ADL 2.0 for C++ Language Reference Manual, Version 1.2

ADL for C++

SunTest

The Open Group Research Institute

The language definition for ADL annotations for the C++ programming language.

ISSUE NUMBER	REASON FOR ISSUE
1.0 Alpha	Document Launch For Review
1.0 Beta	First Revision
1.0 Gamma	Second Revision First snapshot from The Open Group research Institute
1.0 Delta	Third Revision Second snapshot from The Open Group Reserach Institute
1.0	Final revision
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

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.

Trademarks

Change Log

Release 1.2

Complete revision of NLD concepts.

Release 1.1

2- Semantic Annotations

2.1 Removed Non-terminal Constituent.

3. Test Annotations

3.2 Test Function Definition allowed in TDD class body.

3.2 ParameterDeclaration used in TDD factory definition relinquish clause.

5. Complete Grammar

- Changed definition of TranslationUnit to allow parsing exclusive nld file.

- Added Non-terminal IncludeFileList gathering all the IncludeFileDeclarations.

- Changes in definitions of TDD_FieldDeclaration, TDD_FactoryDefinition, TDD_UseDeclaration and AssignmentExpression.

Release 1.0

2. Semantics Annotations

2.1- Use keyword “adlclass” in adl class declaration.

2.3.2 Constructors can also be annotated. Stated some restrictions in constructor specification.

2.3.3 Better explanation of adl inheritance and use of inherited semantics. Added a schema for this.

2.3.5 In adl try statement, ExceptionDeclaration is used.

2.3.5 Explained the use of adl_thrownException and modified examples accordingly.

2.5 Global prologues and epilogues not included in constructor specification.

3. Test Annotations

3.1.2 “tddclass” used for TDD annotated declaration.

3.1.2 Removed external declaration from TDD class body. Replaced with Declaration to specify TDD variables. Specified constraints on these variables.

3.1.2 Test directives are all long. Removed factory declaration.

3.1.2 Added TDD_ADLnewExpression and TDD_ADLExpression

3.1.2 Changed TDD dataset grammar.

3.1.3 Examples changed to conform new test style. Explained use of datasets and factories.

5. Complete Grammar

- Added TypeNameList to be used by ADL_ThrownException.

- Added TDD_ADLnewExpression and ADL_TDDExpression.

- Use of ADL_NamedParamList in ADL_Binding to allow multiple variables declaration.

- ADL_BasicExpression (except “return”) are moved to UnaryExpression instead of PrimaryExpression.

- “return” derive directly from PrimaryExpression to allow it to be post fixed.

- AssignmentExpression extended with ADL_Expression .

- Removed ADL_ArgumentList. Use of ArgumentList instead in “unchanged” expression.

- Removed TDD_DatasetLiteral from PrimaryExpression.

- Removed TDD Long and Short Test directive. Unique definition style for TDDTestDirective

- TDD_FactoryCall used a QualiedId instead of <ID> => allow scope override.

- removed TDD_FactoryDeclaration. use TDD_factoryDefinition instead.

- Changed TDD dataset expressions.

- Types can not be datasets.

Release 1.0 Delta

1. Introduction

No substantive change.

2. Semantics Annotations

2.1: ADL_AnnotatedDeclaration renamed ADL_ClassDeclaration

2.3.1: Added ADL_CallStateExpression.

2.5 : Corrected : Local epilogues are executed before global epilogues. Add restriction about the use of call state operator in Prologues and Epilogues.

3. Test Annotations

.3.1.3 TBD resolved : factories are function definitions.

3.1.7: Importation of tdd files supported with “use” declarations.

3.2.1: Use datasets instead of enum in examples 3.7 and 3.8

3.3: Test directives have two variants : Long test directives and short test directives.

Example 3.5: the semantics of elements of literal datasets has changed: they are evaluated only once (static evaluation). Dynamic behavior is now possible only through factories.

4. NLD Annotations

No substantive change.

5. Complete Grammar

ADL and TDD annotations reviewed.

Release 1.0 Gamma

1. Introduction

No substantive change.

2. Semantics Annotations

2.1: TBD resolved :ADLT allow both external and inline specifications.

2.1: An adl class for each adl file.

2.2.1: Assertion Groups before ADL Specific Expressions

2.2.4: Definition of ADL If statement semantically different from the C++ if statement.

2.3: Semantics are pure adl assertions with no side-effects.

2.3.2: TBD resolved : Adl Inheritance supported.

2.3.4: Catch specification comes next to an ADL_TryStatement

2.4: Use “inline” to declare inline functions behaving like macros.

2.5: Introduction of local and global adl Prologues and Epilogues.

3. Test Annotations

Not modified yet.

TBD: see how to support tdd classes.

4. NLD Annotations

Not modified yet.

5. Complete Grammar

Modified grammar to respect new non-terminal naming model and introduce new concepts.
TBD : Modify the TDD and NLD productions.

5.2: ADL “and”, “or “and “xor” removed. Use C++ operators instead.

Release 1.0 Beta

1. Introduction

No substantive change.

2. Semantics Annotations

2.1: TBD resolved: ADLT will allow only external specifications..

2.3.3: TBD changed: SuperClass semantics reference not implicit; explicit invocation remains a possibility.

3. Test Annotations

Title changed from “TDD Annotations”.

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

3.1.3: Add 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.

3.1.6: Assert moved from language definition to 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 3.12: TBD resolved: multiple reference interpretation changed.

3.3.1: External dataset reference clarified.

4. NLD Annotations

4.1: TBD resolved: inheritance clarified.

Example 4.5: 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

This document describes the enhancements to the ADL Language for the C++ programming language. The ADL Langauge has been revised as part of the ADL 2.0 Project. The purpose of the ADL 2.0 is to extend the technology of the ADL 1.0 project to object-oriented programming languages. Specifically, we intend to target C++, CORBA IDL, and Java, while retaining the capability of specifying ANSI C programs. This extension to object-oriented languages will require a substantial re-implementation. We will take advantage of this opportunity to reduce some of the barriers to adoption of ADL technology. In particular, we will simplify the input syntax of the ADL compiler, and improve its portability by simplifying its internal structure. A migration path for users of ADLT 1 is of utmost importance in this re-implementation.

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 C++ 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.

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 | ABBBC |`** *etc*." and "**`A ( B )+ C`**" is the same as "**`ABC | ABBC | ABBBC |`** *etc*.".
- Non-terminals from the C++ language definition are represented in a sans-serif font (like literal), and the non-terminals that define the ADL augmentation of C++ 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 Semantics Annotations

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

2.1 Describing Semantics Of Interface Operations

ADL provides syntactic constructs to describe semantic behavior of C++ functions. To do this, it provides an extended declaration syntax — the *annotated function declaration* — as shown in the syntax below:

TranslationUnit := IncludeFileList [**ADL_ClassDeclaration]**

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

The above syntax indicates that the non-terminal TranslationUnit is modified to allow the additional construct, **ADL_ClassDeclaration**. ADL introduces the **ADL_InlineDeclaration**, as shown above, to facilitate the writing of *inline function declarations*.

ADL_BehaviorDeclaration := DeclarationSpecifiers FunctionDeclarator “{“**ADL_BehaviorSpecification** “}”

These rules are not complete: they will be refined 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 illustrating the use of these constructs is shown here. Suppose there is a class declared (in file StockBrocker.hh) as:

EXAMPLE 2.1 StockBroker.hh

```
class StockBroker {
  long Cash_Balance(long account);

  long Stock_Balance(long account, char* symbol);

  void Buy(long account, char* symbol, long no_of_shares);
};
```

Then we may describe its behavior with the annotation:

EXAMPLE 2.2 StockBrokerSpec.adl

```
#include “StockBroker.hh”
```

```
# include “StockBrokerAux.hh”

adlclass StockBroker {

inline long cost(char* symbol, long no_of_shares) {
  no_of_shares * price(symbol);
}

void Buy(long account, char* 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;
}

}

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

}
```

In this example, a class with three operations Cash_Balance, Stock_Balance, and Buy is augmented with a description of the behavior of Buy, written in the external function annotation syntax. The two boolean expressions appearing within “semantics {...}” describe legitimate behavior of the Buy Function. In these expressions, “@” is a 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 Function — by default all expressions are evaluated after the execution of the Function.

The first of these boolean expressions make use of the notion of “**`cost`**” of a stock purchase. This is implemented in the interface as an inline function declaration. The inline function declaration in turn requires the notion of the “**`price`**” of a particular share, and this is implemented as a static method “price” from an additional class, StockBrokeAux, which is defined only for purposes of testing and included into the adl file as an external declaration. The main difference between inline and auxiliary function declarations is that the body of inline function declarations is an ADL expression (described fully below) rather than a C++ block statement. The tested functions “**`Buy`**” could be referred to using its fully qualified name “StockBroker::Buy”.

The example above shows that the specification of a class member function is written outside the definition of the class, much like an external implementation.

Behavior descriptions in annotated function declarations may refer to any of the built-in C++ types, and to types declared in the interface of the specified class or in any additional declarations used for testing.

Lastly, it is not possible to define more than one *adl class* in an ADL translation unit, and the name of the file must be the name of the adl class extended with the suffix adl (StockBroker.adl in the previous example).

2.2 ADL Syntax

ADL provides an expression syntax which is an extension of that of C++. The extensions are of two kinds: a few additional primary expressions and operators, and some ADL-specific expression constructions. The ADL extensions will be presented here, without discussion of the standard C++ expressions.

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 is an expression since:

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 (ADL) 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 evaluate to `false`.

UnaryExpression +::= ADL_BasicExpression

ADL_BasicExpression +::= ”return”

Expressions are extended in ADL with the reserved word **`return`**, which is used to refer to the return value of a function. The primary expression **`return`** may be used only in behavior specifications (Section 2.3) of operations with non-void return types and may not appear within an operand 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 ::= **NamedParam** “:” ConditionalExpression

ADL offers a constrained form of quantified expression using which one may 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)
```

With `ADL_long_range(i,j)` which is a function that returns the sequence of `long`’s starting from `i` and ending at `j`. The same applies for `ADL_short_range` and `ADL_int_range` who return sequences of short’s and int’s resp.

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`.

Free variables may not be used within the scope of a call-state or “unchanged” operator (unless the whole quantified assertion itself is inside this scope).

2.2.4 ADL If Statement

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

An “`if` expression” provides a way to conditionally evaluate expressions. Its meaning is quite similar to the “`?:`” operator. The types of both the group expressions of the `if` expression must be the same and this is the type of the `if` expression. If the type of the `if` expression is boolean, then the else part may be omitted and is assumed to be “`else true`”. The conditions (the expressions within parenthesis) must be `bool` 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 function in its simplest form is the function declaration followed by the reserved word “semantics” followed by a list of boolean expressions within braces.

These expressions can refer to the visible state of the system both before and after the execution of the function. The details of the syntax of expressions is presented in Section 2.2.

ADL_BehaviorDeclaration ::= (DeclarationSpecifiers)? FunctionDeclarator
“{“**ADL_BehaviorSpecification** “}”

| DtorCtorDeclSpec CtorDeclarator (ExceptionSpec)?
"{"A**DL_BehaviorSpecification** "}"

Both functions members (with or without a return type) and constructors can be tested.

ADL_BehaviorSpecification ::= "semantics" **ADL_AssertionGroup**

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

The behavior description of `Buy` from Example 2.2 is reproduced below:

```
void Buy(long account,char* 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);
  assertion_2 = (Stock_Balance(account, symbol) == tmp3 +
no_of_shares);
```

Step 4: Determination of consistent behavior:

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

Behavior descriptions can refer to inline function declarations and other function 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 has the effect of evaluating its argument before the call to the specified function.

UnaryExpression +::= **ADL_CallStateExpression**

ADL_CallStateExpression +::="@" UnaryExpression

ADL_BasicExpression +::="unchanged" ArgumentList

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 function, while the second is the value returned by `f` when called after the call to the specified function, but passed parameters whose values are saved from the state before the call to the specified function.

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

Constructor semantics are specified in the following way:

ADL_BehaviorDeclaration +::= DtorCtorDeclSpec CtorDeclarator (ExceptionSpec)? "{" **[ADL_Prologue] ADL_BehaviorSpecification [ADL_Epilogue] (NLD_Annotation)** * "}"

Some constraints are added while specifying a constructor :

EXAMPLE 2.3 StockBroker constructor specification

```
#include "StockBroker.hh"

# include "StockBrokerAux.hh"

adlclass 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 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 of the tested method (the constructor), because it is this 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 +::= “adlclass” <IDENTIFIER> **ADL_SuperClass**
“{“ (**ADL_BehaviorDeclaration**)* “}”

ADL_SuperClass := “:” [“public”] <ID> (“,” [“public”] <ID>) *

Note that the multiple inheritance of ADL classes is supported. Scope override can not figure in the name of the super class since inheriting from nested classes is not supported. 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.hh */

class Bank {
  void openAccount();
  void closeAccount();
}

/* MyBank.hh */

class MyBank : Bank {
  void openAccount();  // overrides Bank.openAccount
  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 MyBankSpec specification

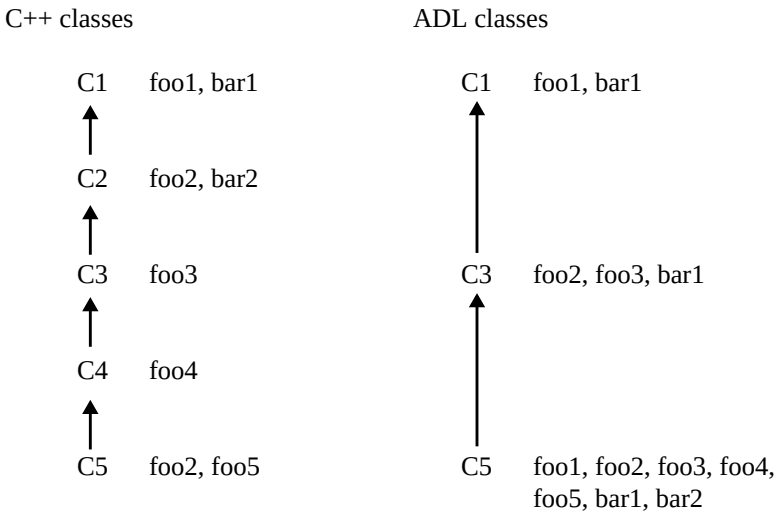
```
# include “Bank.hh”

adlclass MyBank : Bank {

  void openAccount() {
    semantics {
      Bank::semantics; // assertions specific to the method
                       // Bank.openAccount
      <assertion>;  // assertion specific to the method
                       // MyBank.openAccount
    }
  }

}
```

The “:” inheritance clause is quite different from its usual meaning in C++: 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 link the *object* file obtained by transformation of Bank.adl will be accessible, along with the class file generated by transformation of MyBank.adl. Access control keywords is optional and can only be “public” (in comparison of the “public”, “private” or “protected” inheritance in C++).

The assertion “`Bank::semantics;`” is an explicit invocation of the semantics of the superclass Bank 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. The superclass (qualified) name is required to support multiple inheritance. The only side-effects are the generation of test reports for the assertions of this behavior description.

The generated C++ ACO classes for MyBank will in fact inherit from the generated C++ ACO class for Bank. See the ADL 2.0 Translation System design document for more details.

To summarize:



This picture represents a C++ hierarchy (C5 inherits from C4 that inherits from C3...) and a corresponding adl classes hierarchy. The *foo*s and *bar*s are methods defined in C++ classes, some of them being annotated by adl classes. We can notice that:

- the adl classes graph is a subgraph of the C++ classes graph (it is not compulsory to annotate all C++ classes).
- an adl class can annotate a method that is defined in the corresponding C++ class (e.g. foo5 in C5) or overridden (e.g. foo2 in C5) or just inherited (e.g. foo4 in C5).
- the <SuperClassName>::semantics feature can be used only in methods that have already been annotated in an inherited adl class. Specifying the name of the super class prior to the keyword “semantics”, makes the support of multiple inheritance possible in adl. Method under test should have been annotated in the specified super class. For instance in adl class C5, it would be an error to use this feature for methods foo4, foo5 and bar2, and correct for the other methods. Note that this inherited class is not necessarily the direct superclass: for instance in adl class C5, C1::semantics in foo1 is valid.
  Last, this process is recursive: if bar1 in C5 calls C3::semantics and bar1 in C3 calls C1::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” **NamedParamList** “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;
}
```

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

A single binding declaration may introduce multiple variables and initialize them. For example, suppose we had the following function:

```
long foobar(long x, long& y, long& z);
```

The following binding declaration evaluates `foobar` once with the number 5 as its input parameter and "captures" all the values it returns:

```
define long retval, long y, long z with
   retval = foobar(5, y, z);
```

It is also possible for a binding to *rebind* variables introduced in earlier (and possibly more global) bindings.

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" "(" ExceptionDeclaration ")" **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.

A catch specification may name the particular exception it catches and bind it to a local identifier, in which case, this identifier may be used in the following assertion group to

select values returned by the exception. A catch specification may also use the "..." syntax to catch any exception that may be thrown.

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

EXAMPLE 2.6 StockBroker2.hh

```
class BadCall {
  bool bad_account;
  bool bad_stock_symbol;
};

class StockBroker {

  long Cash_Balance(long account) throw (BadCall);

  long Stock_Balance(long account, char* symbol)
  throw (BadCall);

  void Buy(long account, char* symbol, long no_of_shares)
throw (BadCall);

};
```

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

EXAMPLE 2.7 StockBroker2.adl

```
#include "StockBroker.hh"
#include "StockBrokerAux.hh"

adlclass StockBroker2{


inline long cost(char* symbol, long nsh) throw (BadCall) {
  nsh * price(symbol);
}

void Buy(long account, char* symbol, long no_of_shares)
throw (BadCall){
  semantics {
   try {

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

 } // end of semantics

} // end of Buy

}
```

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 comparison use the ADL variable adl_thrownException, 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. 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 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 function 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 function, 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.adl (corrected)

```
#include “StockBroker2.hh”
#include “StockBrokerAux.hh”
adlclass StockBroker2{
inline long cost(char* symbol, long no_of_shares) throw
(BadCall) {
  no_of_shares * price(symbol);
};

void Buy(long account, char* symbol, long no_of_shares)
throw (BadCall) {
semantics {
  try {

   Cash_Balance(account) ==
      @{
         try { Cash_Balance(account) - cost(symbol,
no_of_shares);}
         catch(...) { 0; };
      };
  Stock_Balance(account, symbol) ==
      @{
         try { Stock_Balance(account, symbol);}
         catch(...) { 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 (...) {
      false;
  }

  } // end of semantics

 } // end of Buy specifications

}
```

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 function, hence the 0’s passed from the call state are never really used.

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 function declarations.

2.3.6 Thrown Expressions

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

ADL_BasicExpression +::= **ADL_ThrownExpression**

ADL_ThrownExpression ::= “thrown” “(” (TypeNameList | “...”) “)”

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

“thrown(e)” is equivalent to saying that “adl_thrownException is e”.

Thrown expressions may not be placed within the argument of a call state operator.

2.3.7 Behavior Classification

It is often very useful to broadly categorize the behavior of a function 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 := “[” ((“normal”|”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(...)` and `abnormal` defaults to `thrown(...)`.
- 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 function 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 StockBroker2.adl with behavior classification

```
#include “StockBroker2.hh”
#include “stockBrokerAux.hh”


adlclass StockBroker2{


inline long cost(char* s, long nsh)
{
  try{ nsh * price(s);}
  catch(...) { 0; };
};

inline long _Cash_Balance(long account)
{
  try { Cash_Balance(account);}
  catch(...) { 0; };
};

inline long _Stock_Balance(long account, char* symbol)
{
  try{ Stock_Balance(account, symbol);}
  catch(...) { 0; };
};

void Buy(long account, char* symbol, long no_of_shares)
     throw (BadCall) {
semantics
[ normal = !thrown(...);
  abnormal = thrown(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 (...) { false; };
  }; // end if (normal)
 } // end semantics

} // end Buy

}
```

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 function 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 caused `BadCall` to be thrown.
- The main part of the behavior description of `Buy` is guarded by the "`if`" expression (Section 2.2) "`if (normal)`...". In this case, no exception is expected to be thrown during the evaluate of the assertions, and hence a catch specification is included 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 <:> whose meaning is based on behavior classifications. In the usual usage of this operator, the left operand is the enabler of an exception, while the right operand is a "thrown" expression. The following example illustrates this typical use:

```
bad_account(account) || bad_symbol(symbol) <:>
thrown(BadCall)
```

Informally, A<:>B means that if A is true, then the 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 statuses 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 <:> B` is the same as

```
((A ==> abnormal) and (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 function. It reads: If the file `f` does not exist, then an excep-

tion 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 the exception `not_found` should only be thrown when the file `f` does not exist, and exception `disk_error` should only thrown when 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 members of a class SystemException may be thrown:

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

Now the earlier stock broker example is completed with specification of abnormal behavior. Two auxiliary function declarations — `bad_acct` and `bad_sym` — are added into the StockBroker.hh file:

```
class StockAuxiliary {
public:

bool bad_acct(StockBroker sb, long account);
bool bad_sym(char * symbol);interface StockAuxiliary {
long price(char* s, long nsh);
}
```

EXAMPLE 2.10 StockBroker2.adl with exceptions

```
#include “StockBroker.hh”
#include “StockBrokerAux.hh”


adlclass StockBroker2 {

inline long cost(char* symbol, long no_of_shares) {
  try { no_of_shares * price(symbol);}
  catch(...) { 0; };
};

inline long _Cash_Balance(long account) {
  try { Cash_Balance(account);}
  catch(...) { 0; };
};

inline long _Stock_Balance(long account, char* symbol) {
  try { Stock_Balance(account, symbol);}
  catch(...) { 0; };
};
```

```
void Buy(long account, char* symbol, long no_of_shares)
throw (BadCall) {
semantics
[ normal = !thrown(...);
  abnormal = thrown(SystemException, BadCall); ]
{
  bad_acct(account) <:>
     (thrown(BadCall) && ((BadCall*)adl_thrownException)->
bad_account);
  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 (...) { 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 of the exception conditions to hold.
- It uses an additional auxiliary class to give additional information about accounts and symbols. It is often the case that a class 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.

2.4 Inline Procedure Declarations

Inline macro declarations is other way to define concepts used in behavior descriptions (along with Auxiliary C++ declarations). Their syntax is:

ADL_InlineDeclaration ::= "inline" DeclarationSpecifiers FunctionDeclarator "{"
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 adhoc 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(...) { 0; };
};
```

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

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

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

2.5 Prologues and Epilogues

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

For this purpose, the user can use the "**prolog**" and "**epilog**" features, which provide blocks of "pure" C++ that will be transmitted without any transformation to the generated code. It is up to the C++ compiler to check the correctness of these statements.

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

ADL_ClassDeclaration +::= "adlclass" <IDENTIFIER> [: <ID>]
"{" [**ADL_Prologue**] [**ADL_Epilogue**]
(**ADL_InlineDeclaration** | **ADL_BehaviorDeclaration**)* "}"

ADL_BehaviorDeclaration ::= FunctionDeclaration "{" [**ADL_Prologue**] **ADL_BehaviorSpecification** [**ADL_Epilogue**]"}"

ADL_Prologue ::= "prolog" CompoundStatement

ADL_Epilogue ::= "epilog" CompoundStatement

```
adlclass bankAccount {
```

```
  prolog {
    char* url = “jdbc:odbc:wombat”;
    DbConnection dbcon= DbDriverManager::getConnection(url);
    DbTable dbtbl = dbcon.createTable();
  }

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

}
```

In the generated C++ 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 semantics assertions. The epilog code is copied at the end of this method (a global epilog would be copied right before 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: Call of the tested method
Step 4: Evaluation and saving of call-state expressions
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 behavior specification of a constructor, but the global prologue/epilogue are not included 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.

3.1.1 Re-write

The principle behind TDD2 is that it is processed by re-writing the input to create a test program. The re-write does not remove any information, and a valid program in the target language should not be altered by the re-write. Hence any code fragment in the input which does not use TDD2 features will appear unaltered in the rewritten output.

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:

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

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. Examples of infeasible datasets are programming language types like ‘int’ and ‘float’. 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...) of more than one argument is formally treated as function f_1 of a single argument, AxBxC... – 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:

$$\begin{gathered} F \cong \{D, R, c, d, i\} \\ c \text{ ?Functional? } D \rightarrow R \cup \{\perp\} \\ d \text{ ?Functional? } R \rightarrow \{\varnothing, \perp\} \\ i \text{ ?Functional? } F \rightarrow D \end{gathered} ,$$

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 be 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 semantics 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.

3.1.5 Test Directives

A test directive is ormally 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 Bool 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 Bool expression evaluated for side effect.

An assertion is expressed by a call to the function `tdd_assert(char*, bool)` 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 C++, when the user declares “use bar;”, he can thereafter use for instance the dataset “D1” defined in the file bar.tdd. With “use”, the compiler checks the presence and correctness of the source tdd file. It is however left to the responsibility of the user to ensure that at link-time the object file obtained by transformation of the bar.tdd will be accessible. 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 / C++ Syntax

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

TDD_AnnotatedDeclaration ::= [**TDD_UseDeclaration**] **TDD_ClassDeclaration**

TDD_ClassDeclaration ::= “tddclass” <IDENTIFIER>
“{“ (**TDD_ClassBodyDeclaration**)* “}”

A TDD class declaration is like any C++ class declaration, with the use of the “tddclass” keyword and with the methods access modifiers “public” and “private” removed. It may contain any C++ declaration, plus some TDD constructs.

TDD classes have no inheritance structure, as the concept of “superclass” is not meaningful for a TDD class. TDD class may not extend another, although one TDD class may refer to another. One TDD class may refer to entities (both C++ declarations and TDD declarations) declared in another, either by using the fully scoped name of the external entity or the “use” TDD file feature.

TDD_UseDeclaration ::= (“use” <ID> “;”)+

<ID> refers to the tdd file (without its extension .tdd) to be used.

TDD_ClassBodyDeclaration ::= **TDD_DatasetDeclaration**
| **TDD_FactoryDeclaration**
| **TDD_TestDirective**
| **TDD_FieldDeclaration**
| FunctionDefinition

TDD_FieldDeclaration ::= DeclarationSpecifiers Declarator “=” AssignmentExpression
(“,” Declarator “=” AssignmentExpression)* “;”

In TDD class body, only initialized variable declarations and test fucntion declarations are allowed. Other pure C++ declarations (ExternalDeclaration) can be included using the “# include” feature.

The extra constructs that may occur in a TDD declaration are datasets, factories, test functions and test directives.

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

ADL_NamedParam ::= DeclarationSpecifiers Declarator

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

TDD_FactoryDefinition ::="factory" DeclarationSpecifiers FunctionDeclarator CompoundStatement
["relinquish" "(" ParameterDeclaration ")" CompoundStatement]

A factory is defined just like a function with a C++ compound statement. It must return a pointer to a value (a builtin type or an object). The relinquish clause receives the pointer to the returned value as parameter. It provides a way to free the memory allocated for the returned value. Obviously, the type returned by the factory and the type of the parameter of the relinquish clause must be identical.

TDD_TestDirective ::= [<ID> ":"] "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 transltor can supply a test method declaration specialized for the test framework for which code is being generated.

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

A dataset expression may be used alone in a domain list only if it is a void dataset, which must be produced by a factory; in that case, it denotes the evaluation of the corresponding void factory member for side effect.

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

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

TDD_DatasetSingleton ::= Constant | <ID>

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

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

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

TDD_ADLnewExpression ::= “ADL_new” IdExpression

TDD_ADLExpression ::= “ADL” “(“ PrimaryExpression() “)”

A dataset literal is written in braces. It may be empty. The elements in a literal dataset may be expressions or ranges. 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_DatasetExpression:

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

this definition is correct if f1 is a factory and f2 a method.

3.3 General Syntax & Examples

This section presents the general syntax 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

```
#include “subject.hh”
tddclass t1{
  ADL(subject)->plus(3,4);
}
```

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 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 importation features: `#include` in C++ and `import in Java).`

EXAMPLE 3.2 A Simple Dataset

```
#include “subject.hh”
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 1, from both sides.

3.3.2 Compound Datasets : Factories, Concatenation

EXAMPLE 3.3 Compound Data Construction

```
#include “myio.h”
tddclass{
  factory RandomAccessFile*
           make_file(char* nm, char* mode){
    return new RandomAccessFile(nm, mode);
  } 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”});

  char* buf = new byte[512];

  test (RandomAccessFile* F = F0)
    ADL(F)->read(buf,512);

  test (RandomAccessFile* F=F1) {
    ADL(F)->read(buf,512);
  }
}
```

Dataset F0 has 3x2=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 C++ 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

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, complete with an assert.

EXAMPLE 3.4 Void Datasets

```
use t3;
tddclass{
 factory void setup_system(int condition_code){//..

 } relinquish{...}


 dataset void setup_set = setup_system(1);

 test (setup_set,
       RandomAccessFile* F = F1,// F1 accessible  thanks
                               // to “use t3;”
       char* data={“”, “hello”})
 {
    char* tmp;
    ADL(F)->write(data);
    F->seek(0);
    tmp = ADL(F)->read();
    tdd_assert(“streq(tmp,data)”, streq(tmp,data));
 }
}
```

This example shows the use of an unchecked method (`seek`) in conjunction with some checked methods (`write` and `read`). 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 semantics annotation.

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.

Imported dataset names (through the “`use`” clause) can be unqualified only if defined in the current tdd class. 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” 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).

These rules are also valid for factory importation. Only datasets and factories are importable: the constants 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 =
     { drand48(), drand48(), drand48() };
  factory double rand() { return drand48(); }

  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 directives that reference q_static. But for test directives that use q_dynamic, 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/C++ idiom for a provide test variable. In TDD, any global variable (field) is implicitly constant (`const` in C++) and must be initialized at its declaration. A TDD constant is private: it cannot be imported through the “use” clause.

```
#include “Bank.hh”;

tddclass t6 {

  int SAVINGS = -1, CHECKING = 1, IRA = 7;
  int negative = -10, zero = 0, small = 3, average = 100,
     large = 1000, over_limit = 10000;
  dataset int account_type = {SAVINGS, CHECKING, IRA};
  dataset int size_code =
     {negative, zero, small, average, large, over_limit};
```

```
  factory account* acct(int t, int s) { /* ... */ }

  dataset account* Account1 = acct(account_type,size_code);

  factory int amount(int size_code){/* ...*/}
  Bank* bank = new Bank(/*..*/);

  test (account* act = Account1, int size = size_code ) {
     ADL(bank)->withdraw(act, amount(size_code));
     ADL(bank)->deposit(act, amount(size_code));
     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.

```
#include "Bank.hh"

 tddclass t7 {
  dataset int size_code =
     {negative, zero, small, average, large, over_limit};
  dataset int account_type =
     {checking, savings, IRA, zero, neg, max,over_max};

  factory double amount(int size){/*...*/}
  factory account* make_acct(
           int type_code,
          double size) {/*....*/}
  dataset account* Acct = make_acct(
          acount_type, amount(size_code));

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

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

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

The syntax is:

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

Example:

#include “object_data.h”;

```
tddclass t10{
  Dir1 : test (Object* o = data.obj) {
     ADL(o)->hashCode();
  }
```

A test directive body has the same syntax as statement in the C++ 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.

3.3.7 Void Datasets Use

EXAMPLE 3.8 Void Dataset Use

The syntax for having void datasets is as follows :

```
tddclass t9 {
  dataset int A = { 1,2,3 };

  factory void side_effect(int){/*...*/}
  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.

3.3.8 Advanced Examples

EXAMPLE 3.9 Chaining Factories

```
#include “code.hh”
#include “myio.hh”

tddclass t10 {

  factory char* make_file_name(
     bool absolute,
     bool device,
     bool funny_chars,
     int length_code
  ) { /*...*/ }

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

  dataset char* file_name_set =
     make_file_name(true, false, false, 10);

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

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

  dataset char* legal_open_type = {“r”, “rw”};

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

  dataset char* open_type =
     legal_open_type + illegal_open_type();

  dataset File* File_set =
      make_file(file_name_set);

  dataset RandomAcessFile* Stream_set =
     make_filestream(File_set, open_type);
}
```

This illustrates several techniques for re-using factories.

EXAMPLE 3.10 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.

```
#include “myio.hh”

use t10;

public tddclass t12 {

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

     pos = f->getFilePointer();
     ADL(f)->read(buf);
     ADL(f)->seek(pos);
     ADL(f)->write(buf);
  }

  test (File* f = File_set)
     read_then_write(f, 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 3.11 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.

```
#include “Math.hh”

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.

C++ can be annotated with NL information in several places. Briefly, it can be placed at top level, within a TDD annotation, attached to an annotated function or test statement, or placed after the bindings in an ADL semantics group expression. The translations it provides apply throughout the scope (and enclosed scopes), not just from the declaration point onwards. The examples in this section illustrate some of the annotation attachment locations.

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 C++, this means that NL declarations are inherited from all base classes, and may be overridden in a subclass or implementation class. An error occurs if a conflicting definition arises from this inheritance.

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.

```
/* C++ code */
class C {
  public static int amount;
}

/* ADL source */
adlclass C {
  nld {
     .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 A 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 {
  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 {
public:
  void deposit(int amount);
  void deposit(int amount, int charge);
  void close_account();
};

nld {
  C::deposit(int amount) = “deposit some money”;
  C::deposit(int amount, int charge)
     = “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 an other 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), 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 “`.`” refers 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”;
};

class A {
public:
  static int i;
};

class B {
public:
  static int i;

  nld {
```

```
    .i = “B’s i”;
  };
};
```

An expression using `A::i` 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 `B::i` 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;
};

class B {
public:
  void g();
};

adlclass A {
  nld {
    .i = “A’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. acual 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:
  public int a;
  public 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.

```
strcmp(char*, char*)[!] =
     “string” + $1 + “is equal to string” + $2;
```

With this mapping, an assertion ‘`! strcmp(str, “foo”);`’ will be translated as “string str is equal to string “foo”” instead of “the negation of the value returned by the

function strcmp(char*, char*), invoked with parameters: (str ; “foo”)”, the default translation.

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 Properties

```
void f() {
  semantics { /* ... */ }
  nld {
     . : {
        &includes = “#include <stdlib.h>”;
        &description = “Behavioral description”;
        %purpose = “Short description”;
        %seeAlso = “See the class Foo”;
```

```
    }
  }
}
```

This is equivalent to:

```
nld {
  C::f() : &includes = “#include <stdlib.h>”;
  C::f() : &description = “Behavioral description”;
// ...
}
```

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 annotation 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 C++ 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 C++ 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 C++ scoping, 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 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 Scoped Names

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

nld {
  C::i = "translation for i";
  C::f(int) = "translation for f(int)";
  C::f(int)::i = "translation for 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 (refered 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 C++ language.

NLD_String ::= **NLD_StringElem** ("+" **NLD_StringElem**)*

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

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

5 Complete Grammar

Here is the complete grammar for ADL for C++. Non-terminals in boldface are defined in this document; other non-terminals are part of the C++ language definition.

There may be some discrepancy between this grammar and any particular C++ dialect, for two reaasons: C++ is not standardized, and the ADLT system is not intended to support all of the language. Also, the fact that a C++ construct is parsed by this grammar does not mean that construct will be processed by ADLT.

5.1 C++ productions

TranslationUnit ::= IncludeFileList [**ADL_ClassDeclaration | TDD_AnnotatedDeclaration**] (**NLD_Annotation**)* <EOF>

IncludeFileList ::= (IncludeFileDeclaration)*

IncludeFileDeclaration ::= <SHARP_INCLUDE_FILE> (ExternalDeclaration)* <ENDINC>

ExternalDeclaration ::= DtorDefinition
| CtorDefinition
| FunctionDefinition
| ConversionDeclOrDef
| Declaration
| ";"

FunctionDefinition ::= DeclarationSpecifiers FunctionDeclarator (";" | CompoundStatement)
| FunctionDeclarator (";" | CompoundStatement)

LinkageSpecification ::= "extern" <STRING> ("{" (ExternalDeclaration)* "}" [";"] | Declaration)

Declaration ::= DeclarationSpecifiers [InitDeclaratorList] ";"
| LinkageSpecification

TypeModifiers ::= StorageClassSpecifier
| TypeQualifier
| "inline" | "virtual" | "friend"

DeclarationSpecifiers ::= (TypeModifiers)+
[BuiltinTypeSpecifier (BuiltinTypeSpecifier | TypeModifiers)*
| (QualifiedType | ClassSpecifier | EnumSpecifier) (TypeModifiers)*]
| BuiltinTypeSpecifier (BuiltinTypeSpecifier | TypeModifiers)*
| (QualifiedType | ClassSpecifier | EnumSpecifier) (TypeModifiers)*

SimpleTypeSpecifier ::= (BuiltinTypeSpecifier | QualifiedType)

ScopeOverride ::= ("::") (<ID> ["<" TemplateArgumentList ">"] "::")*
| (<ID> ["<" TemplateArgumentList ">"] "::")*

```
QualifiedId ::= FullyScopedId
              | <ID> [ "<" TemplateArgumentList ">" ]
              | "operator" Optor

FullyScopedId ::= ScopeOverride ( <ID> [ "<" TemplateArgumentList ">" ] | "operator" Optor )

PtrToMember ::= ScopeOverride "*"

QualifiedType ::= FullyScopedType
                | <ID> [ "<" TemplateArgumentList ">" ]

FullyScopedType ::= ScopeOverride <ID> [ "<" TemplateArgumentList ">" ]

TypeQualifier ::= "const" | "volatile"

StorageClassSpecifier ::= "auto" | "register" | "static" | "extern" | "typedef"

BuiltinTypeSpecifier ::= "void" | "char" | "short" | "int" | "long" | "float" | "double" | "signed"
                       | "unsigned" | "bool"

InitDeclaratorList ::= InitDeclarator ( "," InitDeclarator )*

InitDeclarator ::= Declarator [ "=" Initializer | "(" ExpressionList ")" ]

ClassHead ::= ( "struct" | "union" | "class" ) [ <ID> [ BaseClause ] ]

ClassSpecifier ::= ( "struct" | "union" | "class" ) (  ClassMemberList | <ID> [ BaseClause ]
                   ClassMemberList | <ID> [ "<" TemplateArgumentList ">" ] )

BaseClause ::= ":" BaseSpecifier ( "," BaseSpecifier )*

BaseSpecifier ::= [ "virtual" [ AccessSpecifier ] | AccessSpecifier ( "virtual" ) ] ]
                   [ ScopeOverride ] <ID> [ "<" TemplateArgumentList ">" ]

AccessSpecifier ::= "public" | "protected" | "private"

ClassMemberList ::= {" ( MemberDeclaration )* "}"

MemberDeclaration ::= Declaration
                    | EnumSpecifier [ MemberDeclaratorList ] ";"
                    | ConversionDeclOrDef
                    | DtorDefinition
                    | DtorCtorDeclSpec SimpleDtorDeclarator ";"
                    | CtorDefinition
                    | ( DtorCtorDeclSpec CtorDeclarator ";" )
                    | FunctionDefinition
                    | DeclarationSpecifiers [ MemberDeclaratorList ] ";"
                    | FunctionDeclarator ";"
                    | QualifiedId ";"
```

```
                    |   AccessSpecifier ":"
                    |   ";"
                    |   NLD_Annotation ";"

MemberDeclaratorList ::= MemberDeclarator [ "=" <OCTALINT> ]
                         ( "," MemberDeclarator [ "=" <OCTALINT> ] )*

MemberDeclarator ::= Declarator

conversionDeclOrDef ::= "operator" DeclarationSpecifiers [ "*" | "&" ] "(" [ ParameterList ] ")"
                        [ TypeQualifier ] [ ExceptionSpec ] ( CompoundStatement | ";" )

EnumSpecifier ::= "enum" ( "{" EnumeratorList "}" | <ID> [ "{" EnumeratorList "}" ] )

EnumeratorList ::= Enumerator ( "," Enumerator )*

Enumerator ::= <ID> [ "=" ConstantExpression ]

PtrOperator ::= "&" CvQualifierSeq
            |   "*" CvQualifierSeq
            |   PtrToMember CvQualifierSeq

CvQualifierSeq ::= [ "const" [ "volatile" ] | "volatile" [ "const" ] ]

Declarator ::= PtrOperator Declarator
           |   DirectDeclarator

DirectDeclarator ::= |"(" Declarator ")" [ DeclaratorSuffixes ]
                 |   QualifiedId [ DeclaratorSuffixes ]

DeclaratorSuffixes ::= ( "[" [ ConstantExpression ] "]" )*
                   |   "(" [ ParameterList ] ")" [ TypeQualifier ] [ ExceptionSpec ]

FunctionDeclarator ::= PtrOperator FunctionDeclarator
                   |   FunctionDirectDeclarator

FunctionDirectDeclarator ::= QualifiedId "(" [ ParameterList ] ")" [ TypeQualifier ] [ ExceptionSpec]
                             [ "=" <OCTALINT> ]

DtorCtorDeclSpec ::= [ "virtual" [ "inline" ] | "inline" [ "virtual" ] ]

DtorDefinition ::= [ TemplateHead ] DtorCtorDeclSpec DtorDeclarator CompoundStatement

CtorDefinition ::= DtorCtorDeclSpec CtorDeclarator [ ExceptionSpec]
                   ( ";" | [ CtorInitializer ] CompoundStatement )

CtorDeclarator ::= QualifiedId "(" [ ParameterList ] ")"

CtorInitializer ::= ":" SuperclassInit ( "," SuperclassInit )*

SuperclassInit ::= QualifiedId "(" [ Expression ] ")"
```

```
DtorDeclarator ::= [ ScopeOverride ] SimpleDtorDeclarator

SimpleDtorDeclarator ::= "~" <ID> "(" [ ParameterList ] ")"

ParameterList ::= ParameterDeclarationList [ [ "," ] "..." ] | "..."

ParameterDeclarationList ::= ParameterDeclaration ( "," ParameterDeclaration )*

ParameterDeclaration ::= DeclarationSpecifiers ( Declarator | AbstractDeclarator )
                         [ "=" AssignmentExpression ]

Initializer ::= "{" Initializer ( "," Initializer )* "}"
              | AssignmentExpression

TypeName ::= DeclarationSpecifiers AbstractDeclarator

TypeNameList ::= TypeName ( , TypeName )

AbstractDeclarator ::= [ "(" AbstractDeclarator ")" ( AbstractDeclaratorSuffix )*
                       | ( "[" [ ConstantExpression ] "]" )* | PtrOperator AbstractDeclarator ]

AbstractDeclaratorSuffix ::= "[" [ ConstantExpression ] "]"
                           | "(" [ ParameterList ] ")"

TemplateHead ::= "template" "<" TemplateParameterList ">"

TemplateParameterList ::= TemplateParameter ( "," TemplateParameter )*

TemplateParameter ::= "class" <ID>
                    | ParameterDeclaration

TemplateId ::= <ID> "<" TemplateArgumentList ">"

TemplateArgumentList ::= TemplateArgument ( "," TemplateArgument )*

TemplateArgument ::= TypeName
                   | ShiftExpression

StatementList ::= ( Statement )*

Statement ::= Declaration
            | LabeledStatement
            | Expression ";"
            | CompoundStatement
            | SelectionStatement
            | IterationStatement
            | JumpStatement
            | ";"
            | TryBlock
            | ThrowStatement
```

```
LabeledStatement ::= <ID> ":" Statement
                   | "case" ConstantExpression ":" Statement
                   | "default" ":" Statement

CompoundStatement ::= "{" [ StatementList ] "}"

SelectionStatement ::= "if" "(" Expression ")" Statement [ "else" Statement ]
                     | "switch" "(" Expression ")" Statement

IterationStatement ::= "while" "(" Expression ")" Statement
                     | "do" Statement "while" "(" Expression ")" ";"
                     | "for" "(" ( Declaration | Expression ";" | ";" )
                       [ Expression ] ";" [ Expression ] ")" Statement

JumpStatement ::= "goto" <ID> ";"
                | "continue" ";"
                | "break" ";"
                | "return" [ Expression ] ";"

TryBlock ::= "try" CompoundStatement ( Handler )*

Handler ::= "catch" "(" ExceptionDeclaration ")" CompoundStatement

ExceptionDeclaration ::= ParameterDeclarationList |  "..."

ThrowStatement ::= "throw" [ AssignmentExpression ] ";"

AssignmentExpression ::= ConditionalExpression ( ( "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" |
                         "&=" | "^=" | "|=" ) AssignmentExpression )
                       | ConditionalExpression

ConditionalExpression ::= LogicalOrExpression [ "?" LogicalOrExpression ":" LogicalOrExpression ]

ConstantExpression ::= ConditionalExpression

LogicalOrExpression ::= LogicalAndExpression ( "||" LogicalAndExpression )*

LogicalAndExpression ::= InclusiveOrExpression ( "&&" InclusiveOrExpression )*

InclusiveOrExpression ::= ExclusiveOrExpression ( "|" ExclusiveOrExpression )*

ExclusiveOrExpression ::= AndExpression ( "^" AndExpression )*

AndExpression ::= EqualityExpression ( "&" EqualityExpression )*

EqualityExpression ::= RelationalExpression ( ( "!=" | "==" ) RelationalExpression )*

RelationalExpression ::= ShiftExpression ( ( "<" | ">" | "<=" | ">=" ) ShiftExpression )*

ShiftExpression ::= AdditiveExpression ( ( "<<" | ">>" ) AdditiveExpression )*
```

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

MultiplicativeExpression ::= PmExpression ((“*” | “/” | “%”) PmExpression)*

PmExpression ::= CastExpression ((“.*” | “->*”) CastExpression)*

CastExpression ::= “(“ TypeName “)” CastExpression
| UnaryExpression

UnaryExpression ::= “++” UnaryExpression
| “--” UnaryExpression
| UnaryOperator CastExpression
| “sizeof” (“(“ TypeName “)” | UnaryExpression)
| NewExpression
| DeleteExpression
| PostfixExpression
| **ADL_BasicExpression**
| **ADL_CallStateExpression**

ADL_CallStateExpression ::= “@” UnaryExpression

NewExpression ::= [“::”] “new” [“(“ Expression “)”] (“(“ TypeName “)” | NewTypeId)
[ArgumentList]

NewTypeId ::= DeclarationSpecifiers [NewDeclarator]

NewDeclarator ::= PtrToMember CvQualifierSeq [NewDeclarator]
| DirectNewDeclarator

DirectNewDeclarator ::= (“[“ Expression “]”)*

DeleteExpression ::= [“::”] “delete” [“[“ “]”] CastExpression

UnaryOperator ::= “&” | “*” | “+” | “-” | “~” | “!”

PostfixExpression ::= PrimaryExpression
(ArraySuffix | DotAccessSuffix | RefAccessSuffix | ArgumentList
PostDeIncrement)*
| SimpleTypeSpecifier ArgumentList

ArraySufix ::= “[“ AssignmentExpression “]”

ArgumentList ::= “(“ [Expression] “)”

DotAccessSuffix ::= “.” IdExpression [ArgumentList]

RefAccessSuffix ::= “->” IdExpression [ArgumentList]

IdExpression ::= [ScopeOverride] (<ID> | “operator” Optor | “~” <ID>)

PrimaryExpression ::= **TDD_ADLExpression**
| **TDD_ADLnewExpression** ArgumentList
| IdExpression [ArgumentList]
| Constant
| “this”
| **“return”**
| <STRING>
| ParentheizedExpression

ParentheizedExpression ::= “(“ Expression “)”

Expression ::= AssignmentExpression (“,” AssignmentExpression)*

Constant ::= <OCTALINT>
| <OCTALLONG>
| <DECIMALINT>
| <DECIMALLONG>
| <HEXADECIMALINT>
| <HEXADECIMALLONG>
| <UNSIGNED_OCTALINT>
| <UNSIGNED_OCTALLONG>
| <UNSIGNEDDECIMALINT>
| <UNSIGNEDDECIMALLONG>
| <UNSIGNED_HEXADECIMALINT>
| <UNSIGNED_HEXADECIMALLONG>
| <CHARACTER>
| <FLOATONE>
| <FLOATTWO>
| “true”
| “false”

Optor ::= “new” [“[“ “]”]
| “delete” [“[“ “]”]
| “+” | “-” | “*” | “/” | “%” | “^” | “&” | “|” | “~” | “!” | “=” | “<“ | “>” | “+=” | “-=”
| “*=” | “/=” | “%=” | “^=” | “&=” | “|=” | “<<“ | “>>” | “>>=” | “<<=” | “==”
| “!=” | “<=” | “>=” | “&&” | “||” | “++” | “--” | “,” | “->*” | “->” | “(“ “)” | “[“ “]”
| DeclarationSpecifiers [(“*” | “&”)]

ExceptionSpec ::= “throw” [“(“ [ParameterList] “)”]

5.2 ADL productions

ADL_ClassDeclaration ::= “adlclass” <IDENTIFIER> **[ADL_SuperClass]**
“{“ **[ADL_Prologue] [ADL_Epilogue]**

(**ADL_InlineDeclaration** | **ADL_BehaviorDeclaration**) * "}"

ADL_BehaviorDeclaration ::= [DeclarationSpecifiers] FunctionDeclarator "{" **[ADL_Prologue]**
ADL_BehaviorSpecification [ADL_Epilogue] (NLD_Annotation) * "}"
| DtorCtorDeclSpec CtorDeclarator [ExceptionSpec] "{" **[ADL_Prologue]**
ADL_BehaviorSpecification [ADL_Epilogue] (NLD_Annotation) * "}"

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

ADL_BehaviorClassification::= "[" ("normal" "=" **ADL_Expression** ";"
| "abnormal" "=" **ADL_Expression** ";")* "]"

ADL_AssertionGroup ::= "{" (**ADL_Binding**)* (**ADL_InheritedSemantics**)*
(**ADL_Statement**) * (**NLD_Annotation)*** "}"

ADL_InlineDeclaration ::="inline" DeclarationSpecifiers FunctionDeclarator
"{" **ADL_AssertionGroup** "}"

ADL_Binding ::= "define" **ADL_NamedParamList** "with" [<ID> "="] **ADL_Expression ";"**

ADL_SuperClass ::= ":" ["public"] <ID> ("," ["public"] <ID>)*

ADL_Prologue ::= "prolog" CompoundStatement

ADL_Prologue ::= "epilog" CompoundStatement

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

ADL_InheritedSemantics ::= <ID> "::" "semantics" " **;** "

ADL_IfStatement ::= "if" "(" **ADL_ImplExpression** ")" **ADL_AssertionGroup**
["else" (**ADL_AssertionGroup** | **ADL_IfStatement**)]

ADL_TryStatement ::= "try" **ADL_AssertionGroup**
("catch" "(" ExceptionDeclaration ")" **ADL_AssertionGroup**)+

ADL_QuantifiedAssertion ::= **ADL_Quantifier** "(" **ADL_DomainList** ")" **ADL_AssertionGroup**

ADL_Quantifier ::= "forall" | "exists"

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

ADL_Domain ::= **ADL_NamedParam** ":" ConditionalExpression

ADL_NamedParamList ::= **ADL_NamedParam** ("," **ADL_NamedParam**)*

ADL_NamedParam ::= DeclarationSpecifiers Declarator

ADL_Label ::= (<ID> ":")*

ADL_Tags ::= “[“ <ID> (“,” <ID>)* “]”

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

ADL_Expression ::= **ADL_ImplExpression**

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

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

ADL_BasicExpression ::= “normal”
| “abnormal”
| **ADL_AssertionGroup**
| “unchanged” ArgumentList
| **ADL_ThrownExpression**

ADL_ThrownExpression ::= “thrown” “(“ TypeNameList “)”

5.3 TDD Productions

TDD_AnnotatedDeclaration::= [**TDD_UseDeclaration**] **TDD_ClassDeclaration**

TDD_ClassDeclaration ::= “tddclass” <IDENTIFIER>
“{“ (**TDD_ClassBodyDeclaration**)* “}”

TDD_UseDeclaration ::= (“use” <ID> “;”)+

TDD_ClassBodyDeclaration::= **TDD_DatasetDeclaration**
| **TDD_FactoryDefinition**
| **TDD_TestDirective**
| **TDD_FieldDeclaration**
| FunctionDefinition

TDD_FieldDeclaration ::= DeclarationSpecifiers Declarator “=” AssignmentExpression
(“,” Declarator “=” AssignmentExpression)* “;”

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

TDD_FactoryDefinition ::= "factory" DeclarationSpecifiers FunctionDeclarator CompoundStatement
[“relinquish” “(“ ParameterDeclaration “)” CompoundStatement]

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

TDD_DatasetDomain ::= **ADL_NamedParam (“:” |** “=”) **TDD_DatasetExpr**
| **TDD_DatasetExpr**

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

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

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

TDD_DatasetConcatenationExpr ::= TDD_DatasetLiteral
| **TDD_FactoryCall**
| **TDD_DatasetSingleton**

TDD_DatasetSingleton ::= <ID> | Constant

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

TDD_ADLExpression ::= "ADL" "(" PrimaryExpression ")"

TDD_ADLnewExpression ::= "ADL_new" IdExpression

5.4 NLD productions

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**