ADL 2.0 for C Language Reference Manual, Version 1.1

ADL for C

The Open Group Research Institute

SunTest

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
1.0 Delta	Third Revision
1.0	Updated in accordance with version 2.0.2 of the ADL Translation System.
1.1	Updated in accordance with version 2.0.3 of the ADL Translation System

COPYRIGHT AND LICENSE NOTICE

Trademarks

Change Log

Release 1.1

3. Test Annotations

Updated section 3.1.4 and all examples to include the “import” keyword.

Updated examples according to the new syntax for TDD_ADLExpression.

5. Complete Grammar

Replaced IdExpression with Name.

Fixed typo in TDD_DatasetDeclaration.

Updated TDD_FieldDeclaration.

Updated TDD_ADLExpression.

Release 1.0

1. Introduction

No substantive change.

2. Semantics Annotations

Major update of most ADL productions and ADL examples so that it conforms to the last ADL/C grammar. Note this change only concerns the syntax, no other substantive changes have occured.

3. Test Annotations

Major update of most TDD productions and TDD examples so that it conforms to the last ADL/C grammar. Note this change only concerns the syntax, no other substantive changes have occured.

4. NLD Annotations

Major update of most NLD productions and NLD examples so that it conforms to the last ADL/C grammar. Note this change only concerns the syntax, no other substantive changes have occured.

5. BNF

Major update of most ADL, TDD and NLD productions.

Release 1.0 Delta

1. Introduction

No substantive change.

2. Semantics Annotations

2.1: ADL_Inline_Declaration and ADL_FunctionDeclaration gathered in ADL_AnnotatedDeclaration

TranslationUnit changed to contain “ADL Annotated Declaration”.

2.2.2; ADL_ImplExpression removed. It is equivalent to ADL_Expression.

2.5: Added global Prologues and Epilogues in ADL_AnnotatedDeclaration

ADL_Prologue and ADL_Epilogue changed: Statement replaced by CompoundStatement

TBD resolved: a functional example is performed.

3. Test Annotations

3.1.3: TBD resolved. A factory definition is a function definition.

3.1.7: new section for the use importation clause.

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.

Introduction of constants.

3.3: Slight changes in the grammar.

4. NLD Annotations

No substantive change

Release 1.0 Gamma

1. Introduction

No substantive change.

2. Semantics Annotations

Terminology changes: ADL prefix added for ADL Non-Terminals.

2.1: “Annotated function declaration” changed to “ADL function declaration”.

Example 2.2; “define” changed “inline” for the inline declaration.

2.2: ADL Expressions changed to ADL Syntax;

2.2.1: Assertion groups redefined: “group_expression” changed to “ADL_AssertionGroup”

2.2.3 ADL_IfStatement modified and redefined;

“Expression” in ADL_Domain changed to “ConditionalExpression”

2.3: “Multiple Behavior Descriptions For The Same Function” and “Extending the Specification of a function” removed: ADL for C does not support inheritance, C programmer is supposed to ignore this concept..

2.3.1: Added an example for “unchanged” .

2.3.2: “Definitions” changed to “Bindings”.

2.4: ADL_InlineDeclaration modified: “define” changed “inline”

Added an example.

2.5: Added Prologues and Epilogues to perform preliminay initializations.

Added TBD for the example given.

3. Test Annotations

No substantive change

4. NLD Annotations

4.5: Updating the NLD syntax.

Release 1.0 Beta

1. Introduction

No substantive change.

2. Semantics Annotations

No substantive change.

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 removed from language definition.

3.2: TBD resolved: factories are not implicit datasets.

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

Example 3.10: TBD resolved: test directive syntax clarified.

Example 3.12: Rules for multiple reference simplified.

3.3.1: TBD resolved: external dataset reference clarified.

4. NLD Annotations

Example 4.3: TBD resolved: rules on formal argument name clarified.

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

Example 4.9: TBD resolved: Rule on markup (DocBook 3.0 Para entity) clarified.

Release 1.0 Alpha

Initial release.

1 Introduction

This document describes the enhancements to the ADL Language for the ANSI C programming language. The ADL Language 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 ANSI 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 Java appear in **boldface**.
- Lexical tokens and reserved words may appear literally within quotations, or the name of the lexical token may appear in angle brackets like <STRING>.
- The left hand and the right hand sides of productions are separated by the symbol "**::=**". For presentation purposes, the entire right hand side of a production may not be introduced at the same time. The symbol "**+::=**" is used to indicate that the current production is an augmentation of another production with the same left hand side that has been introduced earlier. For example, "**A ::= B**" followed by "**A +::= C**" is the same as "**A ::= B | C**".

2 Semantics Annotations

The ADL extensions that allow the definition of the semantics of a function, or of a collection of functions that constitute a programmer's interface to a given functionality, are discussed in the sections below.

2.1 Describing Semantics Of Functions

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

ADL_TranslationUnit := IncludeFileList
[**ADL_AnnotatedDeclaration** | **TDD_AnnotatedDeclaration**]
(**NLD_Annotation**)*
<EOF>
IncludeFileList := (IncludeFileDeclaration)*
IncludeFileDeclaration := "#include" <INCLUDED_FILE_NAME>
ADL_AnnotatedDeclaration := "adlmodule" (<ID>)?
"{"
[**ADL_Prologue**] [**ADL_Epilogue**]
(**ADL_BehaviorDeclaration** | **ADL_InlineDeclaration**)*
(**NLD_Annotation**)*
"}"

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

Behavior declarations and inline declarations are described in Section 2.3 and Section 2.4 respectively; a simple example illustrating the use of these constructs is shown here. Suppose there is an interface declared (in file StockBrocker.h) as:

EXAMPLE 2.1 StockBroker.h

```
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 ADL specification:

EXAMPLE 2.2 StockBrokerSpec.adl

```
#include "StockBroker.h"
#include "StockBrokerAux.h"

adlmodule StockBrocker {

  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;
     }
  }
}
```

In this example, a file with three functions Cash_Balance, Stock_Balance, and Buy is extended with a description of the behavior of Buy, written in the function declaration syntax. The two boolean expressions appearing within "semantics { ... }" describe legitimate behavior of the Buy function. In these expressions, "@" is an unary operator (referred to as the *call state operator* — see Section 2.3.1) whose sole function is to evaluate its argument prior to the execution of the 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 annotation as an inline 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 an inline declaration is an ADL expression (described fully below) rather than a C block statement.

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

2.2 ADL Syntax

ADL provides an 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

The basic bloc construct of ADL is the *assertion group*, which is a list of *statements.*

ADL_AssertionGroup := "{" (**ADL_Binding** ";")* (**ADL_Statement** ";")* "}"

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 i; this can occur with the inline/define constructs.

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

ADL_Assertion := [**ADL_Labels**] [**ADL_Tags**]
(**ADL_Expression** | **ADL_QuantifiedAssertion**)

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

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

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).They may be given labels and tags. Labels are used when reporting the value of the expression; tags are used by the ADL runtime to selectively evaluate assertions.

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

ADL_BasicExpression +::=**ADL_AssertionGroup**

The following fragment taken from an earlier example is an example of a assertion group:

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

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

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

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

The list of expressions in an assertion group may be preceded by *bindings* (variable declarations and initial value assignments to them).

2.2.2 ADL Specific Expressions

While most ADL specific expressions are described in this section, 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 2.3.2
- The exception operator — Section 2.3.4
- Inline declarations — Section 2.4
- Prologues and Epilogues — Section 2.5

The remainder of this section describes all other 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`. (The exception operator (<:>) is described in the section 2.3.7)

PrimaryExpression +::= "return"

Primaries are extended in ADL with the reserved word **`return`**, which is used to refer to the return value of a function. The primary return may be used only in behavior specifications (Section 2.3) of functions with non-void return types and may not appear within an operand of a call state operator (Section 2.3.1).

2.2.3 ADL Specific Statements

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

"`If` statements" provide a way to conditionally evaluate expressions. The meaning is quite similar to the "`?:`" operator. The types of both the group expressions in 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 parentheses) must be `boolean` valued and are evaluated from top to bottom until the first one that evaluates to `true`. The assertion group of this `true` expression is then evaluated. This is the value of the `if` statement.

The assertion groups of 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.

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

ADL_Quantifier := "forall"
| "exists"

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

ADL_Domain := **ADL_NamedParam** ":" **ADL_DomainExpression**

ADL_DomainExpression := ("ADL_short_range" | "ADL_int_range" | "ADL_long_range")
"(" AssignmentExpression "," AssignmentExpression ")"

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 expression may contain any number of domains. Each domain is specified with the type of the sequence element, a new variable that takes on the values in the sequence one by one, and finally the sequence itself. An example of a domain that iterates over the integers 1 through 10 is:

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

The ADL compiler generates the appropriate code to loop over the sequence of `long`'s starting from `i` and ending at `j`.

In the case of the universal quantifier (**`forall`**), the quantification expression (which must take on a `boolean` value) will be true if the group expression is `true` for all value assignments for quantification variables from their domains. In the case of the existential quantifier (**`exists`**), the quantification expression will be true if the group expression is `true` for at least one set of value assignments for the quantification variables.

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 that says that at least one number between 1 and 10 is divisible by 3 (in fact there are more than one):

```
semantics {
     exists (long i : ADL_long_range(1.10)) {
        i%3 == 0;
     }
}
```

2.3 Behavior Specification

The specification of the behavior of a function in its simplest form is the function declaration followed by the reserved word “semantics” followed by an assertion group.

These assertions 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 := (FunctionDeclarator | DeclarationSpecifiers FunctionDeclarator)
“{“ **ADL_BehaviorSpecification** “}” (**NLD_Annotation**)*

ADL_BehaviorSpecification := “semantics” **ADL_AssertionGroup**

Every time a 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);
report(assertion_1);
assertion_2 = (Stock_Balance(account, symbol) == tmp3 +
no_of_shares);
report(assertion_2);
```

Step 4: Determination of consistent behavior:

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

Behavior descriptions can refer to inline 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 is a unary operator. It has the effect of evaluating its argument before the call to the specified function.

UnaryExpression +::= **ADL_CallStateExpression**

ADL_CallStateExpression := "@" UnaryExpression

UnaryExpression +::= **ADL_BasicExpression**

ADL_BasicExpression+::= "unchanged" "(" **ADL_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 ADL is maintained:

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

is a syntactic sugar for:

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

2.3.2 Bindings

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

ADL_Binding ::= "define" **ADL_NamedParamList** "with" (<ID> "=")* **ADL_Expression**

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

ADL_NamedParam ::= DeclarationSpecifiers Declarator

Suppose the simple stock broker interface was modified to:

```
void Cash_Balance(long account, long *balance);

long Stock_Balance(long account,char* symbol);

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

The essence of the modification is that `Cash_Balance` no longer returns its result as a function return value, rather it returns its result through a pointer parameter. Bindings may be used to bind a local variable to this pointer parameter, and this local variable can then be used in the assertions. The earlier assertion group would be modified to be:

```
{
  define long pre_cash_bal with
     @Cash_Balance(account, &pre_cash_bal);
  define long post_cash_bal with
     Cash_Balance(account, &post_cash_bal);

  post_cash_bal ==
     pre_cash_bal - @cost(symbol, no_of_shares);
  Stock_Balance(account, symbol) ==
     @Stock_Balance(account, symbol) + no_of_shares;
}
```

Even though `Stock_Balance` continues to return its result as a function return value, the second assertion may also be modified to use bindings. The following is equivalent to the above assertion group:

```
{
  define long pre_cash_bal with
     @Cash_Balance(account, &pre_cash_bal);
  define long post_cash_bal with
     Cash_Balance(account, &post_cash_bal);
  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;
}
```

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.

2.3.3 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_BehaviorClassification := "[" (**ADL_NormalBehavior** | **ADL_AbnormalBehavior**)* "]"

ADL_NormalBehavior := "normal" "=" **ADL_Expression** ";"

ADL_AbnormalBehavior := "abnormal" "=" **ADL_Expression** ";"

Normal and abnormal are used to classify the outcome of function invocation. If a function always succeeds and has no way of giving an error indication to the caller, then the specification of that function will have:

```
[ normal = true; ]
```

If, as is common for system functions, there is a special return value that signals to the caller that some problem occurred and the function was unable to perform the requested task, then the abnormal behavior binding will check for that special return value. For example, it is common in Unix interfaces to have

```
[ abnormal = return < 0; ]
```

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 `true` and `abnormal` defaults to `false`.

- 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 2.3 StockBroker2.adl with behavior classification

```
#include “StockBroker2.h”
#include “StockBrokerAux.h”

adlmodule StockBroker2 {
  inline long cost(char* symbol, long no_of_shares)
  {
     no_of_shares * price(symbol);
  }

  int Buy(long account,
          char* symbol,
          long no_of_shares) {
     semantics
     [  normal = return == 0;
        abnormal = return < 0; ]
     {
        if (normal) {
           Cash_Balance(account) ==
                    @Cash_Balance(account)
                    - @cost(symbol, no_of_shares);
           Stock_Balance(account, symbol) ==
                    @Stock_Balance(account, symbol)
                    + no_of_shares;
        };
     }
  }
}
```

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 abnor-

mal behavior of `Buy` in Section 2.3.4. Note that in this example, the main part of the behavior description of `Buy` is guarded by the "`if`" expression (Section 2.2) "`if (normal) ...`". In this case, no behavior is specified unless the function completes normally.

2.3.4 The Exception Operator

ADL provides the exception operator "<:>" whose meaning is based on behavior classifications. It is called the exception operator to reflect the idea that an abnormal behavior condition is an exception to the expected flow of events; in fact, we may refer to such a condition as an exception. It is a binary operator whose syntax is:

ADL_ImplOp +::="<:>"

In usual usage of this operator, the left operand is the enabler of an exception, while the right operand is an exception expression identifying the particular problem. The following example illustrates this typical use:

```
bad_account(account) || bad_symbol(symbol) <:>
stockbroker_errno == EBADCALL
```

Informally, the exception operator states that if the condition specified by the left operand is true, then one of the exceptions specified in the right operand will occur. However if some other exception happens to occur for whatever reason, then the exceptions in the right operand need not occur. This kind of weaker specification is useful because in general exceptions may occur for a variety of uncontrollable reasons and also because the choice of exception communicated to the caller is often left as an implementation decision when it is possible for more than one exception to occur in the same call.

Finally, the exception operator also says that if the exception specified in the right operand occurs, then the left operand must be true.

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) <:> errno == ENOENT;
  disk_full() <:> errno == ENOSPC;
};
```

If we assume that **`abnormal`** is defined as `return < 0`, this assertion group could probably be used to specify a file open function. It reads: If the file `f` does not exist, then errno must be ENOENT. Similarly, if the disk is full, errno must be ENOSPC. However, it does not restrict other exception conditions. But it does say that `errno==ENOENT` should only occur if the file `f` does not exist, and `errno==ENOSPC` should only occur when the disk was full. An interesting consequence is that if both the file does not exist

and the disk is full, errno may take either value. The following assertion group strengthens the above assertion group to require that only these two values of errno or EAGAIN may occur:

```
{
  !file_exists(f) <:> errno == ENOENT;
  disk_full() <:> errno == ENOSPC;
  abnormal ==>
     errno == ENOENT || errno == ENOSPC || errno == EAGAIN;
};
```

Now we complete the earlier stock broker example with specification of abnormal behavior. Two auxiliary function declarations — `bad_acct` and `bad_sym` — are added:

EXAMPLE 2.4 StockBroker2.adl with exceptions

```
#include “StockBroker2.h”
#include “StockBrokerAux.h”


adlmodule StockBroker2 {

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

  int Buy(long account,
          char* symbol,
          long no_of_shares) {
     semantics
     [  normal = return == 0;
        abnormal = return < 0; ]
     {
        bad_acct(account) <:>
        (stockbroker_errno == EBADCALL);
        TranslationUnitTranslationUnitbad_sym(symbol) <:>
        (stockbroker_errno == EBADCALL);
        if (normal) {
           Cash_Balance(account) ==
              @Cash_Balance(account)
              - @cost(symbol, no_of_shares);
              Stock_Balance(account, symbol) ==
              @Stock_Balance(account, symbol)
              + no_of_shares;
        };
     }
  }
}
```

Note the right hand side of the two exception operators refer to the same exception condition, but different additional conditions associated with the condition. This new

behavior description is different from the earlier behavior description in Section 2.3.4 in a few interesting ways, some of which are:

- It makes clear the abnormal behavior.
- This leaves the particular exception condition 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 does not allow for the possibility that `Buy` might return a positive value. Such a condition, where neither `normal` nor `abnormal` is true, is reported as a specification error if it should be observed; similarly for the condition that both `normal` and `abnormal` are true.

2.4 inline declarations

Inline declarations (or inline function declarations) are the other way to define concepts used in behavior descriptions (along with C declarations). Their syntax is:

ADL_InlineDeclaration ::= "inline" declaration_specifiers function_declarator **ADL_AssertionGroup**

The significant difference between inline declarations and C function declarations is that the inline declarations are implemented by ADL assertions, with the additional ADL operators, rather than by C statements.

In other words, Inline declarations are considered as macros in the usual C pre-processor meaning. The call to an inline declaration 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(char* symbol, long no_of_shares)
{
  no_of_shares * price(symbol);
};
```

any expression cost(char* symbol, long no_of_shares) will be replaced by:

{ no_of_shares * price(symbol); }

2.5 Prologues and Epilogues

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

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

Ther are two kinds of prologues/epilogues: either global in ADL_AnnotatedDeclaration or local in ADL_FunctionDeclaration.

ADL_AnnotatedDeclaration := [**ADL_Prologue**] [**ADL_Epilogue**]
(**ADL_InlineDeclaration** | **ADL_BehaviorDeclaration**)*

ADL_BehaviorDeclaration := "{" [**ADL_Prologue**] **ADL_BehaviorSpecification** [**ADL_Epilogue**] "}"
(**NLD_Annotation**)*

ADL_Prologue := "prolog" CompoundStatement

ADL_Epilogue := "epilog" CompoundStatement

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

EXAMPLE 2.5 BankAccount.adl with prologues and epilogues

```
#include "AccountFile.h"

adlmodule AccountFile {

  prolog {
     Char* FileRadix = "/jdbc/odbc/wombat";
  }

  long deposit(long account, long amt) {
     prolog {
       FILE* fd = OpenAccountFile(FileRadix,account);
     }
     semantics {
       return == getBalanceAfterDepositFromFile(fd, amt);
     }
     epilog {
       close(fd);
     }
   }
  }
}
```

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 function, before the code that deals with the semantics assertions. The epilog code is copied at the end of this function (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
Step 2: Execution of the local prologue
Step 3: Evaluation and saving of call-state expressions
Step 4: Evaluation of the assertions and test reporting

Step 5: Execution of the local epilogue
Step 6: Execution of the global epilogue

Note that the global prologue is a purely syntactic construct: variables declared therein are *not* global variables, but variables local to all the specified function — 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 functions whose behavior is specified in the adl file.

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 epilogues. 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.

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 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 Cartesian product 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:

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

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 expression, 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. The annotated functions are looked for in the declared list of imported adl modules.

3.1.5 Test Directives

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

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

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

3.1.6 Assertion

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

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

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

3.1.7 Importation

It is possible to import datasets or factories defined in other TDD files, by using the "use" feature of 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 responsability of the user to ensure that at runtime 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 General Syntax & Examples

This section presents the general syntax along with examples that motivate the design. Several syntactic conveniences are used in the examples:

Expressions as Datasets. A target-language expression can be used as a dataset; the expression is interpreted as a singleton dataset.

Types as datasets. The name of a data type can be used as a dataset; it is interpreted to mean all members of the data type. In C, only `enum` types and `char` are small enough to be useful as datasets.

Dataset Concatenation. The "+" operator is overloaded with dataset concatenation.

3.2.1 Datasets and Data Construction

Some examples of data generation.

EXAMPLE 3.1 t1.tdd : The Simplest Test

```
#include “mymath.h”
import mymath;
tddmodule t1 {
  test ADL(plus(3,4));
}
```

The simplest test is just an invocation of an annotated function. Formally, this test expression is the application of the annotated version of the 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, `plus` is a checked function from “mymath.h”, annotated in adl module “mymath”.

EXAMPLE 3.2 t2.tdd : A Simple Dataset

```
#include “mymath.h”
import mymath;
tddmodule t2 {
  dataset int A = {1,3,5 .. 7};
  test ADL(plus(A,1));
  test ADL(plus(1,A));
}
```

This tests `plus` when adding the constant 1, from both sides.

EXAMPLE 3.3 t3.tdd : Compound Data Construction

```
#include “myio.h”
import myio;
tddmodule t3 {
  factory RandomAccessFile
           make_file(char* nm, char* mode) { /* ... */ };
```

```
  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[512];

  test (RandomAccessFile F=F0) {
     ADL(read(F,buf,512));
  }

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

Dataset F0 has 3x2=6 members, while F1 has 1+2+1=4 members. Note that F1 is the union of several datasets, each produced by a separate invocation of the factory; the example uses "+" as the dataset union operator.

This example shows the full syntax for a test directive, with the datasets listed explicitly as an initialized declaration list. This is the fundamental syntax for a test directive; the shorter procedure call syntax is an abbreviation. Note also that the local variable `buf` and the constant "512" are used as datasets in the expressions in the directive.

EXAMPLE 3.4 t4.tdd : 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.

```
import myio;
use t3;

tddmodule t4 {
  factory void setup_system(int condition_code) {
     // ...
  } relinquish() { ... }
  dataset void setup_set = setup_system;

  test (setup_set,
        RandomAccessFile F = F1,
        char* data={"", "hello"})
  {
     char* tmp;
```

```
    ADL(write(F,data));
    seek(F,0);
    tmp = ADL(read(F));
    tdd_assert( "streq(tmp,data)", streq(tmp,data));
  }
}
```

This example shows the use of an unchecked function (`seek`) in conjunction with some checked functions (`write` and `read`, defined in the adl module "myio".). All three function invocations result in function invocations on the underlying implementation; however, the checked function invocations are relayed through a checking function that implements the semantic checks specified by the ADL semantics annotation. It is an error to invoke the ADL-checked version of a function if that function does not have a semantics annotation.

Imported dataset names (through the "`use`" clause) are used if there is no ambiguity about their origin. Unqualified syntax (`F = F1` in the example) is possible if

- `F1` is defined in the current tdd file, or
- `F1` is defined in at most one of the "used" tdd files.

Since qualified syntax is not used in C, F1 can not be defined in two tdd imported files.

These rules are also valid for factory importation. Only datasets and factories are importable: the constants, test functions and test directives are not.

EXAMPLE 3.5 t5.tdd : 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.

```
tddmodule 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();
}
```

EXAMPLE 3.6 t6.tdd : 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 local: it cannot be imported through the “use” clause.

```
#include “bank_test.h”;
import bank;

tddmodule 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 account_type t,
     int size_code s) { /*...*/ }

  dataset account Account1 =
     acct(account_type t, size_code s);

  factory int amount(int size_code size) { /*...*/ }
  dataset Bank bank = { make_test_bank() };

  test ( Bank b = bank,
         account a = Account1,
         int amounts = amount(size_code) ) {
     ADL(withdraw(b,a,amounts));
  }
  test ( Bank b = bank, account a = Account1) {
     ADL(balance(b,a));
  }
}
```

EXAMPLE 3.7 t7.tdd : Better Test Variables

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

```
#include “bank_test.h”
import bank;

tddmodule 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));

  dataset Bank bank = { make_test_bank() };

  test ( Bank b = bank, account a = Acct,
        int amounts = amount(size_code) ) {
     ADL(withdraw(b,a,amounts));
  }
  test ( Bank b = bank, account a = Acct,
        int amounts = {0.1, 124.1e10, 1125.333} ) {
     ADL(deposit(b,a,amounts));
  }
  test ( Bank b = bank, account a = Acct ) {
     ADL(balance(b,a));
  }
}
```

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.

EXAMPLE 3.8 t8.tdd : Chaining Factories

```
#include “testframe.h”

tddmodule t8 {
  dataset int length_code =
              {ZERO, ONE, MEDIUM, LONG, TOO_LONG};

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

  /* use datatype names as datasets */
  dataset char* file_name_set =
     make_file_name(boolean, boolean, boolean,
                    int length_code);

  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 int File_set =
      make_file(file_name_set);

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

This illustrates several techniques for re-using factories.

EXAMPLE 3.9 t9.tdd : Multiple Data Values

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

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

```
#include “io_test_data.h”
#include “io_test_aux.h”
import myio;

/* last included file defines
 *   struct port_pair {
 *      char* host;
 *      int port;
 *   }
 */

tddmodule t9 {
  factory port_pair make_port_pair(int pp_code) { /*...*/ }
  dataset int port_pair_code = { 0 .. 10 };

  dataset port_pair Ports =
     make_port_pair(port_pair_code);

  test (char* data = io.data_set,
        port_pair pp = Ports) {
     socket_t s = makeSocket(pp.host, pp.port);
     ADL(write(s,data, strlen(data)));
  }
}
```

EXAMPLE 3.10 t10.tdd : Test Directives and Procedures

Simple examples of test directives were given in the previous section. To recap, here are examples of the alternative syntaxes for test directives:

```
#include “some_data.h”;

tddmodule t10 {
  test (data_t d = data_set) {
     hashCode(d);
  }
}
```

The syntax is:

`test (`*`type id = dataset,...`*`) `*`statement`*

A test directive body has the same syntax as *compound 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 in some way.

Local variables are created to range over the specified datasets. Syntactically; it’s 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 be con-

tain a dataset expression denoting a dataset over 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 in text directives. For instance, a goto statement is not a legal test directive statement.

EXAMPLE 3.11 t11.tdd : 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 “testdata.h”
import myio;

tddmodule t11 {
  void read_then_write(FILE* f,
                       char buf[512]) {
     long pos;

     pos = ftell(f);
     ADL(fread(buf,1,512,f));
     ADL(fseek(f,pos,SEEK_SET));
     ADL(fwrite(buf,1,512,f));
  }

  test read_then_write(io.File_set, io.Buf_set);
}
```

This defines and then calls a test procedure that, when executed, will check that the functions `fread`, `fseek`, and `fwrite` operate together correctly when used in this particular way. More exactly, the test procedure will exercise the functions together, giving the assertion-checking code a chance to check the behavior of annotated functions. This is not a good way to test for error handling; it may prove useful when checking the normal operation of an interface.

EXAMPLE 3.12 t12.tdd : 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 use the long form of the test directive.

```
import mymath;
tddmodule t12 {
  dataset int A = { 1, 2, 3 };

  test ADL(plus(A,A));        // 9 evaluations

  test (int a = A) {
     ADL(plus(a,a));          // 3 evaluations
  };
}
```

EXAMPLE 3.13 t13.tdd : Void Dataset Use

Most of the examples have used the procedure-call syntax for the test directive. If the user needs explicit control over the order of selection from datasets, or needs to use void datasets, the longer syntax for a test directive may be used.

```
import adlmod;
tddmodule t13 {
  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.

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.

4.2 Syntax and Semantics

4.2.1 Simple Data Identifier Translation

```
/* C code */
int amount;

/* ADL source */
adlmodule C {
  nld {
     amount = “the correct amount”;
  }
}
```

This declares a translation for the global identifier `amount`. Any expression using an identifier named `amount` will be translated to use the declared string.

4.2.2 A Simple Function Translation

Functions can have translations declared in a similar fashion.

```
int balance();

nld {
  balance() = “the balance of the account”;
}
```

This declares a translation for `balance()`. Any expression using this function identifier will be translated to use the declared string.

4.2.3 Using `semantics` And `nld` Blocks

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

```
int balance(int ac)
semantics {
  ac != 0;
  nld {
     .ac = “the account number”;
  }
}
```

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 {
  balance(int)::ac = “the account number”;
}
```

The formal argument name from the function declaration is used as the name of the local argument, using the "::" scope resolution operator borrowed from C++.

EXAMPLE 4.14 Shadowing or Overriding A Translation

```
int i;

nld {
  i = "the loop counter";
}

void B() {
  semantics { /* ... */ }
  nld {
     .i = "B's i";
  }
}
```

An expression using `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 within B using `i` will be translated as "B's i".

4.2.4 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. If an interpolated identifier name is the same as one of the translation's formal arguments, the translation of the corresponding actual argument is used instead of any translation for the formal argument name.

EXAMPLE 4.15 Invocation Translation

```
int a;
void f(int i);

nld {
  a = "the actual argument";
  i = "the formal argument";
  f(int) = "using " + $1;
};
```

An expression using `f(a)` will be translated as "using the actual argument".

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”; }
```

4.4 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 {
  f() : &includes = “#include <stdlib.h>”;
  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 `nld` 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 functions in one place, rather than forcing the test engineer to distribute NL information throughout the specification files.

EXAMPLE 4.3 Using Fully Scoped Names

```
nld {
  i = “translation for i”;
  f(int) = “translation for f(int)”;
  f(int)::i = “translation for i in f(int)”;
};
```

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.

5.1 C language productions

```
TranslationUnit ::= ( ExternalDeclaration )* <EOF>

ExternalDeclaration ::= Declaration
                    |   EnumSpecifier [ InitDeclaratorList ] ";"
                    |   FunctionDefinition
                    |   Declaration
                    |   ";"

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

Declaration ::= DeclarationSpecifiers [ InitDeclaratorList ] ";"

TypeModifiers ::= StorageStructSpecifier
              |   TypeQualifier

DeclarationSpecifiers ::= ( TypeModifiers )*
                          BuiltinTypeSpecifier ( BuiltinTypeSpecifier | TypeModifiers )*
                            | [ ( Name | StructSpecifier | EnumSpecifier ) ( TypeModifiers )* ]
                      |   BuiltinTypeSpecifier ( BuiltinTypeSpecifier | TypeModifiers )*
                      |   ( ( Name | StructSpecifier | EnumSpecifier ) ( TypeModifiers )* )

TypeQualifier ::= "const" | "volatile"

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

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

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

InitDeclarator ::= Declarator [ "=" Initializer ]

StructSpecifier ::= ( "struct" | "union" )
                    <ID> [ "{" ( MemberDeclaration )* "}" ]
                     | "{" ( MemberDeclaration )* "}"

MemberDeclaration ::= Declaration
                  |   EnumSpecifier [ MemberDeclaratorList ] ";"
                  |   DeclarationSpecifiers [ MemberDeclaratorList ] ";"
                  |   ";"
```

```
MemberDeclaratorList ::= MemberDeclarator ( “,” MemberDeclarator )*

MemberDeclarator ::= Declarator

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

EnumeratorList ::= Enumerator ( “,” Enumerator )*

Enumerator ::= <ID> [ “=” ConstantExpression ]

PtrOperator ::= “&” CvQualifierSeq
              | “*” CvQualifierSeq

CvQualifierSeq ::= [ “const” | “const” “volatile” | “volatile” | “volatile” “const” ]

Declarator ::= PtrOperator Declarator
             | DirectDeclarator

DirectDeclarator ::= “(“ Declarator “)” [ DeclaratorSuffixes ]
                   | Name [ DeclaratorSuffixes ]

DeclaratorSuffixes ::= ( “[“ [ ConstantExpression ] “]” )*
                     | “(“ [ ParameterList ] “)”
                     | “:” ConstantExpression

FunctionDeclarator ::= PtrOperator FunctionDeclarator
                     | FunctionDirectDeclarator

FunctionDirectDeclarator ::= Name “(“ [ ParameterList ] “)”

ParameterList ::= ParameterDeclarationList [ [ “,” ] “...” ] | “...”

ParameterDeclarationList ::= ParameterDeclaration ( “,” ParameterDeclaration )*

ParameterDeclaration ::= DeclarationSpecifiers ( Declarator | AbstractDeclarator )

Initializer ::= “{“ Initializer ( “,” Initializer )* “}”
              | AssignmentExpression

TypeName ::= DeclarationSpecifiers AbstractDeclarator

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

AbstractDeclaratorSuffix ::= “[“ [ ConstantExpression ] “]”
                           | “(“ [ ParameterList ] “)”

StatementList ::= ( Statement )*
```

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

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 ] ";"

Expression ::= AssignmentExpression ( "," AssignmentExpression )*

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

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 ::= CastExpression ( ( "*" | "/" | "%" ) CastExpression )*

CastExpression ::= UnaryExpression
                 | "(" TypeName ")" CastExpression

UnaryExpression ::= PreIncrementExpression
                  | PreDecrementExpression
                  | UnaryOperatorExpression
                  | SizeOfExpression
                  | PostfixExpression
                  | ADL_BasicExpression
                  | ADL_CallStateExpression

PreIncrementExpression ::= "++" UnaryExpression

PreDecrementExpression ::= "--" UnaryExpression

UnaryOperatorExpression ::= UnaryOperator UnaryExpression

UnaryOperator ::= "&" | "*" | "+" | "-" | "~" | "!"

SizeOfExpression ::= "sizeof" ( "(" TypeName() ")" | UnaryExpression )

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

ArraySuffix ::= "[" AssignmentExpression "]"

DotAccessSuffix ::= "." Name

RefAccessSuffix ::= "->" Name

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

PostDeIncrement ::= "++" | "--"

Name ::= <ID>
```

PrimaryExpression ::= **TDD_ADLExpression**
| Name
| Constant
| <STRING>
| ParentheizedExpression
| “return”

ParentheizedExpression ::= “(“ Expression “)”

UnaryPlusMinusConstant ::= (“+” | “-”) Constant

Constant ::= <OCTALINT>
| <DECIMALINT>
| <HEXADECIMALINT>
| <CHARACTER>
| <FLOATONE>
| <FLOATTWO>
| “true”
| “false”

5.2 ADL Productions

ADL_AnnotatedDeclaration ::= [**ADL_Prologue**] [**ADL_Epilogue**]
(**ADL_InlineDeclaration** | **ADL_FunctionDeclaration**)*

ADL_InlineDeclaration ::= “inline” DeclarationSpecifiers FunctionDeclarator
ADL_AssertionGroup

ADL_TranslationUnit ::= IncludeFileList
[**ADL_AnnotatedDeclaration** | **TDD_AnnotatedDeclaration**]
(**NLD_Annotation**)*
<EOF>

IncludeFileList ::= (IncludeFileDeclaration)*

IncludeFileDeclaration ::= “#include” <INCLUDED_FILE_NAME>

ADL_AnnotatedDeclaration ::= “adlmodule” [<ID>] “{“ [**ADL_Prologue**] [**ADL_Epilogue**]
(**ADL_BehaviorDeclaration** | **ADL_InlineDeclaration**)*
(**NLD_Annotation**)* “}”

ADL_InlineDeclaration **::=** “inline” FunctionDeclarator DeclarationSpecifiers **ADL_AssertionGroup**

ADL_BehaviorDeclaration ::= (FunctionDeclarator | DeclarationSpecifiers FunctionDeclarator)
“{“ [**ADL_Prologue**] **ADL_BehaviorSpecification** [**ADL_Epilogue**]
(**NLD_Annotation**)* “}”

ADL_Prologue ::= “prolog” CompoundStatement

ADL_Epilogue ::= “epilog” CompoundStatement

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

ADL_BehaviorClassification ::= “[“ (**ADL_NormalBehavior** | **ADL_AbnormalBehavior**)* “]”

ADL_NormalBehavior ::= “normal” “=” **ADL_Expression** “;”

ADL_AbnormalBehavior ::= “abnormal” “=” **ADL_Expression** “;”

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

ADL_Binding ::= “define” **ADL_NamedParamList** “with” [<ID> “=”] **ADL_Expression**

ADL_NamedParamList ::= **ADL_NamedParam** (“,” **ADL_NamedParam**)*

ADL_NamedParam ::= DeclarationSpecifiers Declarator

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

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

ADL_Assertion ::= [**ADL_Labels**] [**ADL_Tags**]
(**ADL_Expression** | **ADL_QuantifiedAssertion**)

ADL_Labels ::= (<ID> “:”)*

ADL_Tags ::= “[“ <ID> (“,” <ID>)* “]”

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

ADL_Quantifier ::= “forall” | “exists”

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

ADL_Domain ::= **ADL_NamedParam** “:” **ADL_DomainExpression**

ADL_DomainExpression **::=** (“ADL_short_range” | “ADL_int_range” | “ADL_long_range”)
“(“ AssignmentExpression “,” AssignmentExpression “)”

ADL_Expression ::= **ADL_ImplExpression**

ADL_ImplExpression ::= ConditionalExpression (**ADL_ImplOp** ConditionalExpression)

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

ADL_CallStateExpression ::= “@” UnaryExpression

ADL_BasicExpression ::= “normal”
| “abnormal”
| “unchanged” “(“ **ADL_ArgumentList** “)”
| **ADL_AssertionGroup**

ADL_Return ::= “return”

ADL_ArgumentList ::= **ADL_Expression** (“;” **ADL_Expression**)*

5.3 TDD productions

TDD_AnnotatedDeclaration ::= (**TDD_ImportDeclaration**)*
(**TDD_UseDeclaration**)* “tdd_module” [<ID>]
{“ (**TDD_Declaration** | **NLD_Annotation**)* “}”

TDD_ImportDeclaration ::= “import” <ID> “;”

TDD_UseDeclaration ::= “use“ <ID> “;”

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

TDD_DatasetDeclaration ::= “dataset” **ADL_NamedParam** “=” **TDD_DatasetExpression** “;”

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

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

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

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

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

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

TDD_DatasetExpression ::= **TDD_DatasetConcatenationExpr**
(“+” TDD_DatasetConcatenationExpr)*

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

TDD_DatasetSingleton ::= Name | UnaryPlusMinusConstant | Constant

TDD_FactoryCall ::= <ID> "(" [**TDD_DatasetExpression** ("," **TDD_DatasetExpression**)*] ")"

TDD_ADLExpression ::= "ADL" "(" Name ArgumentList ")"

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**