproduct of the input datasets.

Operationally, a factory is implemented by a pointwise function on the elements of the domain. In addition, the implementation of a factory includes a destructor function for elements of the range, and an association from an element of the range to the element of the domain.

The formal definition of a factory is:

$$F \equiv \{D, R, c, d, i\}$$

$$\begin{aligned} c &? \text{Functional? } D \rightarrow R \cup \{\perp\} \\ d &? \text{Functional? } R \rightarrow \{\emptyset, \perp\} \\ i &? \text{Functional? } F \rightarrow D \end{aligned}$$

where D is the domain of the factory, R is the range of the factory, c is the factory's constructor function, d is the factory's destructor function, and i is the inversion function, which can be used to determine the input that gave rise to a given range element.

While several of the target languages provide expression of these notions in their type structure, those expressions may be not available for all types needed for testing; for example, none of the target languages permit extension of the built-in types, and all allow the declaration of types which permit no extension. The factory notion is part of TDD2, outside the target language's type system, so that it can be applied to all types needed for testing.

3.1.4 Checked function

A checked function is a function for which an oracle is available. Calling a checked function produces the same value and outcome as calling the unchecked version of the function, but will report some measurement information as an invisible (within the calling program — not to the user!) side effect.

When running under a debugger, all functions may be said to be checked functions.

In the ADLT system, checked functions are generated from function declarations which have been annotated with semantic specifications. Within a test directive, there is a special convenient syntax for invocation of such an ADL-derived checked function; the class or object on which the method is invoked is enclosed in the ADL pseudo-function.

```
ADL(obj).meth(data);
obj.meth(data);
```

Both method invocations in this example result in invocation of method `meth` on the underlying implementation object `obj`; however, the checked method invocation is relayed through an *Assertion Checking Object* `ADL(obj)` that implements the semantic checks specified by the ADL semantic annotation. It is an error to invoke the ADL-checked version of a method if either the class type of the underlying object has not been annotated (there is no adl class that annotates this type) or if there exists such an adl class but that method does not have a semantic annotation in this adl class.

The scope of the ADL keyword operates on only one method: in the expression `ADL(obj).m1(p1).m2(p2)` the method `m1` is called on the ACO object created by `ADL(obj)` and therefore it is the checked method `m1` that is called; however, this method will create a usual object on which the *unchecked* method `m2` will be called. If the return type of `m1` has been annotated and the user wants to execute the checked method `m2`, he/she must write: `ADL(ADL(obj).m1(p1)).m2(p2)`

It is possible to test a constructor: the syntax `ADL_new Foo(bar)` will create an object of type `Foo` using a checked constructor. The scope is similar to the ADL clause: `ADL(ADL_new Foo(bar)).m(p)` invokes a checked constructor of class `Foo`, that returns a `Foo` object on which is invoked the checked method `m`.

It is possible to test an interface:

```
AnInterf x = new AClass();
ADL(x).meth(data);
```

where `AnInterf` is a Java interface and `AClass` is a Java class that implements `AnInterf`. The notation `ADL(x)` refers to an annotation of `AnInterf` (i.e. there must be an adl class `AnInterf.adl` that describes the behavior of the method `meth`). Even if `AClass` is also annotated, this annotation will not be called in this example: to test the annotation of `AClass`, one should write

```
AClass y = new AClass(); ADL(y).meth(data);
```

In other words, the notation `ADL(obj)` refers to the *static* type of `obj`, not to its *dynamic* type.

3.1.5 Test Directive

A test directive is formally a statement evaluated for side effect. In particular, a test directive normally includes an expression involving one or more calls to checked functions.

Note that a function or method body in a test declaration is subject to the same re-writing as any other code in the test declaration. Hence any call to a checked function, in such a body, will be interpreted as a call to the checked version of the function; and calling such a function or method will have the side-effect of making an observation about the behavior of such checked functions.

A test directive expression is parameterized by the datasets used in the test directive.

3.1.6 Assertion

An assertion is a Boolean expression. However, the test framework takes note of an assertion. An assertion is a postcondition. An assertion contributes to the test result and is reported to the user.

Formally, an assertion is a Boolean expression evaluated for side effect.

An assertion is expressed by a call to the function `tdd_assert(String, boolean)` from the ADLT runtime library. As a stretch feature, the ADLT translator may re-write the assertion to provide better reporting.

3.1.7 Importation

It is possible to import datasets or factories defined in other TDD files, by using the “use” feature of the TDD language. This feature is syntactically similar to the usual importation scheme of the target language: `#include` for C/C++ and `import` (with qualified name) for Java.

Note that this importation clause makes reference to the *source TDD file*, not to the object code obtained after ADLT translation and compilation. In TDD for Java, when the user declares “`use foo;`”, he can thereafter use for instance the dataset “D1” defined in the file `foo.tdd`. There is however an important difference with the “import” clause: with “use”, the compiler checks the presence and correctness of the source `tdd` file; it is left to the responsibility of the user to ensure that at runtime the class file obtained by transformation of the `bar.tdd` will be accessible from the classpath. This is closer to the C semantics, with the distinction between the header file for the compiler and the library at runtime.

3.2 Annotated TDD/Java Syntax

This is not the complete syntax for the TDD extensions to Java, but rather the productions that are additions or modifications from the language standard. Undefined nonterminals and terminals are references to the language standard.

3.2.1 TDD declarations

```
ADL_CompilationUnit ::= ( ImportDeclaration )*
                    ( TDD_UseDeclaration )*
                    TDD_ClassDeclaration
                    <EOF>

TDD_ClassDeclaration ::= [“public”] “tddclass” <IDENTIFIER>
                       “{ ( TDD_ClassBodyDeclaration ) * ”
```

TDD classes have no inheritance structure, as the concept of “superclass” is not meaningful for a TDD class. Therefore a TDD class may not be marked as “final” or “abstract”; all TDD classes are final, just as all Java interfaces are abstract. Neither may a TDD class extend another, although one TDD class may refer to another.

One TDD class may refer to Java entities or TDD datasets and/or factories declared in other compilation units, either using the fully qualified name of the external entity, or by importation (“import” for Java, “use” for TDD, referenced by the incpath or the class-path: see the ADL 2.0 Translation System Design Specification document, chapter 2).

TDD_ClassBodyDeclaration ::= MethodDeclaration
 | TDD_FieldDeclaration
 | TDD_DatasetDeclaration
 | TDD_FactoryDeclaration
 | TDD_TestDirective

The extra constructs that may occur in a TDD class are constant fields, datasets, factories, and test directives.

TDD_FieldDeclaration ::= Type VariableDeclaratorId “=” VariableInitializer
 (“,” VariableDeclaratorId “=” VariableInitializer) * “,”

tdd class fields are constant and must therefore be initialized. The initialization expression must be computable at compile-time (similar to the initialization of static fields in Java).

TDD_DatasetDeclaration ::= “dataset” TDD_SingleDeclarator “=” TDD_DatasetExpr “,”

A dataset is like an initialized declaration, except that a dataset may have type void, and the initializer is a dataset not just a scalar.

TDD_FactoryDeclaration ::= “factory” MethodDeclaration
 [“relinquish” “(“ [FormalParameter] “)” Block]

A factory is declared just like a method.

TDD_TestDirective ::= [<IDENTIFIER> “.”] “test” [“forall”]
 “(“ [TDD_DatasetDomain (“,” TDD_DatasetDomain) *] “)” Statement

A test directive is similar to an ADL quantified expression, and allows a similar syntax. It declares local variables that range over the contents of the specified datasets.

A test directive is implemented by putting it in the body of a method, suitable for invocation by the appropriate test framework. The test method declaration is left implicit, rather than being explicitly written as part of the test directive, so that the ADL translator can supply a test method declaration specialized for the test framework for which code is being generated.

TDD_DatasetDomain ::= TDD_SingleDeclarator (“=” | “.”) TDD_DatasetExpr
 | TDD_DatasetExpr

3.2.2 Dataset Expression Syntax

A dataset expression may be used alone in a domain list (i.e. without being assigned to a single declarator) only if it reduces to a void dataset expression, which must be pro-

duced by a factory; in that case, this notation denotes the evaluation of the void factory member for side effect only (see Section 3.3.3).

TDD_DatasetExpr ::= **TDD_DatasetConcatExpr**
 (“+” **TDD_DatasetConcatExpr**)*

TDD_DatasetConcatExpr ::= **TDD_DatasetSingleton**
 | **TDD_DatasetLiteral**
 | **TDD_FactoryCall**

TDD_DatasetSingleton ::= Literal | Name

TDD_DatasetLiteral ::= “{” [**TDD_DatasetMember** (“,” **TDD_DatasetMember**)* [“,”]] “}”

TDD_DatasetMember ::= ConditionalExpression [“..” ConditionalExpression]

TDD_FactoryCall ::= Name “(” [**TDD_DatasetExpr** (“,” **TDD_DatasetExpr**)*] “)”

A dataset literal is written in braces. It may be empty. The elements in a literal dataset may be expressions or ranges (in a range expression, the two dots “..” must be preceded and followed by blank space(s)). Ranges are only meaningful for integral types. A dataset expression that reduces to a literal or a local field name is converted into a one-element dataset literal. The expressions in a dataset literal are evaluated once for all when the dataset is initialized; they will not be re-evaluated at each selection.

Method invocation is legal only as **TDD_DatasetMember** (for members of a dataset literal) whereas factory invocation is legal only as **TDD_DatasetExpr**:

```
dataset int D = f1() + { f2() };
```

this definition is correct iff *f1* is a factory and *f2* a method.

A **TDD_DatasetSingleton** name must be either the name of a dataset or the name of a local field; a **TDD_FactoryCall** name must be the name of a factory.

3.3 General Syntax & Examples

This section presents the general syntax (for C-like languages) along with examples that motivate the design.

3.3.1 Simple Datasets and Data Construction

Some examples of data generation.

EXAMPLE 3.1 The Simplest Test

```
tddclass t1 {
  test (int i1 = 3, int i2 = 4)
    ADL(subject).plus(i1, i2);
}
```

The simplest test is just an invocation of an annotated function. Formally, this test directive is the application of the annotated function “plus” to the cross-product of two datasets, “{3}” and “{4}”; the promotion from a single value to a one-element dataset is implicit and automatic.

In the example, `subject.plus` is a static method name; `ADL(subject).plus` is the name of a checked version of that static method (we do not precise here what the name “subject” refers to; it is implicit that it has been imported through the target language standard class importation features: `#include` in C++, `import` in Java).

EXAMPLE 3.2 A Simple Dataset

```
tddclass t2 {
  dataset int A = {1,3,5 .. 7};
  test (int i1 = A, int i2 = 1)
    ADL(subject).plus(i1, i2);
  test (int i1 = 1, int i2 = A)
    ADL(subject).plus(i1, i2);
}
```

This tests `plus` when adding the constant dataset `{1}`, on both sides.

3.3.2 Compound Datasets: Factories, Concatenation

EXAMPLE 3.3 Compound Data Construction

```
tddclass t3 {

  factory RandomAccessFile make_file (String name,
                                       String mode) {
    return new RandomAccessFile(name, mode);
    //in TDD for Java
  } relinquish(RandomAccessFile r) { // ...
  }

  dataset File F0 = make_file(
    {"/dev/null", "/dev/tty", "/tmp/foo"},
    {"r", "rw"});

  dataset File F1 =
    make_file("/dev/null", "r") +
    make_file("/dev/tty", {"r", "rw"}) +
    make_file({util.tmpnam()}, {"rw"});

  byte[] buf = new byte[512];
```

```

test (RandomAccessFile F=F0) {
    ADL(F).readFully(buf);
}

test (RandomAccessFile F=F1) {
    ADL(F).readFully(buf);
}
}

```

Dataset F0 has $3 \times 2 = 6$ members, while F1 has $1 + 2 + 1 = 4$ members. Note that F1 is the concatenation of several datasets, each produced by a separate invocation of the factory; the example uses “+” as the dataset concatenation operator.

This example shows the syntax for a test directive, with the datasets listed explicitly as an initialized declaration list.

The optional relinquish clause has similar syntax to a Java catch clause. The relinquish clause takes a single argument whose type must match the return type of the factory method. In the body of the relinquish clause, the user has visibility to all the arguments of the factory method and the system guarantees that values used for the arguments in the preceding call to the factory method to create the return data, are the same when executing the call to the relinquish clause.

3.3.3 Void Datasets

EXAMPLE 3.4 Void Datasets

In order to express the notion of an environment condition that affects the operation of a system under test, without producing an assignable value, the concepts of dataset and factory are extended to allow void pseudo-values. This example imports datasets from the previous one, and shows the use of a block as the body of a test directive, completed with an assert.

```

use t3;

tddclass t4 {
    factory void setup_system(int condition_code) {
        // void factory
        // ...
    } relinquish() {...}

    dataset void setup_set = setup_system(1);

    test (setup_set, // void dataset expression
        RandomAccessFile F = F1, // F1 accessible through
                                // import of t3
        String data = {"", "hello"})
    {
        String tmp;
        ADL(F).writeUTF(data);
        F.seek(0);
    }
}

```

```

        tmp = ADL(F).readUTF();
        tdd_assert("tmp.equals(data)", tmp.equals(data));
    }
}

```

Note that, as the void dataset `setup_set` is defined as a factory call, this factory `setup_system` is called once for each test data instance (see also Section 3.3.4).

This example shows the use of an unchecked method (`seek`) in conjunction with some checked methods (`writeUTF` and `readUTF`). All three method invocations result in method invocations on the underlying implementation object `F`; however, the checked method invocations are relayed through a checking object that implements the semantic checks specified by the ADL semantic annotation.

Imported dataset names (through the “use” clause) can be unqualified if there is no ambiguity about their origin. Unqualified syntax (`F = F1` in the example) is possible if

- `F1` is defined in the current tdd class, or
- `F1` is defined in at most one of the “used” tdd classes.

If two tdd classes are imported (`use c1; use c2;`) such that a dataset `F1` is defined in `c1.tdd` and another in `c2.tdd`, then a call to `F1` must be qualified: `F = c1.F1`. A locally defined dataset name *hides* an imported dataset that has the same name. Importation is not transitive: if tdd class `c0` imports tdd class `c1` and `c1` imports `c2`, then `c0` does not import `c2` (unless it explicitly does so, of course).

The qualified syntax can also be used when the tdd class is not explicitly imported by a `use` clause. These rules are also valid for factory importation. Only datasets and factories are importable: the constants, test functions and test directives of a tdd class are not.

3.3.4 Dataset Elements Evaluation

EXAMPLE 3.5 Runtime Initializers

The elements of a dataset literal are evaluated only once, at initialization time (static evaluation). If the user wants a dataset whose elements are evaluated each time the dataset is referenced (dynamic evaluation), he must use factories.

```

tddclass t5 {
    /* this is not a good dataset; it lacks repeatability */
    dataset double q_static = {
        Math.random(),
        Math.random(),
        Math.random()
    };
    factory double rand() { return Math.random(); }
    dataset double q_dynamic = rand();
}

```

In this example, the dataset `q_static` is initialized with 3 random values that will not change whatever the number of test data instances that reference `q_static`. But for test directives that use `q_dynamic`, each reference to `q_dynamic` in each test data instance will be dynamically reevaluated.

3.3.5 Dataset Constants

EXAMPLE 3.6 Provide Test Variables

This example may be slightly familiar for those familiar with the ADLT1 example programs. The combination of a factory requiring one or more integer parameters with a dataset is the ADL/Java idiom for a provide test variable, just as a collection of static final ints is the Java idiom for an enum. In TDD, any global variable (field) is implicitly constant (static final in Java) and must be initialized at its declaration. A TDD constant cannot be imported through the “use” clause.

```
import Bank; // in TDD for Java

tddclass t6 {

    int SAVINGS = -1, CHECKING = 1, IRA = 7;
    int negative = -10, zero = 0, small = 3, average = 100,
        large = 1000, over_limit = 10000;

    factory account acct(
        int account_type,
        int size_code) { /*...*/ }

    dataset int act_code = { SAVINGS, CHECKING, IRA };

    dataset int size_code =
        { negative, zero, small, average, large, over_limit };

    dataset account Account1 = acct(act_code, size_code);

    factory int amount(int size_code) { /*...*/ }

    Bank bank = new Bank(/*...*/);

    test (account act = Account1, int size = size_code) {
        ADL(bank).deposit(act, amount(size));
        ADL(bank).withdraw(act, amount(size));
        ADL(bank).balance(act);
    }
}
```

EXAMPLE 3.7 Better Test Variables

Here is a more general collection of test variables, showing the increased power of TDD2.

```

import Bank;

tddclass t7 {

    int SAVINGS = -1, CHECKING = 1, IRA = 7;
    int negative = -10, zero = 0, small = 3, average = 100,
        large = 1000, over_limit = 10000;

    factory double amount(int size) { /*...*/ }

    dataset int size_code =
        { negative, zero, small, average, large, over_limit };

    dataset int account_type =
        { checking, savings, IRA, zero, neg, max, over_max };

    factory account make_acct(
        int type_code,
        double size) { /*...*/ }

    dataset account Acct = make_acct(
        account_type, amount(size_code));

    test (account act = Acct, int size = size_code) {
        ADL(bank).deposit(act, {0.1, 124.1e10, 1125.333});
        ADL(bank).withdraw(act, amount(size));
        ADL(bank).balance(act);
    }
}

```

This example is intended to motivate the separation between factories and datasets. The `make_acct` factory can be used to create a dataset with accounts of any size; the `Acct` dataset is the result of applying that factory to a specific set of amount values.

3.3.6 Test Directives

EXAMPLE 3.8 Test Directives and Procedures

Simple examples of test directives were given in the previous sections.

The syntax is:

```
[ label : ] test (type id = dataset,...) statement
```

Example:

```
import object.data;
```

```
tddclass t8 {
  Dir1 : test (Object o = data.obj) {
    ADL(o).hashCode();
  }
}
```

A test directive body has the same syntax as *statement* in the Java grammar; however, “test statement” is a misleading phrase. A label may be placed on a test directive; this will influence the generated code and the generated test documentation in some way.

In the syntax, local variables are created to range over the specified datasets. Syntactically; this is like an initialized declaration, but the initializer is a dataset expression. The declared variable ranges over the members of the dataset during test execution. The list may also contain a dataset expression denoting a dataset of type void, with no variable declared; in that case the dataset member selection, presumably by a factory, is evaluated for side effect only.

Not all programming language statements are legal test directives. For instance, a break statement is not a legal test statement.

EXAMPLE 3.9 Void Dataset Use

The syntax for using void datasets is as follows:

```
tddclass t9 {
  dataset int A = { 1,2,3 };
  factory void side_effect(int i) { /*...*/ }
  dataset void X = side_effect({0 .. 6});
  dataset float F = { f1(), f2(), f3() };
  test (int a = A, X, float f = F)
    ADL(tested).func(f,a);
}
```

In this example, *f* is the loop variable for the inner loop, and varies fastest. The middle loop is a selection over *X*, evaluated only for side effect. The outer test loop varies *a* over *A*.

3.3.7 Advanced Examples

EXAMPLE 3.10 Chaining Factories

```
import testframe.code;

tddclass t10 {
```

```

factory String make_file_name(
    boolean absolute,
    boolean device,
    boolean funny_chars,
    int length_code
) { /*...*/ }

dataset int length_code = { code.ZERO, code.ONE,
    code.MEDIUM, code.LONG, code.TOO_LONG };

dataset String file_name_set =
    make_file_name(true, false, false, 10);

factory File make_file(String file_name) { /*...*/ }

factory RandomAccessFile
    make_filestream(File f, String md) { /*...*/ }

dataset String legal_open_type = { "r", "rw" };

factory String illegal_open_type() { /*...*/ }

dataset String open_type =
    legal_open_type + illegal_open_type();

dataset File File_set =
    make_file(file_name_set);

dataset RandomAccessFile Stream_set =
    make_filestream(File_set, open_type);
}

```

This illustrates several techniques for re-using factories.

EXAMPLE 3.11 Multiple Data Values

In some cases it is useful to produce a group of values with a single dataset expression. Rather than inventing a new syntax for a group of values, we use the data construction mechanism (`class`) already present in the programming language.

For example, to construct a dataset containing pairs of host addresses and ports, you might use:

```

import java.net.*;
import testdata.io;
import foo.port_pair;

/* class port_pair is defined in package foo:

package foo;

```

```

class port_pair {
    String host;
    int port;
}

*/

public tddclass t11 {

    factory port_pair make_port_pair(int pp_code) { /*...*/ }

    dataset int port_pair_code = { 0 .. 10 };

    dataset port_pair Ports =
        make_port_pair(port_pair_code);

    test (byte[] data = { io.mydata }, port_pair pp = Ports) {
        Socket s = new Socket(pp.host, pp.port);
        ADL(s.getOutputStream()).write(data);
    }
}

```

EXAMPLE 3.12 Test By Example

More complex examples bring us to the concept of “Test by Example”: the test code is an example of typical code, or code fragments, the user would write to make use of the interface under test.

```

import java.io.*;
import testdata.io;

public tddclass t12 {

    void read_then_write(RandomAccessFile f, byte[] buf) {
        long pos;

        pos = f.getFilePointer();
        ADL(f).readFully(buf);
        ADL(f).seek(pos);
        ADL(f).writeFully(buf);
    }

    test (File f=t10.File_set)
        read_then_write(f, io.buf_set);
}

```

This defines and then calls a test procedure that, when executed, will check that the `readFully`, `seek`, and `writeFully` operate together correctly when used in this particular way. More exactly, the test procedure will exercise the methods together, giving the assertion-checking code a change to check the behavior of annotated methods.

This is not a good way to test for error handling; it may prove useful when checking the normal operation of an interface.

EXAMPLE 3.13 Multiple Dataset References

A single dataset may be used more than once in a single test directive. This results in independent iterations over the dataset. If the test author wants multiple references to the same value in one directive, it is necessary to declare multiple variables ranging on the same dataset.

```
tddclass t13 {  
  
    dataset int A = { 1, 2, 3 };  
  
    test (int a = A, int b = A)  
        ADL(Math).plus(a, b);           // 9 evaluations  
  
    test (int a = A) {  
        ADL(Math).plus(a, a);           // 3 evaluations  
    };  
}
```

4 NLD Annotations

Natural language annotations can be provided to improve the quality of generated descriptions of ADL and TDD expressions.

4.1 Concepts

The ADLT tool can generate natural language (NL) documentation describing the semantics of functions and the generated test driver. The quality of the generated documents can be improved by annotating the input files with natural language descriptions (NLD). These annotations describe translations for identifier names, and provide other configuration information for the ADLT NL system.

Standard Generalized Markup Language (SGML) is the foundation of the document generation system. ADLT renders ADL and TDD expressions into SGML entity declarations, exploiting any NLD annotations that the test engineer has provided. These entity declarations are processed together with a set of document template entity declarations to form a complete SGML document conforming to the DocBook 3.0 DTD. The final SGML document can be converted to specific output formats such as HTML or Unix manual pages, or incorporated in larger SGML documents. See the NLD and SGML section for more details.

Java can be annotated with NL information in several places. Briefly, it can be placed at top level, within an adlclass or TDD annotation, attached to inline, an annotated function or a test statement, or placed after the bindings in an ADL semantics group expression. The translations it provides apply throughout the scope and enclosed scopes (see Section 4.2).

NLD annotations introduce translation information for identifier names at a specific scope. Translations in outer scopes are shadowed or overridden by translations for the same identifier name within enclosed scopes.

When ADLT comes to generate a natural language rendering of an ADL or TDD expression it takes each identifier in the expression and determines whether the user has provided any NL translations for its name. It searches outwards from the scope declaring the identifier through its enclosing scopes until it finds a candidate translation that satisfies any constraints on usage (such as locale) defined by its predicates. It uses the first one it finds. If more than one satisfactory translation is found at the same scope level a warning is generated and one of the translations is arbitrarily selected.

For example, a translation for an identifier name can be provided at the top level scope and it will be found and used for any identifier with that name in any enclosed scope, unless an alternative translation is provided at a more local scope.

A subclass inherits the NL declarations of its superclass. An NL declaration for an identifier given in the subclass overrides any inherited NL declarations for that identifier name.

4.2 Syntax and Semantics

The inheritance of NL declarations follows the target language; in the case of Java, this means that NL declarations are inherited from the superclass and from any implemented interfaces, and may be overridden in a subclass or implementation class.

4.2.1 Simple Data Member Translation

Translations can be provided very close to where an identifier is declared by using an NL declaration in the same scope as the identifier.

```
/* Java source */
class C {
    public static int amount;
    // ...
}

/* ADL source */
adlclass C {
    nld { // top-level NLD annotations
        .amount = "the correct amount";
    }
}
```

This declares a translation for `amount` in the scope `C` (the dot. before `amount` refers to the current scope where the `nld` block is written, here the global scope of the `adl` class `C`). Any expression using an identifier named `amount` declared within `C` or one its enclosed scopes will be translated to use the declared string.

4.2.2 Simple Function Member Translation

Methods can have translations declared in a similar fashion.

```
class C {
    public int balance() { /*...*/ }
}

adlclass C {
    nld {
        .balance() = "the balance of the account";
    }
}
```

This declares a translation for `balance()` in the scope `C`. Any expression using a function identifier named `balance` declared within `C` or one of its enclosed scopes will be translated to use the declared string.

4.2.3 Out Of Line Translations

Translations do not have to be declared at the same place the identifiers are. The translations in the two previous examples could have been provided out of line, in a `.nld` file, by using fully scoped identifier names.

```

/* NLD source */
nld {
    C.amount = "the correct amount";
    C.balance() = "the balance of the account";
}

```

4.2.4 Translations For Overloaded Methods

As there may be more than one function with a particular name in a scope the function signature must be provided in its NL declaration.

```

class C {
    abstract void deposit(int amount);
    abstract void deposit(int amount, int charge);
    abstract void close_account();
}

nld {
    C.deposit(int) = "deposit some money";
    C.deposit(int, int) = "deposit some money and charge a fee";
    C.close_account() = "close the account";
}

```

These mappings are not preceded by a dot, which means they do not refer to a current scope but describe entities with a full name. The notation `C.deposit(*)` could be used to define a mapping common to all deposit methods. The notation `C.*.amount` could be used to define a mapping common to all local entities (parameter, binding) named "amount", in all methods of C.

4.2.5 Priorities

When several NL mappings are defined for an entity, they are distinguished one from another by an algorithm that detects the more "precise" one:

```

C.deposit(int).amount = "the deposit amount";
// has higher priority than
C.deposit(*).amount = "a deposit amount";
// which itself has higher priority than
C.*.amount = "the amount";
// which itself has higher priority than
C.amount = "the class amount";
// and finally the lowest priority for global scope
amount = "the global amount";

```

With the inheritance mechanism, this algorithm is refined by a prioritized "super class lookup": if `C.deposit(int).amount` is not found, the mapping will first be searched in super classes of C (from parent class of C up to `java.lang.Object`), and if not found the search will be launched on `C.deposit(*).amount` and so forth.

4.2.6 Using semantics And nld Blocks

A method can be annotated with both semantics and NL translations.

```

adlclass C {
    int balance(int ac) {
        semantics {
            ac != 0;
        }
        nld {
            .ac = "the account number";
            . = "the balance of the account";
        }
    }
}

```

The dot notation “.” refer to the current NLD scope (in this case the method `balance(int)`). The notation “`.ac`” is equivalent to using a fully scoped name to refer to the function’s local arguments.

```

nld {
    C.balance(int).ac = "the account number";
    C.balance(int) = "the balance of the account";
}

```

In case of a clash between two equivalent mappings, the final mapping is the *last* encountered one, knowing that NLD files are always parsed *before* ADL/TDD files (except for this rule, NLD files and ADL/TDD files are parsed in the order they appear on the command line). If the two equivalent mappings are defined in the very same file, the last occurrence is retained.

The formal argument name from the function declaration is used as the name of the local argument. The signature of the function must be given in order to disambiguate overloaded functions.

4.2.7 Shadowing or Overriding A Translation

```

nld {
    i = "the loop counter";
}

adlclass A {
    //public static int i; declared in class A
}

adlclass B {
    //public static int i; declared in class B
    nld {
        .i = "B's i";
    }
}

```

An expression using `i` in A will pick up the top level NL declaration for `i` and be translated as “the loop counter”. The NL declaration for `i` within B overrides the top level declaration so an expression using `i` in B will be translated as “B’s `i`”.

4.2.8 Overriding A Non-Local Translation

Translations in other scopes can be overridden too.

```
class A {
    public static boolean i;
    public abstract void f();
}

class B {
    public abstract void g();
}

adlclass A {
    void f() { //...
        nld {
            .i = "f's i";
        }
    }
}

adlclass B {
    void g() {
        semantics {
            A.i;
        }
        nld {
            A.i = "g's i";
        }
    }
}
```

The translation of the reference of `A.i` in the `B.g()` semantics block is “g’s i”, overriding the translation for `i` given in `A`.

4.2.9 Invocation Translation

An invocation translation is used to translate a function call. It provides a mechanism for the translation to refer to the translations of the actual arguments. In order to use this mechanism the function translation must be provided with the full function signature. The notation ‘\$1’, ‘\$2’, etc. in the mapping of a function definition refer to the first, second, etc. actual argument of the function call: the translation of the corresponding actual argument is used instead of any translation for the formal argument name.

```
class C {
    public int a;
    public abstract void f(int i, int j);
}

nld {
    C.a = "the actual argument";
```

```

C.f(int, int) = "using " + $1 + " and " + $2;
C.f(int, int).i = "the first formal argument";
};

```

An expression using `f(a, 3)` will be translated as “using the actual argument and 3”.

4.3 NLD Predicates

Each NL translation associates a list of predicates with an identifier name. Each predicate asserts certain attributes of the translation. The most important attribute is the actual translation text (which must be provided), but other attributes are also defined. Some predicates act as constraints to determine when the translation can be used in the generated documents. SGML entities can also be declared in the predicate list.

The order of predicates in the predicate list is not significant. A predicate can only be used once in a list. Future predicates might include markers for grammatical categories such as tense, gender or number.

4.3.1 Pre-defined Predicates

These predicates (there are currently three defined: call-state, negation and locale) provide a mechanism to select a mapping for a given situation.

For instance, consider:

```

amount = "the amount";
amount[@] = "the former amount";

```

The second mapping will be used to translate the identifier `amount` when it appears within the scope of a call-state (`@amount`) whereas the first one will be used in the other cases. If no mapping with the call-state predicate is defined, an appropriate translation text is synthesized from the basic translation (here `@amount` would be translated as “the previous value of the amount”). This predicate is useful in situations where the synthesized translation is clumsy or inappropriate.

The *negation* predicate (notation “!”) is used in a similar fashion for negation scopes.

Invocation translations apply for call-state and negation translations too.

Different languages require different translations. The `locale(<string>)` predicate can be used to mark a translation as being valid for the specified locale. A translation with the locale predicate is only considered when it matches the current system locale. This is usually configured by setting the `LANG` environment variable. See the `setlocale(3)` manual page for more details. A translation for an identifier name with a locale predicate that matches the current system locale takes preference over a translation with a different or unspecified locale.

It is possible to define a mapping for several predicates (e.g.,

```

amount[!,@,locale("fr")] = "...";

```


To define several mappings with different predicates, it is possible to use the extended syntax:

```
deposit(int, int) : {
  text = "the basic mapping";
  text[@] = "the callstate mapping";
  text[!] = "the negation mapping";
}
```

The notation `deposit(int) = "deposit an amount";` is in fact a shortcut for `deposit(int) : { text = "deposit an amount"; }`

An other possible shortcut is to declare the locale before the translation text:

```
deposit(int) "C" = "the mapping for locale C"; stands for
deposit(int) : {
  text[locale("C")] = "the mapping for locale C"; }
```

4.3.2 User-defined Predicates

A user-defined predicate is a mechanism to assert an attribute to a mapping, so that this mapping can be selected or not elsewhere.

We will give an example in french, a language with explicit gender:

```
maleCat = "chat", [male];
femaleCat = "chatte", [female];
colorOf(int) : "fr.FR" {
  text[$1[male]] = "la couleur du " + $1, [female];
  text[$1[female]] = "la couleur de la " + $1, [female];
}
```

The two first mappings state that `maleCat` and `femaleCat` correspond respectively to a masculine and feminine gender. The notation `$1[male]` is to select a mapping that corresponds to a function call with a first argument that has the "male" predicate. The function call `colorOf(femaleCat)` would therefore be translated as "la couleur de la chatte". The "female" predicate that is defined as an attribute to the `colorOf` method states that the word "couleur" is feminine; thus the expression `colorOf(maleCat)` could be used in a context where an expression with "female" predicate is expected (for a "stupid" example, `colorOf(colorOf(maleCat))` would be rendered as "la couleur de la couleur du chat").

4.4 NLD and SGML

ADLT generates documentation by emitting SGML entity declarations for descriptions of aspects of the annotated functions and test specification. These synthesized and user supplied entity declarations can be used with template entity declarations to produce complete SGML documents for subsequent processing. ADLT supplies templates and synthesizes entities based upon the DocBook 3.0 document type definition for constructing reference manual pages and test specification descriptions.

4.4.1 Reference Manual Document

ADLT processes each annotated function to generate a function file containing SGML entity declarations describing its synopsis, semantics and error conditions. This file can be parsed in conjunction with the supplied reference manual template to produce an SGML document conforming to the DocBook 3.0 `RefEntry` element. ADLT also provides tools to convert the final SGML document into other formats such as HTML or Unix manual pages.

The reference manual template file declares default values for some entities which the function file generated by ADLT can override. Here are the entities for which it is possible to generate a value in nld blocks (we call them “properties”):

`%description`: A general description of the function and/or the class. This can be specified by using the `description` property in the NL declaration for the function/class.

`%includes`: Unlike all other property declarations, the declared text of `includes` is processed before generating the property declaration to escape “<” characters.

`%purpose`: A short description of a function.

`%seeAlso`: A reference.

EXAMPLE 4.1 Using includes and description

```
adlclass C {
    void f() { //...
        nld {
            . : {
                %description = "Behavioral description";
                %purpose = "Short description";
                %seeAlso = "See the class Foo";
            }
        }
    }
}
```

This is equivalent to:

```
nld {
    C.f() : %description = "Behavioral description";
    C.f() : %purpose = ...
    //...
}
```

The implementation of ADLT includes an SGML DTD that defines the structure of these entities. Note that ADLT does *not* preprocess the strings that define these entities: it sends them without any modification, except for “<” and “>” in `%includes` (there is for instance no interpolation mechanism performed on these strings).

4.5 NLD for TDD

Test Data Description sources can also be annotated with nld blocks in order to generate SGML documentation files. There is however an important difference: as there are no assertions in TDD, there are no automated translation of any expression. Therefore the user may only write NLD annotations to provide *properties* (like %description) or SGML entities, that are gathered and rendered in the generated documentation.

EXAMPLE 4.2 NLD Annotations in a TDD class

```
tddclass datasetsCollection {

    nld {
        . : %description = "A collection of datasets.";
    }

    int NEG = -1; int ZERO = 0; int MAX = 100;
    nld {
        .NEG : %description = "a negative value";
        .ZERO : %description = "the null value";
        .MAX : %description = "the greatest value";
    }

    dataset int DEPOSITS = { NEG, ZERO, 7, MAX };
    dataset bank B_SINGLE = make_bank (10,0,DEPOSITS);

    nld {
        .DEPOSITS : %description = "Set of typical values."; }
        .B_SINGLE : %description = "bank.....";
    }

}
```

4.6 NLD and Localization

ADLT chooses translations for identifier names based on the current system locale. Each NL declaration can be marked with a specific locale that determines when the translation can be used. An adl annotation can specify the locale of all the NL declarations grouped within it by using the optional locale marker. Additionally each declaration can use the locale predicate to specify its individual locale. When a locale is specified for a NLD group, any other locale defined for a mapping within this group would be skipped.

If a translation has a locale specified it will only be selected as a candidate when that locale is the system locale. A translation without a locale specification is considered to be in the default locale, and will be selected as a candidate when no other translation specified with the current locale is available.

There are four areas where localization is necessary.

Identifier translations. The locale mechanism provides a way to produce a set of translations for Java and ADL identifiers that are restricted to one locale. They will be selected in preference to translations for the identifiers which do not have a locale specified.

User-specified entity declarations. The locale mechanism can also be used to mark user-supplied entity declarations with a specific locale.

Document templates. The translations and user-specified entities are merged with text in the document template files to produce the final SGML documents. The template files can be localized.

Sentence construction rules. ADLT uses a set of rules to construct descriptions of ADL expressions out of the identifier translation fragments. These rules take the form of a Prolog program that can be localized.

4.7 NLD Syntax

NLD_Annotation ::= "nld" [**NLD_Locale**] "{ (**NLD_Declaration** | **NLD_EntityDeclaration**) * }

NLD_Locale ::= <STRING_LITERAL>

Natural language information is attached to the ADL source with a natural language annotation. An annotation is introduced with the `nld` reserved word, an optional locale indicator and then a group of one or more NL declarations within braces. If the locale indicator is present it acts as if the locale predicate is specified for every translation in the group. For example,

```
nld "C" {
  ...
}
```

acts as if `locale("C")` is specified for each translation.

Each NL declaration is either a translation for a Java or ADL identifier, or a declaration for an SGML entity to be used for document generation.

The left hand side of each kind of declaration can contain a scoped name. In addition to the standard Java scoping `PackagePath.Class.Member` NLD also allows identifier names within a function member to be specified. This makes it possible to give translation information for a method's formal parameters and local ADL bindings. This is useful for specifying translations for identifier names from many packages, classes or methods

in one place, rather than forcing the test engineer to distribute NL information throughout the specification files.

EXAMPLE 4.3 Using Fully Scope Names

```
class C {
    public int i;
    public void f(int i);
}

nld {
    C.i = "translation for i";
    C.f(int) = "translation for f(int)";
    C.f(int).i = "translation for parameter i in f(int)";
}
```

If a function name is overloaded at a particular scope it must have its signature fully specified. Otherwise it can be abbreviated to omit the declarations for the formal parameters and use only the notation “(*)”.

The translation information is entered at the specified scope (referred to as “.”), so an expression rendered at the current scope, or within an enclosed scope can find it.

NLD_Declaration ::= **NLD_ScopedName**
 ([**NLD_Locale**] **NLD_TextAssignment**
 |
 “.” [**NLD_Locale**] (**NLD_Statement** | “{“ **NLD_Statement**⁺ “}”))

NLD_Statement ::= **NLD_PropertyDeclaration** | “%text” **NLD_TextAssignment** “;”

NLD_TextAssignment ::= [**NLD_SelectPred**] “=” **NLD_String**
 [“,” “[“ **NLD_UserPred** (“,” **NLD_UserPred**)^{*} “]”]

NLD_SelectPred ::= “[“ **NLD_Predicate** (“,” **NLD_Predicate**)^{*} “]”

NLD_Predicate ::= **NLD_PredefinedPred**
 | **NLD_ParamNumber** “[“ **NLD_UserPred** (“,” **NLD_UserPred**)^{*} “]”

NLD_PredefinedPred ::= “@” | “!” | “locale” (“(“ **NLD_Locale** “)”

NLD_UserPred ::= <IDENTIFIER>

NLD_ParamNumber ::= “\$”<INTEGER_LITERAL>

NLD_ScopedName ::= “.”
 | **NLD_MethodName**
 | [**NLD_Scope** “.”] **NLD_Identifier**

NLD_MethodName ::= Name **NLD_Signature**

NLD_Signature ::= “((“*” | Type (“,” Type) *) “)”

NLD_Scope ::= “*”
 | “.”
 | Name [“.”]
 | **NLD_MethodName**

SGML entities can also be declared in an NL annotation. The text declared as the value of the entity is not examined by ADLT, it is passed on to the SGML back end uninterpreted and unmodified. For example,

```
&gen-ent = "a general entity";
```

declares a general entity with the specified value.

NLD_EntityDeclaration ::= “&” <IDENTIFIER> “=” **NLD_EntityText**

NLD_PropertyDeclaration ::= **NLD_PropertyName** “=” **NLD_EntityText**

NLD_PropertyName ::= “%description” | “%includes” | “%purpose” | “%seeAlso”

NLD_EntityText ::= <STRING_LITERAL> (“+” <STRING_LITERAL>) *

With the exception of the notation for string literals, the SGML syntax for entity names and values is used. See the SGML Handbook for details. NLD specifies string literals with a notation based upon the Java language.

NLD_String ::= **NLD_StringElem** (“+” **NLD_StringElem**) *

NLD_StringElem ::= <STRING_LITERAL> | **NLD_ParamNumber**

See the Java grammar for descriptions of the *Name* and *Type* nonterminals.

5 Testing CORBA Objects

Writing test suites for CORBA objects whose specifications were supplied using CORBA IDL is mostly the same as developing Java client and server code based on an IDL interface. It is straightforward since part of the IDL to Java mapping is automated by relevant tools. However the test engineer should take care when supplying semantics and test objects to take all the IDL to Java mapping aspects into account. The remaining of this section provides guidelines for the test engineer designing tests for CORBA objects using ADL/Java.

5.1 Mapped built-in types

Test engineer should take care of the mapping constraints for the CORBA built-in types. For example CORBA long type is mapped to Java int type. The original and mapped types are equivalent but it may be somewhat confusing to use mapped types.

The simplest way to avoid getting confused is simply to write ADL tests directly from the mapped Java interface instead of using the original IDL interface.

5.2 Bindings for objects

Take care when using bindings to CORBA objects to specify ADL semantics. CORBA objects are created using a CORBA factory and not using the natural Java “new” operator. While this is an advantage when writing TDD factories, this may be confusing when specifying simple ADL bindings.

For example instead of writing an ADL/IDL binding:

```
define Foo foo with foo = new Foo(1,1);
```

test engineer should write:

```
// ... code to get a reference on a Foo object factory
// called myFooFactory

// Creation of a foo object
define Foo foo with foo = myFooFactory.create(1,1);
```

5.3 Inout parameters passing

This section explains how IDL inout parameters should be managed in ADL/Java. We distinguish two cases: either an isolated variable (non member of a class/struct/union), or a member attribute from a class/struct/union.

5.3.1 Isolated variables

In this case the mapping applied by the ADL/IDL translator is simple: if a variable is used as an out/inout parameter at any time, declare it with its corresponding holder type,

and replace any occurrence of the variable by `variable.value` except when it's used as out/inout parameter where no replacement is needed.

Let's consider the following idl interface and its corresponding java mapping.

EXAMPLE 4.4 Isolated variable used as inout parameter

```
// IDL file - Bank.idl -----

interface Bank {
    void cash_balance(in long account, out long cash);
    void deposit(in long account, in long amount);
}

// Bank.Java obtained after applying idl to java mapping

class Bank {
    void cash_balance(int account, IntHolder cash);
    void deposit(int account, int amount);
}

// ADL file - Bank.adl (ADL/Java) to be written

import Bank;
adlclass Bank {
    void Bank.deposit(int account, int amount) {
        semantics {
            define IntHolder pre_cash_balance with
                @cash_balance(account, pre_cash_balance);
            define IntHolder post_cash_balance with
                cash_balance(account, post_cash_balance);
            post_cash_balance.value == pre_cash_balance.value
                                   + amount;
        }
    }
}
```

5.3.2 Member attribute of interface/structs/unions and inout/out parameters

What we did to use a local variable as an out/inout parameter (changing its type to its associated holder, e.g. `int` -> `IntHolder`) does not work for attributes members of an interface/union/struct. The other approach is to use intermediate holders each time the member attribute is used as an out/inout parameter, and to wrap each occurrence of this with getter and setter methods so that the attribute and the holder are synchronized.

EXAMPLE 4.5 Out/inout parameters and member attributes

```
// Interface in IDL -----
```



```

interface Foo {
    long  member; // member attribute
    long  sideEffects(inout long a);
}

// Generated Java -----

interface Foo {
    int  member();           // get method
    void member(int value);  // set method
    int sideEffects(IntHolder a);
}

// Following is an ADL prolog expressed in pseudo
// ADL/IDL that we would like to express in ADL/Java

//prolog {
// int aValue = sideEffects(member);
//}

// equivalent ADL/Java

prolog {
    // set holder value
    IntHolder memberHolder = new IntHolder(member());
    int aValue = sideEffects(upperBound, memberHolder);
    // get holder value back
    member(memberHolder.value);
}

```

5.4 Mapping for certain nested types.

The rule specified in [IDLJava] is applied. Note it is not a natural way to do.

```

// IDL
module Example {
    interface Foo {
        exception E1 {};
    }
}

// Java
package Example.FooPackage;
final public class E1 extends org.omg.CORBA.UserException {
    ...
}

```

5.5 Interface inheritance

Interface inheritance does not imply implementation inheritance: avoid using the keyword `super` when specifying CORBA object semantics.

5.6 Writing CORBA specific code in TDD/Java.

A CORBA client application, an ADL/Java test suite for example, is mostly the same as a non CORBA application except for two important features: a CORBA client needs to initialize the ORB, and also to get a CORBA entry point, that is a CORBA object from which other objects can be found and used.

After that, a CORBA client application acts just as if it was a non CORBA application, that is it creates, finds, and manipulates objects.

Some good news for test engineers is that TDD factories can easily be built upon existing CORBA factories (CORBA factories is the normal way to create new objects in CORBA applications).

The CORBA API that the test engineer may use to initialize the ORB and get an entry point are any CORBA API provided by the underlying CORBA platform. Using the portable CORBA APIs as defined by the OMG is strongly recommended to develop portable test suites. But since few ORB vendors are fully compliant with these portable APIs, the test developer may use any CORBA API provided with a CORBA platform (high level API such as that of Sun's Joe is ok).

Note that the ADL environment provide a portable CORBA ORB wrapper (see next chapter) that can be used instead of a proprietary ORB in order to reach maximum portability.

As an example let's consider the following example TDD/Java that could be written in TDD/Java to specify a bank factory, which is the CORBA entry point for our bank tests:

EXAMPLE 4.6 Bank factory TDD example

```
import Bank;
import org.omg.CORBA.*;

tddclass bankTest {

    BankFactoryHolder bankFactory =
        new BankFactoryHolder(null);
        // Note a holder is used here
        // because tdd fields shall
        // be constant!

    factory Bank makeBank (int maxAccts,
                          int numAcctsActive,
                          int initDeposit ) {
        org.omg.CORBA.Object obj;
        Bank bank;

        if ( bankFactoryHolder.value == null )
        {
            // Initialize the ORB
            ORB orb = ORB.init();
```

```

// Get a ref on a CORBA bank factory
// (which is the CORBA entry point)
try {
    obj = orb.resolve_initial_references
        ("BankFactoryID")
} catch (org.omg.CORBA.SystemException exc) {
    tdd_end_case();
}

// Narrow the CORBA object reference to its
// actual type
try {
    bankFactory.value =
        BankFactoryHelper.narrow(obj);
} catch (org.omg.CORBA.SystemException exc) {
    tdd_end_case("exception caught when getting"
        + " initial reference");
}

// Create bank
bank = bankFactory.value.createBank(maxAccts);

// Create accounts
for (int x=0; x < maxAccts; x++) {
    try {
        BankAccount ba =
            bank.openAccount(initDeposit);
    } catch (BankException be) {
        tdd_result(ADL_FAIL,
            "exception caught on open");
    }
}
} relinquish (Bank b) {
    if (numAcctsActive > 0) {
        b.close_all_accounts();
    }
}
}

```

5.7 ORBWrapper

In order to provide CORBA test suites with maximum portability, the ADL distribution contains an abstract class called `ORBWrapper` which is the superclass for wrappers for third party vendor Java ORBs.

The concrete subclass of the ORB wrapper is selected using the **adl.orbclass** system property which gives the fully qualified name of the real ORB wrapper.

Internally, the reflection API is used to instantiate the real ORB wrapper. Hence, it is not necessary to have all the real ORB wrapper compiled, only that referenced by the **adl.orbclass** property shall be compiled and available at runtime.

The ORBWrapper class implements a subset of the CORBA ORB methods, which is portable and which maps well on the different Java ORB third party products. The ORBWrapper class provide methods:

- to initialize the ORB (both for an applet and an application)
- method to get and make stringified object reference
- to connect and disconnect CORBA skeleton to the ORB.

This subset of ORB methods is both for client side and server side, but note that an ADL test suite may have to use only the client side methods, for example if the CORBA server is on a distant machine on the network and implemented without using Java but a different language.

5.7.1 ORB Wrapper Design

The abstract class ORBWrapper has the following methods:

```
public static ORBWrapper init(String args[],
                              Properties props);
```

Create a new ORB instance for an application. Parameter args is the application strings from main() method. Parameter props contains application-specific properties. The method returns a reference to the real ORB wrapper implementation.

```
public static ORBWrapper init(Applet app,
                              Properties props);
```

Create a new ORB instance for an applet. Parameter app is the applet. Parameter props contains applet-specific properties. The method returns a reference to the real ORB wrapper implementation.

```
public abstract Object string_to_object(String str);
```

Convert a string to a CORBA object reference¹.

```
public abstract String object_to_string(Object obj);
```

Convert a CORBA object reference to a string¹.

```
public abstract void connect(Object skeleton);
```

Connect a CORBA object to the ORB. All objects created by the user become ORB objects only after connect is called. Parameter skeleton is the object to connect¹.

```
public abstract void disconnect(Object skeleton);
```

1. Note that CORBA objects are passed or returned by the ORBWrapper as java.lang.Object instead of org.omg.CORBA.Object, actual typecast to org.omg.CORBA.Object is let to the wrapper.

Disconnect a CORBA object from the ORB. To stop an ORB object from receiving remote invocations disconnect needs to be called. Parameter skeleton is the object to disconnect¹.

Concrete subclass of the ORBWrapper are in charge of actually implementing the wrapper's methods.

The class diagram is as follows :

```
ORBWrapper  <-- inherits --  SunORBWrapper
              <-- inherits --  VbORBWrapper1
              ...
```

Note: the wrapped Java ORB is required to supply portable classes for org.omg.CORBA.Object (necessary for CORBA references narrowing) and all the built-in types Holders (IntHolder... - necessary for out/inout parameters passing). If not, the application could not easily make use of the generated CORBA stubs and skeletons. For such Java ORBs, direct use of the proprietary API is required in the tests.

5.7.2 Adl environment property to use for the ORB wrapper.

The ADL environment provides:

- a property **adl.orbclass** indicating the real ORB wrapper class to use for the tests,
- a property **adl.initialreferencefile** indicating the file containing the stringified object reference of a CORBA initial object reference. The string_to_object can then be used to get this initial object reference.

Note: how the file containing the stringified initial object reference is created and how the ORB servant objects are registered is not part the test, ORB server can for example be implemented in another language and be located on a distant machine. Anyway in some cases the test scenario shall perform these tasks.

5.7.3 Using the wrapper

A scenario for using the ORB wrapper is described in this section, both client side (the TDD test class) and server side (creation and connection of the CORBA server to the ORB) are showed here. In most cases only the client side should be part of the test suite.

5.7.3.1 Create CORBA objects in TDD factories

Normally, only a single CORBA object (CORBA initial reference) should be obtained from the ORB. This initial reference shall be a CORBA factory from which other CORBA objects can be obtained.

An example of such a TDD file is:

1. Not implemented in this release.

```
import java.io.*;
import org.opengroup.adl.orbwrapper.ORBWrapper;

tddclass bankTest
{
    // the initial object reference
    BankFactoryHolder bankFactory =
        new BankFactoryHolder(null);

    factory bank make_bank (int maxAccts,
                           int numAcctsActive,
                           int initDeposit)
    {
        if ( bankFactory.value == null )
        {
            // create and initialize the ORB wrapper
            ORBWrapper orb = ORBWrapper.init(args,null);

            // Get the stringified initial object reference
            String fileName =
                System.getProperty
                ("adl.initialreferencefile");
            BufferedReader br = new BufferedReader
                (new FileReader(fileName));
            String ior = br.readLine();
            br.close();

            // Get the actual initial object reference
            // here : a bankFactory
            // note bankFactoryHelper is a helper class
            // generated by the IDL to Java compiler
            Object obj = orb.string_to_object(ior);
            bankFactory.value = bankFactoryHelper.narrow
                ((org.omg.CORBA.Object)obj);
        }

        // now use the factory to create banks as in non CORBA
        //test suites
        bank ret = bankFactory.value.makeBank(maxAccts);
        ...
        return ret;
    }

    // these factory does not use the ORB to create objects, it
    // just use a bank object that has been obtained from the
    // top level factory

    factory bankAccount make_account (bank aBank,
                                      int initDeposit)
    {
        bankAccount ret = aBank.openAccount(initDeposit);
        return ret;
    }
}
```

5.7.3.2 Create and connect a CORBA bank server

A bank server class shall also be supplied to create the bank factory and the initial object reference file. Again the portable wrapper can be used.

```
import java.io.*;
import org.omg.CORBA.*;
import org.opengroup.adl.orbwrapper.ORBWrapper;

public class bankServer {

    public static ORBWrapper _orb;

    // Main method.
    public static void main(String[] args)
    {
        try
        {
            // create and initialize the ORB wrapper
            orb = ORBWrapper.init(args, null);

            // create servant and register it with ORB
            bankFactoryImpl servant = new bankFactoryImpl();
            orb.connect(servant);

            // put a stringified reference to the servant in
            // a file for further retrieving by a client
            String iorFileName =
                System.getProperty("adl.initialreferencefile");
            PrintWriter pw = new PrintWriter
                (new FileWriter(iorFileName));
            pw.print(_orb.object_to_string(servant));
            pw.close();

            // wait for invocations from clients
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }
        }
        catch (Exception ex) {
            ex.printStackTrace();
            System.err.println
                ("Bank Server: Exception-> " + ex);
        }
    }
}
```

5.8 References

[IDL/Java] "IDL/Java Language Mapping" OMG TC Document orbos/97-03-01

6 Complete Grammar

Here is the complete BNF grammar for ADL for Java. Non-terminals in boldface are defined in this document; other non-terminals are part of the Java language definition. Lexical entities, names in angle brackets like <THIS>, are references to the Java language standard.

Although, for practical reasons, the grammars for ADL and TDD are mixed in the following grammar, they are in fact distinct. From a TDD annotation, all rules involving an ADL non-terminal will be forbidden (by a semantic check), except for the derivation: UnaryExpressionNotPlusMinus -> ADL_BasicExpression -> PrimaryExpression.

Much of this grammar reproduces the Java expression syntax, with the extensions detailed in this document. The differences are intended as extensions; any Java expression not accepted by this grammar is a bug.

ADL_CompilationUnit ::= (**ImportDeclaration**)*
 (**TDD_UseDeclaration**)*
 (**NLD_Annotation**)*
 ADL_ClassDeclaration | **TDD_ClassDeclaration**
 | (**NLD_Annotation**)*
 <EOF>

ImportDeclaration ::= "import" Name ["." "*"] ",",

TDD_UseDeclaration ::= "use" <IDENTIFIER> ",",

ADL_ClassDeclaration ::= ["public"] "adlclass" Name ["extends" Name]
 "{" [**ADL_Prologue**] [**ADL_Epilogue**]
 (**ADL_InlineDeclaration** | **ADL_BehaviorDeclaration**
 (**NLD_Annotation**)*)* "}"

ADL_InlineDeclaration ::= "inline" ResultType MethodDeclarator **ADL_AssertionGroup**

ADL_BehaviorDeclaration ::= ("static" | "public")* [ResultType] Name FormalParameters
 ["throws" NameList] ["{"] [**ADL_Prologue**] **ADL_BehaviorSpecification**
 [**ADL_Epilogue**] (**NLD_Annotation**)* ("," | "}")

ADL_Prologue ::= "prolog" Block

ADL_Epilogue ::= "epilog" Block

ADL_BehaviorSpecification ::= "semantics" [**ADL_BehaviorClassification**] **ADL_AssertionGroup**

ADL_BehaviorClassification ::= "[(**ADL_NormalBehavior** | **ADL_AbNormalBehavior**)+ "]"

ADL_NormalBehavior ::= "normal" "=" **ADL_Expression** ",",

ADL_AbNormalBehavior ::= "abnormal" "=" **ADL_Expression** ",",

ADL_AssertionGroup ::= {“(**ADL_Binding**)* [“super” “.” “semantics” “,”]
(**ADL_Statement**)* (**NLD_Annotation**)* “}”

ADL_Bindings ::= “define” FormalParameter “with” [<IDENTIFIER> “=”] **ADL_Expression** “,”

ADL_Statement ::= **ADL_IfStatement**
| **ADL_TryStatement**
| **ADL_Assertion**

ADL_IfStatement ::= “if” “(“ **ADL_Expression** “)” **ADL_AssertionGroup**
[“else” (**ADL_AssertionGroup** | **ADL_IfStatement**)]

ADL_TryStatement ::= “try” **ADL_AssertionGroup**
(“catch” “(“ FormalParameter “)” **ADL_AssertionGroup**)+

ADL_Quantifier ::= “forall” | “exists”

ADL_Assertion ::= [**ADL_Label**] [**ADL_Tags**]
(**ADL_Expression** | **ADL_QuantifiedAssertion**) “,”

ADL_Label ::= <IDENTIFIER> “.”

ADL_Tags ::= “[“ <IDENTIFIER> (“,” <IDENTIFIER>)* “]”

ADL_QuantifiedAssertion ::= **ADL_Quantifier** “(“ **ADL_DomainList** “)” **ADL_AssertionGroup**

ADL_Expression ::= **ADL_ImplExpression**

ADL_ImplExpression ::= ConditionalExpression [**ADL_ImplOp** ConditionalExpression]

ADL_ImplOp ::= “==>” | “<==” | “<=>” | “<:>”

ConditionalExpression ::= ConditionalOrExpression
[“?” ConditionalExpression “.” ConditionalExpression]

ConditionalOrExpression ::= ConditionalAndExpression (“||” ConditionalAndExpression)*

ConditionalAndExpression ::= InclusiveOrExpression (“&&” InclusiveOrExpression)*

InclusiveOrExpression ::= ExclusiveOrExpression (“|” ExclusiveOrExpression)*

ExclusiveOrExpression ::= AndExpression (“^” AndExpression)*

AndExpression ::= EqualityExpression (“&” EqualityExpression)*

EqualityExpression ::= InstanceOfExpression ((“==” | “!=”) InstanceOfExpression)*

InstanceOfExpression ::= RelationalExpression [“instanceof” Type]

RelationalExpression ::= ShiftExpression ((“<” | “>” | “<=” | “>=”) ShiftExpression)*

ShiftExpression ::= AdditiveExpression ((“<<” | “>>” | “>>>”) AdditiveExpression)*

AdditiveExpression ::= MultiplicativeExpression (("+" | "-") MultiplicativeExpression)
 MultiplicativeExpression ::= UnaryExpression (("*" | "/" | "%") UnaryExpression)
 UnaryExpression ::= ("+" | "-") UnaryExpression
 | PreIncrementExpression
 | PreDecrementExpression
 | UnaryExpressionNotPlusMinus
 PreIncrementExpression ::= "++" PrimaryExpression
 PreDecrementExpression ::= "--" PrimaryExpression
 UnaryExpressionNotPlusMinus ::= ("~" | "!") UnaryExpression
 | "@" **ADL_CallStateExpression**
 | CastExpression
 | **ADL_BasicExpression**
ADL_CallStateExpression ::= UnaryExpression
 CastExpression ::= "(" PrimitiveType ("[" "]") * ")" UnaryExpression
 | "(" Name ("[" "]") * ")" UnaryExpressionNotPlusMinus
ADL_BasicExpression ::= PostfixExpression
 | "normal"
 | "abnormal"
 | "unchanged" Arguments
 | "return"
 | **ADL_AssertionGroup**
 | **ADL_ThrownExpression**
 PrimaryExpression ::= **TDD_ADLExpression** (PrimarySuffix)
 | **TDD_ADLnewExpression** (PrimarySuffix)
 | PrimaryPrefix (PrimarySuffix)
TDD_ADLnewExpression ::= "ADL_new" Name
TDD_ADLExpression ::= "ADL" "(" PrimaryExpression ")"
 PrimaryPrefix ::= Literal
 | Name
 | "this"
 | "super" "." <IDENTIFIER>
 | ParenthExpression
 | AllocationExpression
 ParenthExpression ::= "(" **ADL_Expression** ")"
 PrimarySuffix ::= BracketSuffix

```

        | DotSuffix
        | Arguments

DotSuffix ::= "." <IDENTIFIER>

BracketSuffix ::= "[" ADL_Expression "]"

Literal ::= <INTEGER_LITERAL>
        | <FLOATING_POINT_LITERAL>
        | <CHARACTER_LITERAL>
        | <STRING_LITERAL>
        | BooleanLiteral
        | NullLiteral

BooleanLiteral ::= "true" | "false"

NullLiteral ::= "null"

Arguments ::= "(" [ ArgumentList ] ")"

ArgumentList ::= ADL_Expression ( "," ADL_Expression ) *

AllocationExpression ::= "new" PrimitiveType ArrayDimensions
        | "new" Name ( Arguments | ArrayDimensions )

ArrayDimensions ::= ( "[" ADL_Expression "]" ) + ( "[" "]" ) *

ADL_ThrownExpression ::= "thrown" "(" NameList ")"

ADL_DomainList ::= ADL_Domain ( "," ADL_Domain ) *

ADL_Domain ::= FormalParameter ":" ConditionalExpression

VariableDeclaratorId ::= <IDENTIFIER> ( "[" "]" ) *

MethodDeclarator ::= <IDENTIFIER> FormalParameters ( "[" "]" ) *

FormalParameters ::= "(" [ FormalParameter ( "," FormalParameter ) * ] ")"

FormalParameter ::= Type VariableDeclaratorId

Type ::= ( PrimitiveType | Name ) ( "[" "]" ) *

PrimitiveType ::= "boolean" | "char" | "byte" | "short" | "int" | "long" | "float" | "double"

ResultType ::= "void"
        | Type

Name ::= <IDENTIFIER> ( "." <IDENTIFIER> ) *

NameList ::= Name ( "," Name ) *

```

TDD_ClassDeclaration ::= ["public"] "tddclass" <IDENTIFIER>
 "{ (**TDD_ClassBodyDeclaration**) * }"

TDD_ClassBodyDeclaration ::= MethodDeclaration
 | **NLD_Annotation**
 | **TDD_FieldDeclaration**
 | **TDD_DatasetDeclaration**
 | **TDD_FactoryDeclaration**
 | **TDD_TestDirective**

TDD_FieldDeclaration ::= Type VariableDeclaratorId "=" VariableInitializer
 ("," VariableDeclaratorId "=" VariableInitializer) * ","

TDD_DatasetDeclaration ::= "dataset" **TDD_SingleDeclarator** "=" **TDD_DatasetExpr** ";"

TDD_FactoryDeclaration ::= "factory" MethodDeclaration
 ["relinquish" "([FormalParameter])" Block]

TDD_TestDirective ::= [<IDENTIFIER> "."] "test" ["forall"]
 "([**TDD_DatasetDomain** ("," **TDD_DatasetDomain**) *])" **NLD_Annotation** * Statement

TDD_DatasetDomain ::= **TDD_SingleDeclarator** (":" | "=") **TDD_DatasetExpr**
 | **TDD_DatasetExpr**

TDD_SingleDeclarator ::= ResultType VariableDeclaratorId

TDD_DatasetExpr ::= **TDD_DatasetConcatExpr**
 ("+" **TDD_DatasetConcatExpr**) *

TDD_DatasetConcatExpr ::= **TDD_DatasetSingleton**
 | **TDD_DatasetLiteral**
 | **TDD_FactoryCall**

TDD_DatasetSingleton ::= Literal | Name

TDD_DatasetLiteral ::= "{ [**TDD_DatasetMember** ("," **TDD_DatasetMember**) * [","]] }"

TDD_DatasetMember ::= ConditionalExpression [".." ConditionalExpression]

TDD_FactoryCall ::= Name "([**TDD_DatasetExpr** ("," **TDD_DatasetExpr**) *])"

ClassDeclaration ::= ("abstract" | "final" | "public") * "class" <IDENTIFIER>
 ["extends" Name] ["implements" NameList]
 "{ (**ClassBodyDeclaration**) * }"

ClassBodyDeclaration ::= StaticInitializer
 | ConstructorDeclaration
 | MethodDeclaration
 | FieldDeclaration

InterfaceDeclaration ::= ("abstract" | "public") * "interface" <IDENTIFIER> ["extends" NameList]
 "{ " (InterfaceMemberDeclaration) * "}"

InterfaceMemberDeclaration ::= MethodDeclaration | FieldDeclaration

FieldDeclaration ::= ("public" | "protected" | "private" | "static" | "final" | "transient" | "volatile") *
 Type VariableDeclarator (" , " VariableDeclarator) * " , "

StaticInitializer ::= "static" Block

ConstructorDeclaration ::= ["public" | "protected" | "private"]
 <IDENTIFIER> FormalParameters ["throws" NameList]
 "{ " [ExplicitConstructorInvocation] (BlockStatement) * "}"

MethodDeclaration ::= ("public" | "protected" | "private" | "static" | "abstract" | "final" | "native" |
 "synchronized") * ResultType MethodDeclarator ["throws" NameList]
 (Block | " , ")

Statement ::= LabeledStatement
 | StatementExpression " , "
 | Block
 | EmptyStatement
 | SwitchStatement
 | IfStatement
 | WhileStatement
 | DoStatement
 | ForStatement
 | BreakStatement
 | ContinueStatement
 | ReturnStatement
 | ThrowStatement
 | SynchronizedStatement
 | TryStatement

VariableDeclarator ::= VariableDeclaratorId ["=" VariableInitializer]

Block ::= "{ " (BlockStatement) * "}"

ExplicitConstructorInvocation ::= "this" Arguments " , "
 | "super" Arguments " , "

BlockStatement ::= LocalVariableDeclaration " , "

```

        | Statement

LabeledStatement ::= <IDENTIFIER> ":" Statement

EmptyStatement ::= ";"

StatementExpression ::= PreIncrementExpression
                    | PreDecrementExpression
                    | Assignment
                    | PostfixExpression

SwitchStatement ::= "switch" "(" Expression ")" "{" ( SwitchLabel ( BlockStatement ) * ) * "}"

SwitchLabel ::= "case" Expression ":"
            | "default" ":"

IfStatement ::= "if" "(" Expression ")" Statement [ "else" Statement ]

WhileStatement ::= "while" "(" Expression ")" Statement

DoStatement ::= "do" Statement "while" "(" Expression ")" ";"

ForStatement ::= "for" "(" [ ForInit ] ";" [ Expression ] ";" [ ForUpdate ] ")" Statement

ForInit ::= LocalVariableDeclaration
        | StatementExpressionList

BreakStatement ::= "break" [ <IDENTIFIER> ] ";"

ContinueStatement ::= "continue" [ <IDENTIFIER> ] ";"

ReturnStatement ::= "return" [ Expression ] ";"

ThrowStatement ::= "throw" Expression ";"

SynchronizedStatement ::= "synchronized" "(" Expression ")" Block

TryStatement ::= "try" Block ( "catch" "(" FormalParameter ")" Block ) * [ "finally" Block ]

Expression ::= Assignment
            | ConditionalExpression

VariableInitializer ::= Expression

LocalVariableDeclaration ::= Type VariableDeclarator ( "," VariableDeclarator ) *

Assignment ::= PrimaryExpression AssignmentOperator Expression

AssignmentOperator ::= "=" | "*" | "/" | "%" | "+" | "-" | "<=" | ">=" | ">>=" | "&=" | "^=" | |=

PostfixExpression ::= PrimaryExpression [ "++" | "--" ]

```

StatementExpressionList ::= StatementExpression (“,” StatementExpression)*

ForUpdate ::= StatementExpressionList

NLD_Annotation ::= “nld” [**NLD_Locale**] {“ (**NLD_Declaration** | **NLD_EntityDeclaration**)* “}”

NLD_Locale ::= <STRING_LITERAL>

NLD_Declaration ::= **NLD_ScopedName**
 ([**NLD_Locale**] **NLD_TextAssignment**
 |
 “.” [**NLD_Locale**] (**NLD_Statement** | {“ **NLD_Statement**+ “}”))

NLD_Statement ::= **NLD_PropertyDeclaration** | “%text” **NLD_TextAssignment** “,”

NLD_TextAssignment ::= [**NLD_SelectPred**] “=” **NLD_String**
 [“,” [“ **NLD_UserPred** (“,” **NLD_UserPred**)* “]”]

NLD_SelectPred ::= “[“ **NLD_Predicate** (“,” **NLD_Predicate**)* “]”

NLD_Predicate ::= **NLD_PredefinedPred**
 | **NLD_ParamNumber** “[“ **NLD_UserPred** (“,” **NLD_UserPred**)* “]”

NLD_PredefinedPred ::= “@” | “!” | “!locale” (“ **NLD_Locale** “)”

NLD_UserPred ::= <IDENTIFIER>

NLD_ParamNumber ::= “\$”<INTEGER_LITERAL>

NLD_ScopedName ::= “.”
 | **NLD_MethodName**
 | [**NLD_Scope** “.”] **NLD_Identifier**

NLD_MethodName ::= Name **NLD_Signature**

NLD_Signature ::= “((“*” | Type (“,” Type)*) “)”

NLD_Scope ::= “*”
 | “.”
 | Name [“.”]
 | **NLD_MethodName**

NLD_EntityDeclaration ::= “&” <IDENTIFIER> “=” **NLD_EntityText**

NLD_PropertyDeclaration ::= **NLD_PropertyName** “=” **NLD_EntityText**

NLD_PropertyName ::= “%description” | “%includes” | “%purpose” | “%seeAlso”

NLD_EntityText ::= <STRING_LITERAL> (“+” <STRING_LITERAL>)*

NLD_String ::= **NLD_StringElem** ("+" **NLD_StringElem**)
NLD_StringElem ::= <STRING_LITERAL> | **NLD_ParamNumber**

end