ADL 2.0 Translation System

ISSUE NUMBER	REASON FOR ISSUE
1.0 Alpha	Document Launch For Review
1.0 Beta	Document Launch For 2nd Review
1.0 Gamma	Update to address design simplification First snapshot from The Open Group Research Institute
1.0 Delta	Additional Command and Control design Parser and code generation design Additional Runtime design Second snapshot from The Open Group Research Institute
1.0	Official IPA delivery
1.1	Revision of the command line interface Description of the documentation generation process.

Trademarks

Change Log

Release 1.1

2. The Driver

2.2: Added mention that NLD files are compiled before any other files.

2.3: Updated command line options and environment variables, removed file suffixes.

2.4: Added section on exit status.

3. Abstract Syntax Tree Design

3.2.1.4: Added note indicating that delegation is not implemented.

3.2.1.5: Added note indicating that externalization is not implemented.

3.3.2: Indicated that SimpleNode extends ADLNode.

4. Input Language Parsers

4.3.2: Added check of ADL() and ADL_new constructions.

6. Documentation Generation Architecture

6.1.6: TBD resolved - the documentation generation process does not use any AST node property.

6.2: Added section describing the general architecture of the documentation generator.

6.3: Added TBD section describing the NL Engine.

6.4: Added section describing the NL Prolog rules.

6.5: Added TBD section on document templates.

7. Runtime Architecture

7.7.2: Fixed some minor error in description and examples of use of tdd_assert.

7.8.1: Updated required version number for JavaCC.

Release 1.0

2. The Driver

2.2: Added architectural design of the driver module.

2.3: Revised options syntax. Added .tdd suffix. Revised C++ file suffixes. Added the extension filter related options. Added default command line when no option is specified. Added configuration file example.

3. Abstract Syntax Tree Design

Removed all references to node additive state.

3.1.1: Added justification to the AST centric design.

3.1.2: Removed “Node state” section.

3.3.2.2: Updated list of basic operations.

4. Input Language Parsers

Added section 4.1.3 on “Extension Parser”.

4.3: TBD resolved - pre-processor directives that are considered are completely described in new section 4.3.1 “C/C++ pre-processing”.

4.3.2: Revised list of generic semantic checks.

4.3.2.2: TBD resolved - list of C++ specific semantic checks.

4.3.3.2: Added reference to ANSI C++ standard for a description of the C++ type compatibility algorithm.

4.3.3.3: Updated type synthesis description.

4.3.3.4: Updated type representation.

4.3.5: TBD resolved - description of the TypeCheck visitor added.

4.4.2: Updated package names.

4.4.3.6: TBD resolved - the credentials management is not implemented in this release.

5. Code Generation

5.2.1.1: Updated the ACO blocks section.

5.2.1.2: Updated the transformation patterns.

5.2.2: TBD resolved - added description of TDD node transformations.

5.2.3: TBD resolved - NLD nodes will probably never be transformed.

5.3.1: Updated declarations of generic unparser fields and methods.

7. Runtime Architecture

7.5: Updated examples.

7.6.1.1: Added missing methods and fixed some others.

7.6.1.2: Added mention of availability of test reporting methods.

7.6.1.3: Added methods ADL_*_range.

7.6.3: Added ADL_setblock function.

7.7.2: Added footnote about availability of bool type. Removed tdd_fail, tdd_Abort, tdd_adl_passed and tdd_get_result methods. Added tdd_skip and infoline.

7.7.4.3: TBD resolved - removed ADL_FORK mode which is not necessary.

7.8.1: Updated ADLT compilation environment.

7.8.3: Updated test build environment.

Release 1.0 Delta

1. Introduction

1.3: Fixed transformation from IDL.

2. The Driver

2.2: TBD resolved - implementationof the driver requires at least a C wrapper.

2.3: Updated list of options. TBD resolved by adding the list of environment variables.

4. Input Language Parsers

Added description of semantic checks and symbol table management.

4.3: TBD resolved: list of semantic checks added in sections 4.3.1, 4.3.2 and 4.3.3.

4.3.1.2: Added TBD for C++ specific checks.

4.3.4: Added TBD for missing description of TypeCheck visitor.

4.4.3.6: Added TBD for improvement of the credential management.

5. Code Generation

5.2: Completely revised with description of ADL AST transformation. Added TBD for TDD and NLD AST transformation.

5.3.2: Revised comments processing section with arguments for not preserving comments in the generated code.

7. Runtime Architecture

Added section 7.8 describing how to work under the TET test harness and consequences for the ADLT system.

7.5: Fixed some examples

7.6: Added description of Java, C++ and C runtime internals.

7.7: Fixed result code names.

7.7.2: Fixed some typos.

7.7.4.3 Removed useless variables. Added TBD for design of ADL_FORK support.

Release 1.0 Gamma

1. Introduction

1.2.2: Relaxed constraint on extensibility of the ADLT system.

1.5: Reverted to using SWI Prolog Interpreter.

2. The Driver

Major simplification of the design of this module.

3. Abstract Syntax Tree Design

Updated to JavaCC version 0.7 pre 5 (signature of some methods changed).

4. Input Language Parsers

4.2.1: TBD resolved by adding a reference to JavaCC on-line documentation.

4.4: More detailed description of how to satisfy external references for the different bindings. Added TBD for complete list of semantic checks. Added TBD for processing of inlines and builtins in C and C++.

4.4.1: Added TBD for complete design of the symbol table.

4.4.2: Added TBD for use of the Type interface.

5. Code Generation

5.1: Added description of the Visitor pattern to be used for most operations on ASTs.

5.3: Complete revision of the design of the unparsing operation according to the Visitor pattern and the new version of JavaCC tools.

6. Documentation Generation Architecture

6.7: Added TBD for property names and types.

7. Runtime Architecture

Incorporated changes to the runtime system description to reflect a more TET-oriented direction for the tools.

Release 1.0 Beta

The document was completely rewritten for this release.

Release 1.0 Alpha

Initial release.

Chapter 1 Introduction

1.1 This Document

This document describes the architecture of the ADLT translation system, version 2. The architecture is described by a mixture of prose descriptions, diagrams (some using the Unified Modeling Language of Booch, Jacobson, and Rumbaugh)[1], code fragments, and design patterns as described in Alexander[2] and in Gamma, Helm, Johnson and Vlissides[3].

The purpose of the document is to enable the reader, a skilled programmer with knowledge of the Java programming language, to understand the ADLT system; to be able to read the code of the system, to find the relevant parts of the system when seeking the answer to a specific question, and perhaps even to program a new module to add to the system.

This document is a work in progress; until the ADLT2 system is finished, the information in this document is subject to revision.

1. Booch, G.; Jacobson, I.; and Rumbaugh, J.: "The Unified Modeling Language for Object-Oriented Development," Documentation Set Version 0.91 Addendum UML Update, September 1996.
2. Christopher Alexander, A Pattern Language, Oxford University Press, 1979. ISBN 0-19-501-919-9.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994. ISBN 0-201-63361-2.

1.2 Design Goals

The goal of the ADLT2 design is to build a code transformation engine out of separable parts.

1.2.1 Tasks

The task performed by ADLT is to translate one kind of program text—specification text of the various kinds of specifications—to another—source text for the targeted execution environment, and SGML source for the documentation.

The ADLT system also includes the runtime libraries and document templates required to make use of the generated code.

ADLT exists in several versions, one for each target language. Each of those versions is independent; it is *not* a requirement that specifications for the various target languages should interoperate in any way.

1.2.2 Constraints

ADLT2 must be at least as powerful as ADLT1; the proof of this is a processor that translates ADLT1 specifications to ADL/C ADLT2 specifications.

It must be possible to write a GUI for ADLT, although writing such a GUI is not part of the core deliverables.

1.2.3 Desiderata

Given the variety of specifications parsed, and the requirement that parties outside the development team be able to add modules, it is desirable to minimize the dependencies between modules of ADLT. In particular, the use of JavaBeans™-like introspection to discover processing modules at runtime will be convenient.

Sharing modules between target languages, or code between modules if they must differ, is desirable.

1.3 Overview

The core of ADLT2 is a language transformer. This transformer reads in programs in one programming language and writes programs in another, similar programming language. The particular transformations performed by the ADLT2 core engine are:

• Annotated Java	->	Java
• Annotated IDL/Java	->	Java
• Annotated C++	->	C++
• Annotated C	->	C
• Annotations	->	natural language documentation

The annotations allowed are described in the ADL language reference manuals for the various target languages. In short, they may be categorized as semantics annotations, which specify the behavior of methods and functions; test annotations, which specify the test procedure for functions, methods, and collections thereof; and documentation annotations, which affect the translation of other annotations into natural language documentation.

In addition to the core translation engine, the ADLT2 system includes a runtime system for controlling and reporting test results, and GUI tools for controlling the translation process, for managing test libraries, and for running tests. Integration between the parts of the translation engine is performed by a central control unit, and data is passed between the parts in a uniform format; this format also has an external representation so that it can be exchanged with external programs.

This document will give some details of the architecture of the ADLT2 system software, with emphasis on the core engine.

1.4 The Parts of ADLT

All of the transformations performed by the ADLT2 core engine may be described by the general pattern: src1 -> src2, where src1 and src2 are programs in (probably different) programming languages. This pattern of operation is implemented by the code pattern: parser->xform->unparser, where parser turns source text into an abstract syntax tree (AST), xform is a transform from one AST to another, and unparser generates source text from an AST.

The ASTs used in ADLT are *annotated*: the tree holds not only the strict syntax tree, which is a reflection of the grammar of the formal language represented, but also is annotated with contextual information like the type of variables, and with derived information like the natural language description of the variables.

The transform pattern is implemented by three modules: a parser module that reads source text and creates the AST, a transformer module that creates a new transformed AST based on the input AST, and an unparser that writes the source text corresponding to the transformed AST.

The action of these modules is coordinated by a driver module, which implements an internal model of the dependencies among the various files known to ADLT. That internal model is also used to generate parts of the test program, especially the make files that control test program compilation.

1.4.1 Parsers

There are four parsers required for ADLT2; one for each of the target languages. Each parser accepts the full range of annotations for the languages: semantic annotations, test annotations, and documentation annotations. The user may of course divide the annotations into separate files if so desired.

1.4.2 Modules

The heart of the work done by ADLT is in the AST transformers; parsing and unparsing are well-understood problems. The overall transformations required may be classified into transformers for three kinds of annotations on the basic source program:

- semantic annotations -> checking functions
- test annotations -> test drivers
- NL annotations -> SGML source

In fact these annotations are layered; the semantic annotations ornament the basic syntax of the target language; the test annotations ornament the semantics AST; the NL annotations ornament the test AST. The degree of interdependency is not constant; the semantic and test annotations depend heavily on the target language AST, while the dependency of the test tree on the semantic tree is minor; the NL tree depends on both the semantic and the test tree.

These relationships could be expressed using an interlaced hierarchy of classes. However, this would lead to a inflexible degree of interdependency between the data structures; derived types would have to know all details of the base type. ADLT1 was built using a shadow mechanism that freed us somewhat from this constraint. ADLT2 will be built with a general AST representation mechanism, which will be used to represent all the different ASTs, as instances of the same data type. The advantage is that the tree transformers can ignore details which do not affect them.

1.4.3 AST Representation

The central representation problem in the construction of ADLT2 is the representation of the ASTs. In general, AST transformers will need to match certain parts of the tree that they understand while leaving other parts untouched.

In addition, the ASTs for the several target languages will have a great deal of commonality, for example in the expression syntax, while not being identical. In order to efficiently support all four target languages, it will be necessary to maximize code re-use.

With these design criteria in mind, we have determined to use a common data type to represent all the various ASTs. The distinction between the different kinds of AST will not be reflected in a formal data type, but will be a difference in instantiation of a single data type.

1.5 Technology

ADLT2 will be built in the Java programming language. Our experience has been that Java programs are much faster to develop and much less error-prone than equivalent C++ programs, due to automatic memory management, to a consistent implementation, and to a sounder language design.

We will make use of two programing tools developed by members of the SunTest team: a parser generator, used to implement the various parsers required, and an AST generation and management tool. These tools replace one of the external tools used in the development of ADLT 1, which proved a source of portability and maintenance headaches.

ADLT2 will make use of the reflection features introduced into Java with the 1.1 release.

ADLT 2 will also make use of the SWI Prolog Interpreter. This freeware tool has proven to be very resilient and portable in recent years, and the natural language translation tools from ADL1 rely upon it. We will extend these tools, but will not move them away from their dependence upon this externally-developed tool.

Chapter 2 The Driver

2.1 Introduction

This chapter describes the general functionality of the ADL Translator. The Translator is intended to take a variety of different inputs and transform them, generating a variety of outputs. Each of these inputs and outputs, as well as the transformation mechanisms involved, are driven by a single driver program. This chapter describes how the driver program operates. No description of the transformation mechanisms involved is made here. Other chapters deal with some of these elements in more detail.

2.2 General Architecture

The diagram in Figure 1 depicts the general architecture of the system. In particular, it shows that although the ADLT system will be perceived by the user as a single tool, it is made of several compilers acting on specifications written in a specific language binding flavor. These compilers implement the transformation mechanisms. Actually, the single tool perceived by the user is the driver. This module allows the selection and operation of the appropriate compiler based on user's directions and the type of inputs. It implements the control of the ADLT system as described below.

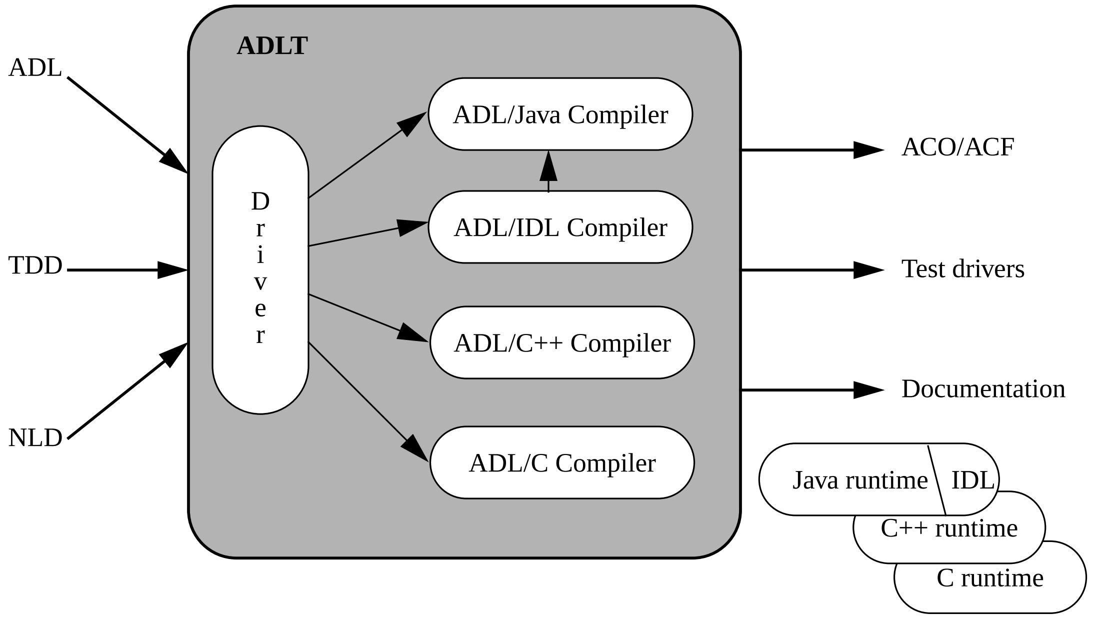


FIGURE 1. General Architecture of the ADL Translator System

Since the Driver module needs to access some environment variables it cannot be written in Java[1]. Or at least it has to be wrapped over by another program. For greater portability and simplicity this wrapper program is developed in C.

The general architecture of the Driver module is depicted below.

The C wrapper program `adlt`:

- reads in environment variables whose name matches "`ADL_<name>`" and converts them as Java properties "`adl.<lowercase name>`";
- extends the class path with the standard directories for ADLT;
- builds a Java command to invoke the main method of the `Adlt` class and executes it with `system()`.

The `Adlt` class:

- loads configuration files: system wide and local or user defined;
- parses command line options;

1. Starting from version 1.1 Java has deprecated the getenv() method which was used to access environment variables. Java developers are rather encouraged to use properties.

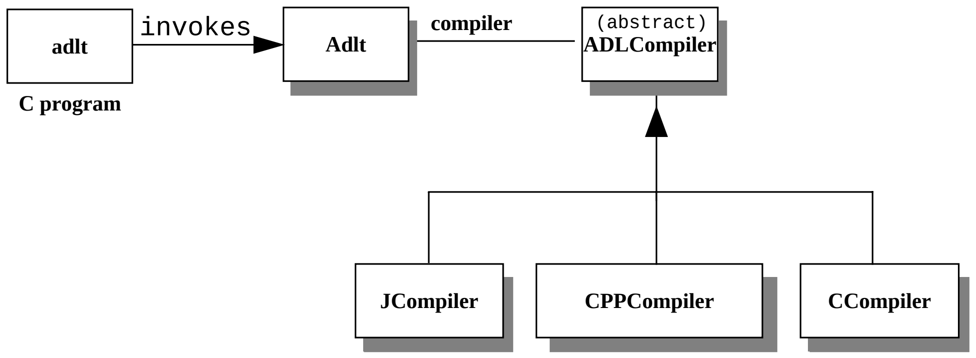


FIGURE 2. General architecture of the Driver module

- asks `ADLCompiler` to create a compiler based on the selected binding language;
- invokes the compile method of the compiler on each input file.

NLD files are compiled before any other file, and only when the documentation generation has been requested (see below).

It also provides other modules with a set of static methods to determine if verbose mode has been selected, and what kind of output has been requested by the user.

The `ADLCompiler` class implements an abstract factory which creates a compiler according to the selected binding language (method `newCompiler`). It also provides the language dependent compilers with a method named `getStream` which returns the input stream for a given file, possibly pre-processing the file through an extension filter.

The language dependent compilers (`JCompiler`, `CPPCompiler` and `CCompiler` classes):

- create all the required modules: parser, translators, unparser;
- compile the input file (method `compile`).

2.3 Control

The operation of ADLT is controlled in general by the command line, which is used to specify user's directions as well as inputs and outputs. More detailed control is available by the use of environment variables, which can be used to set the search path used when looking for a required input, to select the user's locale for generated documentation and to set error and warning options.

The general format for the command line is given below.

```
adlt [-v] [lang_opt] [doc_opt] [output_opts] [env_opts] \
          <input> ...
```

The order of the various options is not important.

The `-v` option allows to operate the ADLT in verbose mode.

The `lang_opt` indicates which specific compiler to use. Possible values are `-c`, `-cpp`, `-java` (default) and `-idl`.

The `doc_opt` option is used to turn on the generation of the documentation (which is off by default). The generated documentation uses the SGML format based on the DocBook DTD version 3.0 from the Davenport Group (http://www.ora.com/davenport). Other formats can be derived from the SGML format by applying DocBook filters (See “Documentation Generation Architecture” on page 97.). The only possible value for this option is `-doc`.

The `output_opts` indicate which kind of output to generate. Specifically, the suffix for each type of input is also used as the switch to request output of that type. In fact the suffix for an input determines the potential outputs that the ADLT can generate. The table below summarizes this first level of control.

Input Suffix	**Output Type**	**Output Option**
`.adl`	. ACO/ACF	`-aco | -acf`
`.tdd`	. Test case classes	`-tc`
	. Test data classes	`-td`
	. Makefile	`[ -dev ] -mk`
	. TET scenario	`[ -noinc ] -scen`

When no `output_opts` is specified, all possible outputs corresponding to the given inputs are generated. On the contrary, the `output_opts` can be used to specifically request one or more of the possible outputs corresponding to the given inputs. This is illustrated by the third column in the table above.`output_opts` can also take another value: `-nocode` which specifically requests the compiler to generate no code. This is useful when only the documentation has to be generated.

The `-dev` option is provided for the sake of test suite developers. It requests that the ALDT generates makefile dependency rules between a generated output and its corresponding input, like, e.g., dependency between an ACO and the ADL source file.

The `-noinc` option indicates that the generated scenario file will not be included in a test suite scenario file.

Files with suffix `.nld` can be passed as input to provide natural language definitions for the symbols described and used in the test specifications.

Finally, the `env_opts` may be used to override some settings defined in environment variables. These include the general ADLT configuration file, `-config <config file>`, the search path used to find included inputs, `-incpath <path>` option, the preferred locale for the user, `-locale <locale>` option. These options override the settings of their equivalent environment variables whose names are made of the option names capitalized with an “`ADL_`” prefix prepended, e.g., `ADL_INCPATH`.

Other environment variables may not be overridden: `ADL2HOME` which is mandatory and references the directory where the ADLT has been installed, `ADL_C_FILTER and ADL_CPP_FILTER` which are useful for the C and C++ language flavors to define the pre-processor command, `ADL_EXTENSION_FILTER` which defines the command to use as an extension filter to pre-parse the input files (like MRI’s extended TDD parser), and ADL_PROLOG which defines the Prolog interpreter to use if the user wants to supply its own instead of the one in `$ADL2HOME/bin`.

Two other environment variables defined for TET may be required for ADLT operation: `TET_ROOT` and `TET_SUITE_ROOT`. `TET_ROOT` references the directory where TET has been installed. `TET_SUITE_ROOT` defines the top level directory of the test suite to build if it is not below `TET_ROOT`.

The following examples are legal ADLT invocations:

```
adlt -c spec.adl
adlt -doc -nocode spec.adlj
```

The first one requests all possible output (the default) from the input “spec.adl” written using the C flavor. The second requests just the output of type documentation, in SGML format, for a java flavor of a specification file.

When no option is passed on the command line and no specific file suffix is used, the default values assumed are equivalent to the following command:

```
adlt -java -aco -tc -td -mk -scen -config adlt.rc
    <input> ...
```

where the file `adlt.rc` contains the default values for the other variables, just like the system wide configuration file `$ADL2HOME/Adlt.rc` which is systematically loaded before any other configuration settings and which contains definitions like:

```
adl.incpath=.:..
adl.c_filter=gcc -E -xc -
adl.cpp_filter=gcc -E -xc++ -
```

2.4 Exit Status

The ADLT compiler indicates the result of its operation to the environment with an exit status. The following values are used to indicate the result of the compilation:

Value	Status
0	No compilation error
1	Documentation generation error
2	Compilation or code generation error
10	No ADL2HOME
11	Incorrect command line
12	Unknown or unreadable file
100	Internal compiler error (program error)

Chapter 3 Abstract Syntax Tree Design

3.1 Introduction

This section describes the abstract syntax tree (AST) mechanisms of ADLT2.

The AST is a central data structure in ADLT2. It is constructed from a set of nodes linked together in interesting ways. The content of the nodes is represented as properties.

The implementation of the AST relies upon certain Java 1.1 features, specifically class reflection for accessing certain properties, and serializability for persistent storage and externalization.

3.1.1 Tree construction

The inputs to ADLT2 are specifications of the interface under test. The specifications are written in a combination of ADL, TDD, NLD and target language expressions. A parser takes the specification and constructs an initial tree representing the input.

To begin with only the structure of the input is represented in the AST. Trees for other representations and properties derived from the interactions of the trees are added as ADLT2 processing continues.

The AST is a collection of data that is shared between various clients. For example, the code generation is an AST client that attach properties to the AST nodes and build new nodes to represent information computed from them.

The AST is designed as data rather than a set of procedures because there are several different kinds of operations that must be performed on it. It is better to keep those actions in external objects rather than in methods on the AST objects so that the AST implementation can remain stable as the actions are developed. This scheme is described by Gamma & Al. in their book Design Patterns as the Visitor Pattern.

3.1.2 Nodes

Nodes are linked together to represent the relationships between them. The information in a node, and the links it has to other nodes are represented as node properties.

Nodes represent syntactic entities in the input and output languages. For example, there are AST nodes to represent input language features such as assertions, test directives, and declarations; and there are nodes for output features such as loops, declarations and block statements.

One important kind of link between nodes is the parent-child relationship. Every node, except the root node has a parent node accessible via `jjtGetParent()` method. Every node has an array of child nodes accessible via `jjtGetChild()` method. The children in this array are part of the fundamental syntax structure of the node and are added as part of the JJTree tree generation process (see below). The chidren array is `null` for leaf nodes.

3.1.3 Properties

Each node in the AST contains a set of properties. A property is a per-node mapping from one Java `Object` (the key) to another `Object` (the value). It is a general mechanism that can be used to represent a link between arbitrary objects. The key is often a string interpreted as the name of the property.

Some properties are provided by all nodes while other properties that a particular node provides are determined by the type of the node and by the data it is representing. For example, all nodes have a “parent’ property, but not all nodes have a “type” property. Each piece of interesting information about a node is available as one of its properties. Some information can also be accessed via native Java mechanisms where convenient or where performance is critical.

There are three basic kinds of property distinguished by their implementation scheme: general table properties are stored in an association table, field properties are an interface to the node object’s fields; and lazy properties are an interface to some of the node object’s methods. A table entry for a property is always preferred over a method or field representation. It is an error to have both a method and a field representation for the same property.

Lazy properties provide a mechanism for parts of the AST to be produced in a demand-driven, lazily evaluated fashion. Such getter methods should be functions, and return the same property value every time they are called.

Some properties are indexable. They are a collection of values that can be accessed by an integer index. An indexable property can also be accessed as a single value, in which case it is represented as an array.

3.2 Requirements

This section describes the requirements that the AST design must meet. It is organized as a set of fundamental AST operations and overall features.

Despite its name, the AST is really a directed graph and not a tree. There are links between nodes in both forward and backward directions (for example, the parent-child link pair). However, for reasons of tradition we'll persist in referring to it as a tree.

The graph is a forest of trees that partially share subtrees. Subtrees are shared because we have several versions of and uses for a subtree, and these versions have more commonalities than differences. In particular, properties of nodes are shared between trees of different structure.

As a design pattern, we treat ASTs as monotonic. This is so several trees can share structure without danger. Most properties are additive and consequently they are added to a node but should not be subtracted nor modified (unless we can prove that no one will be able to tell that we have changed them). Trees are altered by copying and then modifying nodes that must have their properties changed rather than altering the nodes in place.

3.2.1 Basic operations

There is a small number of fundamental operations that the AST must support. More complex tree manipulations can be built out of these basic actions.

3.2.1.1 Node construction

Some nodes are created as the source parser reads the specification and builds the initial AST, and other nodes are created during the transformation phases of ADLT processing.

3.2.1.2 Node copying

Copying a node N from an existent node C means that all properties of C that are not already properties of N have their current values added as properties of N. This means that lazy properties have their values frozen in the new node. This should be safe because lazy properties are supposed to be implemented by function methods.

3.2.1.3 Adding and retrieving properties

Any Java object can be added as the value of a property of a node. Any Java object can be used as the key for a property. A common case of this is objects looked up by a name represented by a Java `String` object.

Lazy properties have their values evaluated on demand when they are required. An AST client can test for the existence of a property and retrieve the value of a property. A list of all the currently defined properties of a node can also be obtained. The list is represented by an `Enumeration`.

3.2.1.4 Delegation

A node N can delegate to another node D for getting property values. This means that if N does not have a local value for a requested property it will attempt to get the value from D. D itself can delegate to another node for the property. Delegation is only performed for getting the value of a property. An attempt to set the value of a delegated property is equivalent to setting a local table property. The new property value is set in N, and does not change the value of the property in D or its own delegates. This local property will now hide the delegated property.

A node delegates to another node by setting its `Delegate` property to the desired delegated node.

Delegation is not currently required. It is therefore not implemented.

3.2.1.5 Externalize

Several external representations of the AST are required. The most complete is provided by the Java 1.1 serialization interface which allows a reader to reconstruct the entire tree. Less complete external formats include reporting information for the runtime, debugging output, and skeleton localization tables.

Nodes implement the Java 1.1 `Serializable` interface. There are no `transient` fields, and no `readObject` or `writeObject` methods; the default system implementations are appropriate. Later evolution of the node design might require these methods to be customized, or for the `Externalizable` interface to be implemented.

The Java 1.1 serialization provides a mechanism for persistent storage of the AST as well.

Externilization is not currently required. It is therefore not implemented.

3.3 Implementation

3.3.1 JavaCC and JJTree

The parsers for the input languages are built using the tools JavaCC and JJTree. JavaCC is a parser generator that produces a top down recursive descent parser from a grammar. JJTree is a companion tool that takes a JavaCC grammar annotated with node information and produces another grammar decorated with actions to build a parse tree from the nodes.

JJTree node objects must implement a simple interface that allows child nodes to be attached to parent nodes. The actions that JJTree inserts into the grammar use this interface to create nodes and to link them together to form the initial parse tree.

3.3.2 Nodes

In addition to the requirements that JJTree places on nodes, they must implement the functionality described in the Requirements section above.

Each node is represented as a Java class object, of a type that inherits from the `SimpleNode` type, which itself inherits from the `ADLNode` type. Distinct node types are used as a means to providing methods to implement lazy properties and field properties, and for type-specific operations such as externalization and unparsing.

Here is a description of the methods on `SimpleNode` objects:

3.3.2.1 JJTree methods

The following methods must be implemented by nodes designed to work with JJTree. Reasonable default implementations are provided in the `SimpleNode` supertype. Often the only methods that need to be customized for a particular AST node class are `jjtClose`() and `jjtCreate`().

The JJTree methods are intended to be called only by JJTree actions. The main exception to this is `jjtGetChild`() which is often called within the implementation of `jjtClose`() so that the child node can be added as a more appropriately named property.

`static Node jjtCreate(int id)`

The actions inserted in the grammar by JJTree call this factory method to create a new node. Its argument is an identifier that was specified in the grammar to indicate what kind of node is required. The method can use this identifier to determine the Java type for the node. Static methods are not inherited in Java, so every AST node type must provide its own implementation of this method. Often this can be as simple as a method that only calls `super(id).`

`void jjtOpen()`

This method is called by JJTree actions to indicate that children can now be added to the node. It provides a mechanism to support nodes that need to make special preparations before their children can be added.

`void jjtClose()`

This method is called once all of the node's children have been added by JJTree. It is an opportunity for the node to store the children in a more convenient way than the default scheme, and to compute information derived from the children.

`void jjtSetParent(Node p)`

`Node jjtGetParent()`

These two methods are used to notify the node of its parent, and provide a programmatic way to find out a node's parent. The node's parent is also available via the `Parent` property.

`void jjtAddChild(Node c, int i)`

The actions inserted by JJTree call this method to add children to a node. It will be called once for each child, after the `jjtOpen`() and before the `jjtClose`() methods have been called. It is the node's responsibility to store the child for later access.

`Node jjtGetChild(int i)`

This method is a programmatic interface for accessing the children of a node. The children are indexed from left to right, starting from zero.

```
int jjtGetNumChildren()
```

This method returns the number of children that JJTree has added to the node.

3.3.2.2 Basic operations

```
SimpleNode(int i)
```

Each node type must implement a constructor with a single integer argument to create an empty node. The new node's association table will be empty, and its field properties will be set to their initial values. The node's `jjtCreate`() method will call this constructor to build the node, passing the node id as argument.

```
void copyFrom(ADLNode source)
```

This method can be used to make a copy of the source node's properties. If a property exists in the source node but does not exist in the current node it is copied. A new property is created and added to the current node's association table with the current value of the source node's property as its value.

```
Object getProperty(Object key) throws
PropertyAccessException
```

Get the value of the specified property. If necessary, follow the delegation chain to retrieve the value. It is an error to attempt to get the value of a nonexistent property.

```
void setProperty(Object key, Object value) throws
PropertyAccessException
```

Set the specified property to the specified value. If the property does not exist it is created and added to the current node's association table, otherwise the property is updated to the new value.

An exception to this is if a lazy property has a setter method, in which case it can be set more than once.

```
Object getProperty(Object key, int i) throws
PropertyAccessException, ArrayIndexOutOfBoundsException
```

```
void setProperty(Object key, int i, Object value) throws
PropertyAccessException, ArrayIndexOutOfBoundsException
```

Get or set the value of the indexed property. It is an error if the property is nonexistent or cannot be indexed.

```
boolean hasProperty(Object key)
```

Test to determine whether the property already exists on this node, following the delegation chain if necessary.

```
void removeProperty(Object key) throws
NoSuchPropertyException
```

Remove the specified property from the node. Any kind of property can be removed regardless of whether it is represented as a table entry, a node field or method, or a delegated property. Be careful.

```
Enumeration getKeys()
```

Return an enumeration that produces the keys of all the properties of the node.

```
void dump(OutputStream o, int level)
```

Produce a textual representation of the nodes and properties of the AST. The level argument controls the amount of detail in the output. A level of `Integer.MAX_VALUE` produces everything.

3.3.2.3 Properties

Node properties are implemented as a combination of an association table, class method calls and field accesses. The association table is always the first place that a property is looked for. If it is not found there the Java core API reflection mechanism is used to determine whether there are accessor methods for the property. Finally, reflection is used to determine whether the node has a field that can be used to access the property value.

The Java Beans introspection API method naming conventions are used for property getter and setter methods. The Java Beans naming convention is as follows. Suppose a node has a property with name `MyProp` and whose value is of type `MyPropType`. A getter method on the node can be defined to retrieve the value of this property like this:

```
MyPropType getMyProp() {...}
```

Similarly a setter can be defined like this:

```
void setMyProp(MyPropType p) {...}
```

Alternatively the value of the property can be directly accessed via a field on the node:

```
MyPropType myProp;
```

Java Beans indexable properties are also supported. The naming conventions for indexable getter and setter properties are:

```
MyPropType getMyProp(int index) {...}

void setMyProp(int index, MyPropType p) {...}
```

Private methods and fields are not considered as implementations for properties.

Chapter 4 Input Language Parsers

4.1 Parsing

The first task of ADLT is to parse the input specification files. The parsers used in ADLT are generated automatically from the grammars for the input languages; the generated parsers construct the basic structure of the required Abstract Syntax Trees.

After construction, the ASTs are checked for semantic consistency; during this processing, references are resolved. After this semantic check, the AST and the parse are complete.

4.1.1 Parser Generation

The parsers are built using the Java™ Compiler Compiler™ parser generator, developed by a member of the ADL team at Sunlabs. JavaCC is a recursive-descent parser generator, similar in spirit to the PCCTS system that was used for ADLT1; however, JavaCC is written in Java and generates a Java program.

One characteristic of recursive-descent parsers is that, in contrast to the tables of an LALR parsers, the parser code is fairly close to the parser source, and fairly easy to read. A consequence is that maintenance of the generated code is easier; it's easier to trace the generated source code back to the input grammar, and easier to read and debug the generated code.

4.1.2 AST Construction

The Abstract Syntax Trees required by ADLT are constructed using the tree-building facilities of JavaCC. These facilities are implemented as a separate grammar pre-processor called `jjtree`. By default, `jjtree` constructs an AST that contains all the non-terminals in the grammar. The input grammar can be annotated to leave out some non-terminals, to specify the class used to represent each node in the AST, and to add non-terminal information; all those annotations are used in ADLT AST construction.

For details of the `jjtree` annotations, refer to the JavaCC documentation set (http://www.suntest.com/JavaCC/DOC/).

4.1.3 Extension Parser

One objective of the ADL translation system is to allow the other partners in the project to develop some extensions to the ADL and TDD languages. Instead of plugging extension parsers/translators into the ADLT compiler, it has been decided to provide extension developers with a mean to pre-process files using the extended syntax before they are fed to the ADLT compiler. Although this solution can only be applied if the extended syntax can be converted to the regular one, it has been found sufficient for the ADL project goals.

The implementation of this solution is based on a system property `adl.extension_filter` (See "The Driver" on page 25.) which defines the command to use to pre-process the input file. This command should generate on its standard output a stream of regular ADL or TDD definitions. The output of the extension filter is passed to the parser to be used as its input stream.

This is implemented by the `DataInputStream getStream(String fileName)` method of the language independent compiler (`ADLCompiler` class). It creates the filter process and get its output stream. If the adl.extension_filter property is not defined, it just returns a `DataInputStream` built from the `FileInputStream` of the file. It the extension filter command fails an error is reported and the compiler aborts its execution.

4.2 Grammars

There are four versions of ADLT, one for each target language. The specification language for each target programming language is described by one grammar; those specification languages are defined in these separate documents:

- *ADL 2.0 for C Language Reference Manual*
- *ADL 2.0 for C++ Language Reference Manual*
- *ADL 2.0 for Java Language Reference Manual*
- *ADL 2.0 for IDL Language Reference Manual*

These documents contain the text of the grammars, as well as explanations of the specification constructs.

4.3 Semantic Checking

The semantic check of an AST is largely symbol table work; linking variable references to variable definitions, associating types with variables and checking that expressions have consistent types, satisfying external references.

The semantic check also needs to verify that the parsed AST is correct according to the ADL semantics, like, e.g., verifying that call state operator and unchanged aren't nested.

For the C and C++ bindings, satisfying external references may require a recursive parsing process, to parse the imported files. The idea is however to rather use the C pre-processor to include all the referenced files and provide the input to the parser. This should allow the compiler to search for external references in its symbol table. The pre-processing implementation is detailed in Section 4.3.1 below.

For the Java binding, resolving external references require that the compiler is able to get information about a referenced class from its Java bytecode. Using the Java 1.1 reflection API to do so would be rather risky since there may be confusion between the classes used by the ADL compiler and those referenced by the parsed files. Instead this will be implemented by reading the constant pool in the Java bytecode for the class.

The results of semantic checking are recorded as properties in the AST.

Section 4.3.3 will present what is to be done with *types* (type synthesis, type checking...), Section 4.3.4 with *names* (scope rules, symbol tables management...), and Section 4.3.2 with miscellaneous syntactic checks.

4.3.1 C/C++ pre-processing

The solution retained to import C/C++ external declarations into the parsed ADL / TDD file is based on the C/C++ pre-processor and its `#include` directive. The system property `adl.filter` defines the pre-processing command. The C/C++ dependent compilers read this system property and create a process to pre-process the source file. The output stream of the process is the parser input stream.

The parser uses the standard `#<line>` directives of the C pre-processor to update its current file name and line number used for compiler error messages.

The parser finally operates in 2 different modes:

- Include Mode: When parsing included C++ files. In this case, parser acts in declarative mode. Its major action is to extend the symbol table with externally defined symbols, hence no semantic checking is performed in this mode. When the included file is completely parsed, all AST nodes created within the Include Mode are cleared and replaced by an IncludeFileDeclaration node in which is saved the included file name. This allows to generate an AST corresponding to the input file before the pre-processing.
- ADL / TDD mode: This the normal mode in which all semantic and type checks are done. All necessary nodes are created and passed to the translator.

Below is an example of the pre-processor output where comments have been added to illustrate how the parser switches between its 2 different operating modes.

```
# 1 “bank.adl”
     // First pre-processor output line
     // Used to set currentAdlFile lexer variable

# 1 “bank.hh” 1
     // Corresponds to “#include <bank.hh>” in bank.adl
     // Lexer send a SHARP_INCLUDE_FILE token to the parser
     // Token image is the included file name “bank.hh”
     // Parser enters IncludeMode

# 1 “bankAccount.hh” 1
     // Corresponds to “#include <bankAccount.hh>” in
     // bank.hh
     // Nested included file => no token send to parser
     // Current file name and line number are updated

//.. bankAccount.hh code

class bankAccount {
//...
}

# 25 “bank.hh” 1
     // bankAccount.hh completely included
     // back to bank.hh

// ... bank.hh specific code here

# 24 “bank.adl” 2
     // bank.hh completely included.
     // back to bank.adl file.
     // END_INC token is sent to parser
     // Parser exits IncludeMode

//... bank.adl code
```

All pre-processor output directives other than the `#<line>` ones are ignored. This is implemented in the C/C++ lexer which skips those directives, and thus, never sends them to the parser. Ignored directives are generally `#pragma`, `#inline` and `#builtin`. However, the last two directives seem to have now disappeared from the output of most pre-processors.

4.3.2 Semantic Checking

In this section are listed the semantic checks that are common to all the languages bindings (except checks related to object oriented paradigm that are not present in the C binding). In forthcoming sub-sections are listed the checks specific to each binding.

Preliminary definition: an assertion group is *nested* if it appears in an *expression* (either in an assertion, a binding definition or a normal/abnormal specification). An assertion is *nested* if it appears in a nested assertion group.

All kinds of checks that deal with names or types will be detailed in next sections.

- In ADL, the inherited adl class *source* file (specified after keyword `extends`) must be accessible from the incpath (See “Control” on page 27.) and readable.
- An adl class and all its behavior declarations are implicitly considered as `public` (this modifier can be explicitly added but is redundant; any other access modifier is not correct).
- In the behavior description of a *constructor*, it is not possible to use call-state or `unchanged` expressions, nor the `return` expression. The `this` expression refers to the object built by the tested constructor.
- The `super.semantics` clause may appear only in the outermost assertion group (after the keyword `semantics` or the potential normal/abnormal definitions).
- `abnormal` (resp. `normal`) must be defined at most once.
- `abnormal`, `normal` and `return` cannot be used in the scope of a call-state expression or in a binding definition.
- `normal` (resp. `abnormal`) cannot be used in the definition of `normal` (resp. `abnormal`).
- A call-state expression or an `unchanged` expression cannot be used within an `unchanged` expression.
- A call-state expression or an `unchanged` expression cannot be used within a prologue or an epilogue.
- A nested assertion group cannot be used within a prologue or an epilogue.
- An `unchanged` expression cannot be used within a call-state expression.
- A free variable of a quantified assertion cannot be used in the scope of a call-state expression.
- A quantified assertion cannot be used inside a nested assertion group.
- The clause `super.semantics` cannot be used when the adl class does not inherit from an other adl class (`extends` clause).
- The behavior description where a `super.semantics` clause is used must be describing a method that overrides a method that was described in an adl class the current adl class is extending (directly or not).
  Note: this check will not be implemented in the first release.
- The name, signature, return type and throws clause of the currently described method must be *exactly* the same as one *public* method of the class under test.
- An expression that invokes an inline must be parsable when replacing the inline invocation by the body of the inline definition.
- A binding cannot be defined inside an inline definition.
- A binding cannot be defined inside a nested assertion group.

- A tdd class and all its datasets and factories are implicitly considered as `public` (this modifier can be explicitly added but is redundant; any other access modifier is not correct).
- A tested method or constructor (invoked through `ADL()` or `ADL_new`) is effectively defined in an accessible adl class.

Warnings

- If `abnormal` or `normal` are used without being initialized, they will be given a default value (see the different language reference manuals).
- A call-state operator used in the scope of a call-state expression or of an `unchanged` expression will be ignored.
- Each method invoked in the scope of a call-state expression should be in a `try` block such that any exception of the throws clause of this method would be caught in a `catch` block (the `try/catch` blocks being inside the call-state expression).
- When a global prologue and/or epilogue is defined and a constructor is described, the global prologue/epilogue block will not be copied in the generated code for the constructor.

4.3.2.1 Java Specific Semantic Checking

- Any constant declared in a tdd class is implicitly considered as `private, static` and `final` (these modifiers can be explicitly added but are redundant; any other modifier is not correct). Their initialization value must be computable at compile-time.
- In ADL, the class under test (specified after keywords `adlclass`) must be a public, accessible (from the runtime classpath) and readable Java class.
- In TDD, each tdd class *source* file specified in a `use` clause must be accessible (from the incpath) and readable.
- In ADL, any exception named in a `thrown` expression must be either part of the `throws` clause of the currently described method, or a `RuntimeException`, or a superclass of one of these exceptions. Its class must be accessible.

4.3.2.2 C++ Specific Semantic Checking

- Any constant declared in a tdd class is implicitly constant `const`. All these variables must be initialized with values computable at compilation time.
- All used symbols must be visible: external declarations must be imported with `#include` directives. Included files are searched in all directories specified in the incpath property (See “Control” on page 27.).
- The method under test must be a public method of the C++ class. It can also be a constructor.
- Nested classes can not be annotated in C++. Scope override can not be used while specifying adl class name or in specifying adl super class names in call for super class semantics.
- Multiple inheritance is supported. Calling the super class annotation is done using the syntax : `<SuperClassName>.semantics`. The super class name should already have been defined as a super class of tested adl class.

4.3.3 Type Checking

Type checking consists in ensuring that everywhere an expression of a type T is awaited from the language specification, the actual expression written by the user has a type T' that is *compatible* with T (type compatibility will be presented in Section 4.3.3.2). Typically, if "`int i;`" was declared, then in assignment "`i=<expr>;`" this expression's type must be compatible with `int`, namely `byte`, `short`, `char` or `int`.

In other words: a type T is compatible with type T' if an expression of type T can be assigned to a variable of type T', i.e. T' is *assignable* from T.

4.3.3.1 Contexts

Type checking is necessary in these contexts:

- Assignment: the rhs expression type must be compatible with the declared type of the lhs variable.
- Method/inline/factory invocation: each actual parameter type must be compatible with its corresponding formal parameter type.
- Test directives: for each dataset that is associated to a parameter, the type of this dataset's elements must be compatible with the declared type of the parameter.
- Method/factory return type: the type of the returned expression must be compatible with the declared return-type.
- Inline definition: the type of the assertion group of an inline definition must be compatible with the declared return-type of this inline.
- Binding definition: the type of the binding expression must be compatible with the declared type of the binding
- Dataset declaration: the type of the rhs (right-hand side) dataset expression must be compatible with the type of the declared lhs variable.
- Dataset domain: the type of the rhs dataset expression must be compatible with the type of the declared lhs parameter.
- ADL `try/catch` feature: the type of a `catch` assertion group must be compatible with the type of the corresponding `try` assertion group.
- `relinquish` clause: the declared type of the parameter must be *exactly* the declared type of the corresponding factory.

4.3.3.2 Type Compatibility

C++ type compatibility is implemented according the ANSI C++ Public Review Document, check this document for further details.

In Java, we will distinguish three distinct sets of types: boolean (which is a singleton), arithmetic types and reference types. In each of these families can be defined a partial order.

The partial order for arithmetic types is as follows:

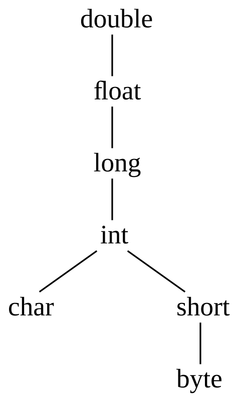


FIGURE 3. Arithmetic types

A skeleton of the partial order for reference types is as follows (suppose C is a class, S its superclass, I an interface that C implements, J a superinterface of I, A an array of a primitive type elements, A[C] an array of class C elements):

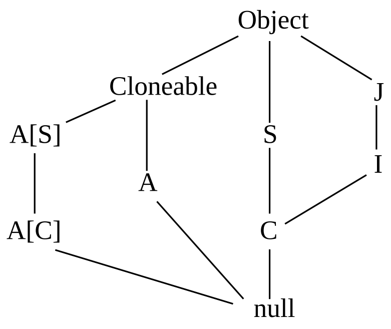


FIGURE 4. Reference Types

This diagram must be read as follows: any class is less than or equal to its superclass or the interface it implements, any array is less than or equal to the interface “Cloneable”, etc. By transitivity, the base type Object is greater than any other reference type.

We can now set two definitions:

- A type T is *compatible* with a type T’ iff T is less than or equal to T’.
- The upper bound of two types T and T’ is the least type that is greater than or equal to both T and T’.

4.3.3.3 Type Synthesis

The type of an expression is computed as follows:

- `normal`, `abnormal`, `unchanged(<expr>)` and `thrown(<name list>)` are boolean expressions.
- The type of the `return` expression in an ADL behavior description is the return-type of the currently described method in the class under test.
- The type of the call-state expression is the type of the enclosed expression.
- An type of an assertion must be boolean, except if:
  1. This is not a quantified assertion.
  2. This is the unique assertion of its enclosing assertion group.
  3. This assertion group is a `try` block *or* a `catch` block *or* an `inline` block.
  4. This assertion group is in the scope of an expression.

  In this case, the type of the assertion is the type of the enclosed expression.
- The type of an assertion group that contains a single assertion is the type of this assertion.
- If an assertion group contains at least 2 assertions, then all its contained assertions must be boolean, and in this case the assertion group is boolean.

- If a dataset member is defined as a range (`<expr> .. <expr>`), both expressions must be integral expressions and the type of the dataset member is the upper bound of these types. Otherwise (not a range), the type of the dataset member is the type of its enclosed expression.
- The type of an empty dataset literal is `null`, otherwise it is the upper bound of the types of its dataset members (see Section 4.3.3.2).
- The type of a dataset expression is defined as follows: If the dataset expression is a dataset variable, then its type is the declared type of this dataset variable.
  1. Else if the dataset expression is a dataset literal, its type is the type of this literal.
  2. Else if the dataset expression is a factory invocation, then its type is the return-type of this factory.
  3. Else if the dataset expression is a basic expression (literal, method invocation or constant), then its type is the type of this expression.
  4. Else if the dataset expression is a dataset concatenation (operator "+"), then its type is the upper bound of the dataset operands.
  5. Else the dataset expression is not correct and a compile-time error results. *Important note: it is not possible to perform a "." (dot) access operation on a dataset expression.*

- The type of a variable expression (resp. a method/factory/inline invocation) is the declared type of this variable (resp. the declared return-type of this method/factory/invocation)
- The type of an explicit cast expression is the named cast type.
  Important note: in ADL or TDD, explicit casts will not be checked. When a user

writes an expression `(MyType)<expr>` the ADLT compiler will not verify that the type of <expr> can be actually casted to type MyType.

- For expressions involving operations, the “classical” rules of the target language will be used: type check of the operands (boolean for logical operations, arithmetic for arithmetic operations, etc.) and numeric promotion (for instance when an addition involves a `long` and a `float` operands, the `long` operand will be “promoted” to `float` and the result-type of the operation is `float`).

4.3.3.4 Type Representation

A data type is represented by a subtype of the TypedNode abstract class. That class is modeled on the representation of type in ADLT 1, the class named ADLTP; in particular, it has these methods:

```
abstract class TypedNode extends SimpleNode {

  /** Returns the defining node of the type, or null. */
  public SimpleNode defn();

  /** Returns the name of this type (if any), or null.  *
  public String getTypeName();

  /** Returns the LType symbol for this type. */
  public LType getType();

};
```

This class is used in conjunction with class LType which is a more generic class, used to represent target language types (both builtin-types and constructed types) in the symbol table. In particular it has these methods :

```
class LType {

  /** Is this type the same type as t? */
  public boolean equals(LType t);

  /** Can a value of type t be assigned to a variable
     * of this type (i.e. is type t compatible with
     * this type)? */
  public boolean assignable(LType t);

  /** Returns a descriptive string representation of the
      * type. */
  public String getName();

};
```

EXAMPLE 1 Interface Type definition

As an example, here is how type check is performed for an assignement node.

```
// here node is an Assignment node made up of two expression
// child nodes
TypedNode leftNode = node.jjtGetChild(0);
TypedNode rightNode =  node.jjtGetChild(1);
if ( leftNode.getType().assignable(rightNode.getType()) )
{
     // type check is ok, set Assignment type to that of
     // left hand expression
     node.setType(leftNode.getType();
}
else
{
     // report a type check error
}
```

4.3.4 Names

4.3.4.1 Scope Rules and Name Checking

- The scope of a binding begins *after* its declaration and goes to the end of the enclosing assertion group.
- The scope of a variable defined in a global prologue begins after its declaration and goes to the end of the compilation unit. The scope of a variable defined in a local prologue begins after its declaration and goes to the end of the behavior declaration.
- The scope of a variable defined in an epilogue (global or local) begins after its declaration and goes to the end of this epilogue.
- The scope of an inline is the complete adl class in which it is declared.
- It is not permitted to define a local variable, a parameter or a binding with the same name and scopes with non-null intersection.
- A binding cannot be used in the scope of a call-state expression, in a prologue or in the global epilogue.
- An inline cannot be invoked in the definition of another inline, in a prologue or in an epilogue.
- An inline cannot have the same name as a method of the class under test.
- It is not permitted to define in the same adl class two different inlines with the same name *and* the same signature.

- The scope of datasets and factories is the entire declaration of the tdd class in which they are declared and of all tdd classes that import (“use”) this tdd class.
- A dataset expression cannot be used outside a dataset definition, a test directive or a parameter in a factory invocation.
- It is not permitted to define in the same tdd class two different datasets with the same name.
- It is not permitted to define in the same tdd class two different factories with the same name.

- It is not permitted to define in the same tdd class a dataset and a constant with the same name.
- A dataset definition cannot be recursive (directly or not, i.e. by mutual recursion with one or more other dataset definitions).
- It is not permitted to define in the same tdd class a factory and a method with the same name and the same signature.

4.3.5 The TypeCheck Visitor

Type checking is performed by the ADLTypeCheckVisitor class. This visitor visits each AST node for which type check is relevant (assignments, arithmetic and logical operation, method calls...) and performs type check and type synthesis. A helper class, the ADLTypeCheckHelper, helps the type check visitor during this process, by providing useful methods for type synthezis (`synthetizeTypes()`) and other type checks (`checkTypesAreAssignable()`and `checkTypesAreSameKind()`).

4.4 Symbol Tables

This section describes the symbol table management system used by the ADL compilers. A HTML document generated by javadoc and describing the classes involved is available.

4.4.1 Concepts

The main concepts used for the symbol table management system are introduced in this section.

4.4.1.1 Symbol Table Manager

The symbol table manager is the main interface the client parser shall use to manage its symbols. The symbol table manager is responsible for:

- allowing the client parser to put and retrieve symbols in and from the symbol table,
- storing a reference to the current scope being parsed,
- managing the client's requests to open or close a new scope (e.g. when entering into a struct or class definition or into a new block of code),
- creating new symbols (abstract factory role).

4.4.1.2 Symbol

A symbol is any identifier that may be encountered while parsing an ADL source files. Amongst symbols are identifiers for types, classes, methods or attributes. The main attribute of a symbol is its name.

4.4.1.3 Scope & Anonymous Scope

A scope defines a unit of visibility for a group of symbols. Such things as a block of code, a class or a package define scopes.

Scopes may be nested within each others, like a block of code inside another one, or a class inside a package. For this reason scopes are organized in a parent-childs tree structure. In some cases this organization may be refined and some scopes may have multiple parent scopes (e.g. a class scope may be related to a package scope as well as to one or several superclass scopes in a child parent relationship).

Some scopes are closely related to symbols, e.g. a class may be considered as a symbol inside a package scope but it also defines a scope. For this reason a scope may be attached to a symbol. Note that some symbols are indirectly related to scopes (e.g. a reference foo to an object of class Bar which definition is class Bar { int x; int y; } is indirectly related to the scope of Bar so that foo.x as well as Bar.x are valid scoped names).

Some scopes are not related to symbols, e.g. a code block scope. These scopes are said to be anonymous.

4.4.1.4 Language Independent Symbols

Since target languages share the same kind of symbols (type, variables, classes...) language independent symbols have been defined to promote code reuse within the ADL parsers.

Language independent symbols are abstract classes for symbols likely to be used within all ADL bindings. Specific behaviour is implemented in language dependent subclasses of the language independent symbol classes. But this is hidden to the symbol table manager client (the parser).

The language independent symbols ar organized as follows:

- target language independent symbols (the L... hierarchy)
- ADL language independent symbols (the ADL... hierarchy)
- TDD language independent symbols (the TDD... hierarchy)

all of these symbols implements the symbol interface, or the scope interface or both.

4.4.1.5 Scoped Name

Scoped names are composite names to identify symbols, each component of the name but the last identifying a scope and the last one identifying a symbol (e.g. java.util.Hashtable, CORBA::ORB). The use of scoped names inside the ADL source files led to the tree structure organization for the symbol table manager.

4.4.1.6 Symbol Listeners

Some symbols can appear in ADL code before they have been defined (e.g. Java methods). In such case, the client parser may supply a SymbolListener object when trying to get a symbol from the symbol table manager. This symbol listener is stored by the symbol table manager, and as the definition of the requested symbol becomes available (or at the end of the parsing), this symbol listener is used to call the client parser back so that it can finish its set up.

Note that symbol listeners are directly inspired from Java AWT1.1 event model.

4.4.1.7 Predicates

Predicates are used to selects symbols in the symbol table, when a same name is shared by several symbol (e.g. to select among different overload of a same method a MethodComparator predicate shall be supplied).

Predicates are supplied to the symbol table manager in addition to a scoped name when symbol selection is necessary.

4.4.1.8 Credentials

Credentials are granted to scopes and enable them to states they have visibility on some other scopes. Credential are for internal use within the symbol table manager.

As an example, a subclass in a Java Package may see protected symbols in its superclasses, and package visibility classes in its own package.

4.4.2 Package organization

The symbol table manager is made of a set of packages:

- the org.opengroup.adl.symboltable package which is a package containing generic components and that defines the interfaces and abstract classes used by all the ADL compilers,
- the compiler specific packages that are used by each specific ADL compiler. These packages are org.opengroup.adl.symboltable.c (for ADL/C and ADL/C++ compilers) and org.opengroup.adl.symbol_table.j (for ADL/java and ADL/IDL compilers).

4.4.3 Generic org.opengroup.adl.symboltable package design

The symbol_table package defines the interfaces and abstract classes that are used by all the ADL compilers to manage symbols.

The class design follows the Composite design pattern (see [Gamma]), in that classes defined to handle symbols and scopes may be composite or not. This depend on the interface the class implements, it may:

- implement only the Symbol interface - to represent simple symbols (e.g. a simple type like int)
- implement both the Symbol and Scope interface - to represent composite objects (e.g. packages, classes, struct)
- implement only the Scope interface - to represent simple scopes (e.g. anonymous blocks of code)

The symbol table manager also follows the Abstract Factory design pattern (see [Gamma]).

4.4.3.1 SymbolTableMgr (abstract class)

SymbolTableMgr is an abstract class that is to be subclassed in each concrete symbol table manager packages. It defines the interface that the symbol table manager client (ADL purser) shall use.

This class is responsible for holding a reference to the scope being parsed (current scope), and managing the client's requests to open or close a new scope (e.g. when entering into a struct or class definition or into a new block of code):

```
void openScope(Symbol s)    // opens the scope owned by
                            // the provided symbol

void openScope()            // opens an anonymous scope
```

This class is also responsible for forwarding the put and get symbols request to the current scope:

```
void putSymbol(Symbol s)
void putAlias(String aliasName, Symbol s)
Symbol getSymbol(String scopedName,
                 UnaryPredicate predicate,
                 SymbolListener symbolListener,
                 Scope  scopeOverride)
```

Predicate may be a MethodComparator, symbol listener is used supplied when the parser is interested in some symbol not yet defined, and scope override is used when default current scope should be overridden to perform the symbol research.

Specialized getSymbol() methods are supplied so that symbols which were classified at parsing time may be easily retrieved, these methods are:

```
LVariable   getVariable(String scopeName)
LType       getType(String scopeName)
LPackage    getPackage(String scopeName)
ADLClass    getADLClass(String scopeName)
TDDClass    getTDDClass (String scopeName)
LExpression getExpression(String scopeName)
```

```
LMethod   getMethod(String scopeName,Predicate predicate)
ADLInline   getInline(String scopeName)
TDDFactory  getFactory(String scopeName)
```

and the likes...

This class is an abstract factory for language independent symbols. That is client parser should create their symbols through calls to the new...() methods. It is then up to the concrete symbol manager to supply either a default language independent symbol or a specialized symbol where necessary. The client parser does not need to know the symbol actual class. It only need to know the language independent symbol interface is implemented. The methods enabling to create these symbols are:

```
LVariable newVariable(String name,
                      int modifier,
                      LType type)
```

which returns a variable with the supplied name, modifier and type.

```
LClass   newClass(String name,
             int modifier,
             LClass[] superclasses,
             LClass[] interfaces)
```

which returns a new class with the supplied name, modifier (bit mask made of ABSTRACT, INTERFACE...), superclasses and interfaces. Note C++ symbol table managers accepts no interfaces while Java symbol table manager accepts only one superclass.

Many other factory methods are defined for other symbols...

To simplify, these methods are a superset of all the methods that should be necessary to create symbols for a particular binding. But since most of the symbols are very close to each other this is not an issue.

Finally the symbol table manager can tell the client parser whether current scope is inside a given scope type or not

```
boolean inScope(Class symbolClass)
```

4.4.3.2 Symbol (interface)

Symbol is the interface that all the symbols shall implement. It's an interface so that concrete symbols may be organized with their own inheritance tree structure.

This interface manages the name of the symbol: `getName()` and `setName()`.

This interface defines methods to gain access to the scope that may be associated to the symbol:

```
boolean hasScopeAccess()
```

Checks whether symbol has access to a scope or not. Note some symbols owns scopes but cannot be seen from any outer scope. An example is a function: it's got its own child scopes, but these scopes are not visible from any outer scope, this means that for function foo(), the scoped name foo.bar should not be allowed!!! This information is stated by the hasScopeAccess() method.

```
Scope getScope()
```

Returns the scope the symbol has access to.

4.4.3.3 Scope (interface)

Scope is the interface that all concrete scopes shall implement.

A scope is responsible for holding its symbols, organizing them in any suitable way (e.g. in a hash table).The methods provided to achieve this are:

```
void putSymbol(Symbol s)
```

which put a symbol into the current scope,

```
void putAlias(String aliasName, Symbol s),
```

which create an alias for symbol s in the current scope (useful for tepidness)

Scopes are also responsible for retrieving symbols, each concrete scope class defines its own policy for this. The most common policy is to search first in a local symbol hashtable and if nothing is found to forward the request to the parent scope (that's the way a method scope or a block scope do). But more subtle policies may be defined: a java class scope forward the requests to its superclasses scopes first and then to its parent package scope. The method used here is:

```
Symbol getSymbol(String name, Predicate p, Credentials c)
```

which gets a symbol in current scope, the symbol shall verify the supplied predicate and be visible to the requestor (which is checked using the credentials).

Furthermore scopes are responsible for forwarding getSymbol() requests to relevant scopes when necessary (e.g. a Class scope forwards the request to its superclasses scopes).

4.4.3.4 Language independent symbols hierarchy

Language independent symbols are abstract classes for symbols likely to be used within all ADL bindings.

The main inheritance tree for the language independent symbols is as follows:

(o-- means implements interface and <-- means inherits)

```
Symbol o-- LPackage

           LNameSpace

           LType    <--  LBuiltinType
                    <--  LPointerType
                    <--  LReferenceType
                    <--  LConstructedType   <--  LStruct
                                            <--  LUnion
                                            <--  LClass
                                            <--  ADLClass
                                            <--  TDDClass

           LCallable      <-- LFunction
                          <-- LMethod
                          <-- ADLInline
                          <-- TDDFactory

           LExpression    <--  LVariable
                          <--  LParameter
                          <--  LField
                          <--  ADLBinding
                          <--  TDDDataset
```

All these symbols classes also implements additional methods, notably to test for assignability of values to variables:

```
boolean LType.assignable(LType t)
```

checks that a variable of type t can be assigned to a variable of this type.

```
boolean LClass.subclassOf(LClass lClass)
```

checks that some classes are subclasses of other classes. E.g. when parsing thrown(E) it must be checked that E is a subclass of Throbs.

In addition, the following classes implements the Scope interface:

```
Scope     o-- LBlockScope
          o-- LTranslationUnit
          o-- ADLTranslationUnit
          o-- TDDTranslationUnit
          o-- LConstructedType
          o-- LCallable
```

The containment relationship in between symbols and scopes are presented here:

(The notation is:
```
  Symbol (type attribute...)
    <Symbol that may be defined in the symbol’s scope>)
```

```
ADLCompilationUnitScope
  LPackage (or LNameSpace)
      LFunction
      LClass (int modifier, LClass superclasses[],
                   LClass[] interfaces ... )
          JField (int modifier)
          JMethod (int modifier, LType returnType,
                       LParameter[] parameters, LClass[] throws )
              LBlockScope
                  LBlockScope...
                      LVariable
  ADLClass ( LClass tested, ADLClass extends )
      ADLInline
      ADLPrologue
      ADLEpilogue
      ADLBehaviourDeclaration
          ADLPrologue
          ADLEpilogue
          ADLSemantic
              ADLBehaviorClassification
              ADLAssertionGroup
                  ADLBinding
TDDCompilationUnitScope
  LPackage
     ...

  TDDClass (TDDClass used )
      TDDConstant
      TDDDataset
      TDDFactory
          ...
```

4.4.3.5 UnaryPredicate (abstract class)

UnaryPredicate is an abstract class that only implements one method

```
      boolean execute(Object o)
```

the predicate is used to check if an object satisfies the predicate or not.

As an example of predicate, consider the LCallableEqualsComparator predicate, which is a subclass from UnaryPredicate, it implements the following methods in to UnaryPredicate:

```
  void  setReturnType(LType type)

  void  addParameterType(LType type)

  void addException(LType exc)
```

This predicate is executed against a LCallable object and if the callable’s return type, parameters and thrown exceptions are equals to the method comparator ones, the predicate returns true. Otherwise it returns false. Note that another comparator LCallableAssignComparator may be used to check a method call is compatible with a method signature (i.e. the expressions types in the method call must be compatible with the formal parameters types).

4.4.3.6 Credentials

The point is that when searching for a symbol, the symbol table manager should check that any suitable symbol is visible from the requesting scope (the current scope when the request is issued).

Imagine for example that SomeClass.somePrivateattribute symbol is requested from a scope being external to SomeClass, then the symbol table manager should not return this symbol.

This issue is all the more complex since many different level of visibility exists (public, protected, package, private...) for many different symbols (public classes, public attributes...). Moreover a request to retrieve a symbol may be forwarded to many different scopes, and the last scope receiving the request may not know about the issuer.

So a general mechanism is required! The solution proposed here is that each scope be granted some credentials. These credentials are used by any scope receiving a request to decide if the requester has visibility over the requested symbol.

The credentials hold the scope originating the request, together with a list of “scope:visibility” pairs that is the scopes for which part or all the symbols are visible.

As an example suppose package BankPackage holds the class Bank and its subclass MyBank, then my bank would be granted the following credentials:

requestor = “MyBank”

credentials = { “Bank:protected” ,“BankPackage:package” }

Then, whenever Bank scope receives a getSymbol() request from MyBank scope it checks that the requested symbol can be returned or not by looking in the credential list at the “Bank:visibility” entry.

Though being somewhat complex, this mechanism is generic enough and scale sufficiently to be used for the ADL/java and ADL/C++ compilers. Furthermore, while a request is forwarded from scope to scope the credential list should grow, the relevant credential being added to the list.

Note: this feature is not implemented in this release. In Java limiting bytecode loading to only public fields and methods is enough for access control. A similar solution might be adopted for C++.

4.4.3.7 Symbol Listeners (callback mechanism)

The point is that some symbols can be used before they have been defined (e.g. Java methods).

A simple solution would be to write a two-pass ADL compiler. But the symbol listener mechanism based on the event model found in the Java AWT1.1 solves this problem and is much more elegant.

Actually, in case a symbol cannot be found by the client parser, possibly because it has not yet been defined, the parser client shall pass a symbol listener object to the symbol table manager. This symbol listener is stored by the symbol table manager, and as the definition of the requested symbol (or at the end of the parsing), this symbol listener is used called back so that the client parser can finish its set up.

Internally, the symbol table manager stores the symbol listener together with the current scope at the moment the symbol listener is received.

After parsing is finished, the symbol table manager process each saved symbol listener, that is try to find the symbol, and call the symbol listener back when the symbol is found.

To use this mechanism, the client parser should supply the symbol table manager with an object implementing the following interface:

```
interface SymbolListener {
    void symbolFound(Symbol symbol);
  }
```

The symbol listener can be implemented as an inner class of the ADLNode. Where methods are likely to be listened to this class could be MethodSymbolListener.

4.4.4 Specific org.opengroup.adl.symboltable.c package design (ADL/C & ADL/C++)

The org.opengroup.adl.symboltable.c package holds the concrete classes responsible for managing the symbols for the ADL C and C++ compilers (C++ symbols are a superset of C symbols).

The responsibility of these symbol table managers is to coordinate symbol searching in between scopes according to the rules defined in the ADL/C and ADL/C++ semantic checks document.

The language independent symbol hierarchy is taken as it is with minor changes. The main constraints on the symbols being:

- a CPP class implements no interfaces
- the CModifier class helps manipulating the C and CPP modifiers (public, protected, private, const, virtual...)

Furthermore, C and C++ typedefs are implemented as symbol aliases.

4.4.5 Specific org.opengroup.adl.symboltable.java Package Design (ADL/Java)

The org.opengroup.adl.symboltable.java package holds the concrete classes responsible for managing the symbols for the ADL/Java compiler.

The ALDSymbolTableMgr (resp. TDDSymbolTableMgr) is the ADL/Java specialized version of the SymbolTableMgr class for the ADL parser (resp. TDD parser).

The responsibility of these symbol table managers is to coordinate symbol searching in between scopes according to the rules defined in the ADL/Java semantic checks document (notably to determine the meaning of a name when retrieving a symbol or before defining a new symbol).

The language independent symbol hierarchy is taken as it is with minor changes. The main constraints on the Java symbols being:

- a class only has one superclass
- the JModifier class helps manipulating the Java modifiers (public, protected, private, abstract, interface...)

However with Java, new issues are raised. Since no pre-processor mechanism exists like in C/C++, Java imported stuff must be accessed at parsing time.

To solve this problem, a companion class is introduced: the symbol loader. The symbol loader (SymbolLoader class) is in charge of loading Java package and classes on demand and populate the symbol table with the symbols loaded.

The client parser uses the symbol loader when import or use clauses are encountered.

Another issue is also raised: loading symbols may be heavy, it is certainly better to load only the necessary stuff. This issue is solved by using proxy symbols. Proxy symbols are based on the Proxy design pattern ([Gamma]).

When requested to load symbols the symbol loader may decide to load only a proxy for the symbol.

The proxy symbol acts like the original symbol (it implements the same interface), that is it tries to respond most of the client request without any help from the original symbols. When the proxy cannot respond by itself, it requests the symbol loader to load the plain symbol, it gets a reference to that plain symbol and forwards the request to the symbol.

Hence, the use of proxy symbols is transparent both to the client parser and to the symbol table manager.

The design for the symbol loader and the proxy is discussed more deeply in the two following section.

4.4.5.1 SymbolLoader

The symbol loader (SymbolLoader class) is in charge of loading Java package and classes on demand and populate the symbol table with the symbols loaded.

The client parser uses the symbol loader when import or use clauses are encountered. Actual loading of the symbol is deferred to appropriate moment (e.g. when a class of an imported package is used).

From the client (parser) viewpoint symbols may be imported through relevant SymbolTableManager calls. That's why the SymbolLoader interface is also implemented by the SymboltableManager. Internally, the symbol table manager collaborates with the symbol loader so that symbols can be loaded as they are requested.

The loaded symbols are discovered via CLASSPATH and/or INCPATH, relevant checks are performed by the SymbolLoader, and finally symbols are put in the symbol table.

The SymbolLoader collaborates with the SymbolTableMgr so that the loaded symbols are put in the symbol table as part of the loading process. Its methods are the following:

```
LPackage importPackage(String name);
```

Imports a package. the processing here is to check that the package is accessible from classpath.

```
LClass importClass(String name);
```

Imports a class (or transparently a proxy for the class).

```
void useClause(String name);
```

Use clause. Makes the named ADL file available for use. Part of the process is to check that the adl file is accessible from incpath.

4.4.5.2 Proxy Symbols

The proxy design pattern shall be used when partial loading of symbols is useful (as part of the import/use mechanism).

Due to the design of the byte code file (.class) a proxy is needed only for Java classes. A Proxy Java Class is created each time a non loaded class symbol is encountered in the byte code (this applies only to public fields and method parameters and return types, superclasses symbols are always loaded).

E.g. a JClassProxy symbol should hold only its class name, respond to request for which this knowledge is enough by itself and forward any request for which more knowledge is required to the real JClass symbol, which have to be loaded before.

Proxy symbols collaborates with the SymbolLoader so that the actual symbols shall be loaded when necessary. The use of ProxySymbols should be transparent to the SymbolTableMgr and to the client parser.

4.4.6 Using the symbol table manager

The example below show how the CSymbolTableMgr is used from the client (ADL parser) view point:

EXAMPLE 2 Using the symbol table.

```
// Initialize symbol table manager

SymbolTableMgr stm = new CSymbolTableMgr();

// Get a handle on int builtin type

LType intT = stm.getType(“int”);

// Declares a typedef type ( typedef int MyInt )

stm.putAlias(“MyInt”,intT);

// Defines a struct type and put it into the current scope

LStruct ComplexT = stm.newStruct(“ComplexT”);

stm.putSymbol(ComplexT);

// Open the scope associated with that struct and defines
// two attributes for that struct (x and y) note that
// CAttribute symbols are constructed using a name and
// a type symbol. This type symbol is obtained from the
// symbol table manager.

stm.openScope(ComplexT);

LField x = stm.newField(“x”,intT);

stm.putSymbol(x);

LField y = stm.newField(“y”,intT);

stm.putSymbol(y);

// Close struct scope

stm.closeScope();

// Search for symbol with ComplexT.x scoped name

Symbol s = stm.getSymbol(“ComplexT.x”);
```

4.4.7 Guidelines for the symbol table manager user (FAQ like)

This section contains guidelines for using a symbol table manager when particular issues are encountered.

4.4.7.1 How to store typedefs in the symbol table (ADL/C & ADL/CPP)

Typedefs are considered as aliases for other types, instead of creating special symbols, alias entries are put in the symbol table. this is achieved using the putAlias() symbol table manager method.

4.4.7.2 How to control that variables are used after they are defined (ADL/Java)

This is an issue for ADL/Java where a variable can be declared anywhere in a block of code. The solution proposed is to force the client parser to check that a variable is defined before being used.

4.4.7.3 How to distinguish in between several overloaded method.

When retrieving a method symbol the parser client shall provide a supplementary LCallableAssignComparator predicate object, that will help the method scope selecting the suitable method symbol.

4.4.7.4 How can I check that only one external functions definition exists (ADL/C)

When retrieving a method symbol the parser client shall provide a supplementary LCallableEqualsComparator predicate object, that will help the method scope selecting the suitable method symbol.

4.4.7.5 How can I check the symbol I encounter is in a suitable scope.

This is to allows to control that symbols are used or are not defined inside a scope of some supplied type. For example to check that inlines are not called in the definition of other inlines.

The solution is to use the inScope() method of the symbol table manager.

EXAMPLE 3 Testing if encountered symbol is in a suitable scope.

```
SymbolTableMgr stm = new SymbolTableMgr();
...
if ( stm.inScope(TDDFactory.class) )
{
   throw new ADLParseError
         (“A factory cannot be called in the definition “ +
          “of another factory”);
}
```

Chapter 5 Code Generation

5.1 Introduction

Code generation is the fundamental task of ADLT. In ADLT, code generation is broken down into two phases: AST generation and unparsing. AST generation creates a new Abstract Syntax Tree to represent the required code; unparsing generates a text file representation of that AST. Code generation is separated into these two phases because many ADL constructs, like, e.g., the call state operator, require to generate some code ahead of their current location in the AST.

In addition, in order to avoid intricate code for both phases into the data structure representing an AST, they are designed around the “Visitor” pattern as described in the book from Gamma et al., “Design Patterns”. This pattern clearly separates the processing from the data structure it operates on by using a simple “accept/visit” protocol. A visitor object implementing a particular processing algorithm, e.g., AST generation, asks a given node in the AST whether it *accepts* to be *visited*. By *accepting*, the node then explicitly requests the visitor to *visit* it. Since version 0.7pre5, the Java^{TM} Compiler Compiler^{TM} tool set, especially `jjtree`, supports the “Visitor” pattern by generating the appropriate methods for the “accept/visit” protocol and an interface to be implemented by any candidate visitor.

Since there is a great deal of commonality between the AST generators for the various parts of a test program, and between the AST generators for various target languages; by separating the syntax details, we are able to re-use parts of the AST generators. Syntax details are mapped onto an AST structure. By simply using similar names in the various target languages grammar for similar syntactic constructs, it becomes possible to design some generic classes that define the appropriate processing methods for the relevant constructs.

Figure 5 shows the overall scheme of the design for code generation and illustrates how it will allow to make maximum re-use of the code within ADLT.

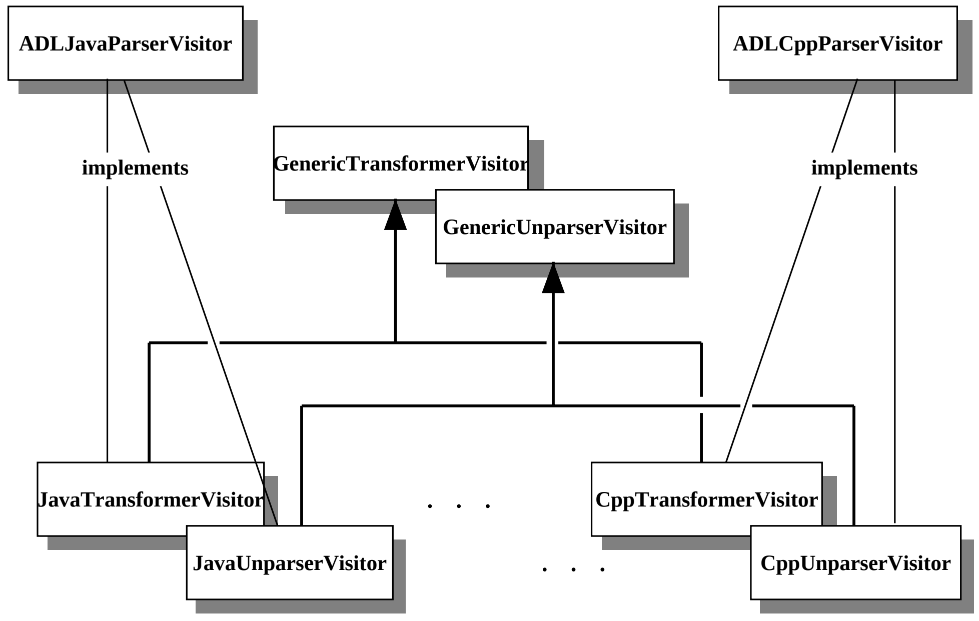


FIGURE 5. Code Generation Architecture Model

The exact content of the generic classes depends on the commonalities between the several target language grammars with respect to the concerned operation.

5.2 AST Transformation

Abstract Syntax Trees for the generated code are derived from the ASTs of the specifications (described in Chapter 3, “Abstract Syntax Tree Design,”). The outline of the generation process is to walk the input AST, generating the new AST as you go, using a “Transformer” visitor. The code pattern of a visit method depends on the type of the currently visited node:

- In some cases, the current node is the root of a subtree that has not to be modified: the visit method does nothing.
- In some cases, the current node has not to be modified, but some of its descendant nodes have to: the visit method simply consists in ensuring that the child nodes will be visited.
- In some cases, the properties of the current node have to be modified, but not its type and structure (it keeps the same number of children).

- Last, in some cases (notably all the ADL_* nodes) the visit method creates a new node that will replace the current node in the tree structure. The child nodes of this newly generated node are built depending on the current node children (visited or not), on its properties and on the type of the new node.

In the next section we detail the transformation patterns with respect to the visited node type, using Java as an example. Although each language binding generates different code, the described patterns are to a large extent reusable for each of them.

5.2.1 ADL Node Transformation

5.2.1.1 Notation Used

In the sequel, we will use the following notation to describe the actions to be performed to transform a node of a given type. This notation is an attempt at providing relevant information in a concise and as precise as possible manner; this is *not* a formal notation.

a/ ATypeNode

Children nodes:

```
ATypeNode1 child1
[ ATypeNode2 child2 ]
( ATypeNode3 child3 )*
```

Created Node:

```
ATypeNode4 :
    ATypeNode1 : child1
    ATypeNode5 :
        [ visit(child2) ]
        ( visit(child3) )*
```

Fields:

```
[ ATypeNode2 field1 = child2 ]
```

Actions:

```
// pseudo-code to precise complicated actions to be done.
```

This must be read as follows: the current node has one first child node of type ATypeNode1, an optional second child of type ATypeNode2, and a possibly empty list of child nodes of type ATypeNode3. It must be transformed into a node of type ATypeNode4 with two children: the first child is the first child of the current node, and the second is a node of type ATypeNode5 which has itself as a first optional node the transformation of the current node second child, and also the list of the transformed ATypeNode3 nodes of the current node. Furthermore, the node child2, if it is present, is stored as a global field: this handle is necessary when the information contained in node child2 is to be used in the generated tree somewhere else that at direct proximity of the present node transformation.

Children nodes are sometimes declared as Node (the interface of all node types) when the dynamic type cannot be known statically: for instance the root node of an expression could be ADL_Expression, EqualityExpression, PrimaryExpression, etc.

b/ ACO Blocks

Some blocks of generated code are always the same, whatever the input program is, or just slightly different (for instance following the name of the adl class).

For this, we define a class ACOauxiliary and the visitor will declare an instance ACO of this class. Some data obtained by visit methods will be stored as ACO fields when they are needed for code generation. The make* methods generate ASTs of the output Java program. For instance, the method makeInitialization generates ASTBlockStatements for the following code:

```
boolean doReport = finalJournalReporting;
boolean normal = true; boolean abnormal = false;
boolean tddRes; boolean wasThrownException = false;
Throwable adl_thrownException = null;
```

and in the case of a constructor or a static method:

```
skipAssertions |= testContext.skipAssertions;
```

else:

```
skipAssertions |= testContext.skipAssertions ||
     realObject == null;
<adlclass> <adlclass>Object = (<adlclass>)realObject;
```

5.2.1.2 Transformation Patterns

a/ ADL_CompilationUnit

```
Children nodes:
    ( ImportDeclaration : impDecl )*
    ADL_ClassDeclaration : adl_ClassDecl
Created node:
    ADL_CompilationUnit :
        ImportDeclaration : <ORG.opengroup.adl.runtime.*>
        ( ImportDeclaration : impDecl )*
        visit(adl_ClassDecl)
```

b/ ADL_ClassDeclaration

```
Children nodes:
    IDENTIFIER : id
    [ Name : name ]  // name of the inherited adl class.
    [ ADL_Prologue : prolog ]
    [ ADL_Epilogue : epilog ]
    (   ADL_BehaviorDeclaration : behavDecl
        |
        ADL_InlineDeclaration : inlineDecl
    )*
Created node:
    ClassDeclaration :
        MODIFIER : “public”
```

```
            IDENTIFIER : id
            Name : extendedClass
            ConstructorDeclaration : ACO.makeConstructor() // build the constructor
            (  visit(behavDecl) )*
    Fields:
        ACO.setExtendedClass(name + “ACO”) // “ObjectACO” by default
        [ Block globalProlog = prolog ]
        [ Block globalEpilog = epilog ]
        ACO.setADLClassName(id) // name of the class under test
        ( ADL_InlineDeclaration inlineDeclaration = inlineDecl )*
        // for inline substitution, see Section 5.2.1.4.
```

c/ ADL_BehaviorDeclaration

```
    Children nodes:
        [ ResultType : resType ]
        Name : name
        FormalParameters : formalParam
        [ NameList : nameList ]
        [ ADL_Prologue : prolog ]
        ADL_BehaviorSpecification : behavSpec
        [ ADL_Epilogue : epilog ]
    Created nodes:
        MethodDeclaration :
            MODIFIER : “public”
            ResultType : ACO.getMetodReturnType()
            MethodDeclarator :
                IDENTIFIER : name
                FormalParameters : formalParam
            [ NameList : nameList ]
            visit(behavSpec)
    Fields:
        [ Block localProlog = prolog ]
        [ Block localEpilog = epilog ]
        ACO.setIsConstructor (<no ResultType>)
        if (isConstructor)
            ACO.setMethodReturnType (name)
        else ACO.setMethodReturnType(resType)
        ACO.setMethodName(name)
        ACO.setMethodFormalParam(formalParam)
        [ ACO.setMethodThrowsList(nameList) ]
```

d/ ADL_BehaviorSpecification

```
    Children nodes:
        [ ADL_BehaviorClassification : behavClass ]
        ADL_AssertionGroup : adl_assGrp
    Created node:
        Block :
            ( BlockStatement : ACO.makeInitialization().child )+
            ( BlockStatement : globalProlog.child.clone() )*
            ( BlockStatement : localProlog.child )*
```

(BlockStatement : saveBlock)*
(BlockStatement : ACO.makeCallBlock().child)+
visit(adl_assGrp)
(BlockStatement : localEpilog.child)*
(BlockStatement : globalEpilog.child.clone())*
(BlockStatement : ACO.makeFinal().child)+

Fields:
ACO.setSuperSemantics(<adl_assGrp has a child ADL_SuperSemantics>)

Actions:
visit(behavClass)
generate default values for (ab)normalSpec fields that have not been defined

Notes:
The saveBlock notation stands for a list of statements that deal with the call-state feature: saveBlock is the assignment of a generated temporary variable whose value is that of a call-state expression (see Section 5.2.1.3.b).
The visit to child adl_assGrp occurs *before* inserting the saveBlock statements: the latter ones are produced during the visit of the former.
The method makeCallBlock will use ACO fields superSemantics, myType, methodName, methodFormalParam, methodThrowsList, normalSpec and abNormalSpec.
The method makeFinal needs methodName and methodThrowsList.

e/ ADL_BehaviorClassification

Children nodes:
(Node : exp)+

Fields:
Expression (ab)normalSpec = exp

Note:
the trees “exp” are *not* visited here: this is will be done in the ACO.makeCallBlock.

f/ ADL_AssertionGroup (nested)

Note:
Conjunction of assertions in an expression
... exp || { <ass_1>; <ass_2>; } ...
To be transformed into:
... exp || (exp_1 && exp_2) ...
where exp_i is the return of visit(ass_i)

Children nodes:
(Node : adl_exp)*

Created node:
visit(ConditionalAndExpression : (adl_exp)*)
or
visit(adl_exp) // if only one child

g/ ADL_AssertionGroup (not nested)

Note:
“real” group of ADL assertions

Children nodes:
(ADL_Binding : adl_Binding)*
(Node : adl_Stmt)*

Created node:

```
Block :
    ( visit(adl_Binding) )*
    ( visit(adl_Stmt) )*
```

h/ ADL_Assertion (not nested)

Children nodes:

```
[ ADL_Label : label ]
[ ADL_Tags : tags ]
ConditionalExpression : adl_Expr
```

Created node:

```
Block:
    BlockStatement :
        <testContext.assertionStart("<assertionString>")>
    IfStatement : // if1
        UnaryNotPlusMinus : // condition if1
            Operator : "!"
            Name : "skipAssertions"
        Block : // then if1
            ( BlockStatement : preAssertion )*
            StatementExpression :
                Assignment :
                    Name : "tddRes"
                    visit(expr) // note: this visit was performed "before",
                            // to generate preAssertion
            IfStatement : // if2
                Name : "debugMode" // condition if2
                Block : // then if2
                    ( BlockStatement :
                        <testContext.infoline(preAssertion.string,
                            preAssertion.tmp)>
                    )*
            BlockStatement :
            <testContext.assertionResult(tddRes ? ADL_PASS : ADL_FAIL)>
        Block : // else if1
            BlockStatement :
                <testContext.assertionResult(ADL_UNEVALUATED)>
```

Fields:

```
[ IDENTIFIER assLabel = label ]
( IDENTIFIER assTags = tags )*
```

Note:

The preAssertion notation stands for a list of statements that deal with the decomposition of the assertion expression in sub-expressions (see Section 5.2.1.3).

Example:

```
testContext.assertionStart("get_active_accounts()==
  @get_active_accounts()+1");
if (!skipAssertions) {
  // preAssertions
  int __ADL_tmp_0 = bankObject.get_active_accounts();
  int __ADL_tmp_1 = __ADL_savTmp_1; // saveBlock tmp variable
  int __ADL_tmp_2 = __ADL_tmp_1 + 1;
  boolean __ADL_tmp_3 = __ADL_tmp_0 == __ADL_tmp_2;
  tddRes = __ADL_tmp_3;
  if (debugMode) {
     testContext.infoline(“get_active_accounts()\t” +
         __ADL_tmp_0);
     testContext.infoline(“@get_active_accounts()\t” +
         __ADL_tmp_1);
     testContext.infoline(“@get_active_accounts()+1\t” +
         __ADL_tmp_2);
     testContext.infoline(“get_active_accounts()==
        @get_active_accounts()+1\t” + __ADL_tmp_3);
  }
  testContext.assertionResult(( tddRes ? ADL_PASS :
     ADL_FAIL ));
}
else {
  testContext.assertionResult(ADL_UNEVALUATED);
}
```

i/ ADL_Assertion (nested)

Children nodes:
    Node : expr
Created node:
    visit(expr)
Note:
    In this case, the assertion is considered as just a boolean expression.

j/ ADL_IfStatement (nested)

Note:
    evaluated as an expression: (cond) ? (thenBranch) : (elseBranch)
Children nodes:
    Node : adl_Expr
    ADL_AssertionGroup : adl_AssGrp
    [Node : elseBranch] // ADL_AssertionGroup or ADL_IfStatement
Created node:

```
ConditionalExpression :
    visit(adl_Expr)
    visit(adl_AssGrp)
#if (#(children nodes) > 2)
    visit(elseBranch)
#else
    BooleanLiteral : “true”
```

k/ ADL_IfStatement (not nested)

Children nodes:
    ConditionalExpression : adl_Expr
    ADL_AssertionGroup : adl_AssGrp
    [Node : elseBranch] // ADL_AssertionGroup or ADL_IfStatement
Created node:

```
IfStatement :
    visit(adl_Expr)
    visit(adl_AssGrp)
#if (#(children nodes) > 2)
    visit(elseBranch)
```

l/ ADL_TryStatement (not nested)

Children nodes:

```
    ADL_AssertionGroup : tryAssGrp
    (   FormalParameters : catchFP
        ADL_AssertionGroup : catchAssGrp
    )+
```

Created node:

```
    TryStatement :
        visit(tryAssGrp)
        (   FormalParameters : catchFP
            visit(catchAssGrp)
        )+
```

m/ ADL_TryStatement (nested)

Children nodes:

```
    ADL_AssertionGroup : tryAssGrp
    (   FormalParameters : catchFP
        ADL_AssertionGroup : catchAssGrp
    )+
```

Created node:

```
    Name : aName
```

Actions:

see the decomposition algorithm (Section 5.2.1.3.c).

n/ ADL_Binding

Children nodes:

```
    FormalParameter : formalParam
    ConditionalExpression : expr
```

Created node:

```
    LocalVariableDeclaration :
        Type : formalParam.Type
        VariableDeclarator :
            VariableDeclaratorId : formalParam.VariableDeclaratorId
            visit(expr)
```

o/ ADL_ImplExpression

Children nodes:

```
    Node : e1, e2
    ADL_ImplOp : op
#if (op == “==>”) // e1 ==> e2  --->  !(e1) || (e2)
```

Created node:

```
    ConditionalOrExpression :
        UnaryExpressionNotPlusMinus :
            Operator: “!”
            ParenthExpression:
                visit(e1)
        ParenthExpression:
            visit(e2)
#if (op == “<==”) // similar (exchange e1 and e2)
#if (op == “<=>”) // e1 <=> e2  --->  (e1) == (e2)
Created node:
    EqualityExpression:
        ParenthExpression:
            visit(e1)
        Operator: “==”
        ParenthExpression:
            visit(e2)
#if (op == “<:>”) // e1 <:> e2  --->
            (e1) ? abnormal : !((e2) && abnormal)
Created node:
    ConditionalExpression:
        ParenthExpression:
            visit(e1)
        Name : “abnormal”
        UnaryExpressionNotPlusMinus
            Operator: “!”
            ConditionalAndExpression:
                ParenthExpression:
                    visit(e2)
                Name : “abnormal”
```

p/ ADL_Expression (and all other Java expression nodes except PrimaryExpression)

Children nodes:
(Node : exp)+
Created nodes:
Name : aName
Actions:
see the decomposition algorithm (Section 5.2.1.3)

q/ ADL_ThrownExpression

Children nodes:
NameList : nameList
Created node:

```
#if (#nameList == 1)
    // thrown(e1)  ---->  (thrownException instanceof e1)
    InstanceOfExpression:
        Name : “thrownException”
        Type : nameList.getChild(0)
#else
    // thrown(e1, e2)  ---->  (thrownException instanceof e1) || (thrownException
    instanceof e1)
    ConditionalOrExpression:
        InstanceOfExpression:
            Name : “thrownException”
        Type : nameList.getChild(0)
    InstanceOfExpression:
        Name : “thrownException”
        Type : nameList.getChild(1)
    ...
```

r/ unchanged

Note:

```
unchanged(expr) ---> (expr == @expr)
unchanged (expr1, expr2 /* , ... */) --->
    ((expr1 == @expr1) && (expr2 == @expr2) /* && ... */)
```

Children nodes:

```
Arguments : args
```

Created node:

```
#if (#args == 1)
    EqualityExpression :
        ConditionalExpression :
            visit(args.ArgumentList.ConditionalExpression)
        Operator: “==”
        ConditionalExpression :
            visit(ADL_CallStateExpression :
                args.ArgumentList.ConditionalExpression)
#else
    ConditionalAndExpression :
        EqualityExpression :
            ConditionalExpression :
                visit(args.ArgumentList.ConditionalExpression1)
            Operator: “==”
            ConditionalExpression :
                visit(ADL_CallStateExpression :
                    args.ArgumentList.ConditionalExpression1)
        EqualityExpression :
            ConditionalExpression :
                visit(args.ArgumentList.ConditionalExpression2)
            Operator: “==”
            ConditionalExpression :
                visit(ADL_CallStateExpression :
                    args.ArgumentList.ConditionalExpression2)
        ...
```

5.2.1.3 Expression Decomposition Algorithm

a/ Rationale

In the ADL runtime, i.e. when evaluating the assertions of the generated ACOs, there is a possible “debug” mode such that all expressions present in the assertions are decomposed in sub-expressions and all these sub-expressions are evaluated separately. Thus, when an assertion fails, the user may set the debug mode in order to have a more refined analysis and detect which sub-expression of the assertion is responsible for the failure.

The debug mode is a *runtime* option, not a *compilation* option, therefore the transformation must foresee this mode: the generated code will systematically decompose expressions in assertions and guard the “debug mode code” by a condition (in fact, only the reporting of the values of the sub-expressions is guarded).

Note that some expressions will not be decomposed:

- try/catch expressions
- call-state expressions
- expressions in quantified assertions

Note also that only expressions in assertions can be decomposed, not the expressions that occur in binding definitions, prologues and epilogues.

b/ Algorithm: General Case

```
Note:
    The algorithm for a PrimaryExpression node (especially when this corresponds to a
    method call) is quite complex: it will be presented separately in Section 5.2.1.5.
Current node:
    Node : exp // an expression node
Children nodes:
    ( Node : subexp )* // any kind of expressions
Action:
#if (isDecomposable)
    String str = <the token string of the current node>
            // stored to be called in infoline, in debug mode
    ( visit(subexp) )*
    make_preassertion // store information: node type, visited node, str
    // return the name of a generated variable stored as a preassertion
Created node:
    Name tmp // variable name returned by make_preassertion
#else
Created node:
    ( visit(subexp) )*
Note:
    isDecomposable = isInAssertion && !(isInTry ||
        isInCallState || isInQuantifiedAssertion)
Example:
    visit(EqualityExpression: a + b == f(c))
        str = “a + b == f(c)”
        visit(AdditiveExpression: a+b)
            str = “a + b” // local variable (does not overwrite preceding str)
            visit(Name: a)
                str = “a”
                // no child node to visit
                make_preassertion(typeOf(a), a, str) -> tmp1
                 // preAssertion: “T tmp1 = a;” where T is typeOf(a)
                return // transformed tree: Name a -> Name tmp1
            visit(Name: b)
```

```
                str = “b”
                // no child node to visit
                make_preassertion(typeOf(b), b, str) -> tmp2
                 // preAssertion: “T tmp2 = b;” where T is typeOf(b)
                return // transformed tree: Name b -> Name tmp2
            // no more child node to visit
            make_preassertion(typeOf(a+b), tmp1+tmp2, str) -> tmp3 // str == “a+b”
            return // transformed tree: AdditiveExpression a+b -> Name tmp3
        visit(PrimaryExpression: f(c))
            str = “f(c)”
            visit(Name: f)
                return // function name: node not transformed
            visit(Name: c)
                str = “c”
                // no child node to visit
                make_preassertion(typeOf(c), c, str) -> tmp4
                return // transformed tree: Name c -> Name tmp4
            // no more child node to visit
            make_preassertion(typeOf(f(c)), f(tmp4), str) -> tmp5 // str == “f(c)”
            return // transformed tree: PrimaryExpression f(c) -> Name tmp5
        // no more child node to visit
        make_preassertion(boolean, tmp3 == tmp5, str) -> tmp6 // str == “a+b==f(c)”
        return // transformed tree: EqualityExpression a + b == f(c) -> Name tmp6
```

During this evaluation, six pre-assertions are stored. Once the assertion evaluation is completed, the transformer will generate the code along with these preassertions: e.g. (see Section 5.2.1.2.d: ADL_BehaviorSpecification):

```
testContext.assertionStart(“a + b == f(c)”);
if (!skipAssertions) {
   int tmp1 = a; short tmp2 = b; int tmp3 = tmp1 + tmp2;
   short tmp4 = c; int tmp5 = f(tmp4);
   boolean tmp6 = tmp3 == tmp5;
   tddRes = tmp6;
   if (debugMode) {
      testContext.infoline(“a”, tmp1);
      // ...
      testContext.infoline(“a+b==f(c)”, tmp6);
   }
   testContext.assertionResult(tddRes ? ADL_PASS :
                                        ADL_FAIL);
} else {
   testContext.assertionResult(ADL_UNEVALUATED);
}
```

c/ Call-State Expression

- A call-state expression is not further decomposed.
- A call-state expression is first saved as a saveBlock, and the generated temporary that corresponds to this saveBlock is then saved as a PreAssertion.

Example:

```
visit(EqualityExpression: a == @(f(b) + c))
    str = “a==@(f(b)+c)”
    visit(Name: a)
        str = “a”
        make_preassertion(typeOf(a), a, str) -> tmp1
        // preAssertion: “T tmp1 = a;”
        return // transformed tree: Name tmp1
    visit(ADL_CallStateExpression: @(f(b)+c))
    // no decomposition of a callstate
        str = “@(f(b)+c)”
        make_saveBlock(typeOf(f(b)+c), f(b)+c) -> savTmp1
        make_preassertion(T, savTmp1, str) -> tmp2 // str == “@(f(b)+c)”
        // preAssertion: “T tmp2 = savTmp1;”
        return // transformed tree: Name tmp2
    make_preassertion(boolean, tmp1==tmp2, str) -> tmp3
    return // transformed tree: Name tmp3
```

Generated code:
As a saveBlock (see Section 5.2.1.2.d: ADL_BehaviorSpecification)

```
int tmpSav1 = 0;
if (!skipAssertions) {
   tmpSav1 = f(b)+c;
}
```

As preAssertion:

```
short tmp1 = a; int tmp2 = savTmp1;
boolean tmp3 = tmp1 == tmp2; tddRes = tmp3;
```

Note:

make_saveBlock generates 2 distinct statements: the declaration of the temporary variable with an initialization to a default value (the Java default value for the type of the variable) and the assignment of the call-state expression to this variable. This is because the evaluation of the expression must be performed only when the runtime is not in a “skipAssertions” mode and because the Java compiler requires that a local variable be always initialized before being used.

d/ “Try/Catch” Case

A try/catch expression is not further decomposed.

Example:

```
visit(EqualityExpression: a + { try { f(b)+c; } catch(Exc e) { 0; } } == 0)
    str = “a+{try{f(b)+c;}catch(Exc e){0;}}==0”
    visit(AdditiveExpression: a+{try{f(b)+c;}catch(Exc e){0;}})
        str = “a+{try{f(b)+c;}catch(Exc e){0;}”
        visit(Name: a)
            str = “a”
            make_preassertion(typeOf(a), a, str) -> tmp1
            return // transformed tree: Name tmp1
        visit(ADL_TryAssertionGroup: try{f(b)+c;}catch(Exc e){0;})
            // no decomposition of a try block
            str = “try{f(b)+c;}catch(Exc e){0;}”
            make_preassertion(typeOf(f(b)+c), try{f(b)+c;}catch(Exc e){0;}, str)
-> tmp2
            return // transformed tree: Name tmp2
        make_preassertion(T, tmp1+tmp2, str) -> tmp3
        return // transformed tree: Name tmp3
    visit(0)
        return // primitive values are not transformed
    make_preassertion(boolean, tmp3 == 0, str) -> tmp4
    return // transformed tree: Name tmp4
```

Generated code:

```
    short tmp1 = a; int tmp2 = 0;
    try { tmp2 = f(b)+c;} catch(Exc e) {tmp2 = 0;}
    int tmp3 = tmp1+tmp2;
    boolean tmp4 = tmp3 == 0; tddRes = tmp4;
```

Note:

here, make_preassertion generates in fact two separate statements: the declaration of the temporary variable and its initialization with a default value, and the try/catch statement where this variable is given the expression value.

5.2.1.4 Inline Substitution Algorithm

a/ Inline call (PrimaryExpression)

Children nodes:

```
PrimaryPrefix:
    Name: inlineName
PrimarySuffix:
    Arguments: inlineArgs
```

Created node:

```
visit(currentInline.ADL_AssertionGroup.clone()))
```

Fields:

```
currentInline = // the inlineDeclaration that corresponds to the current
                // call to inlineName (Section 5.2.1.2.b)
inlineFormalParams = currentInline.MethodDeclarator.FormalParameters
inlineActualParams = inlineArgs
inlining // boolean set to true before visiting currentInline.ADL_AssertionGroup,
         // false after
```

b/ Name

Current node:

Name aName

Actions:

```
if inlining
    if aName <is the nth parameter of inlineFormalParams>
        newnode = inlineActualParams.getChild(n).clone()
```

Created node:

```
visit(newnode)
```

Note:

in other cases, see the usual decomposition algorithm (Section 5.2.1.3).

5.2.1.5 PrimaryExpression: general algorithm

Several cases must be taken into account, for example whether this is an inline call or if the node is encountered during an inline substitution.

```
Current node:
    PrimaryPrefix : prefix
    PrimarySuffix : suffix1
    ( PrimarySuffix : otherSuffixes )*
Actions:
    IF prefix is not a name: usual algorithm (visit all children)
    ELSE
      IF prefix is an inline name
        IF no otherSuffixes: see Section a on page 70
        ELSE return a PrimaryExpression whose prefix is the visited inline call
          (i.e. visit(prefix.suffix1)) and suffixes are visit(otherSuffixes)
      ELSE
        IF inlining
          IF prefix is the name of a formal parameter of the current inline substitution
            return a PrimaryExpression with as prefix the visit to the corresponding
            actual parameter and as suffixes visit(suffix1) and visit(otherSuffixes)
          IF prefix is a qualified name whose first component is the name of a formal
            parameter of the current inline substitution
            IF the corresponding actual parameter is a name, just replace the formal
              parameter by the actual one:
              <Name: <inlineFormalP>”.”<nameRest>><suffix1><otherSuffixes> -->
              <Name: <inlineActualP>”.”<nameRest>><suffix1><otherSuffixes>
            ELSE the rest of the name becomes a DotSuffix:
              <Name: <inlineFormalP>”.”<nameRest>><suffix1><otherSuffixes> -->
              <PrimaryPrefix: inlineActualP><DotSuffix: nameRest><suffix1><oth-
    erSuffixes>
        IF prefix is the name of a component of the tested object, it must be qualified
          by this real object name.
```

5.2.2 TDD Node Transformation

5.2.2.1 Design

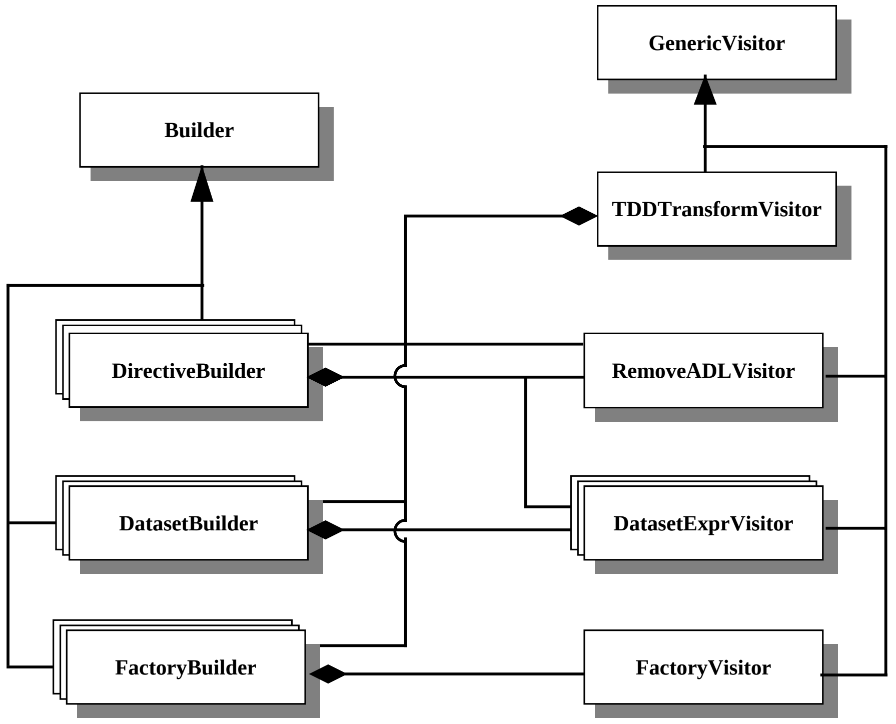


FIGURE 6. TDD Code Generation Architecture Model

Figure 6 presents the UML diagram of the hierarchy of classes used in the code generation code for a tdd class. The main class is the TDDTransformVisitor, which is given the AST root of the parsed tdd class. This visitor will “dispatch” the computation, creating objects for each feature present in the source: a DatasetBuilder object for a dataset definition, a FactoryBuilder for a factory definition, a DirectiveBuilder for a test directive. Each of these builders will have to create an AST that will be unparsed as a separate Java file; they all inherit from an abstract class Builder that initializes such a tree.

Dataset and directive features both contain dataset expressions; when a dataset expression is encountered, one must create a DatasetExprVisitor object that will visit this expression and generate the code that implements it.

Finally, the class RemoveADLVisitor is used to visit the AST of a directive body or a test function to detect ADL and/or ADL_new expressions and replace them with suitable code. FactoryVisitor is used to visit the body of factory definitions in order to replace primitive type expressions in return statements by an equivalent wrapper object (for instance “return 0;” must be replaced by “return new Integer(0);”).

5.2.2.2 TDDTransformVisitor

a/ ADL_CompilationUnit

Children nodes:
```
( ImportDeclaration : impDecl )*
( TDD_UseDeclaration : useDecl )*
TDD_ClassDeclaration : tddClass
```
Created node:
```
CompilationUnit :
    ( ImportDeclaration : importDecls )+
    visit(tddClass)
```
Fields:
```
( ImportDeclaration importDecls += impDecl )*
importDecls += <ORG.opengroup.adl.runtime.*>
```
Note:

“use” declarations are used only by the symbol table manager during parsing.

b/ TDD_ClassDeclaration

Children nodes:
```
IDENTIFIER : id
( TDD_ClassBodyDeclaration : decl )*
```
Created node:
```
ClassBodyDeclaration :
    MODIFIER : “public”
IDENTIFIER : id + “TDD”
Name : “ADLTest” // extends clause
( visit(decl) )*
```
Fields:
```
Name unitName = id
CompilationUnit : constInterface =
    makeConstInterface(constants, unitName, importDecls)
// for the constant interface, see Section f.
```

c/ TDD_DatasetDeclaration

Created node:
```
null // no created node
```
Action:
```
datasets += new DatasetBuilder(node, importDecls, unitName).compilUnit
```
Note:

“datasets” is the handle that gathers all ASTs that correspond to dataset definitions (returned as field “compilUnit” of the DatasetBuilder). It will be used by the ADLT engine that has started the TDDTransformVisitor, so that its AST elements are sent to an unparser.

The principle is the same for both following sections TDD_FactoryDeclaration and TDD_TestDirective.

d/ TDD_FactoryDeclaration

Created node:
```
null // no created node
```
Action:

```
factories += new FactoryBuilder(node, importDecls, unitName).compilUnit
```

e/ TDD_TestDirective

Created node:

```
null // no created node
```

Action:

```
directives += new DirectiveBuilder(node, importDecls, unitName).compilUnit
```

f/ TDD_FieldDeclaration

Children nodes:

```
Type : type
( VariableDeclaratorId : id
  VariableInitializer : init )+
```

Created node:

```
null // no created node
```

Actions:

```
FieldDeclaration constants +=
    MODIFIER : “public”
    Type : type
    ( VariableDeclarator :
        VariableDeclaratorId : id
        VariableInitializer : init
    )+
```

Note:

makeConstInterface is a method called in the visit of TDD_ClassDeclaration, after the visit to FieldDeclaration nodes, which builds the interface that gathers all declarations of the “constants” handle. This is a trivial method that will not be detailed here.

g/ MethodDeclaration

Actions:

Create an RemoveADLVisitor object and make it visit the last child (Block) of the current node (see Section 5.2.2.8).
Return the current node.

5.2.2.3 Dataset Builder

Input:

```
TDD_DatasetDeclaration :
    TDD_SingleDeclarator :
        ResultType : dsType
        VariableDeclaratorId : dsName
    TDD_DatasetExpr : datasetExpr
( ImportDeclaration : importDecls )*
Name : unitName
```

Fields:

```
Name datasetName = dsName
Name className = “S_” + unitName + “_” + datasetName
Type datasetType = dsType
```

Actions:

datasetComponents is computed by visiting the dataset expression datasetExpr through a dataset expression visitor (see Section 5.2.2.6):

```
visitor = new DatasetExprVisitor("tc", "elt");
visitor.visit(datasetExpr);
```

(the first string parameter is used to send the right test-context for newly created datasets, and the second one will initialize the visitor's stack of string strStack)

Output:

```
CompilationUnit :
    ( ImportDeclaration : importDecls )*
    ClassDeclaration :
        MODIFIER : "public"
        IDENTIFIER : className
        Name : findType(datasetExpr) // inherited class name
        ClassBodyDeclaration :
            ConstructorDeclaration :
                MODIFIER : "public"
                IDENTIFIER : className
                FormalParameters :
                    <ADLTest tc>
                ExplicitConstructorInvocation :
                    <super(tc, "<datasetName>");>
                ( visitor.datasetComponents.top )*
                ( BlockStatement :
                    <addElement(elt<i>);>
                )i=1..datasetComponents.top.size()
```

5.2.2.4 Directive Builder

Input:

```
TDD_TestDirective : node
    [ IDENTIFIER : ident ]
    ( TDD_DatasetDomain :
        [ TDD_SingleDeclarator : single ]
        TDD_DatasetExpr : datasetExpr
    )*
    Statement: userStmt
( ImportDeclaration : importDecls )*
Name : unitName
```

Fields:

```
[ Name directiveName = ident ] // or directiveName = "anonymous_<index>"
Name className = "D_" + unitName + "_" + directiveName
( SingleDeclarator singleDecl = single )*
```

Output:

```
CompilationUnit :
    ( ImportDeclaration : importDecls )*
    ClassDeclaration :
        MODIFIER : "public"
        IDENTIFIER : className
```

```
          Name : "TDD" + unitName
          ClassBodyDeclaration :
               ConstructorDeclaration : makeDirectiveConstructor()
               MethodDeclaration : makeMain()
               MethodDeclaration : makeRunInstance()
```

Actions:

`1/ makeDirectiveConstructor -->` generates constructor `"public <className>()"` with body created by:

```
( // loop on dataset expression parameters
   visitor = new DatasetExprVisitor("this", "elt_" + i)
   visitor.visit(datasetExpr);
   ( visitor.datasetComponents.top )*
   BlockStatement :
      "addParameter(elt<i>)"
)i=1..nbDatasetExpr
```

`2/ makeMain -->` generates method:

```
"static public void main(String[] args) {
     testCase.main("<directiveName>", args,
        new <className>());
}"
```

`3/ makeRunInstance -->` generates method `"protected void runInstance(Object[] paramValues) throws Throwable"` with body created by a loop on singleDecl, such that for each SingleDeclarator (a couple type-name), two statements are generated:

- "<type> <name> = (<type>) paramValues[i];"
  (with a special case to consider when the type is primitive: for instance if type==int: then "int <name> = ((Integer) paramValues[i]).intValue();")
- "infoline("Parameter <type> <name> = " + <name>);"

Then the user code of the test directive (userStmt) is visited by an RemoveADLVisitor (see Section 5.2.2.8) and the return of this visit is added after these statements.

5.2.2.5 Factory Builder

Input:

```
TDD_FactoryDeclaration : node
     MethodDeclaration :
          ResultType : factType
          MethodDeclarator :
               IDENTIFIER : factName
               FormalParameters :
                    ( FormalParameter :
                         Type : typeParam
                         VariableDeclaratorId : nameParam
                    )*
          [ NameList : factThrowsList ]
          Block : factBlock
```

```
        [ [ FormalParameters : relinquishParams ]
          Block : relinquishBlock ]
    ( ImportDeclaration : importDecls )*
    Name : unitName
```
Fields:
```
    Name className = "F_" + unitName + "_" + factName
```
Output:
```
    CompilationUnit :
        ( ImportDeclaration : importDecls )*
        ClassDeclaration :
            MODIFIER : "public"
            IDENTIFIER : className
            Name : "ADLFactorySet"
            ClassBodyDeclaration :
                ConstructorDeclaration : makeFactoryConstructor()
                MethodDeclaration : makeFactoryProvide()
                MethodDeclaration : makeFactoryRelinquish()
                MethodDeclaration : makeProvide()
                MethodDeclaration : makeRelinquish()
```
Note:

The constructor and the methods makeFactoryProvide, makeFactoryRelinquish and makeRelinquish are easily built: direct and obvious tree creation using available information (no subtree to visit, no particular algorithm or data structure to take into account).
The makeProvide (resp makeRelinquish) method generates the code of a method that is exactly the same as the factory code written by the user (resp the relinquish block), except:

- the name of the method ("provide_<factName>" instead of "<factName>").
- if the factory has a primitive return type (e.g. int), the last child (Block) of the factory definition must be visited by a FactoryVisitor object: see Section 5.2.2.7.

5.2.2.6 Dataset Expression Visitor

a/ Algorithm

The dataset expression visitor is used by two builders: DatasetBuilder and DirectiveBuilder. It is given as input an AST that corresponds to a dataset expression, but it does not modify this tree: instead it creates a set of trees that will be used in different ways by the caller.

The transformation of a dataset expression (either in a dataset or in a directive definition) consists in declaring and initializing a Java variable. The class type of this object is either an existing external dataset implementation class or factory implementation class, or an ADLLiteralSet (initialized by an array containing the values of the literal set expression), or an ADLConcatenationSet. The goal of the dataset expression visitor is to build the trees corresponding to these declarations.

There are two kinds of dataset expressions: the *primitive* expressions (name or literal) and the *aggregated* ones (concatenation or factory call). In the case of aggregated expressions, the dataset is composed of dataset sub-expressions: the dataset *components* (arguments of a factory call or elements of a dataset concatenation). These dataset components must be sent to the parent object for its initialization. This process may of course be recursive, and therefore requires to use stack-oriented data-structures: datasetComponents is a stack of vectors of nodes (datasetComponents.top corresponds to the components of the current dataset) and strStack is a stack of strings to give names to new declarations; this stack is initialized by a string given as a constructor parameter by the object that uses a dataset expression visitor.

The algorithm to generate the declarations corresponding to dataset components is as follows:

```
makeComponents( (node) nbNodes)
    // node stands for dataset components (operands of a "+" or arguments
    // of a factory call)
    datasetComponents.push(new())
    for i = 1..nbNodes
      strStack.push(strStack.top + "_p" + i)
      visit(TDD_DatasetExpr : node)// recursive visit
      strStack.pop()
    endfor
    result = datasetComponents.top
    datasetComponents.pop()
    return result;
```

b/ Name (of a dataset)

add to datasetComponents.top the declaration:

“<name> <datasetName> = new <name> (<testContext>);”

```
FieldDeclaration : createdNode
    Type : fullName
    VariableDeclarator :
        Name : strStack.top
        AllocationExpression :
            Name : fullName
            Arguments :
                Name : <testContext>
```

Fields:

fullName = makeName(name) // name: current node

// makeName gives the complete dataset name (e.g. from “D2” to “S_auxlib_D2”)

Actions:

```
datasetComponents.top += createdNode
```

c/ Literal (singleton literal dataset)

“Cast” in a real literal dataset.

Actions:

visit(TDD_DatasetLiteral : currentNode)

d/ Name (of a constant)

Note:

considered as a literal

Actions:

```
    // "Cast" in a real literal dataset.
    visit(TDD_DatasetLiteral : currentNode)
```

e/ TDD_DatasetExpr (concatenation of datasets)

Example: // in tdd class "examples"

```
    D1 + D2 // D1 is defined in "auxlib" and D2 in the current tdd class
        -->
    ADLConcatenationSet <myname> =
        new ADLConcatenationSet(<testContext>, null);
    S_auxlib_D1 <myname>_p1 = new S_auxlib_D1(tc);
    S_examples_D2 <myname>_p2 = new S_examples_D2(tc);

    <myname>.addElement(<myname>_p1);
    <myname>.addElement(<myname>_p2);
```

Children nodes:

(TDD_DatasetConcatExpr : child)^{nbChildren}

Actions:

```
    #if ANONYMOUS
    FieldDeclaration : createdNode1
        Type : "ADLConcatenationSet"
        VariableDeclarator :
            Name : strStack.top
            AllocationExpression :
                Name : "ADLConcatenationSet"
                Arguments :
                    Name : <testContext>
                    StringLiteral : "null"
    datasetComponents.top += createdNode1
    #endif
    datasetComponents.top +=
        makeComponents( (child )^nbChildren )
    #if ANONYMOUS
    (   Statement : createdNode2
            "<strStack.top>.addComponent(<strStack.top>_p<i>);"
        datasetComponents.top += createdNode2
    )_i=1..nbChildren
    #endif
```

Notes:

- The special case "dataset T D1 = D2;" is considered as a concatenation of datasets with a single operand.
- When the concatenation operation is at the outermost level of a dataset declaration (e.g. "dataset int D = D1 + D2"), i.e. when the result is not "anonymous", only the "makeComponents" part of this algorithm is necessary: the ADLConcatenationSet declaration and the "addComponent" statements (in fact addElement for concatenation or addParameter for factory call) are performed by the caller, dataset or directive builder. These operations occur only in the case of complex expressions such as

"fact(D1, D2+D3)": here the expression "D2+D3" is anonymous and must therefore be declared as a dataset concatenation and initialized with the components D2 and D3.

f/ TDD_FactoryCall

Example:

```
Fact(D2)
    -->
F_examples_Fact <myname> = new F_examples_Fact(<testContext>, null);
S_auxlib_D2 <myname>_p1 = new S_auxlib_D2(tc);
<myname>.addParameter(<myname>_p1);
```

Children nodes:

Name : factName

(Node : factArg $)^{nbArgs}$

Actions:

```
#if ANONYMOUS
FieldDeclaration : createdNode1
    Type : factFullName // makeName(factName)
    VariableDeclarator :
        Name : strStack.top
        AllocationExpression :
            Name : factFullName
            Arguments :
                Name : <testContext>
                StringLiteral : "null"
datasetComponents.top += createdNode1
#endif
datasetComponents.top +=
    makeComponents( (factArg )^nbArgs )
#if ANONYMOUS
(   Statement : createdNode2
        "<strStack.top>.addComponent(<strStack.top>_p<i>);"
    datasetComponents.top += createdNode2
)i=1..nbArgs
#endif
```

Note:

ANONYMOUS: cf second note for concatenation above (Section e).

g/ TDD_DatasetLiteral / TDD_DatasetMember

The transformation is performed in three steps:

- initialize a vector with the values of the literal set
- transfer this vector into an array
- declare the dataset as an ADLLiteralSet constructed with this array.

Example:

```
{3, 5 .. (int)(Math.random()*10)+5, 20 .. (int)(Math.sqrt(1000))}
    -->
java.util.Vector <myname>_v = new java.util.Vector();
```

```
<myname>_v.addElement(new Integer(3));
int minexp, maxexp;
minexp = 5; maxexp = (int)(Math.random()*10)+5;
for (int i = minexp; i <= maxexp; i++)
    <myname>_v.addElement(new Integer(i));
minexp = 20; maxexp = (int)(Math.sqrt(1000));
for (int i = minexp; i <= maxexp; i++)
    <myname>_v.addElement(new Integer(i));
int <myname>_a[] = new int[<myname>_v.size()];
for (int i = 0; i < <myname>_v.size(); i++)
    <myname>_a[i] = ((Integer)<myname>_v.elementAt(i)).intValue();
ADLLiteralSet <myname> = new ADLLiteralSet(tc, <myname>_a);
```

Actions:

All these declarations and statements are to be generated in datasetComponents. We will not detail here the transformation, which is simple but tedious... Just note that in the case of a single-valued dataset literal, we generate a simple "<myname>_v.addElement" statement, whereas for a range dataset literal we generate a loop from the first value of this range to the second one.
Note also that if a dataset member calls a static method "meth" of the current tdd class, the generated code must call "<tddClassName>TDD.meth".

h/ Example

Let's consider the following dataset definition (in tdd class bTest):

```
dataset bank TEST =
     make_bank(DEPOSITS, BANKSIZE + DEPOSITS, INT_VALUES);
```

The transformation of this dataset declaration will generate a class S_bTest_TEST that inherits from the class that implements the factory make_bank, and where the dataset expression `make_bank(...)` is transformed into:

```
S_bTest_DEPOSITS elt_p1 = new S_bTest_DEPOSITS(tc);
// 2nd parameter: anonymous --> recursive call
ADLConcatenationSet elt_p2 =
     new ADLConcatenationSet(tc, null);
S_bTest_BANKSIZE elt_p2_p1 = new S_bTest_BANKSIZE(tc);
S_bTest_DEPOSITS elt_p2_p2 = new S_bTest_DEPOSITS(tc);
elt_p2.addElement(elt_p2_p1);
elt_p2.addElement(elt_p2_p2);
S_bTest_INT_VALUES elt_p3 = new S_bTest_INT_VALUES(tc);
// Components are then added as parameters of the factory
// make_bank:
addParameter(elt_p1); addParameter(elt_p2);
addParameter(elt_p3);
```

5.2.2.7 FactoryVisitor

The aim of this visitor is to visit a Block AST in order to replace primitive type expressions in return statements by an equivalent wrapper object (for instance "`return 0;`" must be replaced by "`return new Integer(0);`").
This class overrides only one visit method, for the ReturnStatement node:

Children nodes:
  [Node : returnExpr]
Created node:
  ReturnStatement :
    [AllocationExpression :
      Type : <JavaWrapperType(returnExpr)>
      Argument : returnExpr]

5.2.2.8 RemoveADLVisitor

The aim of this visitor is to replace ADL/ADL_new expressions that occur in directive bodies or test functions.

```
ADL(<object name>)<suffixes>*  --->
     new <object type>ACO(<object name>, this)<suffixes>*

ADL(<class name>)<suffixes>*  ---> // test of a static method
     new <class name>ACO(null, this)<suffixes>*

ADL_new <name>(<args>)<suffixes>*  ---> // constructor test
     new <name>ACO(null, this).<name>(<args>)<suffixes>*
```

This class only overrides the visit methods for nodes PrimaryExpression, TDD_ADLExpression and TDD_ADLnewExpression.

5.3 Unparsing

Generating the text of the generated file is comparatively simple. The only difficult part is laying out the source so it’s easy to read. Unparsing an AST, or other data structure, to produce human-readable text is often called “prettyprinting”.This may require two passes; one to count the length of a generated line, another to generate the line with any required linebreaks. Or, assuming the current column in the output stream is available to unparser method, a single pass where required linebreaks are generated when it becomes impossible to dump the next node.

According to the visitor pattern as described in introduction above, the unparser visitor provides a visit method for each node type. The visit method generates the text for the node, using hard and optional line breaks. The first unparse pass uses only hard line breaks; if a line of text is too long, then it is regenerated with optional line breaks.

For example: the code to unparse a “for” statement might be (assuming as above that the parts of the for statement are available as instance variables)

```
void visit(ForStatement fs) {
  boolean wrapped = false;
  if (currentColumn >= lineLimit - 5) {
     wrapped = true;
     newline(); // emit a newline and reset currentColumn
     addIndent(subIndent); // add subIndent to indent
     indent(); // emit the current indent
  }
  emit(“for (“); // emit the start of the for statement

  // ask the forInit member to unparse itself
  fs.forInit.jjtAccept(this);
  emit(“; ”);
  // ask the forTest member to unparse itself
  // it will take care of remaining space to decide whether
  // to emit a newline or not
  fs.forTest.jjtAccept(this);
  emit("; ");
  // ask the forUpdate member to unparse itself
  fs.forUpdate.jjtAccept(this);
  emit(")");

  fs.forBody.jjtAccept(this);

  if (wrapped) {
     delIndent(subIndent);
  }
}
```

EXAMPLE 4 Unparsing for statement

This code is certainly not complete; it may be improved by adding a special treatment when the body of the for statement is not a block. It relies on several information and methods in the current unparser visitor that are not described in this example. Additionally, it provides no code to deal with comments in the code. A specific section below describes the design for comments processing. Nonetheless, the pattern of the unparse code should be evident.

5.3.1 Unparser Visitor Classes

Unparsing is done by an unparsing visitor class, which is the dual of the parser for the AST type in question. Unparser visitors are derived from the abstract class GenericUnparserVisitor (see Figure 5). This class provides specific unparser methods with current unparser state information and generic methods.

The relevant declarations are:

```
abstract class GenericUnparserVisitor {
  ...
  protected PrintWriter out; // print stream to write to
  protected String outputDirectory; // root dir for output

  private StringBuffer currentIndent;

  protected int subIndent; // indent level for sub expr
  protected int regIndent; // indent level for stmt
  protected int currentColumn; // current col in print stream
  protected int lineLimit; // maximum length of line
  protected void addIndent(int extraIndent) { ... }
  protected void delIndent(int extraIndent) { ... }
  protected void indent() { ... }
  protected void setIndent(int newIndent) { ... }
  protected int getIndent() { ... }
  protected void newline() { ... }
  protected void emit(String str) { ... }

  private String addUnicodeEscapes(String str) { ... }

  protected GenericUnparserVisitor(String toolname,
                                   String version,
                                   int linelimit,
                                   String regindent,
                                   String subindent) {
  ...
  }

  protected GenericUnparserVisitor(String toolname,
                                   String version) {
  ...
  }
}
```

EXAMPLE 5 GenericUnparserVisitor Skeleton

A simple rule for writing the specific UnparserVisitor classes is that there should be a `visit` method for each grammar rule that consume a token; parse rules that do not consume token may not need emit methods for the corresponding Node class. If one Node class is used to represent more than one grammar rule, then the code for the emit method will have to distinguish those nodes by looking at the node properties.

Before a string can be emitted on the print stream, each character out of the standard ASCII character set should be replaced by its corresponding UTF-8 encoded character. This operation is performed by the `addUnicodeEscapes` method. Such replacement might be conditioned by a system property indicating the type of character set used by the underlying operating system (ASCII or UTF-8).

5.3.2 Comments Processing

If comments are to be reproduced in the generated code they require a special treatment during the parsing, and they need to be copied into the target AST generated by the AST transformation phase.

As far as parsing is concerned, the simplest way to handle comments is to apply the general pattern provided in the Java^{TM} Compiler Compiler^{TM} examples.

In this pattern, comments are usually parsed as SPECIAL_TOKENS.

As these special tokens get attached to the real token immediately following them, each non-terminal node needs in turn to reference its first token to be able to access potential comment strings preceding it.

Some terminals in the ADL grammar (generally keywords, operators and separators) may not be represented by nodes in the AST. They are however parsed as tokens and can therefore reference a preceding comment string. Ideally, each non-terminal node making use of such terminals in its production rule should keep a reference to those. But this is not always possible since the terminal may in fact be used in a different production rule than the one where the non-terminal node is built. The solution is then to keep the last token of a non-terminal, as shown in the examples above, so that it can be used to get the next token in the token chain, i.e., the terminal's token. Figure 7 depicts the actual links between the generated AST and the token chain for a simple input text. This input text has been chosen because it illustrates the problem of comments associated to terminals without corresponding nodes.

Input Text "(a+2 // comment 1
, b // comment 2
)"

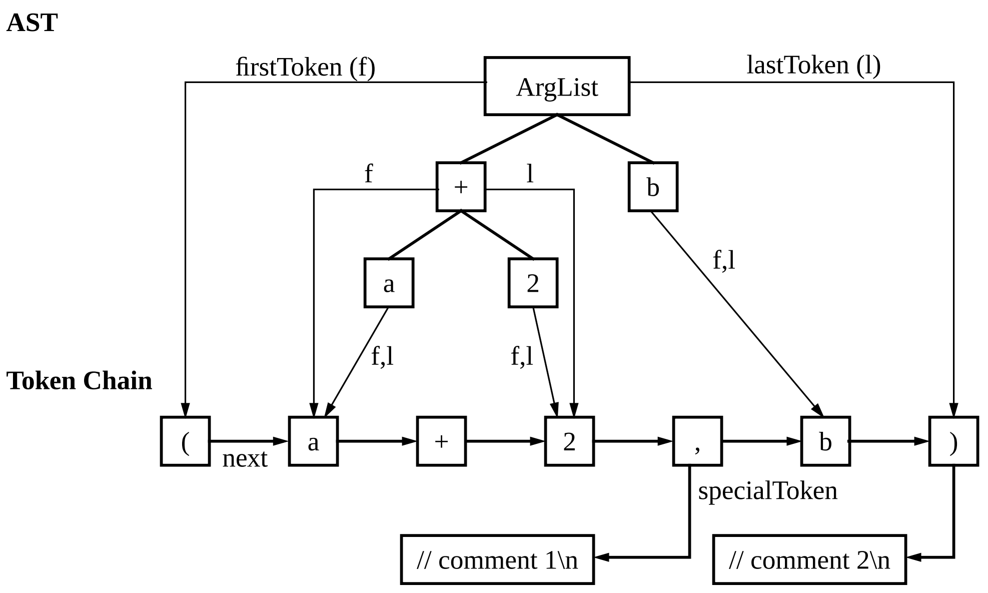


FIGURE 7. Links between AST and Token Chain

The typical algorithm used to unparse text in such a situation is kind of *token-driven*. Starting from the first token in the token chain, each token is unparsed in turn. This is rather incompatible with the *node-driven* algorithm suggested by the AST data structure and the visitor pattern model.

Furthermore, it should be noted that the depicted links are valid right after parsing. And while AST transformation can preserve them as much as possible, it will never be able to establish such links for generated nodes.

And another even more serious problem is related to the memory usage of the parser. As long as one non-terminal node keeps a reference to its first token, all remaining tokens returned by the lexer.from the parsed file are referenced and will therefore never be garbage collected, or at least until the tree is transformed. This means that a really huge amount of tokens are kept in memory, together with the whole AST. JavaCC users have already reported that this may cause Java to run out of memory.

For all these reasons, we have decided to currently not preserve comments in the generated code.

Chapter 6 Documentation Generation Architecture

6.1 Introduction

In addition to code to implement the test, ADLT2 generates natural language to describe the specification and data that it is to be tested with.

This chapter describes the requirements that the ADLT2 natural language generation subsystem should satisfy and the design of its implementation.

6.1.1 Generated documents

There are two kinds of document produced as output. The first is reference manual documents to describe the interface of the function under test. They are modeled on the standard Unix manual page entries and include descriptions of the function synopsis and the assertions from the ADL specification.

The second document type is the test description. This document describes the data that the function is to be tested against, and the sequence of annotated functions that constitute the test.

Some of the content of these documents comes from templates, some is generated automatically from the ADL and TDD sources, and some is directly provided by user annotations.

6.1.2 Expression translation

ADLT2 can generate natural language translations for expressions in ADL and TDD. The default translations are rather stilted and unnatural. The translations can be improved by providing alternative translations for source language identifiers. NLD annotations are the mechanism for associating translations with source identifiers.

Each natural language annotation provides information about the translation for an identifier. It does this in the form of a list of predicates that assert facts about the translation. Some predicates define the actual translation text for the identifier, while others provide contextual constraints such as the appropriate locale for the translation. SGML entities can also be declared in the predicate list.

6.1.3 Sentence construction

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 Prolog program is specific to the natural language being generated, since it has rules that construct sentences in the language. One candidate for the generation of Japanese sentences is the Language Toolbox, which is an existing Prolog system for generation of Japanese.

6.1.4 Internationalization

ADLT2 must support the generation of documentation in languages other than English.

ADLT2 provides internationalization mechanisms in three areas: the NLD translation annotations can be marked with their locale (by using a locale predicate), or stored in locale-specific files; the document templates can be locale-specific files; and the sentence generation rules can be modified to support a specific locale (such as the Japanese Language Toolbox).

6.1.5 SGML support

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.

ADLT supplies templates and synthesizes entities based upon the industry standard DocBook 3.0 document type definition for constructing reference manual pages and test specification descriptions.

The synthesized entity declaration are taken with user supplied entity declarations together with template entity declarations to produce complete SGML documents for subsequent processing. They can be converted with auxiliary tools into more conventional formats such as HTML and Unix manual page format. Alternatively, the SGML document can be incorporated into larger SGML documents.

The SGML that is generated is standard SGML and can be manipulated, transformed and rendered with standard SGML tools. Although ADLT2 will make use of freely available tools to convert the generated documentation to HTML and Unix manual page formats[1], users will be able to use their own SGML tools for custom processing.

1. See http://www.gr.opengroup.org/adl/papers/sgml_setup.html.

6.1.6 Abstract syntax tree interface

Natural language information need not be made available to other parts of the ADLT system. Hence there is no need to attach the translations for identifiers and expressions directly to nodes in the AST. Moreover, as it is described below, there is no AST while processing a NLD file.

6.2 General Architecture

Figure 8 below depicts how NLD annotations are processed in the ADLT system to generate the Natural Language Specifications (NLS) and Tests Description (NLTD).

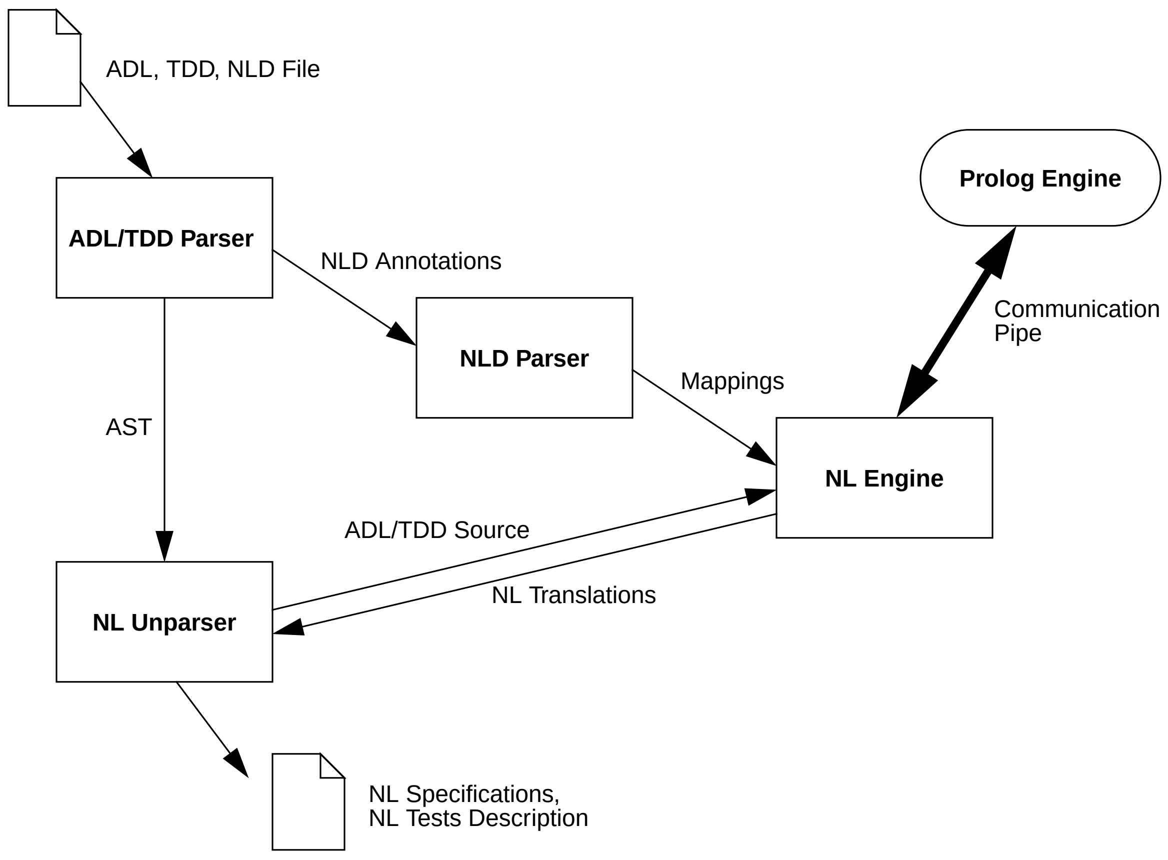


FIGURE 8. NLD Annotations Processing

Basically, the ADL compiler passes each input file to the ADL/TDD parser of the current language binding. NLD annotations are extracted from the input file and sent to the Prolog engine as mappings, i.e., associations between symbols in a given context and translations.

The resulting AST at the end of the parsing of the input file is then passed to the NL unparser which generates the associated documentation: NL specifications for an ADL file, NL Tests Description for a TDD file.

NLD files don't generate any AST. Their purpose is just to provide some additional mappings to feed into the Prolog engine. Since these additional mappings may be necessary to generate the documentation of any ADL or TDD file, NLD files are parsed before any other file. This is done by reorganizing the list of input files in the ADLT system driver class: `Adlt` (See "The Driver" on page 25.).

6.2.1 ADL/TDD Parser

The key idea in separating the ADL/TDD parser from the NLD parser is that the syntax of NLD annotations is almost the same for all binding languages. Each ADL/TDD parser ignores the NLD annotations and passes them to the NLD parser.

NLD annotations start with the `nld` keyword, followed by a brace-delimited block. The exact syntax is expanded below:

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

This syntax is used to define a token in each input language parser to ignore the NLD annotations while still keeping its value.

In case of a pure NLD file, no AST is generated by the input language parsers. The NLD annotations it contains are just used to extend the set of mappings of the Prolog engine.

Obviously, when the user has requested not to generate the documentation, the `NLD_Annotation` tokens are just discarded.

6.2.2 NLD Parser

The role of the NLD parser is to convert NLD annotations in the source files into proper calls to the NL Engine methods which extend the set of mappings known by the Prolog engine. For this task, there is no need to create an AST representing the annotations. The parsing consists in extracting the various information attached to an annotation: predicates, scope, translation; interpreting and passing them as arguments to the NL engine methods.

A single NLD annotation grammar is used for all supported language bindings. The syntax of NLD strings has hence been extended to cover both C, C++ and Java. For instance, it is possible in any case to define a concatenation string as in C++:

```
aMapping = "a first string" ", a second one concatenated";
```

or as in Java:

```
aMapping = "a first string" + ", a second one concatenated";
```

The complete NLD annotations grammar is reproduced in the language reference manuals for all bindings. An interesting feature of NLD annotations in ADL 2 is that they may define properties that characterize them, and be qualified with some additional predicates. This features is less important in English than in other languages. The example below demonstrates its usefulness on a French translation example (note that it does not use the exact syntax).

```
Color(Object[gender(male)]) = “la couleur du $1”,
                                  gender(female);

Color(Object[gender(female)]) = “la couleur de la $1”,
                                  gender(female);

Color(Object[number(>1)]) = “la couleur des $1”,
                                  gender(female);

Color(Object) = “la couleur de l’objet $1”,
                                  gender(female);
```

EXAMPLE 6 Use of predicates and properties for NLD annotations

While parsing the NLD annotations contained in the source files, the NLD parser also filters them to only retain those defined in the appropriate locale. The user’s locale, if specified, is available in the `adl.locale` system property. Otherwise, it defaults to the `C` locale.

6.2.3 NL Engine

Upon creation, the NL engine opens a connection to the Prolog engine using the `FilterProcess` class. It selects the appropriate set of rules to pass to the Prolog engine according to the user’s locale (see above). Rules are stored in the `$ADL2HOME/lib/nl/<locale>/nl_rules` file. Any user can provide additional locale-specific rules by putting them in the appropriate file and directory.

There is a single instance of the NL engine, with a permanent connection to the Prolog engine, during the entire lifetime of the ADLT compiler. It serves all requests for translations for all compiled files.

Upon creation, the NL engine sends a default mapping to the Prolog engine using the `assertz/1` predicate. This default mapping ensure that even un-annotated symbols will get a translation: the symbol name.

All other mappings are sent to the Prolog engine by using the `asserta/1` predicate.

Translation requests are sent to the Prolog engine using the `xlate/3` predicate.

6.2.4 NL Unparser

Since the NL unparser works on the AST generated by the input language parser, there should be one NL unparser for each binding language.

Its main purpose is to extract specification phrases as source code, and request the NL engine to translate them to generate the appropriate documentation. Basically a specification phrase is an assertion. From a node representing an assertion in the AST, the NL unparser visits its children recursively to build the specification phrase. Each node visited should return a string describing how to translate that node, which in turn is used by the parent node to build its own translation request string. When the visit of the children ends, the assertion node asks the NL engine to translate the resulting phrase.

The NL unparser also builds a complete documentation out of a template that is loaded from a source file in the `$ADL2HOME/lib/nl/<locale>` directory, according to the user's locale and the type of file that is currently unparsed: ADL or TDD. There is one template for each kind of documentation, i.e., one for NL Specifications named `nl_specs.tpl`, and one for the NL Tests Description named `nl_tests.tpl`. These templates define the general layout of a document using SGML and predefined entities referring to the various parts generated during the NL unparsing: class description, method description, method semantics, etc. The NL unparser class contains various placeholders where it keeps the generated translations for the predefined entities used in the templates.

6.3 NL Engine

The NL Engine is a Java class that is used by both the NLD parser and the NL unparsers as the single interface with the Prolog engine. Its role is to:

- launch a SWI Prolog process, initialize it with the *locale* prolog rules (`$ADL2HOME/lib/nl/<locale>/nl_rules`), and maintain a connection with this process. This is done via the `FilterProcess` class.
- during parsing, maintain information on the local scope in which an NLD annotation appears.
- allow communication with the Prolog engine for *mappings* (from the NLD parser) and *queries* (from the NL unparsers), manipulating strings accordingly to be coherent with the string formats used by the different modules.

TBD *Complete description of the NLEngine class.*

6.4 NL Rules

6.4.1 Basic Principles

Basically, the role of the Prolog engine is twofold: during NLD parsing it has to store *mappings* (association of a symbol with its translation) and during NL unparsing it takes as input an *expression* and the *context* of this expression, and returns the *translation* of this expression in this context, possibly using the stored mappings.

Consider for instance a variable "foo" in a class AClass, where foo is both a class field and the parameter of a method "bar"; suppose that in the ADL specification there is a mapping "a stupid translation for foo" for the parameter foo of bar (see the ADL for

Java or C++ Reference Manual, chapter 4), but no mapping for the field foo. Consider now the ADL assertion “foo == 0”. If this assertion is in the specification for method bar, it refers to the *parameter* foo, and the user wants this assertion to be translated to something like “a stupid translation for foo is equal to 0” in the generated NL specification document of AClass; conversely, if this assertion is in another method, it refers to the *field* foo, hence it has no proposed mapping, and the translation will be a default one “foo is equal to 0”.

This basic mechanism is extended in two directions: scope and inheritance.

- Saying that the parameter foo of method bar has a mapping is not accurate. In fact, there may be a mapping defined for “AClass.bar(int).foo”. When translating an assertion in bar that contains foo, the Prolog engine will search for such a mapping. If this mapping does not exist but there is a mapping for “AClass.bar(*).foo”, then this mapping has to be taken. If it does not exist, search for “AClass.*.foo”, and so on.
- In Java and C++, we must take inheritance into account. If there is no mapping defined for field foo but foo is in fact a field from class ASuperClass, ancestor of AClass, and there is a mapping for ASuperClass.foo, then this mapping must be used.

The inheritance mechanism has a higher priority than the scope mechanism. In the case of the assertion in bar that involves foo, and if there is no mapping for “AClass.bar(int).foo”, then a mapping for “ASuperClass.bar(int).foo” will be preferred to the mapping “AClass.bar(*).foo”.

6.4.2 Mappings

During NLD parsing, the mappings defined in NLD annotations are stored in a Prolog database as clauses “map” via the “assert” pre-defined clauses.

The clause map has the following signature:

```
map(Symbol, Scope, Translation, UserPred, SystemPred)
```

where Symbol is the symbol’s name, Scope represents the scope of the symbol as it is defined in the NLD annotation, Translation is the proposed translation for the symbol. UserPred and SystemPred are for user/system predicates and will be discussed later (see Section 6.4.4).

Example:

```
asserta(map(‘maxact’, [‘#<EXTERN>’, ‘bank_example’, ‘bank’,
      [‘bank’, ‘(int)’]], [‘le nombre max de comptes’],
      [], [])).
```

This defines a translation “le nombre max de comptes[1]” without predicates for “`bank_example.bank.bank(int).maxact`”, i.e. the variable maxact defined

1. the max number of accounts

in the constructor bank(int) of class bank in package bank_example. #<EXTERN> is a "trick" to unify with C++. Single quotes are used to avoid clashes with Prolog variables.

For methods and functions, we must also precise how the translation of the arguments is used in the translation of the method call, and if the mapping is defined for arguments that obey some predicates.

```
asserta(mapfn('is_active(bank, long)',
      ['#<EXTERN>', 'bank_example', 'bank'],
      ['le compte', 2, 'de la banque', 1, 'était actif'],
      [['female'], []], [], ['#CALLSTATE'])).
```

This defines a mapping for `bank_example.bank.is_active(bank, long)` when the first argument has the predicate "female" and the method call occurs in the scope of a call-state. In such a case, the expression `@is_active(aBank, 100)` where aBank has a mapping "Société Lyonnaise" and predicate female will be translated as "le compte 100 de la banque Société Lyonnaise était actif[1]".

Finally, there is also a simple clause for NLD entities:

```
asserta(mapEntity('bank', ['#<EXTERN>', 'bank_example'],
  description, 'Class defining the interface of a bank.')).
```

This defines the entity "description" for class `bank_example.bank`.

6.4.3 Translations

The basic Prolog clause for translation is "natural":

```
natural(+Expr, +Ctxt, -Transl, +ScopeList,
      -UserPredList, +SystemPredList)
```

Note: we follow the Prolog usage: an argument denoted as "+" is passed to the clause, whereas an argument denoted as "-" is computed by this clause.

Expr is the expression to be translated, Ctxt represents the local scope (the enclosing method, for an assertion) and ScopeList the current class and its ancestors. Transl is the returned translation and UserPredList a list of user predicates, used for translation of method calls.

6.4.3.1 Mapping Resolution

When the expression to be translated is a simple symbol, the "natural" clause calls the "search" clause (which has the same signature).

```
search('maxact', ['#<CLASS>', ['bank', '(int)']], Transl,
  [['#<EXTERN>', 'bank_example', 'bank'],
   ['#<EXTERN>', 'java', 'lang', 'Object']],
  UserPredList, SystemPredList).
```

1. account 100 of bank Société Lyonnaise was active

The second argument (Ctxt) means “the current scope is the method bank(int) of the current class” and the fourth one (ScopeList) describes this current class and its ancestors.

This clause will first call the searchInherited clause, that will try to detect a mapping for “bank(int).maxact” in the current class or its ancestors. This is done by replacing the keyword #<CLASS> by the complete class name (first element of ScopeList) and search for

```
map(’maxact’, [‘#<EXTERN>’, ‘bank_example’, ‘bank’,
     [‘bank’, ‘(int)’]], Transl, UserPredList,
     SystemPredList).
```

and if this map does not exist in the assert database, then replace bank_example.bank by the next element in ScopeList (java.lang.Object here), and so on.

Note that even if there is no mapping defined for maxact, this call to map will return for argument Transl the default translation: maxact, i.e. the symbol itself. Hence, saying “the map does not exist” is more precisely “the map is the same as the given symbol”.

If searchInherited has failed (i.e. the returned translation is still the initial symbol), then we trigger the second searching mechanism: scopes. For this, we first use a clause that “enlarges” the Scope argument and we re-run the search clause.

In the previous example, enlarging the scope means replacing [‘#<CLASS>’, [‘bank’, ‘(int)’]] by [‘#<CLASS>’, [‘bank’, ‘(*)’]], hence searching for a mapping “bank_example.bank.bank(*).maxact”. The next steps would be:
[‘#<CLASS>’, ‘*’] for “bank_example.bank.*.maxact”
[‘#<CLASS>’] for “bank_example.bank.maxact”

For a further scope enlargement, ScopeList must also be modified:
[‘#<CLASS>’], [[‘#<EXTERN>’, ‘bank_example’, ‘bank’], [‘#<EXTERN>’, ‘java’, ‘lang’, ‘Object’]] must be changed in [‘#<CLASS>’], [[‘#<EXTERN>’, ‘bank_example’, ‘*’], [‘#<EXTERN>’, ‘java’, ‘lang’, ‘*’]] in order to search for “bank_example.*.maxact”, and so on.

The search for method mappings is similar, except that the “searchfun” clause has one argument more than search, an argument that deals with argument predicates. See Section 6.4.4.

6.4.3.2 Decomposition Algorithm

The sentence that is passed from the NL unparser to the Prolog engine corresponds to an assertion, i.e. a complex expression. This expression is decomposed by “natural” rules, most of them very simple (such as <the translation of expression “e1 + e2”> is <the translation of expression “e1”> “plus” <the translation of expression “e2”>). The “search” clauses are used for atomic expressions.

There are also rules to deal with ADL specificities, for instance to translate “thrown(e1, e2)” by “at least one of the following exceptions is thrown: e1, e2”.

Note that such rules depend on the locale and should be translated if one wishes to rewrite them for a different locale.

6.4.4 Predicates

There are two kinds of predicates, and their use in the translation rules require different behaviors. A user predicate is an “ascendant” information: it is present in the mappings and goes “up” until a clause uses it, whereas a system predicate is a “descendant” information that goes “down” to the mapping.

6.4.4.1 User Predicates

```
asserta(map(‘aBank’,
     [‘#<EXTERN>’, ‘bank_example’, ‘bank’],
     [‘Société Lyonnaise’], [female], [])).
```

(bank is a feminine noun in french...)
This mapping defines a translation for field aBank of class bank_example.bank and says that it has a “female” predicate. Hence, when the search clause will find this mapping, it will also transmit this information.

In an assertion such as “aBank != null”, this information on the female predicate of aBank has no use, but in an assertion that contains the expression “is_active(aBank, 100)”, a clause “mapfun” is activated:

- mapfun first searches the argument aBank and therefore the information “aBank has female predicate” is synthesized. No predicate is found for the second argument.
- this information is propagated up to mapfun.
- mapfun then searches for a method “is_active” with the information “the first argument has female predicate” propagated down to the map clause, which will unify with the mapping defined for is_active in Section 6.4.2.

6.4.4.2 System Predicates

The mechanism for user predicates purely lays down on Prolog unification, that is why the user can freely define its own predicates (in the previous example, one just has to ensure that the word “female” used in the mapping for aBank is exactly the same as the word used in the mapping for is_active, otherwise they will of course not be unified...)

The mechanism for system predicates is different because it is triggered by the NL rules. In ADLT2, two system predicates are defined, one for negation and one for call-state. When entering an expression that begins by “!” (resp. by “@”), the “natural” clause adds the keyword #NOT (resp. #CALLSTATE) to the argument SystemPredList. These predicates can be used in two different ways:

- either by a “natural” rule (for instance in the rule for expression “e1 > e2”, if #NOT is in the SystemPredList, then the returned translation will be “the translation of e1 is less than or equal to the translation of e2”. Note that the translation for sub-expressions is launched *without* #NOT in their SystemPredList).

- either by the mapping resolution if a mapping was defined with a system predicate in the NLD annotation (unification with the “map” clause).

6.5 Document Templates

TBD *Definition of the default templates for NLS and NLTD.*

Chapter 7 Runtime Architecture

7.1 Introduction

The ADL 2.0 runtime architecture broadly defines two entities; the ADL runtime library and the test program code generated by the ADL Translator. The code generated by the ADL translator is heavily dependent on the ADL runtime library to report the results of a test to the user, to control test execution, and to assist with data generation. The linking of the generated code, the implementation being tested and the ADL runtime library is the end goal of the ADL Translator; a portable executable program that exercises and verifies that a specific implementation conforms to its test and interface specifications.

7.2 Runtime Architecture Goals

Reporting is the end goal of test creation; the test report represents the value of the test program. Some of the design goals of the runtime architecture are:

Minimal interference. The reporting scheme should minimize the impact on the tested code. For example, the reporting mechanism should minimize its use of potentially scarce resources like memory and (on Unix) file descriptors.

Unattended operation. ADLT is targeted at the acceptance test market; hence it is important that an ADLT test be able to operate in batch mode, with after-the-fact inspection of results.

Minimal code generation. The code generated by ADLT should take extreme advantage of the runtime library. Minimal code should be emitted by ADLT to create a test program.

Preservation of user written test code. An ADL generated test will contain rewritten user test code. The rewritten code should preserve as much as possible the syntax of the original input. The user should be able to easily identify original source from perusal, compilation and execution of the rewritten source code.

7.3 Design Notation

- The source examples and design description contained in this chapter are written in the Java language. Any implementation details of a particular target language (C, C++, IDL, or JAVA) relevant to this design description will be noted.
- Most of the diagrams presented in this chapter loosely follow the Unified Modeling Language(UML)[1].
- In presenting TDD examples

  - comments are bold italicized: ***`// this is comment in TDD examples`***

  - tdd specific syntax is bold: **`test`**`(int a = DS){ //...}` ***`// this is`***
  `// a test directive`

  - all other source is in normal “code” font

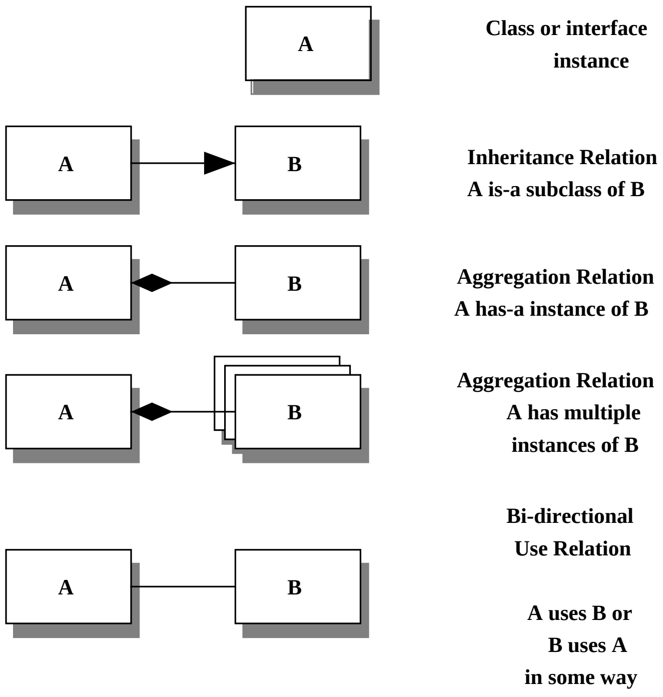
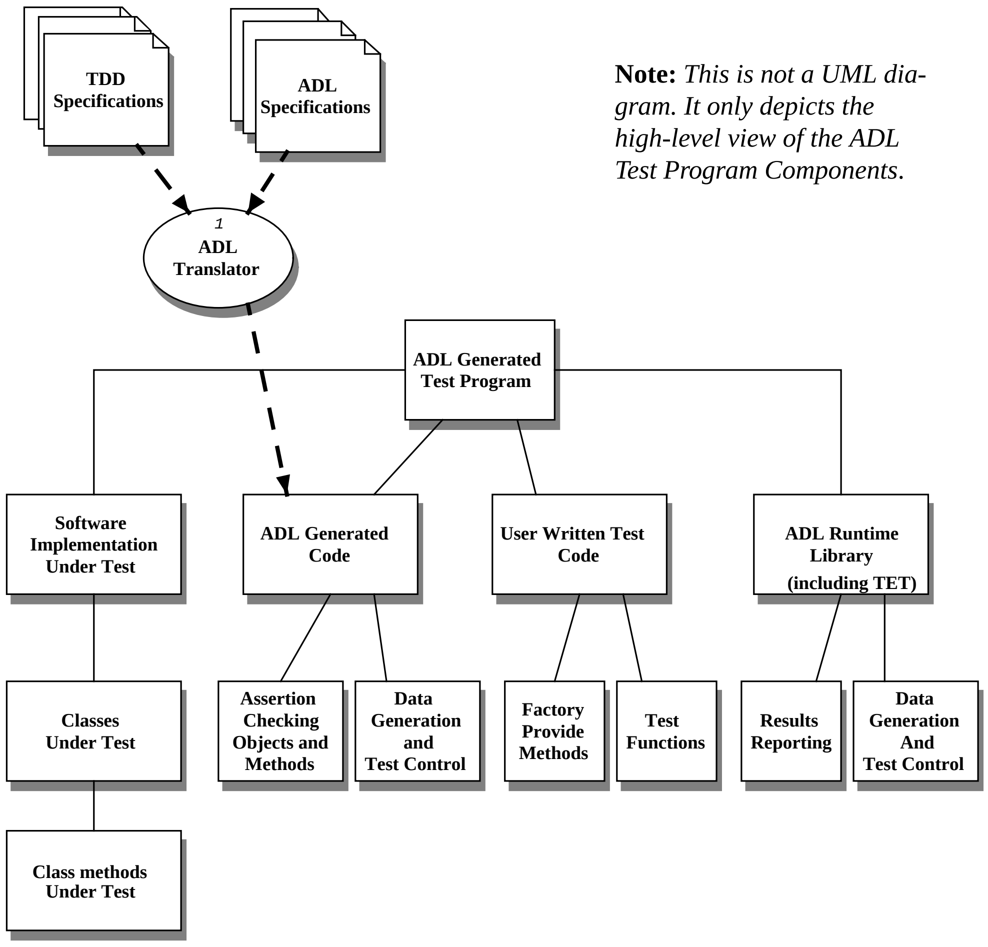

A
Class or interface instance
A
B
Inheritance Relation
A is-a subclass of B
A
B
Aggregation Relation
A has-a instance of B
A
B
Aggregation Relation
A has multiple instances of B
Bi-directional Use Relation
A
B
A uses B or B uses A in some way

FIGURE 9. UML Notation Key

1. Booch, G.; Jacobson, I.; and Rumbaugh, J.: “The Unified Modeling Language for Object-Oriented Development,” Documentation Set Version 0.91 Addendum UML Update, September 1996.

7.4 ADL Generated Test Program Design

The ADL generated test program is a self contained executable that embodies the test and verification intentions of the input ADL and TDD specifications. It consists of user written test code, the implementation under test, the ADL runtime library and the generated test code from the ADL Translator.


TDD Specifications
ADL Specifications
1
ADL Translator
ADL Generated Test Program
Software Implementation Under Test
ADL Generated Code
User Written Test Code
ADL Runtime Library (including TET)
Classes Under Test
Assertion Checking Objects and Methods
Data Generation and Test Control
Factory Provide Methods
Test Functions
Results Reporting
Data Generation And Test Control
Class methods Under Test

Note: *This is not a UML diagram. It only depicts the high-level view of the ADL Test Program Components.*

FIGURE 10. ADL Test Program Components

Some of the source code that comprise an ADL test program is generated by the ADL translator from input TDD specifications and optional ADL interface specifications. There is no dependence on ADL annotated functions for generating a useful working test. The Assertion Checking Functions (ACF) or Assertion Checking Objects (ACO) generated by ADLT check an implementation against its corresponding ADL specification. An ACO is a mirror image class definition of an ADL annotated object that contains the same method names and signatures of the ADL specified original. The methods of the ACO object are the delegating ACF methods that test whether the corresponding methods of the object under test conform to their ADL specification. A constructor of the ACO takes the object under test as an argument. After the ACO object is created with this constructor, the calls to methods of the ACO will invoke the corresponding (ACF)s of the method under test. In the ADL Translation System, Version 2.0 the generated ADL verification methods are available to the user to be called explicitly from the test code in a TDD specification file. ACFs and ACOs offer a more thorough and in most cases a more convenient method of ensuring expected behavior of the system under test. However, all verification and assertion checks can be explicitly written in the TDD test code. If desired, ADL specifications and generated ACOs can be ignored entirely. It is up to the user to determine what is more effective for their testing and specification requirements.

7.5 Overview of TDD Language constructs

ADLT generates a test program from an input TDD file. Specifically, a single test program is generated for each of the TDD test directives defined in a TDD environment scope. All user written source code appearing inside a TDD environment class or scope is rewritten or transformed as various output source code files that make up the test program. All true target language source code appearing in a TDD file will remain unaltered except for some special defined TDD library methods which are slightly remapped to include original source location and contextual information. The TDD library methods explained in Section 7.7.2, “Test Reporting Library,” on page 125 are available to test writers to affect the control, reporting and outcome of a test.

7.5.1 The TDD environment scope

All TDD declarations occur inside a TDD environment scope. In ADL for Java and ADL for IDL the TDD environment scope is declared as a tdd class:

```
// TDD environment scope definition (in Java)
public tddclass simpleTest {

  // declare a dataset
  dataset int VALUES = {1,2,3,4,5};

  // declare a test directive
dir1 : test(int i=VALUES) {
    // a sample TDD assertion check
    tdd_assert (”the compiler is hosed”,
                i * 0 == 0);
  }
}
```

In all other ADL target languages, the TDD environment scope is declared as a named source block.

EXAMPLE 7 Sample TDD Environment scope declaration (for Java)

```
// TDD environment scope definition (in C/C++)

simpleTest:tdd {

  // declare a dataset
  dataset int VALUES = {1,2,3,4,5};

  // declare a test directive
  test1 : test(int i=VALUES) {
    // a sample TDD assertion check
    tdd_assert (”the compiler is hosed”,
                i * 0  == 0);
  }
}
```

EXAMPLE 8 Sample TDD Environment scope declaration (for C/C++/IDL)

In a TDD environment scope you can declare test data (as TDD datasets), and TDD tests using test directives. In addition, you can include ordinary methods used in calls from a test directive. Except for some minor additions, the source code in the TDD environment scope is expressed in the target language.

In the implementation of the generated test, the TDD environment scope maps to a single class definition (in ADL for Java and ADL for IDL). Below is the Java output of ADLT that represents the TDD environment scope class presented in simpleTest Example above:

```
//*********************************************************
//   File Generated by ADLT Version 2.0
//
//   Date: Feb 4, 1997 20:58:47
//   TDD file: simpleTest.tdd
//*********************************************************
public class D_simpleTest_test1 extends simpleTestTDD {

  /* some snipped house keeping declarations */

 public void runInstance (Object[] paramValues)
              throws Throwable {

     int i = ((Integer) paramValues[0]).intValue();
     infoline(“Parameter int i = “ + i);

     /*USER CODE STARTS HERE*/
     tdd_assert (“the compiler is hosed”,
                  i * 0 == 0);
  }
}
```

EXAMPLE 9 ADLT Generated TDD environment class file simpleTest.java

7.5.2 Datasets and Factories

One of the main goals of the ADL System is to offer an easy and powerful way to express the number and type of data to be used in the testing of interfaces or software under test. The fundamental data specification construct in TDD is the dataset. The dataset is used to describe a set of values of a particular type that are to be used consecutively in a test of a system under test. In TDD there are several kinds of datasets; A literal, an expression, and a factory. Each type of dataset expresses a finite set of values that can be used as parameter input to test directives or as input to factories.

```
public tdd class bankTest {
  // ====================================================
  //       literal integer dataset
  // ====================================================
  dataset  int   DEPOSITS    = {0,1,10,100};
  dataset  int   WITHDRAWS   = {0,1,15,100};


  // ====================================================
  //      expression data set of type bank
  // ====================================================
  dataset bank BANK_VALUES = {new bank(1),
                              new bank(2),
                              new bank(3)};
  // ====================================================
  //      a factory that creates objects of type
  //      bankAccount.
  // ====================================================
  factory bankAccount
  make_account  ( bank theBank,
                  int initDeposit) {

    if (theBank == null) {
      tdd_end_case("can't test with null bank object");
      return null;
    } else {
        return theBank.open_account (initDeposit);
    }
  }

  // declaring a dataset that uses a factory
  dataset bankAccount BA1 =
      make_account(BANK_VALUES,DEPOSITS);
}
```

EXAMPLE 10 Sample TDD Dataset declarations

A TDD factory is a special type of dataset used to define a function that generates an instance of data based on the values of its parameters. A TDD factory definition can optionally include a relinquish clause that is called at the end of a test instance to allow the reclamation of any resources that are consumed by the generated data. Below is an example of a TDD factory definition:

```
public tddclass bankTest {

  factory bank
  make_bank (  int maxAccts,
               int numAcctsActive,
               int initDeposit) {

      bank ret = new bank(maxAccts);
      for (int x=0;x < maxAccts && x < numAcctsActive;x++) {
         try {
            bankAccount ba = ret.open_account(initDeposit);
         }  catch (bankException be) {
            tdd_result (ADL_FAIL,"exception caught on open");
            tdd_end_case(null);
         }
      }
      return ret;
  } relinquish (bank b) { // optional relinquish clause
      if (numAcctsActive > 0) {
         b.close_all_accounts();
      }
  }
}
```

EXAMPLE 11 Sample TDD factory definition

All TDD reporting methods defined in Section 7.7.2, “Test Reporting Library,” on page 125 are available to the user when writing the body of a factory. The optional relinquish clause has similar syntax to a Java catch clause. The relinquish clause takes a single argument whose type must match the return type of the factory method. In the body of the relinquish clause, the user has visibility to all the arguments of the factory method and the system guarantees that values used for the arguments in the preceding call to the factory method to create the return data, are the same when executing the call to the relinquish clause.

7.5.3 Test Driver and Test Directives

The test driver is the “main” of the ADL generated test program. It is responsible for program initialization, test execution and the gathering and processing of the test results information generated from the successive runs of test instances. The test driver can run one or more test directives that are declared inside a TDD environment scope.

A test directive is declared in the TDD specification file and specifies a list of data set values which are assigned to normal target language identifiers of the appropriate type. The body of a test directive is ordinary target language code that can use and refer to the current values of the specified data set arguments. Below is an example TDD file that demonstrates a test directive declaration:

```
public tddclass bankTest {

  //(see dataset declarations in Example 10 above.)

  // example of a TDD test directive
  test_bank_3:  test (  bankAccount ba=BA1,   <--  Dataset
                        int d = DEPOSITS,     <--  Arguments
                        int w = WITHDRAWS) {  <--

     int save_val = ba.balance();

     // invoke the Assertion Checking Object
     //for the bankAccount Object ba to call the
     // Assertion Checking Functions (ACF)s
     // deposit and withdraw.
     ADL(ba).deposit(d);
     ADL(ba).withdraw(w);

     // call tdd_assert method when both d and
     // w are equal to ensure the saved balance
     // is equal to the current balance

     if ( d == w) {
        tdd_assert(“ba.balance() == save_val“,
                        ba.balance() == save_val);
     }
  } // end of test_bank3 test directive
```

EXAMPLE 12 TDD Test Directive declaration

In the execution of a test, the test driver iterates over the dataset arguments, generates or selects the appropriate values and assigns the values to the argument identifiers. The test driver then uses these data to exercise the interface(s) referenced in the test directive. This continues until all dataset values are exhausted.

7.5.4 Invocation of Assertion Checking Objects and Assertion Checking Functions

As previously noted, an ADL generated test program does not implicitly invoke an ACF or an ACO as was the case in ADL Translation System, Version 1.0. In 2.0, ADLT generated ACOs and ACFs must be explicitly invoked by the user from either a test directive or some test code function inside the TDD environment scope. This is accomplished by calling the TDD defined method ADL (<object under test>) that creates and returns the ACO object. The returned ACO object has exactly the same method signature as the object being tested and indeed is a true delegation object that will ultimately forward the method call to the object and method under test.

In Example 12, "TDD Test Directive declaration," above, we see the use of the **ADL**(Object) method being used to call the ACFs for a previously created bankAccount object:

```
...
     ADL(ba).deposit(d);
     ADL(ba).withdraw(w);
...
```

The ADL method returns the ACO object wrapper of the object being tested and the ACF of the method under test is invoked directly. If the goal of a test directive is to test a single ACF, the ADL method can be used in a test directive:

```
public tddclass bankTest {
...

  // Note: the names of the datasets BA1 and WITHDRAWS are
  // used explicitly in the test directive below

  test_bank_4: test (bankAccount b = BA1, int w = WITHDRAWS)
     ADL(b).withdraw(w);
}
```

If testing a constructor of an object is desired, the system generates special constructor methods which create an object under test. test the conformance with the ACF and return the object as a result of the call to the special constructor method. The system creates the same number and signature of the test constructor methods as contained by the source ADL annotated class definition. The testing constructor methods are named using the following naming convention:

```
     ADL_new <class_name>(<constructor arguments>)
```

For example you can test the constructor and ACF in a single directive like:

```
public tddclass bankTest {
...
   test_bank_4: test (int d = DEPOSITS, int w = WITHDRAWS)
      ADL(ADL_new bankAccount(d)).withdraw(w);
}
```

7.6 Internal Runtime Architecture Design

The ADL Translation System, Version 2.0 Runtime Architecture relies heavily upon the Test Environment Toolkit (TET). TET supplies the command line handling, results tracking, and result reporting functions. Thus, the Runtime Architecture of this system is relatively straightforward:

- Each Test Directive maps to a single, executable Test Case.
- Each Test Data Instance of a Test Directive is an Invocable Component (IC) of that Test Case. The IC number is determined by assigning an ordinal number to each unique combination of Test Data.
- Within each IC there is a single Test Purpose. That Test Purpose is the sequence of testing defined in the Test Directive.
- For each assertion represented by that Test Directive, a variety of infoline calls will be made to report intermediate status information. When the assertion is complete, the ACO will call assertionResult with the result of the execution of that individual assertion. TET will automatically handle determining the aggregate result from these intermediate results upon completion of the Test Directive for that Test Data Instance.

The code to implement this policy is made possible by a variety of extensions to TET that were made in version 3.2.

Within each Test Purpose of a generated test, the structure will be something like:

1. Determine the test data instance by evaluating the ordinal IC number against the set(s) of relevant test data sets. Output the selected test data items via tet_infoline.
2. Call the appropriate provide factory methods with the appropriate arguments. The factory methods are encouraged to make use of TET primitives to output information, synchronize with remote processes, etc.
3. Execute each ACO referenced in the Test Directive.
4. Within each ACO, evaluate each assertion not tagged as “untestable”.
5. Call the appropriate relinquish factory methods.

7.6.1 Java Internals

The architecture of the Java runtime environment for the ADL Translation System version 2.0 is depicted below.

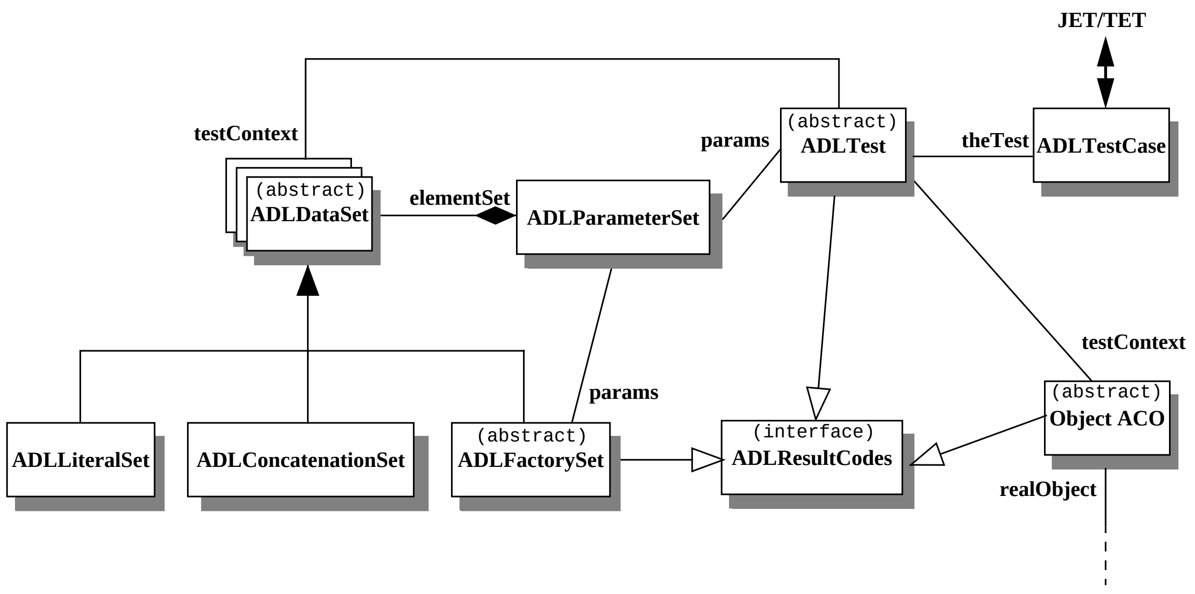


FIGURE 11. Architecture of the Java runtime environment

Basically, there are three major classes: `ADLTest`, `ObjectACO` and `ADLDataSet`. The first one is the root class for all test directives. The second one is the root class of all assertion checking objects and the last one is the root class for all TDD datasets. When used as parameters, e.g., for a test directive or a factory, TDD datasets are grouped in an `ADLParameterSet`.

In addition to these classes, the `ADLTestCase` makes the link with the TET environment through JETpack (Java Enabled TETware Package). And `ADLResultCodes` provides the result codes used to report to TET. It is implemented as an interface which all the classes that needs them *implements*.

Many of these classes are abstract since they should never be instantiated directly. They are subclassed by generated code which will effectively be instantiated.

The following sections describe the major classes of this architecture.

7.6.1.1 Class `ADLTest`

The highest level interface between the generated test cases and the runtime system is the `ADLTest` class. This class has within it all of the methods that a generated test will use. These methods include:

`void addParameter` - adds an `ADLDataSet` type parameter (see below) to the matrix of test data parameters available to this test. This method will be used by the generated test to inform the test about the test data that it has available to exercise the interface(s) under test.

`void infoline` - sends a String to the journal file.

`String getvar` - gets a configuration variable value from the Test Environment Toolkit.

`void setblock` - increments the block number for the Test Environment Toolkit.

`void result` - reports an intermediate result to the Test Environment Toolkit.

`void assertionStart` - indicate to the runtime that the processing of an assertion has started.

`void assertionResult` - posts a result value for an assertion. This method will be used by generated tests to inform the runtime system of the results of each assertion evaluation.

`void factoryResult` - posts a result value and an optional message to the journal from within a data factory. This method should only be used to indicate some wort of warning or failure encountered while setting up the test parameters.

The `ADLTest` class also implements the test reporting methods described in Section 7.7.1 on page 124.

The ADLT compiler generates one subclass of `ADLTest` for each TDD file, and one subclass of this for each test directive. Test functions are placed in the `ADLTest` subclass. Test directive subclasses define a `main` method so that they are invocable.

Static constants in a TDD file are placed in an interface which any class representing a dataset or factory using them implements.

7.6.1.2 Class `ADLDataSet`

An `ADLDataSet` represents a TDD dataset, i.e., a variable with a provide method to get a value from an ordinal number, and a relinquish method to free that value. It is generally labelled and is used within a test context. It is an abstract class that cannot be directly instantiated. It is subclassed as: `ADLLiteralSet`, which represents datasets made of static values; `ADLFactorySet`, which is an abstract class for factories which build values at each invocation based on their parameters (that can be `ADLDataSet`s); and `ADLConcatenationSet`, which is built by concatenating other datasets. Each subclass defines the following abstract methods:

`Object provide` - provide the nth value in the dataset.

`boolean relinquish` - free the nth value in the dataset (exact meaning of *free* depends on the actual dataset).

These two methods are called by the `ADLParameterSet` attached to an `ADLTest` or `ADLFactorySet`.

To provide dataset reusability, except when the dataset is anonymous, the classes `ADLLiteralSet` and `ADLConcatenationSet` are not instantiated directly, but rather subclassed.

The scheme for a factory is to generate one subclass of `ADLFactorySet` to define the actual provide and relinquish functions, and one subclass of this for each dataset built using the factory. The `ADLFactorySet` class also implements all the test reporting methods described in Section 7.7.1 on page 124, except `tdd_assert`.

7.6.1.3 Class `ObjectACO`

The `ObjectACO` class is an abstract class from which any generated ACO inherits. It just provides them with a generic constructor that gets some information from the TET runtime environment (reporting level selected) and some attributes which are useful when evaluating assertions and dealing with inherited specifications.

Generated ACOs define their own constructor - which just calls super - and assertion checking wrappers deduced from the specifications.

The `ObjectACO` class also provides the runtime environment with the methods to generate an enumeration for integer ranges of all kinds (ADL`_short_range`, ADL`_int_range` or ADL`_long_range`).

7.6.2 C++ Internals

The overall architecture of the C++ runtime environment is similar to the Java one depicted in Figure 11 to a large extent. The slight differences come from the different nature of both languages.

For instance there is no interface concept in C++ but rather multiple inheritance. Hence static constants in a TDD file are in this case placed in a class from which any class representing a dataset or factory using them inherits. And for the same reason the ADL standard result codes are placed in a header file used by both the C++ and C runtime environments.

Another slight difference is due to the lack of the equivalent of the Java `String` class in C++. This makes formatting messages a lot more tedious. To avoid having to deal with formats and memory, a additional class has been defined: `ADLString`. Its use is described below.

7.6.2.1 Class `ADLString`

This class implements the standard operators generally used with C++ streams to make formatting message easier. It is an extension of the standard `ostrstream` class where the `<<` operator has been redefined to return an `ADLString&` instead of an `ostream&`.

It also defines a `reset` method to reinitialize the string to be formatted. When `reset` is not called, remaining arguments are appended to the previously formatted message.

Finally it defines a set of conversion methods to and from a `char*`.

The `ADLString` class is instantiated only once for a static instance called `adlstr` (just like `cin`, `cout` or `cerr` in the stream classes). The typical use is demonstrated below:

```
infoline(adlstr.reset() << anInt << “ and “ << aFloat);
```

7.6.3 C Internals

The C Language has a flat namespace. This means that it is not possible to isolate the methods associated with the runtime within the various classes, as in the other runtime environments. Consequently, all of the C Runtime methods are prefixed with the string “ADL_” for ADL-related functions, and “tdd_” for methods that are called only from within the TDD test declarations. The “tdd_” prefix is lowercase because it is likely that these methods will be called directly by users in the specifications and should be as easy to type as possible. The “ADL_” functions are most likely only called by the generated code, although the ADL_infoline and ADL_printf functions might be called in user-written provide functions. The functions available within the C Language runtime include:

`void ADL_infoline` - sends a char * to the journal file.

`void ADL_printf` - sends a formatted char * to the journal file.

`char* ADL_getvar` - gets a configuration variable value from the Test Environment Toolkit.

`void ADL_setblock` - increments the block number for the Test Environment Toolkit.

`void ADL_result` - reports an intermediate result to the Test Environment Toolkit.

`void ADL_assertionStart` - indicates to the runtime that processing of an assertion is starting.

`void ADL_assertionResult` - posts a result value for an assertion. This method will be used by generated tests to inform the runtime system of the results of each assertion evaluation.

`void ADL_factoryResult` - posts a result value and an optional message to the journal from within a data factory. This method should only be used to indicate some wort of warning or failure encountered while setting up the test parameters.

And all reporting methods defined in Section 7.7.1 on page 124.

7.7 External Runtime Architecture Design

7.7.1 Test Result Reporting

The final result generated from an ADL test is computed from the tabulated results of all test instances executed from a given run of the test. The results are computed as a hierarchy of granularity ranging from the program result as the least granular to an assertion result as the most granular level. The following diagram depicts the test results hierarchy

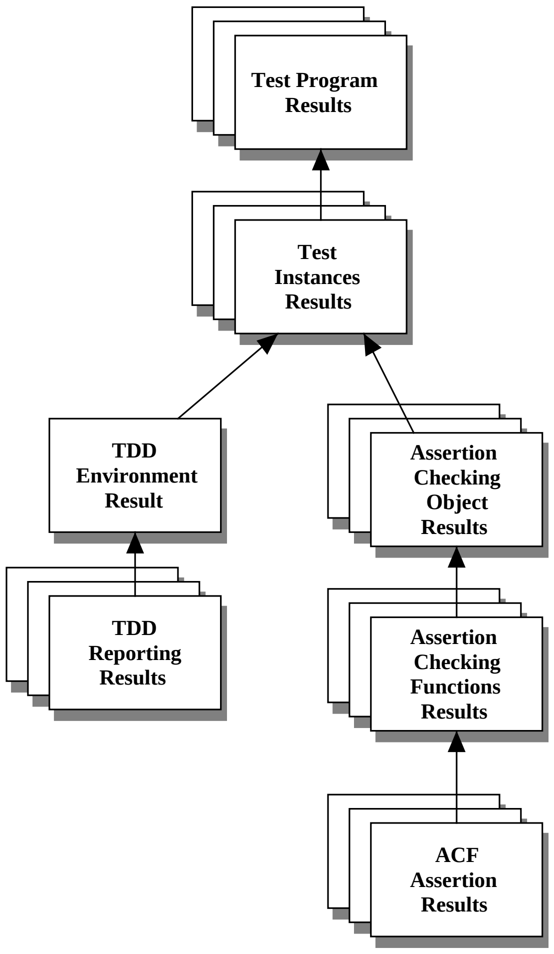


FIGURE 12. Test Results Hierarchy

Test results reporting occurs at the end of the test execution. The level of reporting detail is determined by the reporting level switch set either by default or from the command line of the test program. The results of a test are defined by a code number, name, and precedence value and are defined by the system or optionally by the user. The precedence value is used to determine the overall result of a given component of the tests based on the aggregate results of the sub-components. A test code with a higher precedence value overrides a lower precedence result. Below is the table defining the ADL System test results:.

Result Code Name	Precedence Value	Legal Value	Meaning
ADL_PASS	1	0	Test has passed
ADL_FAIL	MAX_VALUE	1	Test has failed
ADL_UNRESOLVED	MAX_VALUE - 1	2	Evaluation of test could not be determined.
ADL_NOTINUSE	2	3	The test is not evaluated in this environment.
ADL_UNSUPPORTED	2	4	The test is not supported
ADL_UNTESTED	2	5	The test was not done
ADL_UNINITIATED	MAX_INT - 1	6	The test was not started
ADL_NORESULT	MAX_INT - 2	7	The test did not yield a result
ADL_UNEVALUATED	0	32	The assertion was not evaluated
ADL_AMBIGUOUS	3	33	Abnormal and normal both evaluated to true
ADL_UNDEFINED	3	34	Abnormal and normal both evaluated to false
User Defined Result Codes	4	as defined by the user	as defined by the user

TABLE 1. ADL System Test Result Codes

The ADL System result codes are predefined and can not be overridden. However, you can add your own code values to system with a results code file. See Section 7.7.3, “Result Code File,” on page 126 for details.

7.7.2 Test Reporting Library

With no intervention by the user, the code generated by ADLT can report on the results of tests, the results of assertions, and the value of expressions within assertions and within inline functions. The user can control test execution and can provide more in the test report by using the ADLT runtime library, which contains these functions (note that in the C and C++ runtimes, the functions of type “boolean” here are declared as type “int”[1]):

1. Or “bool” according to the C++ standard.

```
void infoline(String message);
```

`Infoline` emits a message in the test journal.

```
boolean tdd_result(int result_code, String reason);
```

`Tdd_result` sets the result of the test instance. It returns a boolean which is FALSE iff the result is a terminating result code, in the opinion of the test framework. The effective result code after this call might be different from the `result_code` parameter, because test result codes are ranked in a precedence hierarchy, so that a later PASS cannot hide an earlier FAIL. Reason may be null.

```
boolean tdd_assert(String expr_text, boolean expr)
```

`Tdd_assert` takes a boolean expression as the second parameter. The expression text is reported to the user and the result of the test is set to ADL_PASS if the expression evaluates to true, and ADL_FAIL otherwise. In any case the test continues with the next statement in the test expression.
Expr_text should not be null. Expr is returned as the value of the function.

```
boolean tdd_skip();
```

Skip the current test case evaluation. It is primarily intended to be called from a factory to ignore a test instance. The result of the test is `ADL_SKIPPED` and all assertions are reported as `ADL_UNEVALUATED`.

```
boolean tdd_end_test();
```

`Tdd_end_test` ends the test upon invocation. The result of the test is the current result of the test at the time of the call to `tdd_end_test`(). `Tdd_end_test` does not return; it is declared as a boolean so that it may be used in expressions like
`tdd_assert(Text, Condition) || tdd_end_test()`

```
boolean tdd_end_case(String reason);
```

`Tdd_end_case` ends the current test instance upon invocation. The result of the test instance is the current result of the test instance at the point of the call to `tdd_end_case`(). Reason may be null. `Tdd_end_case` does not return; it is declared as a boolean so that it may be used in expressions like
`tdd_assert(Text, Condition) || tdd_end_case()`

7.7.3 Result Code File

The method result is normally called with a test result code defined in a result code file. The result code file location is defined by the environment variable TET_RESCODES_FILE.

The format of the result code file is as defined by TET: each line of the file defines one result code, giving its number, its name (as a quoted string), and an optional action. The action controls what the test driver should do when this result code is encountered; legal values are `Continue` and `Abort`. Blank lines and comment lines (which start with a '#') are legal in the result code file. The name of the result code is used in the test report.

7.7.4 Test Program Control

7.7.4.1 Command line invocation

User control of the test program is achieved via the command line and via TET configuration variables. Test Programs are invoked from TET's test case controller - the options of that program can be found in the TET documentation (see http://tetworks.opengroup.org).

7.7.4.2 Configuration Variables

The user can control the test program by use of configuration variables. These are variables that can be set by a variety of mechanisms and read by the test program; some are read by the ADL library and control reporting. User-written code can read test variables and control the test; for example, a configuration variable could be used to set the name of a temporary directory for a file system test.

Configuration variables can be read by user-written code by use of the `getvar` function, which is part of the ADLT runtime library. Its declaration is:

```
String getvar(String propName);

char *getvar(char *propName); // C++

char *ADL_getvar(char *propName); /* C */
```

The name of the configuration variable is the sole argument to `getvar`; it returns the value of the variable if it is set, or `null` if the variable is not set.

Configuration variable settings are available in a file named by the TET_CONFIG environment variable.

7.7.4.3 System Configuration Variables

Certain configuration variables are read by the ADLT runtime library and control the operation of the test program. All configuration variables with names beginning with ADL are reserved for use by the ADL system.

Variable	Legal Values	Default Value	Meaning
ADL_RPT_DETAIL	LONG SHORT	SHORT	Reporting detail level

TABLE 2. System Configuration Variables

User-written code may read the values of these system configuration variables, as well as any other user-defined configuration variables.

7.7.5 Reporting Formats

The format of ADL reports is that of the Test Environment Toolkit. Within these reports, ADL-generated tests will adopt the convention that each assertion will report its information in a separate “block” (a TET reporting abstraction). This will permit easy post-processing that could produce per-assertion reports, should those be needed.

7.8 Building, Executing, and Cleaning Tests under TET

In order to ensure that ADL and ADL generated tests work well in the expected environments, the requirements for those environments and the behavior of the build and run environments needs to be very clear. This section documents these behaviors.

7.8.1 The ADLT compilation environment

In order to build the various ADL translators, the platform must have at least the following:

- A JDK-compliant java compiler and classes at the 1.1 level or better.
- A version of GNU make.
- A variety of POSIX.2-conforming utilities, including the shell, cp, rm, mv, and touch.
- TETware 3.2 or the freely available TET 3.2.
- JavaCC and JJtree version 0.8 pre 1 or better (if parser regeneration is needed).
- An ISO C compiler.
- A draft ANSI C++ compiler (gcc 2.7.2).
- A library build tool (Note - there should be an option to build the ADL libraries as shared libraries to speed execution and reduce the size of compiled tests).

The makefiles in this environment will use a master Makefile.vars in the top level adl2 directory that has platform-specific options in it.

Examples for these will be included in a config directory. Typing “make all” at the top of the adl2 source tree will recurse through the source tree, compiling the different elements. At each level the makefiles will use the GNU make `include` directive to bring in the variables settings from the Makefile.vars at the top level. The makefiles can also take some variables from the environment if they are not set in the Makefile.vars file (e.g. `TET_ROOT`, `ADL2HOME`).

7.8.2 ADLT execution environment

When the user is running ADLT, certain facilities need to be available. These include the following:

- A JVM and the core java classes conforming to JDK 1.1 or better.
- A prolog interpreter such as SWI prolog to assist in natural language generation

7.8.3 Generated Test Build Environment

Once a user has used ADLT to generate the tests, these tests can be exported and run on any platform that supports the language in which the tests were generated. This environment needs at least the following facilities:

- A POSIX.2 conforming make.
- (possibly) some POSIX.2 conforming tools including a shell, cp, rm, and mv.
- TETware 3.2 or the freely available TET 3.2 with JETpack installed if using Java.
- The ADLT runtime environment, including libraries, headers, and tools that assist TET's compilation of the tests.
- A compiler for the language(s) used by the generated tests:
    - An ANSI C Compiler for C Language Tests.
    - A Draft ANSI C++ Compiler for C++ Language Tests.
    - A JDK 1.1 or better compliant compiler and class libraries for Java tests.
    - A IDL/Java 1.0 compliant class library implementation for ADL/IDL tests.

7.8.4 Generated Test Execution Environment

Once the generated tests have been built, they can be distributed in their pre-compiled form and executed elsewhere. In order to accomplish this, the system under test must have at least the following facilities:

- TETware 3.2 or the freely available TET 3.2 (in the case of remote testing, this can be only the executables tccd, tetxresd, and tetsyncd) with JETpack installed if using Java.
- The ADLT runtime environment, including libraries and tools that assist TET's execution of the tests (note that these tools, written in C, require the XPG facilities getenv and putenv in addition to the standard C language libraries).
- If the test author chooses, a POSIX.2 conforming make that will be used in conjunction with the ADLT runtime environment to start the test after setting environment parameters, or
- A requirement that the test user's environment be set up appropriately before executing the tests through traditional means that do not include the use of make.

7.8.5 How it all hangs together

The purpose of this design, and the overall goal for ADL2 in general, is to have an environment in which it is possible to iteratively develop test specifications, tests, and implementations. Once the user has run adlt once for a given TDD test specification, they should only have to type “make” in the directory to have the appropriate files, including the makefiles, regenerated as their specifications and implementations change. Unfortunately, getting this to work correctly in all instances will require a reasonably sophisticated user. More on that later. For the typical user in the typical environment, here is the sequence of events for each phase of TET usage (as it relates to developing specifications and tests):

7.8.5.1 Running ADLT for the first time

When a user has created a specification file and a test directive file, they can call the adlt translator for the relevant language. This translator is responsible for parsing the test data definitions and assertion definitions, turning them into source code for the test driver, the assertion checking object(s), and the translation files for the natural language versions of the assertions. These are well understood. The structure of the makefiles it generates, however, are somewhat confusing:

1. First, there is the Makefile. In the directory in which adlt was run, a master Makefile is created or modified. The translator places a single line in the master Makefile for each test directive in the parsed TDD file. This target has as its method a call to the real makefile for that test directive - named something like `tdd_file.mk.` It also passes into that makefile the name of an (optional) user-provided test configuration file, the name of the target to work on, and the mode of operation (build, exec, or clean).
2. In the generated makefile `tdd_file.mk`, there are targets for each of the generated test directives for each it the modes (three targets per test directive[1]).
3. The build target has dependencies on the source files used by the test directive. It uses default rules to regenerate the executable test case files from the sources if needed. If adlt was run with the -dev option, the source files generated by adlt will in turn have dependencies on their interface specification source or test description source file, as appropriate. It will use other default rules to regenerate the source files from their specification files before compiling.
4. The execute target has dependencies as above (thus automatically ensuring that the tests are built before executing them).
5. The method for the execute target is to hand control off to the test case (via the JVM in the case of Java test cases), passing along the arguments that might have been passed by TET's test case controller (e.g. the invocable components to execute).
6. The clean target has no dependencies, per se. However, it will remove the executable test cases and any intermediate files.
7. An additional “realclean” target will perform the clean as above, and also remove the generated files (except the generated makefiles).
8. ADLT also generates a TET scenario file that describes the various tests generated by ADL. By default this scenario file is a complete TET scenario file called `tetsyncd`. If the -scen_include option is selected, the generated scenario file is designed to be collected later by a scenario file generation tool. This option would be used when generating tests that are an element of a test suite, rather than a complete test suite themselves.

7.8.5.2 Performing subsequent actions via TET

Now that the makefiles have been created, we can use make and/or TET to do builds and runs of the tests. (Note that, in general, you should always execute the tests via TET's

1. In Java only the clean target has to be defined. Build and exec targts are defined in the generic makefile.

tcc so that the environment is properly set up at test execution time. It is possible to simulate this environment, but it is almost always better to just let TET do its job.) In order to teach TET to do this, we provide the following information and tools to TET:

First, we need to establish configuration files. These files tell TET how to perform in each of its modes of operation. Since we cannot know where in a directory hierarchy the ADL user is building their tests, and since these configuration files need to be at the root of a test suite tree, we do not generate them from ADLT automatically. Instead, the user needs to be in the appropriate directory and ask specifically that they be generated. Examples are also provided with the ADL2 source that can be copied, if that is more desirable. The configuration files specify the following:

- The name of the tool to use in the mode (e.g. TET_BUILD_TOOL). This tool is an ADL-provided C program that should build and run on any platform. It performs some rudimentary checking and then calls make.
- The correct settings for TET flags for each mode (e.g. TET_API_COMPLIANT, TET_PASS_TESTCASE_NAME).
- The default set of test results.
- The tools need to be in the user's path. We place these tools in the ADL bin directory by default. However, if they are included in the ADL runtime environment for use on platforms without ADL present, they should be installed in the TET bin directory for simplicity.
- If the test suite was generated as a collection of parts using the -scen_include option (as described above), then a tet_scen file needs to be generated using an ADL-provided tool.
- Now everything is set up. We can use TET's tcc to build the tests, execute them, and clean up. The following is a description of the sequence of execution if TET were used in build mode in the environment described above for a test suite named mysuite:
- The user enters the command tcc -b mysuite
- The tcc does some initial setup, reports the path to the journal file it will generate, and examines the tet_scen and tet_build.cfg files for mysuite
- For each test case specified in the scenario named all, tcc does the following:
  1. Establishes a build lock (to prevent simultaneous compilation by multiple users)
  2. Calls the ADL-provided BUILD_TOOL with the test case name as an argument
  3. The ADL-provided BUILD_TOOL (let's call it adlbuild) looks in the current working directory and the test suite root directory for a file called adl_options.mk. This is a file in which the test suite author can specify options on a per-directory and/or test suite wide basis. These options will work in conjunction with the default make rules. Examples of such options are the name of the library where the implementation is stored, additional elements for the CLASSPATH, etc.
  4. adlbuild then calls make: make ADLCONFIG="-f path_to_options" ADLTARGET=build test_case_name
  5. The master makefile's entry for test_case_name then calls make again, using the real makefile name and passing in the ADL default rules and the test suite-specifi

options: make -f $(ADL2HOME)/lib/generic_java.mk $(ADLCONFIG) -f tdd_file.mk TARGET=$@ $(ADLTARGET)

6. Make reads the ADLCONFIG file (if any) and the default make rules, and then reads in the makefile for this tdd file. In the tdd_file.mk file there is a target for each test_case that reacts appropriately given the value of ADLTARGET (as described above).
7. That target has dependencies value of TARGET, which in turn has dependencies on each component of the test case, including optional dependencies that might get resolved from data in the adl_options.mk file.
8. make ensures that all of the dependencies are satisfied, and finally executes the default loader rule that will create the executable test case file.

The output from this activity is recorded in the journal file.

Once all of the test cases have been generated, tcc exits. Note that, because of the distributed nature of TET, it is possible to perform any TET operations across the network. In the case of a build, if the tet_scen file's all scenario were enclosed within a :remote 1,2,3: option, the build would be done on three systems simultaneously.

In the execute and clean modes, the operations would be similar.

7.8.6 ADL2 and the sophisticated user

The default mode of operation in ADL2 is target at the typical user with typical needs. We believe this user is someone who has a reasonable number of tests that exercise an implementation. The tests are probably being specified separately from the actual development. Consequently, they user needs to tell ADL and TET where to find their implementation. They also may need to tell ADL where to find relevant header files, and potentially the source files for the implementation. All of these things can be specified in the adl_config.mk file. There can be a single version of this file in the test suite root directory. There can also be a version of the file in any directory in the test suite hierarchy - overriding the test suite wide version in the root directory. In most instances, this file is not needed at all. When it is needed, it is usually only needed in the root directory. Only the most sophisticated test suites will have configuration files on a per-directory basis.

The format for this file is exactly like that of a makefile. However, we really expect that the user will only specify a certain set of options in that file. They could theoretically also specify additional make rules and dependencies for their implementation, but ADL's default rules should be sufficient for that.

The following table describes the variables that can be specified in this configuration file.

[Note: *The contents of this table have yet to be designed. It will depend upon the way generated tests are constructed at link time. See ADL1 for some ideas in this area. -spm]*