Using ADL for Regression Test Development in a Distributed Object Environment

by

Eric J. Ding

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY June 1996

© Eric J. Ding, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ........................................ .............. ...................

Department of Electrical Engineering and Computer Science

May 23, 1996

Certified by ........................................ ...........................

Stephen Garland
Principal Research Scientist
Thesis Supervisor

Accepted by ........................................ ...........................

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUN 11 1996
Using ADL for Regression Test Development in a Distributed Object Environment

by

Eric J. Ding

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1996, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I used a formal specification language in the design and testing of objects in a distributed object environment built atop an implementation of the Object Management Group’s Common Object Request Broker Architecture. Specifically, I designed and implemented ADLRunner, a graphical tool which allows users to input test data and run automated functional tests based on specifications of the objects under test. ADLRunner provides an incentive for formal specification in the design stage, as well as a structured method for functional program verification.

Thesis Supervisor: Stephen Garland
Title: Principal Research Scientist
Acknowledgments

Thanks abundantly to my Lord and Savior Jesus Christ, in whom are hidden all the treasures of wisdom and knowledge. By His grace I have been saved from sin and death; by His strength have I not only survived but thrived at MIT; by His Spirit am I continually being molded to be more and more like Him. *Soli Deo gloria!* Jude 24,25.

Thanks to my parents, who have supported and loved me more than I can appreciate or fathom. Your wisdom, counsel, and encouragements have been and continue to be an immeasurably important part of my life.

Alan Noble, Colm Gavin, and Furqan Khan were especially helpful as I wrestled to understand the details of CORBA, VL, Vital, ROAR, and every other acronym, it seems, known to the human race. Thanks to Schlumberger ATE for the opportunity over the last three years to work and learn.

Sriram Sankar, Mark Hefner, and the rest of the Primavera group at Sun Labs were instrumental in making this thesis a reality. Your eagerness to make ADL into a useful tool was the only reason ADLRunner ever got beyond the idea stage; your openness about your work made it easy to learn what I needed to know.

Thanks to Steve Garland, whose encouragements, comments, patience, and signature (heh) helped me finish this thesis just in time.

Thanks especially to my dear brothers and sisters in Christ in CBF who have prayed with me, sharpened me, and continued to walk with me as “aliens and strangers” on this campus for His sake. Thank you for constantly asking how my thesis was going, not because you had to, but because you cared enough to ask. “Expect great things from God; attempt great things for Him” – William Carey. May the Lord use us mightily in these times for His glory. *Maranatha!*
List of Figures

1-1 The CORBA ORB functions as a software bus ........................ 9
1-2 Example of an IDL Interface ........................................ 9
1-3 Example of an ADL/IDL specification ............................... 11

2-1 Overview of ADLRunner program flow ............................ 14
2-2 Main window of ADLRunner ........................................... 15
2-3 Editing module data in ADLRunner ................................. 16
2-4 Example of test data input by user ................................ 17
2-5 Example of error output during test data syntax check .......... 18

3-1 Object Modeling Theory diagrams of ADLRunner program architecture .. 22
Chapter 1

Background

1.1 Introduction

In Schlumberger’s drive to stay at the top of the automatic test equipment (ATE) market, the graphical software that drives the semiconductor test equipment manufactured by the company has become a vital component of its ATE package. In the last two years, this software has been presented with the challenges of managing distributed, multi-client usage and of allowing client extensions, such as graphical tools to design new kinds of hardware tests. So far, it has not been able to handle these demands effectively. Taking a step toward addressing both challenges, Schlumberger has been developing a new software architecture that takes advantage of emerging standards in distributed object technology, which provide a stable and long-term software architecture strategy, as well as possibilities of multi-platform inter-operability.

Schlumberger’s new Revised Object Architecture (ROAR), built on top of an implementation of the Object Management Group’s CORBA [2] standard, answers both challenges. However, the introduction of a new, object-oriented methodology and architecture requires new answers to questions at two crucial steps of the design process: first, how and at what level of granularity does one specify the behavior of objects in the architecture (such objects may represent, for example, hardware test parameters), and second, how does one then effectively test the behavior of those objects?

At this point in time, neither question has been sufficiently answered. Both specification and validation phases are informal. Prose documentation is written to describe the behavior of client-side and server-side applications, but no formal method of validation has been
established for the software being written. The development of new applications, or porting of existing ones to the new architecture, is governed by a maturing design process, but the underlying architecture is only tested to the extent that the applications using it exercise the objects within it.

The lack of a formal process of unit testing in ROAR, along with the concurrent development of a formal specification language being specialized for use with CORBA, prompted this project to develop ADLRunner. ADLRunner is a Motif-based graphical application which, using formal specifications written in ADL (Assertion Definition Language) and user-inputted test data, automates the unit testing of objects in a CORBA-based architecture such as ROAR. By formalizing some aspects of the specification stage of the development process, the use of ADL should also ease the implementation phase by resolving ambiguities unaddressed or even introduced by informal high-level specifications.

1.2 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard for distributed object environments, developed by the Object Management Group (OMG). CORBA’s Application Programming Interfaces (API) enable distributed object interaction within a specific implementation of an Object Request Broker (ORB). CORBA does not specify how the ORB should be implemented; in principle, it may be a set of runtime libraries, a set of daemon processes, a server machine, or part of an operating system. [1]

The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network ... The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object’s interface. [2]

The Interface Description Language (IDL), as part of the CORBA specification, is used to define interfaces to objects. IDL gives a simple description of an object, allowing flexibility in implementation. It also gives no information about the semantics of an interface or the methods associated with it (see Figure 1-2).
Figure 1-1: The CORBA ORB functions as a software bus, allowing clients to invoke methods on server objects without regard to physical location, implementation language, or other system aspects of the server object.

Remote objects view a CORBA object purely in terms of this interface. IDL provides encapsulation of an object’s implementation behind a formal IDL interface which is independent of implementation language, implementation algorithm, location, machine architecture, operating system, network technology and even ORB architecture. This separation of interface and implementation allows CORBA to be viewed as a “software bus”, and is one of the most powerful aspects of CORBA. [1]

```idl
interface longArray {
    void addHigh(in long elt);
    string remHigh();
};
```

Figure 1-2: Example of an IDL Interface

1.3 Assertion Definition Language

Because IDL does not specify the behavior of CORBA interfaces, the development of Assertion Definition Language (ADL) and its potential to be used to describe the semantics of CORBA interfaces attracted the attention of ROAR developers. ADL is a language framework designed to be used for the specification and validation of software components. The ADL approach to program validation is to perform black-box tests using test harness
functions which have been generated automatically from formal specifications. Correct behavior is determined by examining the results of the program or function in light of the semantics given in the specification.

In most published work on testing, the details of [determining if the program behaves as intended on a given set of test data] have not been dealt with. Rather, an oracle is assumed to exist... Lately, however, much work has been done to realize this oracle. In nearly all cases, this has been achieved by comparing the behavior of a program against its specifications that have been written in some formal specification language... The design of ADL is well-suited for testing – generating a test oracle from an ADL specification is a straightforward process.[5]

The concepts of ADL are specialized for use with a particular programming language by rendering these concepts into a syntax similar to that of the language. In adopting the approach of designing a language framework that is specialized for particular languages, the creators of ADL have used a methodology similar to that of other specification, programming, and interface description languages such as Larch [3], IDL, and Rapide [4]. ADL has already been specialized by the Primavera group at Sun Microsystems Laboratories, Inc. for use with ANSI C; the specialization of ADL for use with CORBA IDL (discussed later) was developed by the group as a prototype to prove the feasibility of ADL, and further development began concurrently with this project.

ADL is similar to well-known specification languages such as Larch and Z [6] that all are high-level languages which focus on expressivity; however, ADL differs in that its designers have also made “implementability” a priority, allowing better tool support at the cost of lesser expressivity. Where it may be difficult to express semantics in terms of ADL constructs, it is possible to write auxiliary functions which can be “called” when the ADL is used for functional testing.

Among the features of ADL are:

- ADL specifications are post-conditions on operations (or functions) of software components. That is, they constrain the program state at the time of termination of operation evaluations.
- ADL specifications may be partial. The complete details of a function need not be written in ADL.

- ADL is well-suited for the purpose of testing software components. Difficult to evaluate constructs such as quantifiers have been excluded for the time being.[7]

1.4 ADL for IDL

ADL allows for partial specification, which is especially useful in some CORBA environments, where only portions of the functionality of the system can be described in terms of IDL methods and attributes. For example, an interface may represent a monolithic block of functionality which does not directly interact with other CORBA objects. The ability to write auxiliary functions enables developers to loosely document and test portions of the system otherwise not visible from the IDL level. Additionally, because most of the constructs in ADL/IDL come directly from IDL, CORBA developers who have already written IDL interfaces should be able to use ADL/IDL fairly quickly.

```c
void longArray::addHigh(in long elt)
semantics{
  size() == @size()+1;  // new size = former size + 1
  get(size()-1) == elt;  // element at index size-1 (0-indexed) is now elt
}
```

Figure 1-3: Example of an ADL/IDL specification

The developers at Sun Microsystems Laboratories, Inc. have developed a tool which parses ADL/IDL, generating black-box “Assertion Checking Functions” (ACF’s) that are used to automate the unit testing of CORBA objects. ACF’s are functions called as hooks before and after the functions or methods to which they correspond are actually run. Also on the horizon (and already developed for ADL for ANSI C) is an ADL/IDL translator which would allow ADL/IDL to be translated into natural language documents, further helping developers to use interfaces without having to understand their implementation.
1.5 Using ADL for IDL

ADL/IDL's adoption of IDL-like syntax encourages its use not only as a program verification tool, but also as a part of the design stage of the software engineering process. By providing a formal semantics for IDL interfaces, ADL/IDL allows for greater detail in the design stage of a CORBA-based system than would be available with only a high-level prose program specification. At the same time, ADL reduces the possibility of ambiguity in the specification of IDL interfaces, making clearer the task of the programmer who implements the interface.

Using ADL for IDL should also encourage code and object re-use by lessening the need for engineers to examine low-level code to understand the functionality of existing parts of a system. By providing a simple yet formal framework for specification, ADL should help programmers to quickly document the semantics of their interfaces, and also enhance the re-usability of those interfaces by lessening the need to read high-level and possibly ambiguous natural language documentation. ADL/IDL should also increase CORBA's appeal as a cross-platform, cross-language object-oriented architecture standard.
Chapter 2

ADLRunner

2.1 Introduction

ADLRunner is a graphical application designed to help software engineers develop and manage regression tests based on ADL specifications of IDL interfaces. ADLRunner was developed on a SunOS 4.1.3 Sparcstation in the X11R5 window environment with the Motif 1.2 graphical user interface widget set. Although it can theoretically be generalized to be used with any ORB that conforms to the CORBA standard (such as Sun Microsystems’ NEO), ADLRunner has been written specifically for use with IONA Technologies’ Orbix, a full implementation of the CORBA 1.1 standard.

The entirety of ADLRunner is implemented in C++. The graphical front-end was written using a C++ class library that encapsulates the Motif widget set. ADLRunner incorporates the functionality of the ADL parser under development at Sun Microsystems Laboratories, as well as managing the test data to be run by the test harness functions generated by the parser.

2.2 Functionality of ADLRunner

2.2.1 Overview

As shown in Figure 2-1, ADLRunner is responsible for several tasks:

- generating test code from ADL specifications,
- helping the developer to supply and organize test data,
Figure 2-1: Overview of ADLRunner program flow. Highlighted boxes, from left to right, show functions integrated in ADLRunner. “File” boxes and dotted lines depict the data flow of files associated with ADLRunner.
2.2.2 Creating Test Modules

When the user selects the “Add Module” button, ADLRunner prompts the user with a file selection dialog box to select the appropriate ADL file for test driver generation, then creates test “modules” from them which organize test data corresponding to selected IDL interfaces. For example, after selecting the `string` and `longArray` ADL specification files, the user would see a main window similar to Figure 2-2. To delete a module from the current Project, the user can simply select the appropriately named button, and press “Remove Module” in the button bar.

For each module, the user can enable or disable the generation of test code by toggling the “Active” toggle button. When the toggle button is active, the program will attempt to generate test drivers for the given module whenever the “Build Tests” button is selected. When the toggle button is not active, then ADLRunner will simply ignore the associated module when it is building and running tests.

ADLRunner keeps track of whether any test data has been specified for modules, and indicates whether a module is ready for tests to be built for it via the “Ready/Not Ready” label. If the user has yet to specify test data with correct syntax for each active method...
associated with a given module, then ADLRunner will indicate that the module is not ready. If all the active methods of an module have data specified, then ADLRunner will indicate that that module is ready.

In some instances, the user may desire to change the data specified for a given module without overwriting existing test data sets. In that case, by selecting the “New Config” button, the user can create new data sets (configurations) or copy existing ones for the selected module, rather than reading in the same specification file again. The configuration of a module can be selected or changed with the option menu to the right of the “Ready/Not Ready” label; for each configuration, the user can specify a comment or description, displayed to the right of the configuration name. To remove a configuration (unless it is the only one), the user can select “Delete Config” from the button bar.

2.2.3 Specifying Test Data

![Figure 2-3: Editing module data in ADLRunner](image)

To edit test data for an object, the user selects the appropriate module and the “Edit Data” button from the button bar main window. ADLRunner then pops up a dialog box which corresponds to the methods associated with the selected module (see Figure 2-3). The layout of the Edit Data dialog window is similar to that of the main window; configurations correspond again to data sets, only at a smaller level of granularity, with each configuration containing the data for a given method rather than for an entire module.

In order to edit test data for a given method, the user selects the Edit Data button, and ADLRunner spawns an external editor in which the user may enter test data. Test data is written out in C++ syntax (see Figure 2-4) for simplicity of implementation, allowing the program to check the syntax of inputted “data” by simply compiling it.
Figure 2-4: Example of test data input by user

After the user finishes and exits the editor, the program parses the data to check that it is syntactically correct. To do so, ADLRunner wraps the test data code in a class declaration, and compiles the modified code:

```cpp
#include <longArray.h>

class SublongArray : public longArray
{
 public:
   SublongArray()
   {
     addHigh(0);
     addHigh(1);
     addHigh(2);
   }
};
```

The "wrapper" subclass eliminates the need to write code that instantiates an object of the class being edited and calls functions on the object. For example, for the generated code above, since SublongArray is a subclass of longArray, all the public member methods of longArray can be directly called within member methods (such as the constructor) of SublongArray. This approach even allows the test data code to be included in the file with a #include C pre-processor directive, rather than being literally inserted in the code.

If the compiler exits with an error, then ADLRunner relays any error output to the user (see Figure 2-5), and does not save the new version of the test data.

### 2.2.4 Generating and Running Test Executables

Because ADL’s method for code verification focuses on testing, verifying code correctness involves actually instantiating and calling methods on the objects under test, with parser-generated Assertion Checking Functions (ACF’s) being called in appropriate places to monitor the program state. ADLRunner automates this task, first determining which methods
and modules have appropriate test data specified, then inserting hooks into code for those methods.

Inserting test harness calls into code involves several steps:

- First, the corresponding IDL interfaces are compiled. This phase involves invoking the IDL compiler, which translates IDL specifications into code to marshal and dispatch object requests. This stub code, written in C++, is later linked with the eventual test data invocations to allow the test program to interact with the server objects. For example, the stub code generated for `longArray::addHigh` would be:

```cpp
void longArray::addHigh (long elt, CORBA::Environment &IT_env) {
    if (IT_env || m_isNull) return;
    CORBA::Request IT_r (this, "longArray", IT_env, 1, 0);
    if (!IT_r.isException (IT_env)) {
        IT_r << elt;
    }
    IT_r.invoke (CORBA::Flags(0), IT_env);
    if (IT_r.exceptionKind () == CORBA::StExcep::SYSTEM_EXCEPTION)
        IT_r.checkEnv (IT_env);
}
```

- After the stub code has been generated, ADLRunner calls a perl script which processes the code, inserting calls to ACF’s. These functions, generated by Sun Labs’ ADL parser, examine the state of the system before and after a method is called to verify its correctness, and send output to either a file or standard output, depending on how they are called.

The perl script inserts a call to a function which checks the state before `addHigh` has
actually done anything, and a call to another function which checks the post-state of
the function and outputs to either a file or standard output, depending on how it is
called:

```cpp
t void longArray:: addHigh (long elt, CORBA::Environment &IT_env) {
    // beginning of ADLRunner insertion
    _ACF_pre_longArray_addHigh();
    // end of ADLRunner insertion

    if (IT_env || m_isNull) return ;
    CORBA::Request IT_r (this, "longArray",IT_env,1,0);
    if (!IT_r.isException (IT_env)) {
        IT_r << elt;
    }
    IT_r.invoke (CORBA::Flags(0),IT_env);
    if (IT_r.exceptionKind () == CORBA::StExcep:: SYSTEM_EXCEPTION)
        IT_r.checkEnv (IT_env);

    // beginning of ADLRunner insertion
    _ACF_post_longArray_addHigh();
    // end of ADLRunner insertion
}
```

Under the version of NEO, Sun’s implementation of CORBA, being worked on by the
Primavera group at Sun Microsystems Labs, this step is done by the IDL compiler
simply by calling the compiler with an additional flag.

• Once the script has finished, ADLRunner generates C++ code which instantiates the
objects under test, calling the appropriate methods on them with specified test data. It
then compiles and links the code with the stub code to generate a working executable.
Similarly to when checking syntax of test data, ADLRunner creates the appropriate
code by declaring subclasses of the interfaces being tested, and instantiating them.
For example, for the longArray interface, ADLRunner generates:

```cpp
class SublongArray : public longArray
{
public:
    SublongArray() {
        addHigh(0);
        addHigh(1);
        addHigh(2);
    }
};
```
ADLRunner also generates a main function which calls all the appropriate “create” functions.

- After the executable has been run, ADLRunner displays the output of the ACF’s in a dialog box, and gives the user the option of analyzing code coverage via an external tool. By default, no such tool is spawned; the user may change default options to link in tools which perform dynamic code analysis if he desires.

2.3 Persistence of Test Projects

Projects can be saved into and loaded from ASCII format text files. The saved project includes not only all information about test data, active/inactive modules, etc., but also all the options (i.e., compile command, compile-time options, etc.) associated with a given project.
Chapter 3

Design and Implementation

3.1 Program Architecture

The underlying structure of ADLRunner is implemented using the \textit{RWOrderedVector} class of RogueWave Software’s Tools++ library. \textit{RWOrderedVector} is a pointer-based collection class which can contain pointers to classes which are subclassed from \textit{RWCollectable}. In the initial design, every class directly contained an \textit{RWOrderedVector} data member. After a preliminary design review, it was decided to re-design the class hierarchy by introducing a general \textit{TestCollection} class to capture the duplicated container functionality of many of the classes.

\textit{TestCollection}, from which all other container classes in the architecture are subclassed (see Figure 3-1), contains a private \textit{RWOrderedVector} data member, and handles functionality having to do with container operations (see Appendix A for individual class specifications). For example, \textit{getElement}, \textit{setElementName}, and \textit{getTestElementNames} are all inherited from the \textit{TestCollection} class, while subclass-specific operations like \textit{TestProject::getActiveStatus} and \textit{TestModule::setComment} are implemented in the subclasses.

Because it is possible at several levels to have multiple configurations (i.e., multiple data sets per method or per module), as well as several different modules in each TestProject, the inherited container functionality of \textit{TestCollection} is general enough to be needed by all its subclasses. At the same time, where there is a need for more specific knowledge of the contained class (as in the \textit{copyElement} method), each subclass augments the inherited behavior with more specific functionality. For example, the \textit{TestProject::copyElement} method not only copies a contained object, but also copies its “ActiveStatus.”
Figure 3-1: Object Modeling Theory (OMT) diagrams of ADLRunner program architecture
The TestElement class, which simply stores a name attribute, is a superclass of TestCollection (and hence, indirectly a superclass of all container classes in the program) and TestData. To use R沃OrderedVector, it is also necessary that all contained classes be subclassed from RWCollectable (see Figure 3-1); because TestElement inherits from RWCollectable, all other classes indirectly inherit from RWCollectable as well.

File I/O operations are restricted to TestProject and TestModule, which need to load and (for TestProject) save data to and from files. Operations such as generating test driver code and parsing test data are left to the application interface code to make it easier for developers to change the way those tasks are performed if necessary, and to separate the functionality of parsing and containing data, which is left to the container classes to do, from the processes of generating data and test drivers, which is delegated to the user interface code. This modular approach should allow for minimal impact on the underlying program architecture if, for example, ADLRunner is ported to be used with another CORBA implementation.

3.2 Inputting and Processing Test Data

One question that was faced early in the design process was how to allow the user to edit data, and then to process the input. One alternative was to design and implement a limited editor for inputting test data; this choice was abandoned, however, because of the significant complexity it would have added to the project. Another option explored was to design a parser to check a specified syntax for test data; because of limited experience in implementing parsers, and again because of its significant complexity, this option was abandoned.

Ultimately, it was decided that both functions could be performed adequately by existing programs. The task of getting user input for test data was easily delegated to an external editor of the user’s choice; the default editor was vi, or the value of the EDITOR environment variable if set. To check syntax, ADLRunner spawns a script which compiles an augmented version of the test data (see Section 2.2.3). A beneficial (though untested and mostly undocumented) side-effect of this choice is that users may be able to write C++ code within the test data “code” that will be run during the test, as long as it is accepted by the compiler.
3.3 Evaluation

ADLRunner is a user-friendly graphical user interface which makes it easy for software engineers to organize and develop test drivers for CORBA objects. Many of the objects in ROAR have already been specified using ADL; it remains, then, for the developer to write up test data for the interfaces and then to compile and link modified stub code with existing ROAR client tools.

The GUI functionality of ADLRunner has been tested and demonstrated thoroughly, and its underlying functions of generating and checking test data code and client stub code have been implemented and tested. Along the way toward completion, some other libraries used by ROAR were exercised as well:

- ADLRunner’s approach of creating, deleting, then adding more graphical widgets when modules are added or deleted uncovered and prompted the fixing of a bug in the C++ widget library used for ROAR tools.

- Poorly documented interfaces were highlighted and noted; a majority of interfaces now have ADL specifications as well.

3.4 Remaining Issues

What remains to be completed in ADLRunner are the test generation and execution steps of the testing process. Because of time restraints and the author’s unfamiliarity with Orbix system internals, the C++ code which instantiates, binds to, and calls methods on an actual CORBA object has yet to be written. Once this part has been completed, it will be possible to actually generate, run, and evaluate the ADL-based tests.

Also, ADLRunner’s implementation is very specialized for Orbix. In order to port ADLRunner to another vendor’s CORBA implementation, such as Sun Microsystems’ NEO, it would be necessary to modify the way that ACF’s are generated and inserted into client-side stub code, and to write the code as discussed above which instantiates, binds to, and calls methods on CORBA objects. Because the Primavera group has successfully augmented the NEO IDL compiler’s functionality to automatically insert calls to ACF’s (see Section 2.2.4), the task of porting ADLRunner to a NEO architecture may be significantly easier than porting to any other CORBA implementation; in fact, the primary task remaining would be the
same task which remains for the Orbix version of ADLRunner.
Appendix A

Code Specifications

A.1 TestElement

class TestElement : public RWCollectable
{
  public:

    TestElement& operator=(const TestElement& te);
    RWBoolean operator==(const TestElement& te) const;
    virtual RWBoolean isEqual(const RWCollectable *te) const;

    // these methods are used for persistence
    virtual void saveGuts(RWFile &)
      const;
    virtual void saveGuts(RWvostream &)
      const;
    virtual void restoreGuts(RWFile &);
    virtual void restoreGuts(RWvistream &);

    const RWCString& getName() const;
    // effects Returns the name of this.

    virtual void setName(const RWCString& config_name);
    // modifies this
    // effects Sets the name of this to be config_name.
};

A.2 TestCollection

class TestCollection : public TestElement {
  public:

    RWBoolean operator==(const TestCollection& tc) const;

    // these methods are used for persistence
    virtual void saveGuts(RWFile &)
      const;
    virtual void saveGuts(RWvostream &strm) const;
    virtual void restoreGuts(RWFile &);
    virtual void restoreGuts(RWvistream &strm);
virtual TestElement *newElement(const RWCString& element);
// effects Creates a new TestElement in this with its
// name set to element, and returns a pointer to it,
// unless a TestElement with the name element already
// exists; otherwise, returns NULL.

virtual RWBoolean isEqual(const RWCollectable *te) const;

virtual TestElement *copyElement(const RWCString& oldName,
const RWCString& newName);
// requires oldName is the name of a valid TestElement of this.
// effects Creates a new TestElement named newName which
// contains the same data as oldName, and returns it,
// unless a TestElement named newName already exists.
// If newModule already exists, returns NULL.

void deleteElement(const RWCString& element);
// requires element is the name of a valid TestElement in this.
// effects Deletes the TestElement named element.

RWBoolean setElementName(const RWCString& element,
const RWCString& newName);
// requires element is the name of a valid TestElement in this.
// effects Changes the name of the TestElement named element
to newName. If newName is already used as a name of a
TestElement in this, returns FALSE and does not rename
element; otherwise, returns TRUE.

TestElement *getElement (const RWCString& element) const;
// effects If element is found in this, then returns it. Otherwise,
// returns NULL.

TestElement *getCurrentElement() const;
// effects Returns the current TestElement in this.

TestElement *setCurrentElement(const RWCString& element);
// effects If element is found in this, then sets the current
// TestElement to the TestElement named by element,
// and returns the element. Otherwise, returns NULL.

RWOrdered getTestElementNames( const;
// effects Returns an RWOrdered of the names of the valid
// TestElements in this.

int entries() const;
// effects Returns the number of elements in this.
class TestProject : public TestCollection {
    // Overview: A TestProject is an ordered collection of TestModuleCollections.
	nothing:

public:
    TestProject& operator=(const TestProject& tm);

    TestModuleCollection *newModuleCollection(ifstream &adl_file);
    // effects Creates a new TestModuleCollection based on the
    // interface described in adl_file. If element with same
    // name exists, or adl_file is badly formatted, then
    // returns NULL. Otherwise, returns the newly created
    // element.

    virtual TestElement *newElement(const RWCString& name);
    // effects This should never be called!!!
    // Prints to diagnostic output and returns NULL.

    virtual TestElement *copyElement(const RWCString& oldName,
                                    const RWCString& newName);
    // requires oldName is the name of a valid TestModule of this.
    // effects Creates a new TestModuleCollection named newName which
    // contains the same data as oldName, unless a
    // TestModuleCollection named newName already exists.
    // If newName already exists, returns NULL;
    // otherwise, return the new element.

    // these methods are used for persistence
    virtual void saveGuts(RWFile &f) const;
    virtual void saveGuts(RWvostream &)
    virtual void restoreGuts(RWFile &f);
    virtual void restoreGuts(RWvistream &);

    RWBoolean getActiveStatus(const RWCString& name) const;
    // requires mod is a TestModuleCollection in this.
    // effects Returns TRUE if mod is active, FALSE otherwise.

    RWBoolean toggleActiveStatus(const RWCString& module) const;
    // requires name is the name of a Module in this.
    // effects Toggles the active status of mod. Returns the
    // active status after toggle (see getActiveStatus).

    TestElement *getElement(const RWCString& name) const;
    // returns a TestModuleCollection

    TestElement *getCurrentElement() const;
    // returns a TestModuleCollection

    RWOrdered getActiveModuleNameo() const;
    // effects Returns an RWOrdered of the active modules in this.
    // If there are no modules, returns an empty
    // RWOrdered. Modules are returned in the order
    // in which they will be executed.

    void setOptions(const ADLOptionStruct& opts);
    // effects Sets the options for the project.
ADLOptionStruct getOptions() const;
// effects Returns the options for the project.

RWCString writeTestDrivers() const;
// effects Magically post-processes the IDL stubs, adding in
calls to the appropriate ACF's.

A.4 TestModuleCollection

class TestModuleCollection : public TestCollection {
  // Overview: A TestModuleCollection is a collection of TestModule's,
  // all of which correspond to the same ADL/IDL interface.

public:

  TestModuleCollection& operator=(const TestModuleCollection& tmc);
  RWBoolean operator==(const TestModuleCollection& tmc) const;
  virtual RWBoolean isEqual(const RWCollectable* te) const;

  // these methods are used for persistence
  virtual void saveGuts(RWFile &f) const;
  virtual void saveGuts(RWvostream &strm) const;
  virtual void restoreGuts(RWFile &f);
  virtual void restoreGuts(RWvistream &strm);

  RWBoolean parseADLFile(ifstream& adl_fp);
  // requires adl_fp points to a readable file named filename.
  // effects If file is in correct format, parses file pointed
  // to by adl_fp, sets name and methods of this to the
  // name and methods of the interface in the file,
  // initializes a default configuration with name
  // "Default", and returns TRUE. Otherwise, returns
  // FALSE.

  void setIDLFile(const RWCString& filename);
  // effects Sets the internal idl_file data member to filename.

  RWCString getIDLFile();
  // effects Returns the internal idl_file data member.

  RWCString writeTestDrivers();
  // requires This is a TestModule that is ready for test, i.e.,
  // has at least a minimum set of test data specified
  // for its methods in the current TestModule.
  // effects Returns a string containing C++ function calls
  // to ACF's using the specified test data for each
  // method in this.

  virtual TestElement *newElement(const RWCString& module);
  // effects Creates a new TestModule in this with its
  // name set to module, and returns it, unless a
  // TestModule with the name module already exists;
  // otherwise, returns NULL.
virtual TestElement *copyElement(const RWCString& oldName,
    const RWCString& newName);
// requires oldModule is the name of a valid TestModule of this.
// effects Creates a new TestModule named newName which
// contains the same data as oldName, unless a
// TestModule named newModule already exists.
// If newModule already exists, returns NULL;
// otherwise, return the new element.
};

A.5 TestModule

class TestModule : public TestCollection {
    // Overview: A TestModule is a module containing test data for an
    // ADL/IDL interface.

    public:

    TestModule& operator=(const TestModule& tm);
    RWBoolean operator==(const TestModule& te) const;
    virtual RWBoolean isEqual(const RWCollectable* te) const;
    virtual void saveGuts(RWFile &f) const;
    virtual void saveGuts(RWvostream &strm) const;
    virtual void restoreGuts(RWFile &f);
    virtual void restoreGuts(RWvistream &strm);

    TestElement *getElement(const RWCString& method) const;
    // returns a TestMethod *
    TestElement *getCurrentElement() const;
    // returns a TestMethod *

    RWBoolean parseADLFile(ifstream& adl_file, RWCString& config);
    // requires adl_file points to a readable file.
    // effects If file is in correct format, parses file pointed
    // to by adl_file, sets methods of this to the
    // methods of the interface in the file, sets config
    // to the name of the interface, and returns TRUE.
    // Otherwise, returns FALSE.

    void setComment(const RWCString& comnt);
    // effects Sets the comment string of this to be comnt.

    const RWCString& getComment() const;
    // effects Returns the comment string of this; if no string
    // has been set, returns the empty string.

    RWBoolean toggleActiveStatus(const RWCString& method);
    // modifies this
    // requires method is the name of a valid TestMethod in this.
    // Effects Toggles the active status of method. Returns the
    // active status after toggle (see setActiveStatus).

    RWBoolean setActiveStatus(const RWCString& method) const;
    // modifies this
    // requires method is the name of a valid TestMethod in this.
// effects Returns TRUE if method is active, FALSE otherwise.

RWOrdered getActiveMethodNames() const;
// effects Returns an RWOrdered of names of the active methods of this. If there are no active methods in this, returns an empty RWOrdered. getMethodsO returns the methods in the execution order of this.

RWCSString writeTestDrivers() const;
// requires this is a TestModule that is ready for test, i.e., has at least a minimum set of test data specified for its methods.
// effects Returns a string containing C++ function calls to ACF's using the specified test data for each of the methods in this.

int getReady() const;
// effects If all active methods of this are ready, returns TRUE.
// If not all active methods are ready, returns FALSE.
// If there are no active methods, returns -1.

};

A.6 TestMethod

class TestMethod : public TestCollection {
    // Overview: A TestMethod contains the test data for a method; that is, it contains one or more test data configurations for the method.

public:

    TestMethod& TestMethod::operator=(const TestMethod& tm);
    RWBoolean operator==(const TestMethod& tm) const;
    virtual RWBoolean isEqual(const RWCollectable* te) const;

    // these methods are used for persistence
    virtual void saveGuts(RWFile &f) const;
    virtual void saveGuts(RWvostream &strm) const;
    virtual void restoreGuts(RWFile &f);
    virtual void restoreGuts(RWvistream &strm);

    virtual TestElement *newElement(const RWCString& element);
    virtual TestElement *copyElement(const RWCString& oldName, const RWCString& newName);

    const RWCString& getHelpComments() const;
    // effects returns the help "header" comments for the method.

    RWBoolean setArgs(const RWCString& args);
    // effects Sets the arguments for the method, and constructs help header comments for them, if the string is formatted correctly, and returns TRUE. Otherwise, returns FALSE, arguments are not set, and header comments are not changed.

    const RWCString& getArgs() const;
    // effects Returns the arguments for the method.
RWCString unparsedata() const;
// effects Returns a string containing the test data
// corresponding to the current configuration, in the
// format specified in scenario.txt (including comments).

A.7 TestData

class TestData : public TestElement {
    // Overview: A TestData contains a test data configuration for a method.

public:
    TestData& operator=(const TestData& td);
    RWBoolean operator==(const TestData& td) const;
    virtual RWBoolean isEqual(const RWCollectable* te) const;

    // these methods are used for persistence
    virtual void saveGuts(RWFile& f) const;
    virtual void saveGuts(RWvostream& strm) const;
    virtual void restoreGuts(RWFile& f);
    virtual void restoreGuts(RWvistream& strm);

    const RWCString& getData() const;
    // effects Returns the test data.
    void setData(const RWCString& data_lines);
    // effects Sets the test data to data_lines.

    RWBoolean getReady() const { return ready; }
    // effects If data is "ready" (i.e., not the empty string), returns
    // TRUE. Otherwise, returns FALSE.
Bibliography


