INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

COM-Tester:
A Script Testing Tool for MS COM

Yongqiang Chen

A Thesis

in

The Department

of

Computer Science

Presented in partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

May 2000

ABSTRACT

COM-Tester:
A Script Testing Tool for MS COM

Yongqiang Chen

The purpose of this thesis is to develop a tool to test the COM object. Although COM development is so important and so popular, there is no general tool to test it.
In industry, the cost to test COM–related software is so high that there is a need to develop a simple and general tool for the test. This thesis focuses on developing such a tool called COM-Tester.

COM-Tester is a small scripting language tool designed for writing test scenarios. The main goal of the tool is to reduce the costs related to the testing activities of Microsoft COM-based software. The core distinctive features of COM-Tester are:

- A high-level simple interpreted language where the notion of a test is directly supported by a specific syntax.
- Primitives to create and work easily with COM components.
- Merging of source code and documentation.
- Automatic generation of testing report.

Acknowledgements

Sincerely, I would like to express my deepest respect and gratitude to my thesis supervisor Dr. Peter Grogono for his guidance, invaluable suggestions, encouragement and various prompt help throughout the course of this research work. I am very thankful for the opportunity that I had to work with Dr. P. Grogono, and for everything he taught me during my master's studies.

I would like to give a special thanks to my wife. It was her love, patience and continued support that made this thesis possible.

Table of Contents

Chapter 1 Introduction

1.1 Overview of COM

The Component Object Model (COM) is Microsoft's distributed object architecture. In COM, functions defined by a COM interface may be called by objects on other machines, written in different languages and running on different operating systems.

COM is tightly integrated in Microsoft's operating systems, Windows 95/98, Windows NT and Windows 2000, and it underpins Microsoft's view of current and future application development.

Microsoft is heavily committed to the use and reuse of binary software components across their systems. As a demonstration of this commitment, they have split much of their own software into COM component – for example the Office suite and Internet Explorer.

Let's review the reason why component-based development is such a good idea – namely, code reuse.

Code reuse can be as simple as copying and pasting source code. But whenever you change the original source code, to add features or to fix a bug, you need to paste the revised code into every application.

To improve source code sharing, object-oriented programming recognizes object relationships and object-oriented languages automate reuse of parent class code, and allow polymorphic method calling.

But this too has its limitations – any changes in the parent classes necessitate recompilation of any application that reference child classes.

The next stage along is use of precompiled libraries; in Windows there are Dynamic Linked Libraries (DLLs). This solves an additional problem – how to protect the code on which you have worked so hard from being stolen. But the problem of keeping DLL versions up-to-date is still left unsolved.

The industry solution to this is to define a unique `interface` to an object, which has the following attributes:

- Unique identification;
- Interface set in perpetuity;
- Exposes public methods, and acts as a level of indirection between the caller and the object. Thus, it supports strong encapsulation.

This is implemented through a binary-level compatibility specification. This specification is based on the use of what COM calls a `virtual table`, or `vtable`.

When it comes to that bug fix, you can recompile the new, improved COM object to support the same `interface`, and no-one will ever know, except by the disappearing errors – as it supports the same `interface`.

Anytime you want to expand the functionality of an object, you can create a new `interface`. This will have a new name and a new `Identifier`. And because COM objects support multiple `interfaces` (as a means of supporting different sets of functionality), your COM object can still support the old `interface`.

1.2 Objective and Scope of the Thesis

The objective of this thesis is to develop a tool to test the COM object. Although COM development is so important and so popular, there is no general tool to test it.
In industry, the cost to test COM–related software is so high that there is a need to develop a simple and general tool for the test. This thesis focuses on developing such a tool called COM-Tester.

COM-Tester is a small scripting language tool designed for writing test scenarios. The main goal of the tool is to reduce (in the short as well as in the long term) the costs related to the testing activities of Microsoft COM-based software. The core distinctive features of COM-Tester are:

- A high-level simple interpreted language where the notion of a test is directly supported by a specific syntax.
- Primitives to create and work easily with COM components. `Interface` information (function prototypes) and data type information (classes and structures) are automatically extracted from TLBs (type libraries) or the standard `IDispatch interface`.
- Merging of source code and documentation. This enables documentation to be extracted from the code, thus easing maintenance related to synchronization of specification and actual code.
- Automatic generation of a VTR (Validation Test Report, or Log).

1.3 Organization of the Thesis

Chapter 2 of the thesis provides a literature review of COM technique. The emphasis is on the `interface` which is the bridge for the client and server to communicate with

each other. The chapter introduces two kinds of `interfaces`: `custom` and `dispatch interface`.

Chapter 3 gives the reason why we need such a tool like COM-tester to test COM based software.

Chapter 4 describes the design and implementation of COM-Tester. This is the core of this thesis. First of all, it presents the language elements (including `literal`, `variable`, `data structure`, `assignment`, `unification` etc.). Secondly, it shows the grammar of COM-Tester. Thirdly, it describes the architecture of COM-Tester. Then, it provides COM-Tester Abstract Machine (CAM). Finally, it gives the design and implementation of COM-Tester.

Chapter 5 shows the results from COM-Tester and includes a simple scripting test file.

Chapter 6 presents our conclusions and proposes future extensions to COM-Tester.

Appendix A contains a larger example of a script than the one in Chapter 5. Appendix B shows the result of running the script in Appendix A. Finally, Appendix C Lists some of the instructions executed by the CAM while running the script.

Chapter 2 An Introduction to COM

The Holy Grail of computing is to put applications together quickly and cheaply from reusable, maintainable code – preferably, code written by someone else. For many years now, experience and research have shown that object-oriented languages have a marked effect on the ability of software developers to write this kind of code. The ability to abstract concepts from a problem, and to turn them into classes and objects in a way that is fundamentally supported by the programming language, is a powerful draw for software engineers. The benefits of object-oriented techniques are there for all to see.

On its own, however, an object-oriented programming language is not sufficient for widespread reuse. As soon as we go beyond the idea of having a single developer or group of developers, the real world comes crashing in. The first problem we see is that developers throughout the world are programming in different languages.

As much as some well-known Californian companies would like us to simplify things by standardizing around a single programming language, it's never going to happen. The reasons for the diversity of languages in the world today are complex, beyond one company's control, and in many cases well founded. Some languages are better suited than others to a particular problem domain; some programmers have a natural preference for a particular language because it more closely reflects the way they think; some languages relate well to particular hardware. New languages and tools replace old ones because they're based on new ideas or take advantage of processor power that wasn't previously available. The popularity of a language responds to fashion and hype, and even to the quantity of good books teaching the subject. We live, and will continue to live, in a world of many tongues.

Now, a multitude of languages has its benefits, but the problem it presents to us is that it fragments the marketplace for reusable components. A Java class is of little use to a C++ developer, and a chunk of Visual Basic code won't help a COBOL programmer. If somebody writes a system in C++ today, will that effort be superceded five years from now by the arrival of a new programming language, as yet undreamed? It was issues like

these that drove the authors of the Component Object Model (COM) to their solution, which is to use language-neutral, binary components. Subject to a few considerations, this methodology allows developers to write components in whatever language they choose. It's the compiled code that matters, not the source code.

Of course, COM is neither the first nor the only way of reusing compiled code. C-style DLLs, for example, have been used extensively in Windows programming for a very long time, and their advantages are manifold **[1]**:

- They allow parts of an application to be swapped out or upgraded without the need for recompilation
- Code can be loaded on a just-in-time basis, so that it doesn't take up any memory if it's never needed
- Code can be shared between processes, which can be more memory-efficient than linking it statically (that is, compiling it into the application)

Given this list of important features, it will come as no surprise that COM components can themselves be packaged as DLLs. However, things are not as simple as that, C-style DLLs also have a number of disadvantages that COM must address:

- They expose only simple functions – they are not object-oriented.
- Traditionally, DLLs have been loaded by filename, which means that if the location or the name of a DLL Change, the application will not be able to load it.
- It is difficult to provide different versions of a DLL in the same system, because doing so can cause conflicts between different vendors' products.

Fortunately, COM overcomes all of these problems, and provides a number of other facilities we've yet to mention. So what is COM?

2.1 One sentence description of COM

Although COM is a complicated topic, we can write a simple one-sentence description that outlines its most important features **[2, 3]:**

"COM is a specification and a set of services that allow you to create modular, object-oriented, customizable and upgradeable, distributed applications using a number of programming languages."

Let's look more closely at that overlong sentence and its implications to get a fuller picture of the kinds of facilities offered by COM.

- COM is a specification
  The COM specification describes the standards that you need to follow in order to create interoperable COM components. This standard describes what COM components should look like and how they should behave.

- COM is a set of services
  The specification is backed up by a set of services or APIs. These services are provided by the COM library, which is part of the operating system on Win32 platforms, and available as a separate package for other operating systems.

- COM allows modular programming
  COM components can be packaged as DLLs or EXEs – COM provides the communication mechanism to allow components in different modules to talk to each other.

- COM is object-oriented
  COM components are true objects in the usual sense: they have identity, state and behavior, they support encapsulation. In certain circumstances, COM components can be treated polymorphically. Although they don't support implementation inheritance, they do support `interface` inheritance.

- COM enables easy customization and upgrades to your applications
  COM components link with each other dynamically, and COM defines standard ways of locating components and identifying their functionality, so individual components are swappable without having to recompile the entire application.

- COM enables distributed applications **[6]**
  COM provides a communication mechanism that enables components to interact across a network. More importantly, COM provides location transparency to

application (if desired) that enables them to be written without regard to the location of their components. The components can be moved without requiring any changes to the application.

- COM components can be written in many programming languages
  COM is a binary standard. Any language that can cope with the binary standard can create or use COM components. The number of languages and tools that support COM is sizable, with C, C++, Java, Jscript, Visual Basic, VBScript, Delphi, PowerBuilder, and MicroFocus Cobol forming just part of the list.

COM is not about any particular type of application. It's not about controls (that's Active X) [7]; it's not about compound documents (that's OLE); it's not about data access (that's OLE DB and ADO); and it's not about games and graphics (that's DirectX). But COM is the object model that underlies all these technologies. From the above list, you can see, an understanding of COM is vital to programming any of these technologies successfully.

2.2 The Component Object Model

General speaking, the component object model is built around the notions of components (often called `coclasses`), objects, and `interfaces`. These three different entities are defined and related as follows [3]:

- Loosely, as we have just been discussing, a `coclass` (named from component object class) is a piece of binary code that implements some kind of functionality. `Coclasses` can be distributed in DLLs, or in executable files. It is possible for a single module (DLL or executable) to contain more than one `coclass`.
- A COM object is an instance of a `coclass` that gets loaded into memory. (By the same token, one might say that a `coclass` is a 'blueprint' for a COM object.) It is not unreasonable (indeed, it is quite common) for more than one object of a given `coclass` to be active at a time.
- COM `interfaces` are the means – the only means – by which other components and other programs get access to the functionality of a COM component. An

`interface` is a set of definitions of logically-related methods that will control one aspect of the component's operation. Each component can have one or more `interfaces`.

Among these, the most important one is the COM `interface`. Let's have a closer look at it.

2.3 The Importance of Interface

Depending on the context in which it's being used, the word "`interface`" can have slightly different meanings. In the broadest possible terms, a COM `interface` is a group of definitions of methods that are usually related in the operations they perform. The methods in an `interface` can be called by using a pointer to that `interface`, and doing so results in the execution of the code in a COM object **[4]**.

A "human" component, for example, might have `interfaces` called `Imouth` and `Ihand`. These `interfaces` would group different methods – for example, `Imouth` might contain methods called `Eat()`, `Talk ()`, and `Kiss ()`, and `Ihand` might contain `Write()`, `Point ()` and `Scratch()`.

Choosing our words a little more carefully, we can say that an `interface` is an abstraction. The definition of an `interface` includes the syntax of the methods it contains (return types, parameter types and calling conventions), and the semantics of how to use them. To see how the latter can be important, consider that there are often restrictions placed upon implementers and users of an `interface` that just can't be described in code. A requirement such as the need to call an `Init()` method on the `interface` before calling any other method needs to be clearly documented, and forms the semantic part of the `interface` definition.

More carefully still, an `interface` actually has a very specific structure: it is any array of pointers to the implementations of the functions it contains – this is the binary standard that we mentioned earlier. Because the implementation of each function in the

interface is accessed by a pointer in an array, the precise order of the items in that array is an important part of the interface's definition.

Notice that the definition of an interface does not include an implementation of its methods. When a component says that it's going to implement an interface, it's up to that component to do so in a way that is both appropriate to itself and in accordance with the semantics defined for that interface. This separation leads to more robust design: interfaces can be reused in different situations, and a component that makes a particular interface available can be swapped with another component that makes available (exposes) the same interface. As a client, if you know that you need the functions for a particular interface, all you need to do is to find an object that implements it.

We began this chapter by talking about the desire to build reusable components, but from the point of view of the user (usually called the client), the important aspect is not what the component is, but what it can do. Because interfaces are the only way of making a component do anything at all, we can say that the functionality of a component is defined by the interfaces it exposes. For example, if you want to say that a coclass is both a lawyer and a philanthropist (and if you don't feel that's an oxymoron), you can do so by having it expose interfaces called (say) Ilawyer and Iphilanthropist.

The COM specification [2] includes details of a number of standard interfaces that Microsoft has defined. By implementing one of these interfaces, a component states that it supports some kind of functionality, or that it will work in some given situation. For example, a coclass that implements IsupportErrorInfo is able to return rich error information, while a component implementing IdataObject is capable of allowing data to be pasted or dropped into another application.

2.3.1 Interface as contract

As explained above, COM enforces complete encapsulation of the data and implementation of a component. A client can only call methods on the interfaces exposed by a component; it never gets direct access to the component's data.

This fact is what makes interfaces so fundamental, and when we link it to our earlier assertion about COM allowing easy customization and upgrading of applications, we can reach a couple of important conclusion;

- An interface, once defined, must never change. Published interfaces are immutable.
- Once a component has said that it exposes an interface, any future version of that component should also support that interface, to avoid existing clients from malfunctioning.

The interfaces that a component exposes represent a 'contract' between the component and its clients. A consequence of the second of these points when taken in the context of the first is that changes made to the contract in order to 'update' a component will surely break any existing clients, and so any revisions must be made with care.

2.3.2 Interface type

Broadly speaking, there are two types of interface in COM, categorized according to how the methods in the interfaces are accessed – static or dynamic invocation. Static invocation is the mechanism used by custom interfaces, and the traditional way the COM objects talk to each other. Dynamic invocation, on the other hand, is the means by which Automation interfaces go about their business.[3]

Static invocation is a contract between the client and a server object. The client knows exactly the number of methods in an interface, and the signatures of those methods. The object, for its side of the contract, must implement the methods described by the interface; if it does not, the two will not be able to communicate.

With custom interfaces, the only negotiation involved is the client querying for an interface with QueryInterface (), and therefore the client can only ask a server for interfaces that it already knows about. There is no other negotiation possible – the client can not ask the object to list the interfaces it supports, nor can it ask the object to tell it about the methods on those interfaces. Furthermore, if the object

only support interfaces that the client does not know about, the client cannot access the object.

Automation interfaces use dynamic invocation. Basically, Automation allows the client to ask an object to return information about the interfaces that it supports. Through type information, the object can list all the interfaces it supports and, when queried, it can return information about the methods on a specified interface. Using this information, the client can invoke a method dynamically – in other words, the client can package the parameters in a generic way, and then tell the object to call a particular method with those parameters. This invocation is done on the fly, and can be performed with no prior knowledge of the object.

2.3.3 Automation Mechanism

Automation objects use a standard interface called Idispatch. You can tell whether an object can be 'automated' by querying for this interface **[5]**.

Server objects implement Automation by allowing clients to use a method on the Idispatch interface to call other methods. The collection of methods it makes available in this way is called a dispinterface (short for 'dispatch interface'). Idispatch allows an object to indicate what dispinterface methods it supports in two different ways. It can be done at runtime, through other Idispatch methods; or else the object can be a little more expressive and maintain information about its dispinterfaces that a client can use at compile time. The client can still ask the object for information about its dispinterfaces, and can call those interface methods dynamically.

The information that the object provides must include details of all the disprinterfaces it supports, all the methods on those dispinterfaces, and all the parameters of those methods. For an object to be an Automation object, a client must be able to get access to this information, which the object can choose to supply either directly or (more typically) through the type library files(with tlb extension) which are being usefully employed as suppliers of the type information that's used by Automation clients.

A type library is effectively a tokenized version of the IDL file, and it describes all the `interfaces` supported by the object, all the methods, and the parameters of those methods. Since a type library describes exactly what an `interface` can do, it can be used to marshal data between processes.

The rules of COM allow an `Automation` object to expose its `interfaces` both through `Idispatch` and through the `vtable` of a `custom interface`. Such an `interface` is called a `dual interface`. Nowadays, most projects implement `dual interfaces`, so the calling object can take the advantage of different `interface`.

2.3.4 The need for `Automation`

When we are programming for clients that only support `Automation interfaces`, we have to use `Automation` rather than `custom interfaces`. Such clients include those that use VBA and VBScript etc.

For example, when a web developer writes VBScript code, the code is not compiled and so there is no checking on the methods that are called or on the data types of the parameters that are passed. Since a VBScript client passes parameters without knowing beforehand what the types of those parameters are, this check must be done at runtime by the object, using type information. If the object does not support a particular method, then the call will fail. If the data passed from the client is of wrong type, the object can attempt to coerce the data to the required type.

Since COM-Tester(We will see it soon) is a kind of script language, we will use `Automation` to design and implement it.

Chapter 3 The Need for COM-Tester

3.1 Testing cost is high

From last chapter, we can see how important COM is in modern programming. Today COM is everywhere, COM programming is the trend and center of Microsoft-related software.

The software development cycle, according to the theory of software engineering,[8] includes three major phases: requirements/design; implementation; and testing. The cost of each phase for different types of software systems is different. However, the figures (Table 3.1) given by Boehm (1975) [9] are probably still approximately correct.

Table 3.1 Relative costs of software systems

System type	Phase costs(%)		
	Requirements/design	Implementation	Testing
Command/control systems	46	20	34
Spaceborne systems	34	20	46
Operating systems	33	17	50
Scientific systems	44	26	30
Business systems	44	28	28

The inference which can be drawn from the above figures is that the software development costs are greatest at the beginning and at the end of the development cycle. This suggests that a reduction in overall software development costs is best accomplished by more effective software design and effective testing. Reducing testing cost is the main concern after implementation.

3.2 Why we test software

We test software in order to:

1. Provide programmers information that they can use to prevent bugs.
2. Give management information it needs to rationally evaluate the risk of using the object.
3. Achieve an object as bug-free as warranted by the situation.
4. Falsify the object with respect to stated and unstated requirements; also called, "breaking the software".
5. Validate the object; that is, show that it works.

Anything written by people has bugs. Not testing something is equivalent to asserting that it's bug-free. Programmers may make mistakes – especially of all the possible interactions between features and between different pieces of software.

The penultimate objective of testing is to gather management information. Given enough testing, we can make reasonably comfortable predictions about the software's fitness for use.

The highest goal of testing is to support quality assurance: to gather information that, when fed back to programmers, results in avoidance of past mistakes and in better future software.

3.3 COM-Tester comes to the rescue

Normally, to test a component object, the developer or tester has to develop a client using some kind of programming language. This is really cost ineffective, because, each time you develop a new component, you have to develop a new client to test it. This increases the cost. Meanwhile, it is painful to maintain the consistency of the software specification.

In the COM world, the `interface` is everything. The only way for the application to communicate with a COM object is through the object's `interface`. This feature provides us with the opportunity to develop a general tool to test the component object with the following general requirement:

- A way to specify unitary test cases;
- A way to specify more complex functional/integration test cases;
- A driver for a test campaign;
- A way to mix test coding and documentation in a single source file.

This general tool is COM-Tester. The next chapter will describe its design and implementation.

Chapter 4 Design and Implementation

COM-Tester is a small script testing tool designed for writing test scenarios, especially for the testing of Microsoft COM-based software. The main goal of the tool is to reduce (in the short as well as in the long term) the costs related to the testing activities of Microsoft COM-based software. The core distinctive features of COM-Tester are:

- A high-level simple interpreted language where the notion of a test is directly supported by a specific syntax.
- Primitives to create and work easily with COM components. `Interface` information (function prototypes) and data type information (classes and structures) are automatically extracted from TLBs (type libraries) or the standard `IDispatch interface`.
- Merging of source code and documentation. This enables documentation to be extracted from the code, thus easing maintenance related to synchronization of specification and actual code.
- Automatic generation of a VTR (Validation Test Report, or Log).

The following sections in this chapter will show the detailed description of design and implementation of COM-Tester. It will cover the topics from language elements, grammar, architecture to the COM-Tester Abstract Machine (CAM) as well as the class level description of the implementation.

4.1 Language Elements

This section presents the lexicon, syntax and semantics of COM-Tester, going from the basic elements to more complex functionalities.

4.1.1 Literal

There are three types of simple (primitive) data in COM-Tester: boolean, numbers and strings.

Booleans

The two boolean literal are the reserved keywords **true** and **false**.

Numbers

Numbers can be written in decimal or hexadecimal following the notation of the C programming language.

Strings

Strings follow the conventions of C too. That is, they are written using double quotes (") and meta-characters such as '\t', '\n', etc. are supported.

4.1.2 Identifiers and variables

Identifiers are written as in C: combined with letters, digits and underscore, not beginning with a digit. Identifiers are used to name variables, functions and tests. The underscore alone (_) is reserved for the anonymous variable. The anonymous variable stands just for a memory slot which is never referred to - a "don't care"; more on it below. Variables are not declared. They are simply introduced, either by binding them to some expression or by passing them as unbound arguments to a function:

```
HelloStr = "Hi, everybody!"
```

An important note is that variables get their scope from the context in which they first appear. Consequently, the lifetime of a variable is the same as the activation time of the scope in which it first appeared. Variables with the same name used in

unrelated scopes are also unrelated. That is, a `variable` used for the first time in a test or a function has nothing to do with a `variable` with the same name in another test or function. This follows from the lexical scoping rules of COM-Tester.

The fact that `variables` are not declared before their use implies that it is impossible to distinguish between local and global `variables`. Hence, a `variable` in an outer scope is NOT immediately visible in the inner scope. To make it clearer, this means that global `variables` are not available in functions or tests. To access a global `variable` in a function or a test, the '$' operator alone must be used in front of the `Identifier`. For example:

```
program_name = "Namer Server"

test USE_GLOBAL_1
header = program_name     // Probably unexpected behaviour,
                          //program_name is unbound here.
   // ...
end

test USE_GLOBAL_2
header = $program_name  // OK, header is "Namer Server"
                        // now.
     // ...
end
```

Of course, at the top level, the '$' operator may be omitted.

4.1.3 Lists and Data Structures

The only compound `Data Structures` that can be defined in COM-Tester are `Lists`. A `list` is a sufficiently general structure to represent any other structure. A

`list` is an ordered collection of optionally named elements. Elements are separated by commas (,) and can contain any value, in particular a nested `list`. Here is an example:

```
L1 = [1, "Wrong password", [0xFE, 0xA9, 0x9000]]
```

Naming `list` elements is done with the '=' operator:

```
status = 0x9000
L2 = [status, ub, count=1, message="The data is wrong",
result = [0xFE, 0xA9]]
```

Here, the first element of the `list` takes the value of the `variable` 'status' (i.e. 0x9000); the second has neither a value nor a name, it is an empty slot; the third has the name 'count' and the value 1; etc.

Another way to specify empty slots is to write the special `literal` '_' which represents an anonymous always unbound `variable`:

```
[count=1, _, str=" Hello World!"]    \\ unnamed empty slot
```

or

```
[unbound=_, count=1]   \\ named empty slot
```

Elements can be accessed through table indexing (starting from 1):

```
L1{2} → "Wrong password"
```

Or with the dot operator '.':

```
L2.message → "The data is wrong"
```

Negative numeric indexes specify a position from the end of the `list`(starting from –1):

```
L1{-3} → 1
```

Element access operators all lead to a lvalue. That is, they can be assigned to:

```
L2.message = "This program has a high security risk"
```

Note however that the name of an element cannot be changed once it has been specified in the `list literal`. Loosely speaking, those two access modes let one view a `list` as either an array of values indexed with integers or an object with members accessed through the conventional scope access operator ".".

The empty `list` is written as might be expected: []. It is a runtime error to try to access an element of the empty `list`.

4.1.4 `Assignment` and `Unification`

The usual `Assignment` operator '=' is present in COM-Tester. The expression on its left must evaluate to a `variable` reference (i.e. a memory slot) - either a `variable` name or a `list` element. On the right hand side, the expression could be anything. `Assignment` is right associative and the result of an `Assignment` expression is the value of the expression on the right hand side.

The `Unification` operator '<=>' behaves as in Prolog. That is, two expressions unify either if they evaluate to the same value (a `number`, a `boolean`, a `string` or a `list`) or at least one of them is an unbound `variable reference`. Note that the `Unification` of two values fails if they are not of the same type (i.e. both are `numbers` or both are `strings` or `booleans` or `Lists`). Said in other words, even

though variables do not have types, values do, and, moreover, there's no automatic conversion from one type to another. Lists unify if and only if their heads unify and their tails unify recursively. The result of the Unification operator is a boolean (true or false).
This language decision (Unification) relates to the special COM testing case generation. Normally, the functions in COM interfaces return HRESULT, which is simply a 32-bit number containing a structured error code. For example, S_OK(0x00000000), S_FALSE(0x00000001), etc. In COM-Tester, we need to compare the result of function call with HRESULT to know if the test steps pass or not.

It may seem uncommon and somewhat messy to allow Assignment and Unification at the same time. Indeed very "unstructured" programs can be written in COM-Tester. However, the goal of COM-Tester is not to have clean semantics, but to be useful. With both Assignment and Unification, side effects percolate very easily. Consider :

```
f(param) is
     ...
     param = "Hello world"
end

a <=> b
b <=> c
d <=> a

f(d)
```

Assuming that a, b, c and d are initially unbound, the last line in the above example will put the string "Hello world" in all four variables a, b, c and d. If we replace the Unification (<=>) by Assignment (=), we would end up with only d being modified, because assigning to a variable binds it only to a value and NOT to another

variable. The "special case" of assigning to an unbound variable is handled simply by unbounding to assigned variable:

```
str = "Here is a bound variable: str"  //"str" gets a value
str = unbound                          //"str" looses its value
```

4.1.5 Arithmetic and Other Common Operations

In this version of COM-Tester, number arithmetic, bitwise, string and list operations are supported only through predefined functions. See the list of those in **Section 4.1.8** below.

4.1.6 Functions

In COM-Tester, a function's definition and behavior resemble logic programming rules. The body of a function is simply a sequence of expressions. As soon as an expression fails to execute (for example because some Unification failed), the function exits.

Parameters are always passed by Unification. Going from left to right, as soon as an actual parameter does not unify with a formal one, the function call fails.

A notable difference of COM-Tester functions from conventional logic rules is that they may have return values. Return values are specified through Assignment to the reserved keyword **retval**. Assigning to **retval** does NOT cause the function to exit – it simply specifies the value to be returned as the result if none of the expressions in the function fails. Multiple Assignments of **revtal** are permitted and only the last one is taken.

An important point about return values is that they are really values, not references. Whereas a variable reference passed as a parameter may have its value changed inside the body of the function (a side effect), the result of a function, even if it's a variable reference, may not be changed by the caller. In short, a function never returns an *lvalue*.

Functions may be called recursively, but this is not supported in this version of COM-Tester because of the absence of backtracking and conditionals which gives no way to stop the recursion.

For the precise syntax of a function definition, please see the COM-Tester grammar below(Section 4.2). Note that, even though COM-Tester functions are modeled after Prolog rules, the specific syntax of function definition (rule declaration) departs a little bit. In particular, the if ':-' is replaced with an 'is' and there are no commas to separate expressions. This is intentional since the presence of global `variables` and `Assignment` in COM-Tester breaks heavily the clear logical semantics of the ':-' and ',' operators of Prolog.

4.1.7 Conditionals And Iteration

In this version of COM-Tester, there's no provision for conditionals ('if' statements) and iteration ('for', 'while', 'do'). They will be added as the need arises, but, ideally, backtracking should be implemented. This is a key point to be decided.

4.1.8 Predefined Functions

This section documents the built-in functions available in COM-Tester. Due to the current lack of operators (`arithmetic`, `list` and `string` manipulation), they would undoubtedly prove essential to any serious and useful COM-Tester test script. All of the built-ins can be overridden by simply defining a function with the same name.

`trace(X)`

Write the object X in the trace file (specified on the command line). X may be anything from a `Boolean` value to a `list` with several levels of nesting. Unbound references are written as underscores (_). This function won't fail or produce a runtime error under any circumstance. If a trace file is not specified when the COM-Tester interpreter is invoked,

the function call is ignored during compilation and there are no performance penalties. The return value is the object X itself and it is available even if COM-Tester is invoked without trace command specified.

`list(N)`

Creates a new `list` with N unbound, unnamed elements. If N is not a positive integer, a runtime error is reported.

`head(L)`

The head(L) function is equivalent to L[1]. It returns the first element of `list`. If L is empty or it is not a `list` at all, a runtime error is reported.

`tail(L)`

Returns the tail of the `list` L. That is, the `list` L without its first element. If L is not a `list`, a runtime error is reported. If L is empty, the result is the empty `list` too.

`append(L1, L2)`

Returns a newly constructed `list` which is the concatenation of L1 and L2. If one of L1 or L2 is not a `list`, a runtime error is reported.

`reverse(L)`

Returns a newly constructed `list` which is formed by reversing the order of the elements of L. If L is not a `list`, a runtime error is reported.

`insert(L, Elem, Pos)`

Returns a newly constructed `list` obtained by inserted the element Elem at position Pos in the `list` L. If L is not a `list` or Pos is not a valid integer position in L, a runtime error is reported.

`sublist(L, Start, End)`

Returns a newly constructed `list` obtained from the elements of L from position Start to position End included. If L is not a `list` or Start > End or one Start, End is not a valid integer position in L, a runtime error is reported.

`remove(L, Start, End)`

Returns a newly constructed `list` obtained from L after removing all elements from position Start to position End included. If L is not a `list` or Start > End or one Start, End is not a valid integer position in L, a runtime error is reported.

`flatten(L)`

Returns a newly constructed `list` obtained from L after expanding all nested `Lists` at their positions. If L is not a `list`, a runtime error is reported.

`copy(L1, Start, End, L2, Pos)`

Copy the elements of `list` L1 from position Start to position End included at position Pos in the `list` L2. If either L1 or L2 is not a `list` or Start > End or one of Start, End, Pos is not a valid integer position in the respective `Lists`, or L2 is not big enough to hold all elements a runtime error is reported.

`add(N, M)`

Return the result of adding the `numbers` N and M. If either N or M is not a `number`, a runtime error is reported.

`sub(N, M)`

Return the result of subtracting the `numbers` N and M. If either N or M is not a `number`, a runtime error is reported.

`mul(N, M)`

Return the result of multiplying the `numbers` N and M. If either N or M is not a `number`, a runtime error is reported.

`div(N, M)`

Return the result of dividing (integral quotient) the `numbers` N and M. If either N or M is not a `number`, a runtime error is reported.

`mod(N, M)`

Return the remainder of the division of N by M. If either N or M is not a `number`, a runtime error is reported.

`concat(S1, S2)`

Returns a newly constructed `string` which is the concatenation of `string` S1 and S2. If one of S1 or S2 is not a `string`, a runtime error is reported.

`substring(S, Start, End)`

Returns a newly constructed `string` which is part of `string` S starting from Start and ending at End. If S is not a `string`, a runtime error is reported.

`length(X)`

Returns the length of X which may be a `list` (in which the `number` of elements of the `list` is returned) or a `string` (in which the `number` of characters is returned). If X is neither a `list` nor a `string`, a runtime error is reported.

The following functions related to COM:

`com_get_object(L) L=[S1,S2,S3]`

There are three arguments in the argument `list` L, they are class ID S1, `interface` ID S2 and type library path S3 respectively. If class ID or `interface` ID is empty `string` or not a `string` at all, a runtime error is reported.
This function call will return a newly created com object ID.

`com_get_instance(L) L=[S1,S2,S3]`

There are three arguments in the argument `list` L, they are class ID S1, `interface` ID S2 and type library path S3 respectively. If class ID or `interface` ID is empty `string` or not a `string` at all, a runtime error is reported.
This function call will return a newly created com `interface` object ID.

`com_release(N)`

This function call will release the com object identified by object id N.

`com_call(N,S)`

This function call will call the method specified by S in the com object identified by object id N.

4.1.9 Tests

Tests are simply sequences of expressions. A test is specified by the reserved keyword **test**, followed by the name of the test, followed by a sequence of expressions and terminating with the keyword **end**. If any of the expressions in the test body fails, the test fails too. So, essentially tests are, for now, based on the process of setting up parameters and return values, and calling COM object functions to unify with the results. Again, please read the usage example in next chapter and the grammar (section 4.2) for the complete picture.

4.1.10 Comments and Documentation Generation

There are two types of comments in COM-Tester: code comments and documentation comments. Code comments are ignored by the parser whereas documentation comments are extracted for the generation of test specifications and validation test reports. Code comments are as in C++ with the slight modification that multi-line comments can be nested; they can appear anywhere. Documentation comments are arbitrary Lists of white-space separated tokens and are delimited by the special character sequences '<<' (for

beginning) and '>>' (for end); they may only appear in meaningful places. The spaces inside documentation comments are only separators and are ignored in a document production. A documentation comment is always associated the syntactic construct that follows it. Thus a <<...>> comment appearing immediately before the keyword **test** is taken as the description of the test. If it appears before an expression in a test, it is taken as the description of the test step (As it would be written in a Validation Test Specification).

Generated documentation is in HTML format because it is an universal, simple yet sufficient choice for the kind of typesetting that we want. It is also possible to generate plain text which might prove desirable if further processing is needed and/or if the use of an HTML browser proves too heavy.

For comment examples, see **Chapter 5**.

4.2 Grammar

The following is a complete grammar of COM-Tester. Production names are surrounded with the '<' and '>' signs. Tokens are written in double quotes. Alternatives in a production are separated by a '|', and a whole production is terminated with ';'. The floating production *<opt-doc.>* indicates places where a documentation comment may appear.

```
<test-file>          : <opt-doc> <definition-list>
                     ;
<definition-list>    : <opt-doc> <definition> <definition-list>
                     |
                     ;
<definition>         : <function-definition>
                     | <test-definition>
```

```
                    | <variable-binding>
                    | <include-directive>
                    ;
<include-           : "include" <string>
directive>          ;
<variable-binding>  : <var_ref> <assignment>
                    ;
<test-definition>   : "test" <id> <expression-list> "end"
                    ;
<function-          : <id> "(" <formal-arguments> ")" "is"
definition>                    <expression-list> "end"
                    ;
<expression-list>   : <opt-doc> <expression> <expression-list>
                    |
                    ;
<expression>        : <literal>
                    | <var_ref> <opt-assignment>
                    | <function-call>
                    | <expression> "<=>" <expression>
                    ;
<opt-assignment>    : <assignment>
                    |
                    ;
<assignment>        : "=" <expression>
                    ;
<var_ref>           : <opt-global-spec> <id> <opt-access-spec>
                    | "_"
                    ;
<opt-global-spec>   : "$"
                    |
                    ;
```

```
<opt-access-spec>   : "." <var_ref>
                    | "{" <expression> "}" <opt-access-spec>
                    |
                    ;
<function-call>     : <var_ref> "(" <funcparam-list> ")"
                    ;
<funcparam-list>    : <funcparam> <opt-funcparam-rest>
                    |
                    ;
<opt-funcparam-     : ","  <funcparam> <opt-funcparam-rest>
rest>               |
                    ;
<funcparam>         : <literal>
                    | <var_ref>
                    ;
<literal>           : <number>
                    | <string>
                    | <list>
                    | "_"
                    ;
<list>              : "[" <list-elements> "]"
                    ;
<list-elements>     : <list-elem> <opt-list-rest>
                    |
                    ;
<opt-list-rest>     : "," <list-elem> <opt-list-rest>
                    |
                    ;
<list-elem>         : <id> "=" <literal>
                    | <literal>
                    ;
```

```
<formal-arguments>  : <formal-arg> <opt-args-rest>
                    |
                    ;
<opt-args-rest>     : "," <formal-arg> <opt-args-rest>
                    |
                    ;
<formal-arg>        : <id>
                    | <literal>
                    ;
```

4.3 Architecture

Here is a list of the principal modules developed for this first version of COM-Tester together with a brief description of each:

4.3.1 COM Manager

This module is responsible for the management of COM object: creation, destruction, reading the TLBs or calling the `IDispatch interface`, function invocation, mapping COM typed values to COM-Tester values and vice-versa (e.g. COM structures are simply represented as `Lists` with named elements which can be accessed with the usual "." operator), etc.

4.3.2 Interpreter

The Interpreter module is the core of COM-Tester. It defines the run-time model, memory domains, primitive operations and virtual machine. It uses the COM Manager module and the Doc. Generation Module.

4.3.3 Parser

The Parser module reads a source file in textual form and transforms into intermediary form for input to the Interpreter and the Doc. Generation Module.

4.3.4 Doc. Generation Module

The Doc. Generation Module contains the extracted documentation comments in a structure following the test file. It also keeps an execution trace input by the Interpreter. The module is thus able to generate a VTS, a VTR and a trace.

4.3.5 Main Driver

The Main Driver simply brings all the pieces together. It reads command line arguments, reports parsing errors to the user and manages input/output files.

4.4 The COM-Tester Abstract Machine

4.4.1 Overview Of The Abstract Machine

The COM-Tester Abstract Machine (CAM) is somewhat inspired from Warren's Abstract Machine (the WAM) **[10]** which provides a well-known basis for Prolog like languages. In our first version only certain elements of the WAM are borrowed and care has been taken to ensure that the CAM will scale up to a full-featured backtracking machine with an execution model a lot closer to the real WAM. The specifics of COM-Tester force us, however, to abandon the letter of the WAM and to follow just the spirit. One of the major sources of modification is the presence of `assignment` and global `variables`. Because evaluation of expressions may in theory consume a lot of space and have a lot of side effects, `assignment` alone causes complications to the original WAM design. Another is the COM-Tester `list` construct with named elements, acting as a `list`, an array or an object (structure).

The CAM follows the common pattern of other virtual machines. It is organized around several memory domains, an instruction set and an execution driver. The most notable memory domain is the *Data Stack* (or *Heap*) (see section 4.4.2) which is presently used for pretty much everything – permanent store, evaluation stack, procedure activation stack. The other important memory domain is the *Code Area*. As its name suggests, there resides the compiled (COM-Tester → CAM) code.

The following global registers are used during execution:

DST – the Data Stack top, containing the address in the *Data Stack* of the next available cell.

AST – the Address Stack top.

DFP – the Data Stack frame pointer containing the address in the *Data Stack* of the beginning of local `variables` for the current procedure being executed.

AFP – the Address Stack frame pointer.

PC – the program counter, containing the address in the *Code Area* of the current instruction being executed.

4.4.2 Memory Domains

Data Stack

The data stack is organized as a vector of data cells. Each data cell is a structure of 3 elements:

- A `tag` indicating the type of the data cell.
- An `integral value (IValue)` which can represent a `number` constant, a pointer elsewhere in the data stack or a count of something.
- A `string` value (`SValue`) which can represent either a `string` constant, a name of a `variable` or a name of a function.

The following table summarizes the possible `tags` together with the corresponding interpretation of the IValue and SValue fields:

Tag	IValue	SValue
BOOL	`0 for false, 1 for true.`	`Unused.`
NUMBER	`The number constant.`	`Unused.`

STRING	Unused.	The string constant.
VAR	The index (pointer) in the Data Stack of the value of the variable. It may a link to another VAR cell (created by unification) or to a constant or a list (created by unification or assignment). By convention, an unbound variable is a one pointing to itself.	The name of the variable or the empty string if this is simply an unnamed reference (for example an unnamed list element).
LIST	The number of element in the list. See below for a complete description of how lists are represented.	Unused.

Given the above table, the representation of `Boolean`, `number` and `string` constants as well as of `variables` is obvious. `Lists` are a bit trickier because we would like to make both array-like operations (integer indexing) and standard `list` operations (head and tail) efficient. Moreover, we want `list` elements to be seen as memory slots which can be set and reset to different values preferably in constant time. This last fact forces us to have a VAR (reference) cell for each element in a `list` in addition to a constant value cell as is the case for the classic WAM. Now we can organize those reference cells in two ways: either as a chain of (value, next) pairs which offers an efficient representation for mutating `list` structures (as is the case in the WAM and most implementations of `list` structures); or as a vector of consecutive cells which is efficient for array like indexing. We choose the later because the syntax and predefined functions of COM-Tester encourage precisely this style of programming. That is, array indexing is easy to write and is likely to be common when working with data buffers whereas all predefined `list` functions (except head and tail) construct a new `list` and there is no provision for mutating an existing one. Hence, a `list` might be represented as a `LIST`-tagged data cell followed by N (the `number` of elements in the `list`) consecutive reference cells each one pointing either to itself (when the element is unbound) or to another cell containing the value of the `list` element. This is indeed the direction we take, but with a slight modification in order to make the tail operation efficient: what is immediately below the `LIST`-tagged cell is a reference cell pointing to the beginning of the `list` elements. The `LIST`-tagged itself contains (in its IValue) the `number` of elements.

Here is an example of a `list` representation in the data stack. The first column stands for addresses in the Data Stack, the second for data cell `tags` and the third for data cell values where both the IValue and the SValue are given or only one of them when the other one is unused:

```
L = ["s", 2, t = 3, [n = 55, "humpty"], msg = "forget it"]
→
```

0	STRING	"s"
1	NUMBER	2
2	NUMBER	3
3	NUMBER	55
4	STRING	"humpty"
5	VAR	"n", 3
6	VAR	"", 4
7	LST	2
8	STRING	"forget it"
9	VAR	"", 0
10	VAR	"", 1
11	VAR	"t", 2
12	VAR	"", 8
13	VAR	"msg", 9
14	LST	5

Address Stack

Another meaningful name for this memory domain could the "Evaluation Stack". The cells are simply pointers (integral values) into the Data Stack.

COM Objects Representation

COM objects are accessed through `interfaces`. For a given instance, we might have multiple `interfaces` in use at any given time. CAM maintains a run-time table of currently active COM `interfaces` each of which is identified by a unique integer (assigned at run-time). It is through this unique id that the `variables` in COM-Tester refer to a COM `interface`.

Code Area

The code area is a vector of instruction cells. The instruction cells consist of an opcode and a `list` of arguments. The section on CAM instructions presents a `list` of all instruction together with their arguments and semantics.

4.4.3 CAM Instructions

`putvar <string>`	Create a new unbound variable with the given name on the Data Stack.
`putstring <string>`	Create a new string constant on the Data Stack and push its address on top of the Address Stack.
`putnumber <number>`	Create a new numeric constant on the Data Stack and push its address on top of the Address Stack.
`putbool <true | false>`	Create a new boolean constant on the Data Stack and push its address on top of the Address Stack.
`putref <string>`	Create a new reference cell with the given name on top of the Data Stack. The cell will point to the address specified on top of the Address Stack. The latter is removed as a final step of the operation.
`putlist <n>`	Create a new list pair of cells on the Data Stack and push the address of the LIST-tagged cell on top of the Address Stack. The reference cell points to the one immediately below.
`push <n>`	Push the address specified by <n> on top of the Address Stack.
`pop`	Remove the top of the Address Stack.
`getlocal <n>`	Push the Data Stack address of the nth local variable in the current

	activation scope onto the Address Stack.
getarg <n>	Push the Data Stack of the nth actual argument to the current procedure onto the Address Stack. Arguments are numbered starting from 1.
set	Bind the next top address in the Address Stack to the top address in the Address Stack. Remove the top of the Address Stack.
unify	Unify the data cells pointed to by the top and next to top addresses in the Address Stack. Used only for the output parameters.
Equal	Compare the data cells pointed to by the top and next to top addresses in the Address Stack. Pop 2 times the Address Stack. Create a new Boolean cell on the Data Stack containing the result of the operation and push its address onto the Address Stack.
Not equal	Compare the data cells pointed to by the top and next to top addresses in the Address Stack. Pop 2 times the Address Stack. Create a new Boolean cell on the Data Stack containing the result of the operation and push its address onto the Address Stack.
ideref	If the top address in the Address Stack does not bind to an integer or the next to top address does not bind to a list, report an error; else find the position in the list specified by the integer, reporting a run-time error if not existing, and push its address on the Data Stack after removing the top addresses.
sderef <string>	If the top of the Address Stack does not point to a list, report a run-time error. If the specified <string> is not the name of an element of the

	list, report an error. Otherwise, push the Data Stack address of the element specified by <string> on the Address Stack after removing its top element.
call <n>, <string>	Call a function with <n> arguments. If <string> is non empty, it is taken as the global name of the function. Else the function will be looked up via the Data Stack element pointed to by the (n+1)th next to top element on the Address Stack.
ret	Return from a function.
runtest <string>	Run the test specified by <string> in the current environment of global variables.
putcom <n>	Create a new com object with the given id on top of the Data Stack and push its address on top of the Address Stack.

4.4.4 Compilation (COM-Tester → CAM)

Constants

A constant (Boolean, integer or string) is compiled by simply pushing it onto the Data Stack via the corresponding instruction:

```
putbool <value> or
putstring "..."
```

Variables

There are several instructions relating to the treatment of variables. The putvar and getlocal instructions are specifically designed for that. The putvar instruction will create a new variable on the Data Stack. The main program as well as every test and function

begin with a sequence of putvar instructions to create the global (for the main program) or local (for tests and functions) `variables`. References to global `variables` in expressions are compiled with the push instruction because their addresses are known at compile time whereas references to local `variables` are compiled with the getlocal instructions because their absolute addresses are only available at run time (via the frame pointer register). The necessity of getlocal comes from our simple addressing scheme (only absolute addressing allowed).

A special case is the anonymous `variable` '_' . It has a special status in that it is always unbound. To ensure this, we generate a separate unbound, unnamed `variable` for each occurrence of '_' in the source text.

Variable References

A `variable` reference is a top-level `variable` followed optionally by one or more access specifiers. The top-level `variable` is compiled as specified in the previous section. Access specifier can be either integers or names. A special instruction is provided for each case. For the array indexing form ({}), the expression inside the braces is compiled and then a 'ideref' instruction is generated. The check that the result is indeed an integer is made at run time. Similarly for the name access (with '.'), there is the special instruction 'sderef <`string`>' expecting the top of the stack to evaluate to a `list` and <`string`> to be a valid name for an element of this `list`. For example, if the `list` L is at address 100 and 'i' at address 77, then the expression L{i}.n is compiled as follows:

```
push 100
push 77
ideref
sderef "n"
```

Variable Bindings

Compilation of `variable` bindings amounts to compiling the `variable` reference on the left of the `Assignment` operator, compiling the expression on the right and generating a 'set' instruction.

```
N = 255  →
        push <address of 'N'>
        putnumber 255
        set
```

Lists

`List literal` are compiled by first compiling the expressions for all sub-elements. This leaves the Address Stack with (say) n Data Stack addresses on top pointing to the n elements of the `list`. Then for each element, a new reference is created on the Data Stack via the putref instruction. Note that because putref operates on the top of the Address Stack, the references are actually created in reverse (to the source code) order. Finally, a `list` cell with the `number` of `list` elements is created on top of the Data Stack. For example:

```
[1, x = 2, msg = "Hi"] →
      0:putnumber 1
      1:putnumber 2
      2:putstring "Hi"
      3:putref "msg"
      4:putref "x"
      5:putref ""
      6:putlist 3
```

Unification

`Unification` is compiled the same way as `variable` binding except that the 'set' instruction is replaced by the 'unify' instruction.

Function Calls

Calling a function is done as in any other language. Compile the expressions forming the actual call parameters. This leads to all parameters being on top of the Data Stack. Then generate a 'call' instruction with the function's (usually address, but in our case) name:

```
Fun(<e1>, <e2>, ..., <en>) →
      <code for e1>
      ...
      <code for en>
      call n, "Fun"
```

A complete example

Here is the sample output from Chapter 5 sample test program(in cam.txt)

```
---> Main program   <---

putvar          S_OK
putvar          namer_object_guid
putvar          iget_guid
push            0
putnumber       0
set
push            1
putstring       {39257021-1402-11d1-8AAD-0020182A59AB}
set
push            2
putstring       {39257022-1402-11d1-8AAD-0020182A59AB}
set
```

```
runtest         NRM_GET

---> TEST: NRM_GET  <---
putvar          iget_object
putvar          buffer
<<STEP:  Create an instance of Iget >>
getlocal        0
push            1
push            2
putstring       atl_server.tlb
putref          tlb
putref          interfaceid
putref          classid
putlist         3
call            1, com_get_instance
set
pop
<<STEP:  Call GetBuf function in the interface>>
getlocal        1
getlocal        0
sderef          GetBuf
putnumber       81
call            1,
set
pop
<<STEP:  Compare string to verify the result >>
getlocal        1
putstring       COM-Tester Version 1.1
equal
pop
```

Appendix C is the example generated by CAM from the example of **Appendix A** :

4.5 The Design of COM-Tester

This section presents the low-level C++ design of the COM-Tester interpreter. The important classes are described, their responsibilities, their interaction and their implementation.

To make life easier, our description will follow the principal modules developed for this version of COM-Tester:

4.5.1 COM Manager Module

There are two important classes in this module: com_method and com_manager. They are responsible for the management of COM object: creation, destruction, reading the TLBs or calling the IDispatch `interface`, function invocation etc.
Com_manager class is a singleton for housekeeping of COM objects and Type libraries.
Com_method class manages a COM method given an object that exposes it. A type descriptions needed in the call.

In addition, `camcom` class maps COM typed values to COM-Tester values and vice-verse (e.g. COM structures are simply represented as `Lists` with named elements which can be accessed with the usual "." operator), etc.

4.5.2 Interpreter Module

The Interpreter module is the core of COM-Tester. It defines the run-time model, memory domains, primitive operations and virtual machine. It uses the COM Manager module and the Doc. Generation Module.
The major classes in this module include cam, ct_object and instr_cell.
Cam class represents the actual CAM interpreter. It manages the memory domain and CAM objects. It handles the main interpretion loop, interaction between various COM-

Tester runtime objects (such as functions and tests). It provides a main execution driver and event driven `interface` to the document generation and debugging facilities.
Ct_object class contains the COM-Tester objects. COM-Tester objects are tests, user defined functions and builtin functions. Each object has its own code segment and documentation, manages local `variables` and is capable of executing itself (given a camT object) and printing itself to an output stream.
Instr_cell class implements all instructions of the CAM (COM-Tester Abstract Machine). Please refer to **Section 4.4.3** for a precise description of each instruction and the CAM architecture.

4.5.3 Parser Module

The Parser module reads a source file in textual form and transforms into intermediary form for input to the Interpreter and the Doc. Generation Module.
In this module, there are two important classes: ct_parser and slex.
Ct_parser implements a strightforward recursive descent parser. It follows exactly the grammar specified in this documentation. That is, to every grammar production there corresponds a function that recognizes that
production and generates the corresponding CAM (COM-Tester Abstract Machine) code.
Lexical analysis is done with the 'slex' and related classes. Code is generated using 'code_segmentT' objects which act as arrays of CAM instructions. Every parsing function (except the top-level 'TestDefinition' and 'FunctionDefinition') returns a 'code_segmentT' containing the code has been generated. If this code segment is empty, that means that the function did not recognize the input as matching the corresponding grammar production.
The global `variable` 'current_object' contains at whenever the current test or function definition is parsed. This enables the parser to be aware of the current scope at any given time during processing.
Slex is a simple lexical analyser for common languages. It is a set of classes for simple lexical analysis of languages with C-like lexical elements. Many things are predefined and the configurable all have defaults.

4.5.4 Doc. Generation Module

The Doc. Generation Module contains the extracted documentation comments in a structure following the test file. The module is thus able to generate a VTS, a VTR and a EVTR.

There are three major classes in this module, they are `vtrgenT`, `vtsgenT` and the combination of this two `evtrgenT`.

Class `vtrgenT` handles generation of a Validation Test Report. It acts as a finite state machine responding to events sent by the CAM. It registers for those events at construction time.

Class `vtsgenT` handles generation of a VTS (Validation Test Report) given a CAM (COM-Tester Abstract Machine) and a file name. Objects of this class don't have state for now and all function could have been made static. But, in order to allow for future extensions, we decided against this.

Class `evtrgenT` handles the generation of a EVTR (Extended Validation Test Report) which is, by our definition, a VTR + a VTS after the check `list`. Links from the checklist to the specification of the corresponding test are provided and, in case of failure, from the failed step to the description of this step.

Naturally, the class derives from both `vtrgenT` and `vtsgenT`.

4.5.5 Main Driver Module

The Main Driver simply brings all the pieces together. It reads command line arguments, reports parsing errors to the user and manages input/output files.

4.5.6 Others

There are some other classes in this program, among them, camcom class is very important due to its role that sits in between interpreter and COM manager module. camcom class integrates the CAM and the COM_MANAGER. It encapsulates the cooperation of the two. The main task of the class is calling COM methods from COM-Tester while taking care of the mapping between COM-Tester and COM data types.

Chapter 5 The Usage and The Result

To test the COM_Tester, a very small COM object is built , which has only one interface Iget with one function GetBuf. The sample test script is following:

```
// BEGIN OF Get TEST

S_OK = 0
namer_object_guid = "{39257021-1402-11d1-8AAD-0020182A59AB}"
iget_guid         = "{39257022-1402-11d1-8AAD-0020182A59AB}"

<< To test that get interface works with com_tester>>
test NRM_GET

     << Create an instance of Iget >>
     iget_object = com_get_instance([classid =
$namer_object_guid,
                                     interfaceid =
$iget_guid,
                                     tlb =
"atl_server.tlb"])

    << Call GetBuf function in the interface>>
     buffer = iget_object.GetBuf(81)

    << Compare string to verify the result >>
     buffer <=> "COM-Tester Version 1.1"

end

// END OF Get TEST
```

And the output (to a html file) from this test program is in next page if there is no function call failure:

Extended (+VTS) Validation Test Report

Test Checklist

Test Name	Description	Success Status	Step that failed
NRM_GET	To test that get interface works with com_tester	OK	N/A

TEST NRM_GET

Description: To test that get interface works with com_tester
Test Procedure:

1. Create an instance of Iget
2. Call GetBuf function in the interface
3. Compare string to verify the result

top

This document was generated by ComTester Version 2.0 on Wed May 10 16:00:27 2000

If, for some reason, for example, we have not register our Namer server, the test will fail on step 1. In the Extended Validation Test Report, the "Success Status" will be "KO", and "Step that failed" will point to step 1. You can see the output for this test below:

Extended (+VTS) Validation Test Report

Test Checklist

Test Name	Description	Success Status	Step that failed
NRM_GET	To test that get interface works with com_tester	KO	1

TEST NRM_GET

Description: To test that get interface works with com_tester
Test Procedure:

1. Create an instance of Iget
2. Call GetBuf function in the interface
3. Compare string to verify the result

top

This document was generated by ComTester Version 2.0 on Wed May 10 16:00:10 2000

For this version of COM-Tester, a test campaign will consist of a single source file defining a set of tests to be performed. The file is read, compiled and then the definition statements in it are interpreted in the order in which they appear. Note that function definitions are not interpreted; only `variable` bindings and test definition are. See **Appendix A** for a complete and complex example of 3 tests of the Write method of the IFileAccess `interface`.

When the example is run, the output consists of two files. One will contain the list of all tests together with the result of the execution. The format of this file is set up to contain the test name, description (if provided via the <<...>> special form) and the result (OK or KO). The other file will be a complete trace of the execution steps with input, output and return values; even, the hidden interactions that COM-Tester performs behind the scenes (such as releasing a COM object when it goes out of scope) are logged in this file.

The sample output for the Extended VTR (VTR +VTS) corresponding to the example is attached in **Appendix B**.

Chapter 6 Conclusion and Future Extension

6.1 Conclusion

COM-Tester is a powerful tool for testing COM based software. The script language is simple for the tester to master; the tool can automatically generates the test report as well as the test specification. This solves the test specification consistency problem and ease the maintaining of test specification. It greatly reduces the cost of the test activities of COM related software. Another benefit from COM-Tester tool is now the developers can use it as a helper to find the bugs in their COM products.

6.2 Future extension

COM-Tester and the description here may appear quite incomplete. The specification was kept minimal in order to avoid making key decisions before the first implementation effort. Beside more powerful common language features, ideally there should be

1) a way to define tests in a more generic manner, i.e. to separate the logical test rule from the data used to perform the test;
2) a way to logically group tests and a graphical interface to support the tool.

Also, there are some improvements to do. For example, to test some COM object, there is a need to deal with some kind of User Defined data Type. In this version of COM-Tester, we do not have such functionality.

References:

[1]. "The Advantages of Using DLLs", Visual C++ Programmer's Guide, MSDN Library, April, 1999.

[2]. "The Component Object Model Specification ", Draft Version 0.9, October 24, 1995. Microsoft Corporation and Digital Equipment Corporation.

[3]. "Inside COM", Dale Rogerson, Microsoft Press,1997.

[4]. "The Component Object Model: A Technical Overview", MSDN Library, October 1994.

[5]. "Inside OLE", Kraig Brockschmidt, Microsoft Press, 1995

[6]. "Professional DCOM Programming", Dr. Richard Grimes, Wrox Press Ltd, 1997.

[7]. "ActiveX Controls Inside Out", Adam Denning, Microsoft Press, 1997.

[8]. "Software Engineering", I. Sommerville, Addison-Wesley, 1985

[9] " Computer performance evaluation : report of the 1973 NBS/ACM workshop" Edited by Thomas E. Bell and Barry W. Boehm, and S. Jeffery. National Bureau of Standards, USA, 1975

[10]. "Warren's Abstract Machine, A Tutorial Reconstruction" Hassan Ait-Kaci, http://www.isg.sfu.ca/~hak/documents/wam.html.

Appendix A: Example Of Usage

Here is a complete example of tests of the Write method of the IFileAccess interface:

```
// BEGIN OF Write TEST


include "Const.ct"  // error and other constants

<< To test that Write works with specific valid handle, data
length and mode>>
test NRM_WRITE

    << Create an instance of IFileManage >>
    file_manager = com_get_instance([classid =
$file_manage_guid,
                          interfaceid = $ifile_manage_guid,
                          tlb = $sp_tlb])

    << Create A Buffer for Reading file >>
    file_buffer = file_manager.CreateByteBuffer(1)

    << Attach to reader "Reader Name">>
    $S_OK <=> file_manager.AttachByIFD("Reader Name",1)

    << Create an instance of IFileAccess via IFileManage >>
    file_access = file_manager.CreateFileAccess()

    << Call Open method with absolute path and file ID>>
    f_hnd = file_access.Open($TYPE_BY_ID,"\\0200\\0004")
```

```
    << Convert BSTR to bytebuffer, prepare for Writing>>
    $S_OK <=>
file_manager.ConvertHexBSTRToByteBuffer("02",buffer)

    << Verify that Write can write the specific opened
file>>
    $S_OK <=> file_access.Write(f_hnd,1,buffer,0)

    << Select the file location>>
    $S_OK <=> file_access.Seek(f_hnd,0,$SEEK_FROM_BEGINNING)

    <<Verify that Read can read the specific opened file>>
    $S_OK <=> file_access.Read(f_hnd,1,buffer,0)

    << Convert bytebuffer to BSTR, prepare for comparing the
writing result >>
    string = file_manager.ConvertByteBufferToHexBSTR(buffer)

    << Compare string >>
    string <=> "02"

    <<Select the file location>>
    $S_OK <=> file_access.Seek(f_hnd,0,$SEEK_FROM_BEGINNING)

    << Convert BSTR to bytebuffer, prepare for writing
back>>
    $S_OK <=>
file_manager.ConvertHexBSTRToByteBuffer("01",buffer)

    <<Write back the bytebuffer, recover the original file>>
    $S_OK <=> file_access.Write(f_hnd,1,buffer,0)
```

```
    <<Close the File>>
    $S_OK <=> file_access.Close(f_hnd)

end



<< To test that Write doesn't work with specific invalid
handle >>
test CTX_WRITE_1

    << Create an instance of IFileManage >>
    file_manager = com_get_instance([classid =
$file_manage_guid,
                            interfaceid = $ifile_manage_guid,
                            tlb = $sp_tlb])

    << Create A Buffer for Reading file >>
    buffer = file_manager.CreateByteBuffer(1)

    << Attach to reader "Reader Name">>
    $S_OK <=> file_manager.AttachByIFD("Reader Name",1)

    << Create an instance of IFileAccess via IFileManage >>
    file_access = file_manager.CreateFileAccess()

    << Call Open method with absolute path and file ID>>
    f_hnd = file_access.Open($TYPE_BY_ID,"\\0200\\0004")

    << Convert BSTR to bytebuffer, prepare for Writing>>
```

```
    $S_OK <=>
file_manager.ConvertHexBSTRToByteBuffer("02",buffer)

    <<Verify that Write can not write with invalid handle>>
    $S_OK != file_access.Write(10,1,buffer,0)

    <<Close the File>>
    $S_OK <=> file_access.Close(f_hnd)

end

<< To test that Write doesn't work with specific invalid
data length >>
test CTX_WRITE_2

    << Create an instance of IFileManage >>
    file_manager = com_get_instance([classid =
$file_manage_guid,
                          interfaceid = $ifile_manage_guid,
                          tlb = $sp_tlb])

    << Create A Buffer for Reading file >>
    buffer = file_manager.CreateByteBuffer(1)

    << Attach to reader "Reader Name">>
    $S_OK <=> file_manager.AttachByIFD("Reader Name",1)

    << Create an instance of IFileAccess via IFileManage >>
    file_access = file_manager.CreateFileAccess()

    << Call Open method with absolute path and file ID>>
    f_hnd = file_access.Open($TYPE_BY_ID,"\\0200\\0004")
```

```
    << Convert BSTR to bytebuffer, prepare for Writing>>
    $S_OK <=>
file_manager.ConvertHexBSTRToByteBuffer("02",buffer)

    <<Verify that Write can not write with invalid data
length>>
    $S_OK != file_access.Write(f_hnd,4000,buffer,0)

    <<Close the File>>
    $S_OK <=> file_access.Close(f_hnd)

end

<< To test that Write doesn't work with  non-existing mode>>
test CTX_WRITE_3

    << Create an instance of IFileManage >>
    file_manager = com_get_instance([classid =
$file_manage_guid,
                            interfaceid = $ifile_manage_guid,
                            tlb = $sp_tlb])

    << Create A Buffer for Reading file >>
    buffer = file_manager.CreateByteBuffer(1)

    << Attach to reader "Reader Name">>
    $S_OK <=> file_manager.AttachByIFD("Reader Name",1)

    << Create an instance of IFileAccess via IFileManage >>
    file_access = file_manager.CreateFileAccess()
```

```
    << Call Open method with absolute path and file ID>>
    f_hnd = file_access.Open($TYPE_BY_ID,"\\0200\\0004")

    << Convert BSTR to bytebuffer, prepare for Writing>>
    $S_OK <=>
file_manager.ConvertHexBSTRToByteBuffer("02",buffer)

    <<Verify that Write can not write with non-existing
mode>>
    $S_OK != file_access.Write(f_hnd,1,buffer,5)

    <<Close the File>>
    $S_OK <=> file_access.Close(f_hnd)

end


// END OF Write TEST
```

Appendix B: Sample Output

The sample output for the Extended VTR (VTR +VTS) corresponding to the example in Appendix A is as following:

Extended (+VTS) Validation Test Report

Test Checklist

Test Name	Description	Success Status	Step that failed
NRM_WRITE	To test that Write works with specific valid handle, data length and mode	OK	NA
CTX_WRITE_1	To test that Write doesn't work with specific invalid handle	OK	NA
CTX_WRITE_2	To test that Write doesn't work with specific invalid data length	OK	NA
CTX_WRITE_3	To test that Write doesn't work with non-existing mode	OK	NA

TEST CTX_WRITE_1

Description: To test that Write doesn't work with specific invalid handle

Test Procedure:

1. Create an instance of IFileManage
2. Create A Buffer for Reading file

3. Attach to reader "Reader Name"
4. Create an instance of IFileAccess via IFileManage
5. Call Open method with absolute path and file ID
6. Convert BSTR to bytebuffer, prepare for writing
7. Verify that Write can not write with invalid handle
8. Close the File

top

TEST CTX_WRITE_2

Description: To test that Write doesn't work with specific invalid data length

Test Procedure:

1. Create an instance of IFileManage
2. Create A Buffer for Reading file
3. Attach to reader "Reader Name"
4. Create an instance of IFileAccess via IFileManage
5. Call Open method with absolute path and file ID
6. Convert BSTR to bytebuffer, prepare for writing
7. Verify that Write can not write with invalid data length
8. Close the File

top

TEST CTX_WRITE_3

Description: To test that Write doesn't work with non-existing mode

Test Procedure:

1. Create an instance of IFileManage
2. Create A Buffer for Reading file

3. Attach to reader "Reader Name"
4. Create an instance of IFileAccess via IFileManage
5. Call Open method with absolute path and file ID
6. Convert BSTR to bytebuffer, prepare for writing
7. Verify that Write can not write with non-existing mode
8. Close the File

top

TEST NRM_WRITE

Description: To test that Write works with specific valid handle, data length and mode

Test Procedure:

1. Create an instance of IFileManage
2. Create A Buffer for Reading file
3. Attach to reader "Reader Name"
4. Create an instance of IFileAccess via IFileManage
5. Call Open method with absolute path and file ID
6. Convert BSTR to bytebuffer, prepare for writing
7. Verify that Write can write the specific opened file
8. Select the file location
9. Verify that Read can read the specific opened file
10. Convert bytebuffer to BSTR, prepare for comparing the writing result
11. Compare string
12. Select the file location
13. Convert BSTR to bytebuffer, prepare for writing back
14. Write back the bytebuffer, recover the original file

15.Close the File

top

This document was generated by Com-Tester Version 2.0 on Thu Apr.11 12:21:37 2000

Appendix C: CAM Instruction trace

Here is the trace example generated by CAM from Appendix A: the example of usage (we only present here part of the extraction as an illustration):

```
---> Main program   <---

putvar          S_OK
putvar          sp_tlb
putvar          file_manage_guid
putvar          ifile_manage_guid
putvar          TYPE_BY_NAME
putvar          TYPE_BY_ID
.
.
.
push            0
putnumber       0
set
push            1
putstring       sp.tlb
set
push            2
putstring       {99A52A52-0E0C-11D3-9FE5-0008C7A185BF}
set
push            3
putstring       {5E586211-5A09-11d0-B84C-00C04FD424B9}
set
.
.
runtest         NRM_WRITE
```

```
runtest         CTX_WRITE_1
runtest         CTX_WRITE_2
runtest         CTX_WRITE_3

---> TEST: CTX_WRITE_1   <---
putvar          file_manager
putvar          buffer
putvar          file_access
putvar          f_hnd
<<STEP:  Create an instance of IFileManage >>
getlocal        0
push            2
push            3
push            1
putref          tlb
putref          interfaceid
putref          classid
putlist         3
call            1, com_get_instance
set
pop
.
.
<<STEP: Close the File>>
push            0
getlocal        3
sderef          Close
getlocal        4
call            1,
equal
pop
```