Kalimetrix **LOGISCOPE**

# TestChecker Getting Started

Before using this information, be sure to read the general information under “Notices” section, on page 107.

# *Table of Contents*

# Chapter 1

# *About this manual*

## Audience

This manual introduces Kalimetrix Logiscope™ *TestChecker* and get you started. Within one hour you will be familiar with the tool main features and concepts. Step-by-step instructions will show you Logiscope *TestChecker* from different points of view.

## Overview

Throughout this document you will observe how to take advantage of test coverage measurements produced by Logiscope *TestChecker* to improve testing strategy for the software application under test and how to generate automatically test coverage reports. Applications can be written in Ada, C, C++ or Java and all these cases will be seen throughout this manual.

This consists in the following phases:

1. An introduction to the notion of Test Coverage presenting the various levels of coverage produced by Logiscope: e.g. Decision To Decision Path (DDP) coverage, Multiple Conditions / Decisions Coverage (MC/DC).
2. Building C++ Instrumented Code with Logiscope **Studio**.
   In this second phase you will discover Logiscope **Studio** (if you are not yet familiar with it) and you will produce your program instrumented binary through a *TestChecker* project.
3. Testing on a Host Machine with Logiscope **TestChecker**.
   You will create your first test sessions, generate and analyse your application first test coverage results.
4. Viewing Results from Logiscope **Studio** and **Viewer**.
   Previous results will be viewed in different types in these two tools. You will get acquainted with Logiscope *Viewer* if you never met it.
5. Building C Instrumented Code. In this part, you will enhance your knowledge of *Studio* and *TestChecker* through the a C example introducing the MCDC.
6. Testing on a Target Machine with *TestChecker* **TcGatway**.
7. Building and Testing an Ada Instrumented Code.
8. Building and Testing a Java Instrumented Code.
9. Using *TestChecker* in Command Line mode.

# Related Documents

Additional information can be found in:

- *Kalimetrix Logiscope TestChecker - Testing on a target machine*.
- *Kalimetrix Logiscope Studio Reference Manual*.

# Conventions

The following writing conventions are used in this manual:

- **bold**: names of commands (e.g. **vcs**), files and folders (e.g. **LogiscopeProjects**), and file extensions (**.res**)
- *italic*: names of user-defined textual elements (*version_1*, *component_2*), notes,
- `typewriter`: screen messages (`Reference filename`) requiring user action,
- keycaps (<Enter>).

**<InstallationDir>** will refer to the *Rational Logiscope* installation directory.

**<Version>** will refer to the Logiscope current version: e.g. 6.6 or upper.

*Note: Screen displays in this manual can be slightly different from those you get when running the Getting Started.*

# Contacting Kalimetrix Software Support

If the self-help resources have not provided a resolution to your problem, you can contact Kalimetrix Support for assistance in resolving product issues.

## Prerequisites

To submit your problem to Kalimetrix Software Support, you must have an active support contract

To submit your problem online (from the Kalimetrix Web site) to Kalimetrix Software Support, you must additionally:

- Be a registered user on the Kalimetrix Software Support Web site. For details about registering, go to http://support.kalimetrix.com
- Be listed as an authorized caller in the service request tool

## Submitting problems

To submit your problem to Kalimetrix Software Support:

1. Determine the business impact of your problem. When you report a problem to Kalimetrix, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

| Severity | Description |
|---|---|
| **1** | The problem has a *critical* business impact. You are unable to use the program, resulting in a critical impact on operation. This condition requires an immediate solution. |
| **2** | The problem has a *significantl* business impact. The program is usable, but it is severely limited. |
| **3** | The problem has a *some* business impact.The program is usable, but less significant features (not critical to operation) are unavailable. |
| **4** | The problem has a *minimal* business impact.The problem causes little impact on operations or a reasonnable circumvention to the problem was implemented. |

**2.** Describe your problem and gather background information, When describing a problem to Kalimetrix, be as specific as possible. Include all relevant background information so that Kalimetrix Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

  To determine the exact product name and version, use the option applicable to you:

  - Start your product, and click **Help** > **About** to see the offering name and version number.
- What is your operating system and version number (including any service packs or patches)?
- Do you have logs, traces, and messages that are related to the problem symptoms?
- Can you recreate the problem? If so, what steps do you perform to recreate the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
- Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

3. Submit your problem to Kalimetrix Software Support. You can submit your problem to Kalimetrix Software Support going online to the Kalimetrix Software Support Web site at http://support.kalimetrix.com .

# Chapter 2

# *Notion of Test Coverage*

## 2.1 Suggested Approaches

In order to evaluate the completeness of the tests performed, Logiscope measures the test coverage as the following ratio:

$$\frac{\text{number of objects executed}}{\text{number of objects to be executed}}$$

which just leaves the notion of object to be defined.

Test coverage represents the percentage of objects exercised by executed tests.

As execution paths cannot be identified automatically, objects are considered as portions of the execution paths.

The larger the size of these portions, the more they integrate control structure combinations. The effort to obtain maximum test coverage is greater but the risk of software failure is reduced.

Logiscope proposes several types of approach to measuring test coverage.

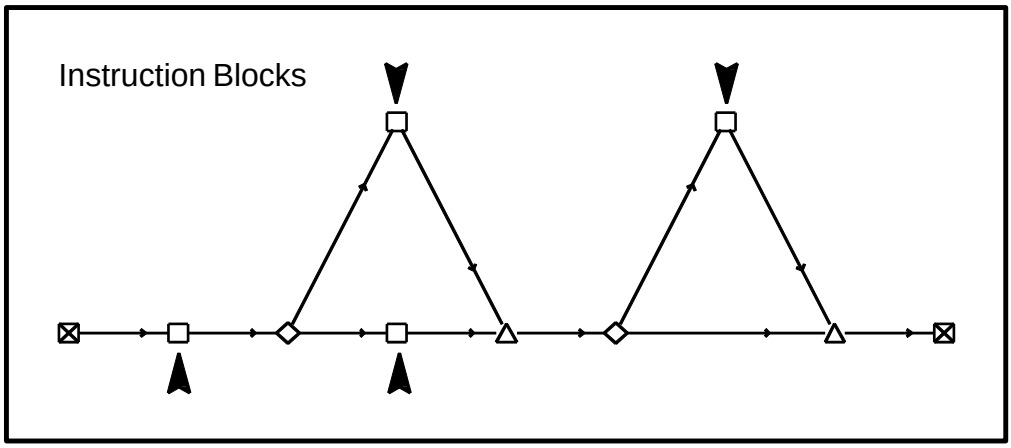
Objects considered are:

- **Instruction Blocks (IBs),**
- **Decision-to-Decision Paths** (**DDPs**),
- **Modified Conditions/Decisions** (**MC/DC**s).

# 2.2 Instruction Blocks

Instruction Blocks (or IBs) represent sequential instructions in the function such that the execution of the first instruction block leads to the execution of the last. They are symbolized on the control graph by squares.

**Example:**




*Control graph with 4 IBs*

**Note :**

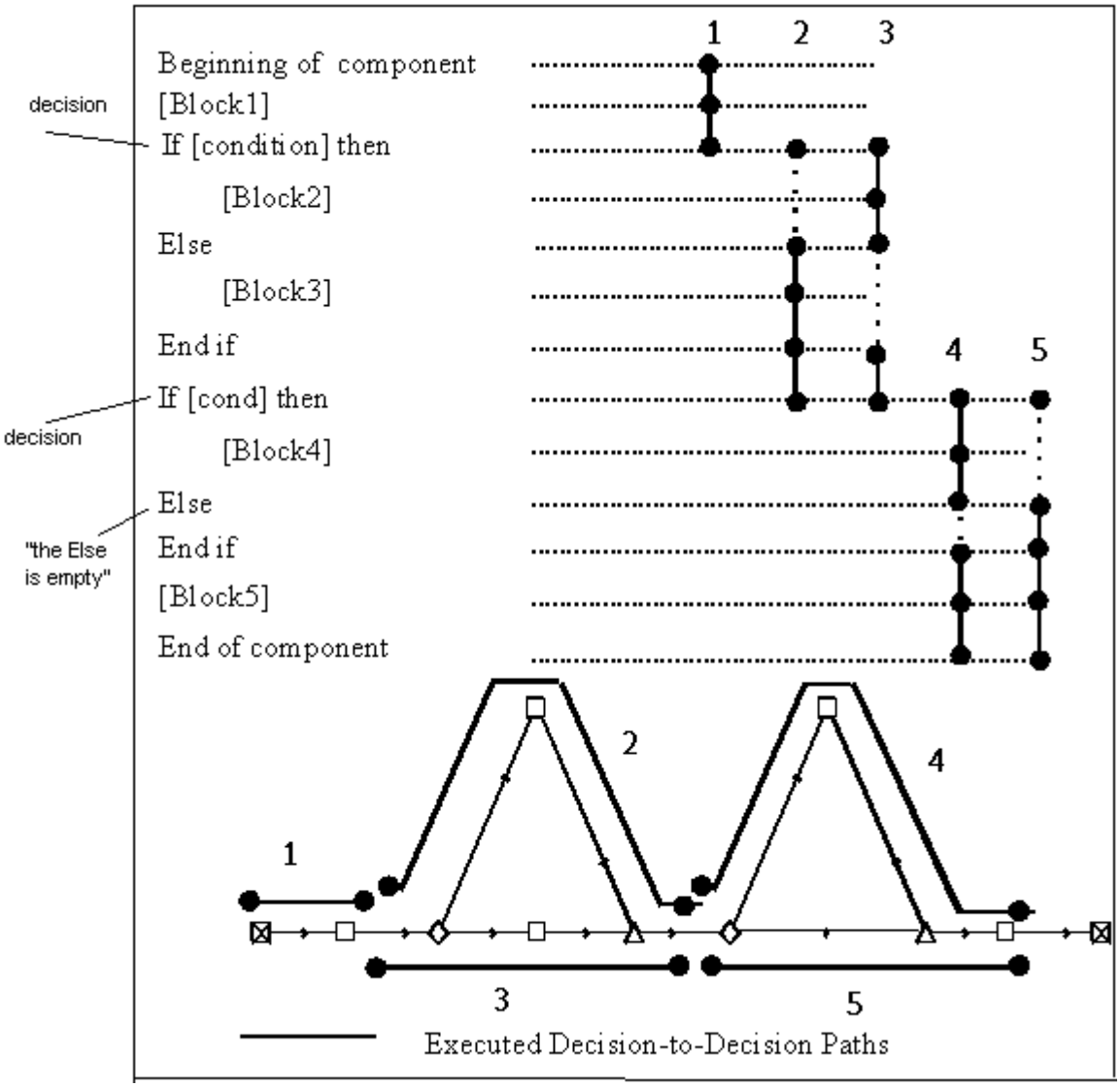
Only Logiscope **Viewer** makes possible the display of IB coverage in graphic and tabular form. For details, refer to the Logiscope Viewer online help.

# 2.3 Decision to Decision Paths

A Decision-to-Decision Path (or DDP) is a sequence of instructions whose origin is the entry point of the function or a decision (if, while,...) and whose end is the exit point of the function or the next decision. No decision should be found between the point of origin and the end point. Control instructions are symbolized on the graph by nodes (◇). Components beginning and end, symbolized on the graph by nodes (⊠), are taken as decisions feedback.

**Example:**

In the component represented by the pseudo-code illustrated below, the following five DDPs and control graph are detailed:




*DDP example*

Logiscope **Viewer** enhances the qualitative aspect provided by Logiscope TestChecker by displaying DDP coverage in graphic (see control graph above) and tabular form. It also indicates necessary conditions to execute non-tested DDPs. For details, refer to the Logiscope Viewer online help.

# 2.4 Modified Condition/Decision

The Modified Condition/Decision Coverage (MC/DC) provides most of the benefits of multiple–condition testing while keeping the number of required tests from growing exponentially.

The DO-178B standard (*Software Considerations in Airborne Systems and Equipment Certification*) defines testing objectives according to the application degree of criticality, as it relates to real-life aircraft failure conditions.

DO-178B classifies software according to consequences of failure ranking from Level A: the most critical to Level E. Level A corresponds to software whose failure "would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft."

For the verification process, DO-178B states that level A software requires 100% of Modified Condition/Decision Coverage .

## 2.4.1 Definition

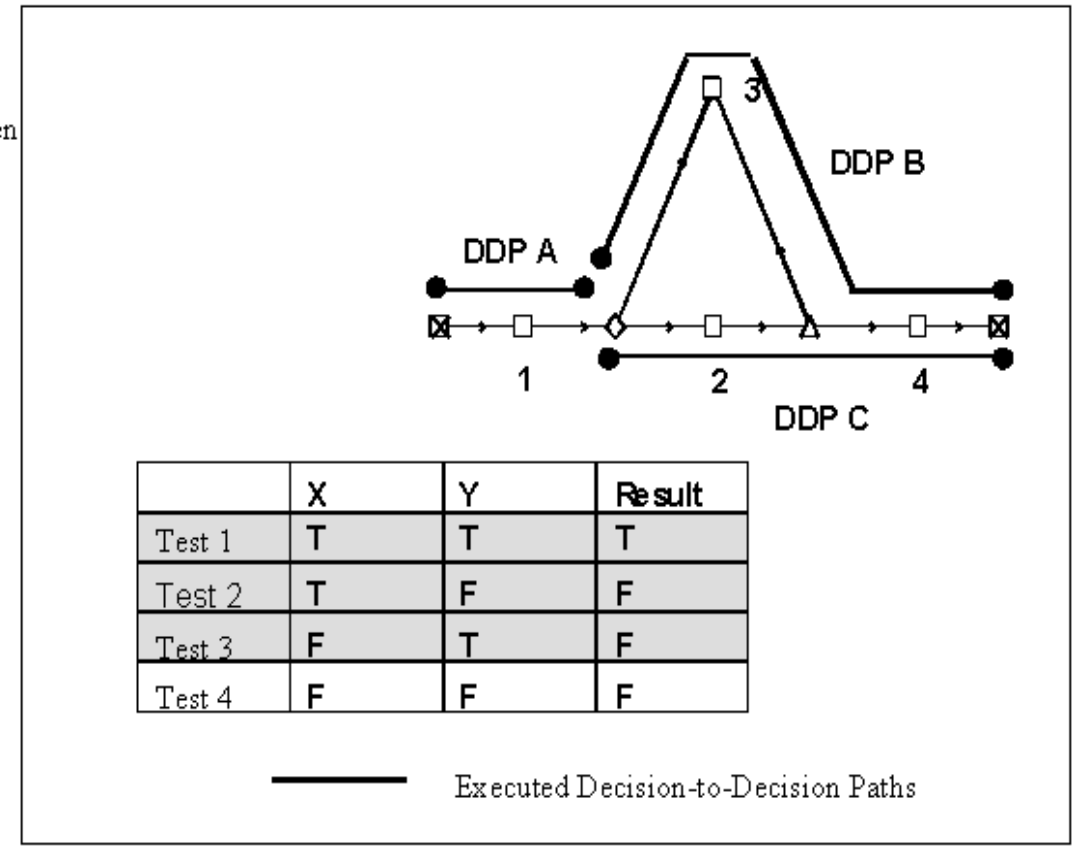
The Modified Condition/Decision Coverage criterion is satisfied if the following requirements are met:

- **Requirement 1**: every entry and exit point of the program module under consideration has been invoked at least once, and each program decision has switched to all possible outcome values at least once.
- **Requirement 2**: program decisions having been broken down into basic Boolean conditions connected by logical operators (AND, OR, etc.), every one of these conditions has taken all possible outcome values; every condition has acted on the outcome of the decision independently. In other words, the outcome of a decision has changed as a result of changing a single condition.

**Example:**

The following example illustrates a simple Modified Condition/Decision Coverage test:

```
begin
  [block 1]
if (X and Y) then
  [block 2]
else [block 3]
  [block 4]
end
```




| | X | Y | Result |
|---|---|---|---|
| Test 1 | T | T | T |
| Test 2 | T | F | F |
| Test 3 | F | T | F |
| Test 4 | F | F | F |

*Modified Condition/Decision Coverage*

In this basic example a 100% DDP (Decision-to-Decision Path) coverage rate is achieved: test 1 with X = T (True) and Y = T (True) covers DDPs A and C, and test 3 with X = F (False) and Y = T (True) covers DDPs A and B. The outcome of these tests also satisfies requirement 1. Note that expression Y has not taken on the value F, which means that more testing has to be conducted to satisfy requirement 2. Tests 1 and 3 have not shown that Y independently affects the outcome of the decision.

The truth table in Figure 2 shows that the (T,T) test is required, as it is the only one that will allow to reach the T value. The (F,T) test is also required as it is the only test that will change the value of X as well as the outcome of the decision, thus showing the independence of X. Similarly, the (T,F) test is required to show the independence of Y. Therefore, three tests are required to meet requirements 1 and 2.

## 2.4.2 Test Coverage

The table below details tests for the example decision (X and (Y or Z)).

| | X | Y | Z | Result | X | Y | Z |
|---|---|---|---|---|---|---|---|
| Test 1 | T | T | T | T | 5 | | |
| Test 2 | T | T | F | T | 6 | 4 | |
| Test 3 | T | F | T | T | 7 | | 4 |
| Test 4 | T | F | F | F | | 2 | 3 |
| Test 5 | F | T | T | F | 1 | | |
| Test 6 | F | T | F | F | 2 | | |
| Test 7 | F | F | T | F | 3 | | |
| Test 8 | F | F | F | F | | | |

*Pair table for (X and (Y or Z))*

The first column lists test case numbers, and three columns on the right are used to pair tests relevant for conditions X, Y and Z. This table shows that the pair of a given test case for a condition is the test case which establishes the independence of this condition. From Figure 3 it can be demonstrated that test case 1 (T,T,T) and test case 5 (F,T,T) can be paired to show the independence of X. Consequently, test case 1 is the unique pair for test case 5 as far as condition X is concerned.

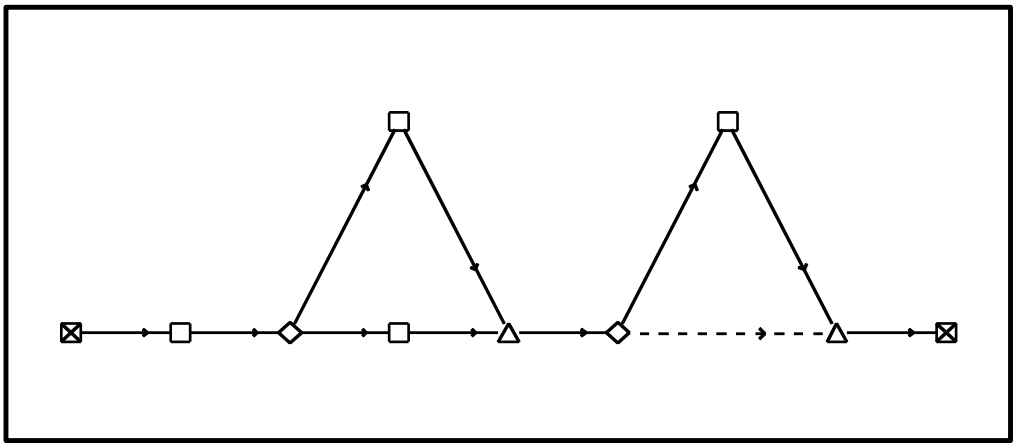
As mentioned above, the pair table indicates that test case 1 (T,T,T) and test case 5 (F,T,T) show the independence of X. Similarly, test case 2 (T,T,F) and test case 4 (T,F,F) show the independence of Y, and test case 3 (T,F,T) and test case 4 (T,F,F) can be paired to show the independence of Z. As a result, the test set {1,5,2,4,3} satisfies the Modified Condition/Decision Coverage for the expression X, Y and Z. Obviously, this is not the only possible combination.

Coverage rates are obtained by calculating, from executed tests, a set of tests sufficient to demonstrate that all conditions of the decision are indeed independent. Coverage is obtained from the result of the following ratio:

$$\frac{\text{number of tests}}{\text{number of tests} + \text{minimal number of tests required}}$$

# 2.5 Coverage Precision

Let us illustrate this with the control graph of the previous figure. In the testing phase executed test cases have made it possible to pass through some structures in the control graph. In the control graph, paths taken are represented by a continuous line and paths that have not been taken are represented by a broken line.



*Control graph of paths taken*

With respect to the various objects, running these test cases has covered:

| objects | proportion | coverage |
|---|---|---|
| IBs | 4 out of 4 | 100% |
| DDPs | 4 out of 5 | 80% |
| MC/DCs | 3 out of 6 | 50% |

These three approaches correspond to three degrees of measurement precision, and the choice of which approach to use will depend on the criticality of the software to be tested and the objective to be reached.

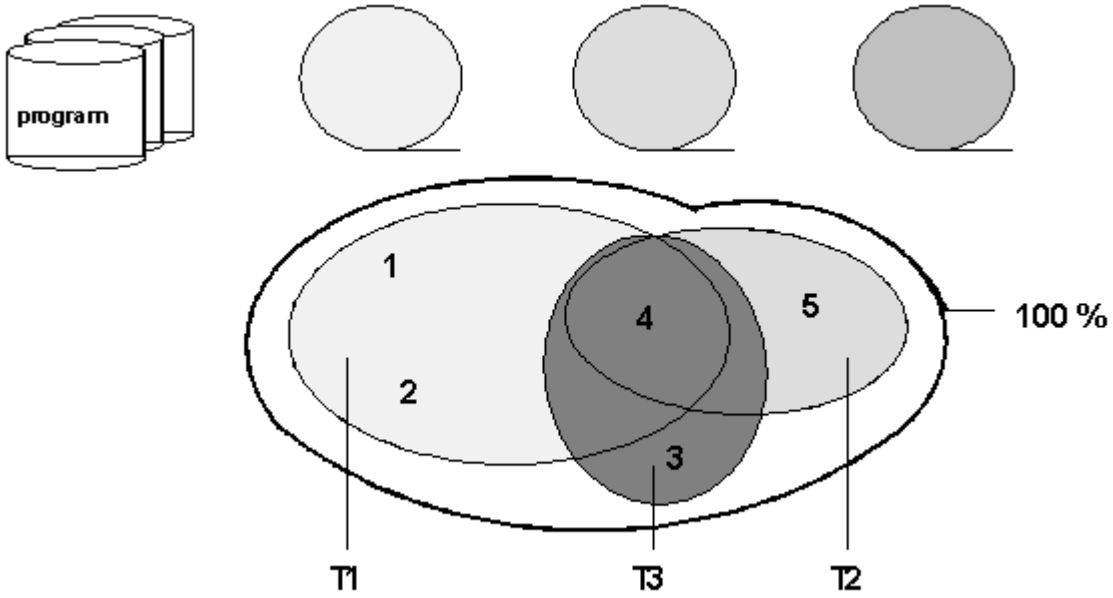
- for a trivial application, an IB coverage rate of 100% may be sufficient,
- for a critical application, a DDP coverage rate close to 100% may be advisable,
- for a very critical application, a MC/DC coverage rate close to 100% may be required.

# 2.6 Coverage Gain

Gain, or improvement represents the percentage of objects specifically executed during a test. In other words it gives the percentage of objects executed exclusively by a test. This notion is a dynamic one. Each time a test is added (executed) or removed using Logiscope *TestChecker*, the gain of each test may decrease or increase accordingly.

## 2.6.1 Example 1

Here is an example to illustrate the notion of gain. Suppose to analyze a program containing a function which has 5 DDPs, and 3 tests (T1, T2 and T3) executing some of these DDPs for this function. The table below provides DDP coverage, the percentage covered by each test or by the sum of all tests, and gain for each test.




**X** (in uppercase) indicates a DDP which is executed by one test only. Gain is positive for this test.

**x** (in lowercase) indicates a DDP which is executed by several tests. Gain is null for this test and for this DDP.

| DDP / Tests | 1 | 2 | 3 | 4 | 5 | Coverage | Gain |
|---|---|---|---|---|---|---|---|
| T1 | X | X | | x | | 60 % | 40 % |
| T2 | | | | x | X | 40 % | 20 % |
| T3 | | | X | x | | 40 % | 20 % |
| T1+T2+T3 | X | X | X | X | X | 100% | |

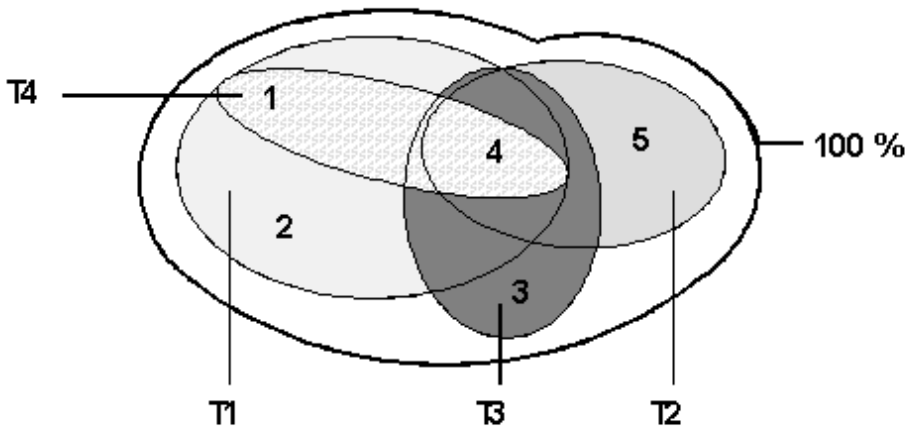
## 2.6.2 Example 2

Assuming the execution of another test (T4) at this point. Look at the resulting change in the gain column below. In fact, this new test will not increase global coverage because all covered DDPs have already been executed by another test, in other words, gain will be 0%.

In addition to this, gain for T1 will decrease because 2 out 3 of the DDPs covered by T1 have already been executed by other tests: T4 will execute DDP1, which has already been tested by T1.

In such a case, deleting T4 will not affect global coverage. The list of "efficient" tests will still be T1, T2, T3.

| DDP / Tests | 1 | 2 | 3 | 4 | 5 | Coverage | Gain |
|---|---|---|---|---|---|---|---|
| T1 | x | X | | x | | 60 % | 20 % |
| T2 | | | | x | X | 40 % | 20 % |
| T3 | | | X | x | | 40 % | 20 % |
| T4 | x | | | x | | 40 % | 0 % |
| T1+T2+T3+T4 | X | X | X | X | X | 100% | |



# Chapter 3

# *Building C++ Instrumented Code with Logiscope Studio*

## 3.1Before you start

Along with this chapter, you are provided with a program written in C++ language, an implementation of an ATM machine. The program has been carefully designed for you to use all features of Logiscope *TestChecker*.

Source files of this program are stored in the directory **<InstallationDir>\samples\Tchk\C++\ATM.**

As a precaution to keep original files safe, it is highly recommended that you copy this subdirectory into a working directory of your own: e.g. **C:\ATM**. on Windows, **$HOME/ATM** on UNIX.

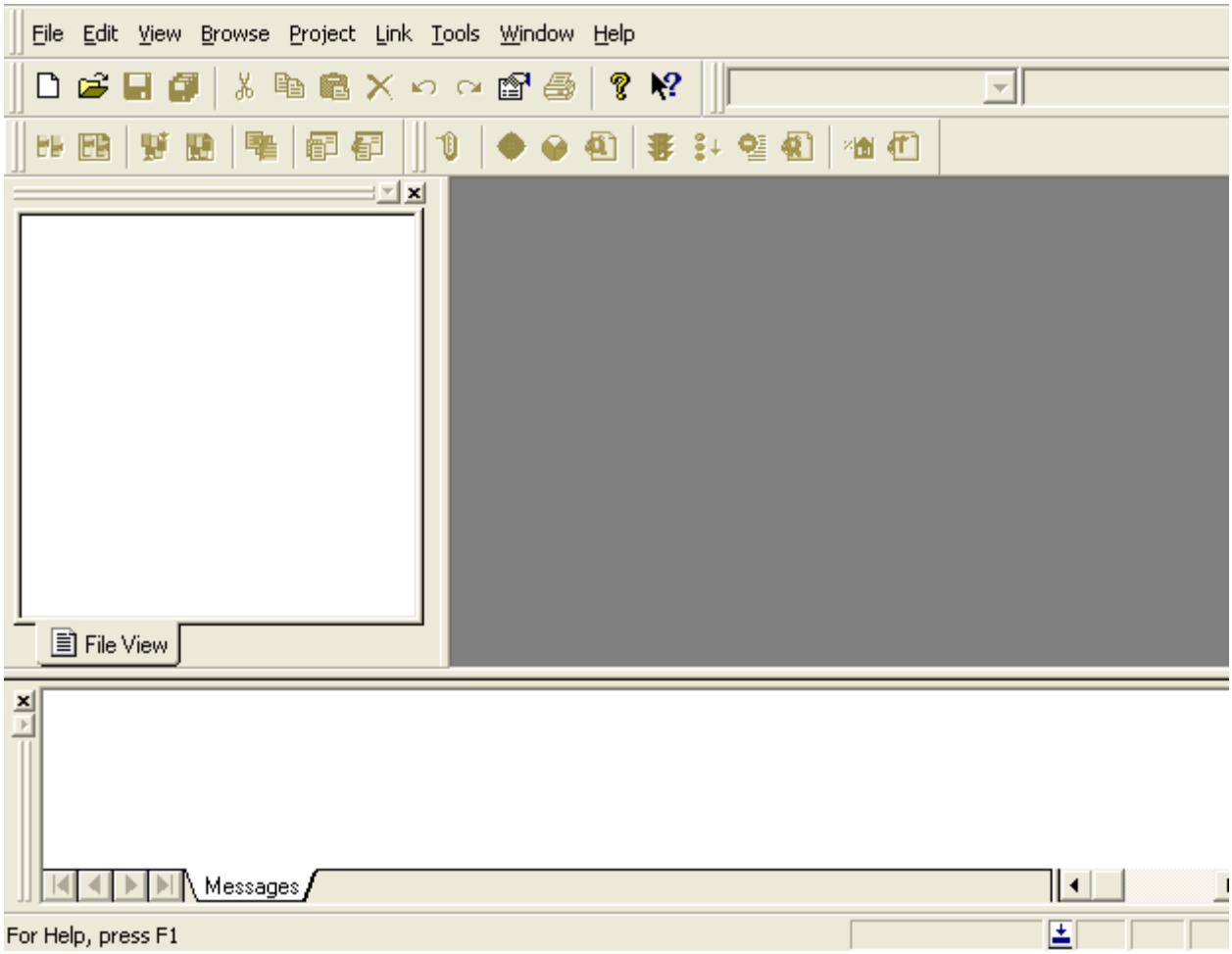
In addition, you will create Logiscope projects and associated repositories: i.e. sets of files containing internal data used by Logiscope. It is recommended to a create a dedicated directory to store these data: e.g. a folder named **LogiscopeProjects**.

## 3.2 Starting a Logiscope Studio Session

1. To begin a Logiscope **Studio** session:

- On Windows:
  - click the **Start** button and select the **Kalimetrix Logiscope <version>** item in the **Kalimetrix** Programs Group.
- On UNIX (i.e. Solaris or Linux):
  - launch the **vcs** binary .

The Logiscope splash screen is first displayed and then the Logiscope **Studio** main window appears as follows:




# 3.3 Creating a *TestChecker* Project

First, you shall define a Logiscope *TestChecker* project which mainly consists in:

- the list of source files to be first instrumented and then being tested for test coverage analysis,
- applicable source code instrumentation options according to the compilation environment,
- the special traces that will be generated by the Logiscope libraries during the execution of the test cases on the instrumented application.

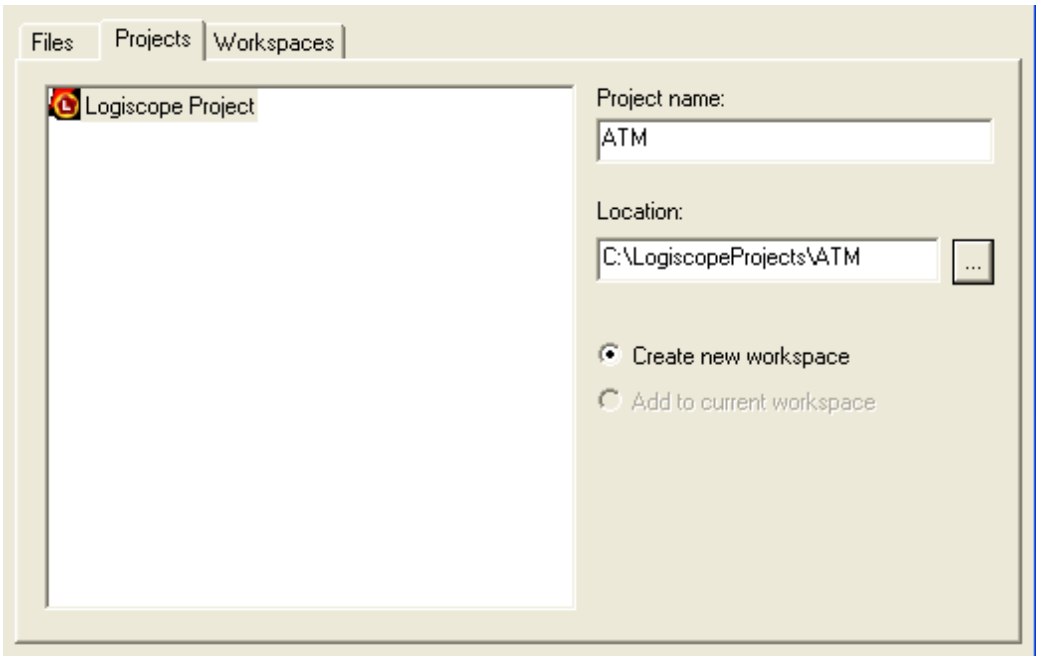
2. In the **File** menu, select the **New**... command or click the  icon.

The **New** Logiscope Projects dialog box appears.

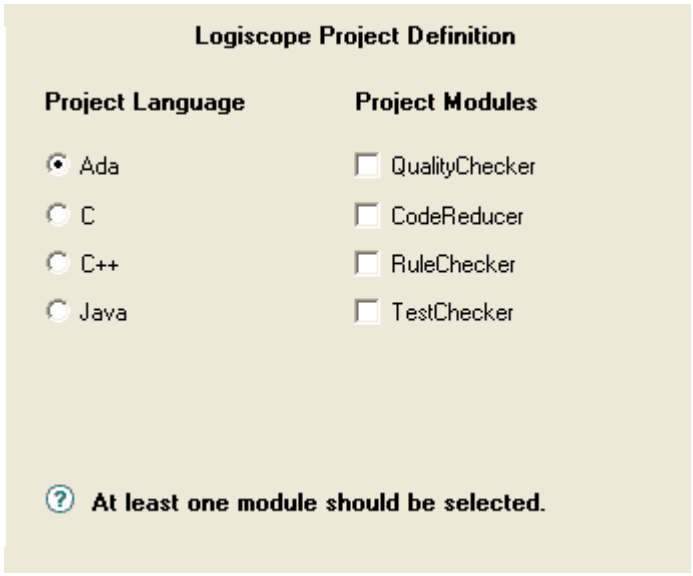
3. In the **Project name:** pane, enter the name for the new Logiscope project to be created. In the context of the guided tour, this simply can be the name of the application under test: e.g. *ATM*.
   The information provided in this pane will be then refer as the <ProjectName>.

4. Then select its **Location:** i.e. the directory where the Logiscope project (i.e. a "**.ttp**" file) and the associated Logiscope repository will be created; the Logiscope repository is a folder in which Logiscope internal analysis result files are generated.
   The information provided there will be then refer as the <LogiscopeRepository>.

Note: By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.




5. Click **OK** to access to the **Logiscope Project Definition** first window.




6. Select the **Project Language:** i.e. the programming language in which are written the source code files to be analysed.
For the ATM project, select **C++**.

Note: Only one language can be selected. If your application contains source code files written in several languages, you should create several distinct Logiscope projects: one for each language.

7. Select the **Project Modules:** i.e. the verification modules to be activated on the source files of the project .
For the guided tour, select **TestChecke**r.

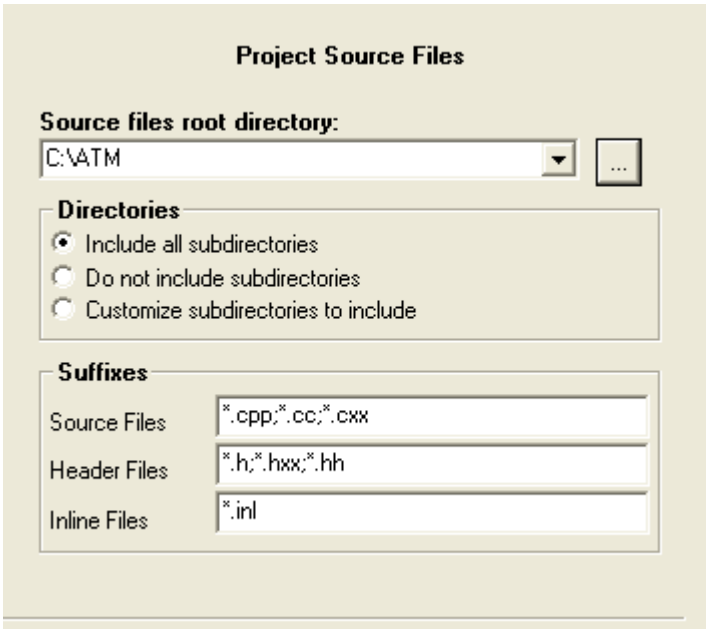
Notes: At least one module should be selected. The *TestChecker* module cannot be selected with another module.

For more details on *QualityChecker* and *RuleChecker* modules, please refer to *Kalimetrix Logiscope QualityChecker & RuleChecker Getting Started.*

For more details on *CodeReducer* module, please refer to *Kalimetrix Logiscope CodeReducer - Identifying Code Similarities* .

8. Click the **Next** button to continue the creation.

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.




.

**Source files root directory** shall specify the location directory of the source files to be analyzed.

9. Browse to select the directory where the *ATM* sample source files are located: i.e. in the **samples/Tchk/C++/ATM** folder of the Logiscope installation directory or in the directory where the source files have been copied as recommended in previous section: e.g. **C:/ATM**

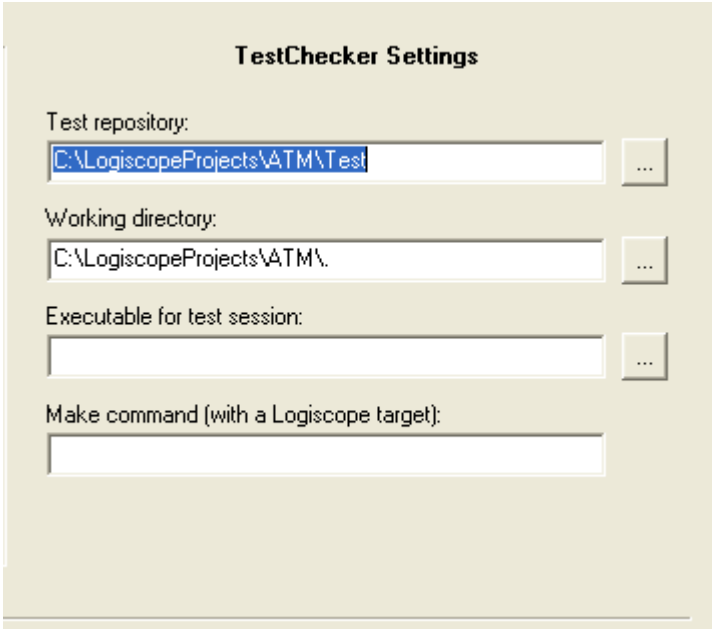
The **Directories** choice allows to select the list of repertories covering the application source files.

- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

**Suffixes** choices allow to specify applicable source, header and inline file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

10. Click the **Next** button.

The **TestChecker Settings** dialog box is now displayed. It allows to specify some of the key settings of a Logiscope *TestChecker* project.




11. The **Test repository:** is the directory in which instrumented code and traces files generated when executing the instrumented executable will be saved.
    Keep the default location i.e. a Test folder to be created in the Logiscope repository specified in the **New** Logiscope Projects dialog box (see Item 3.).
12. The **Working directory**: is the directory where the make file can be found and where the executable will be generated (unless otherwise specified by the make file).
13. The **Executable for a test session:** shall specify the instrumented executable.
    In this context, the executable is not yet generated and will be chosen later.
14. The **Make command** file shall contains the command to build the instrumented executable. Type the following:
    **on UNIX: make lgatm**
    **on Windows, cmd /c MakeLog.bat**

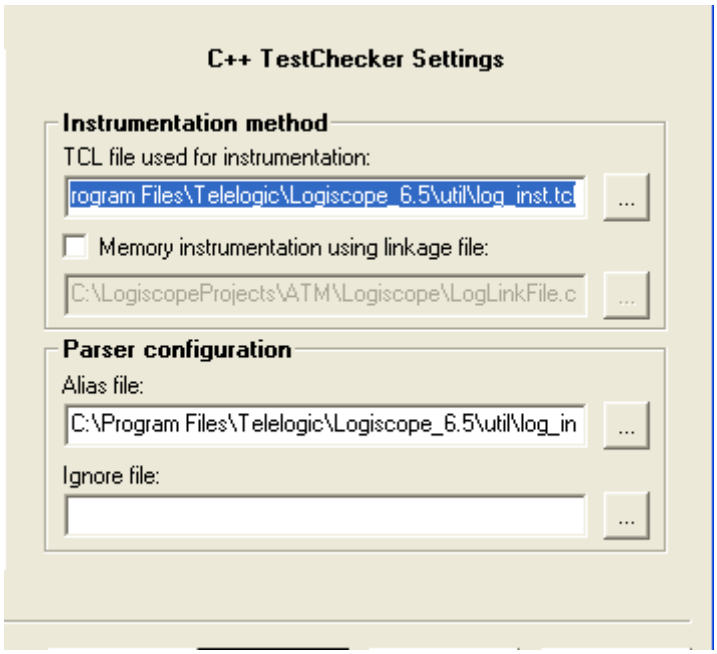
    *Note: According to your DOS version, use the equivalent of the 'cmd' command.*

The Make command will launch the make file in which a Logiscope target has been defined, to compile and link-edit together the instrumented source files and the instrumentation library file located in **<InstallationDir>/instr/src/vlgtchk.c**.
In the next section "Building the Instrumented Executable", you will be prompted to edit and modify the make file specified in this pane to adapt it to your compilation environment.

15.Click the **Next** button.

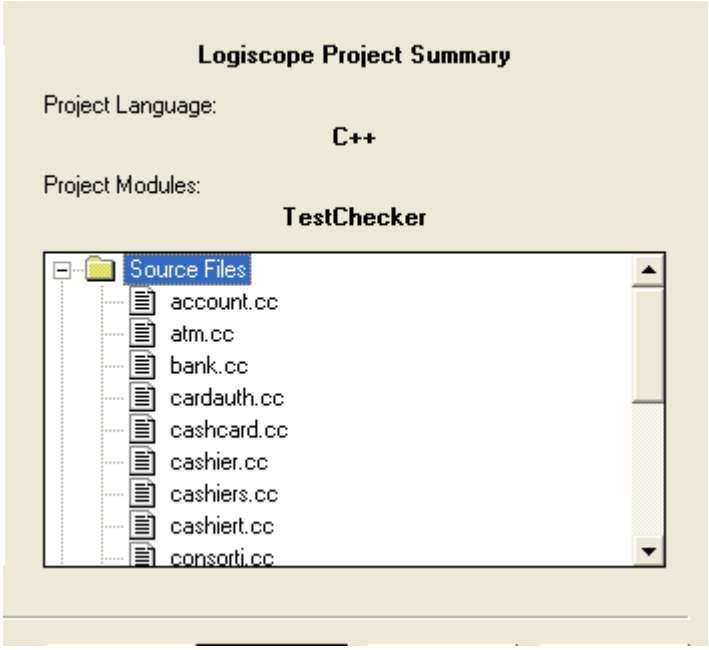
The following wizard box will allow you to complete the project specification with some specifics of the C++ language .

In the **Instrumentation method** part, you can choose **Memory instrumentation** checkbox but it is an advanced usage for some targets only.
In the **Parser configuration** part, you can choose the tool in charge of parsing and instrumenting the source code.
TCL and alias files are defined by default (included in Logiscope product).You can find details about updating the alias file in section 4.5 *Updating the alias file*.




16.Click the **Next** button to confirm.

The last wizard window is displayed. You can check if all files are correct by expanding folders.



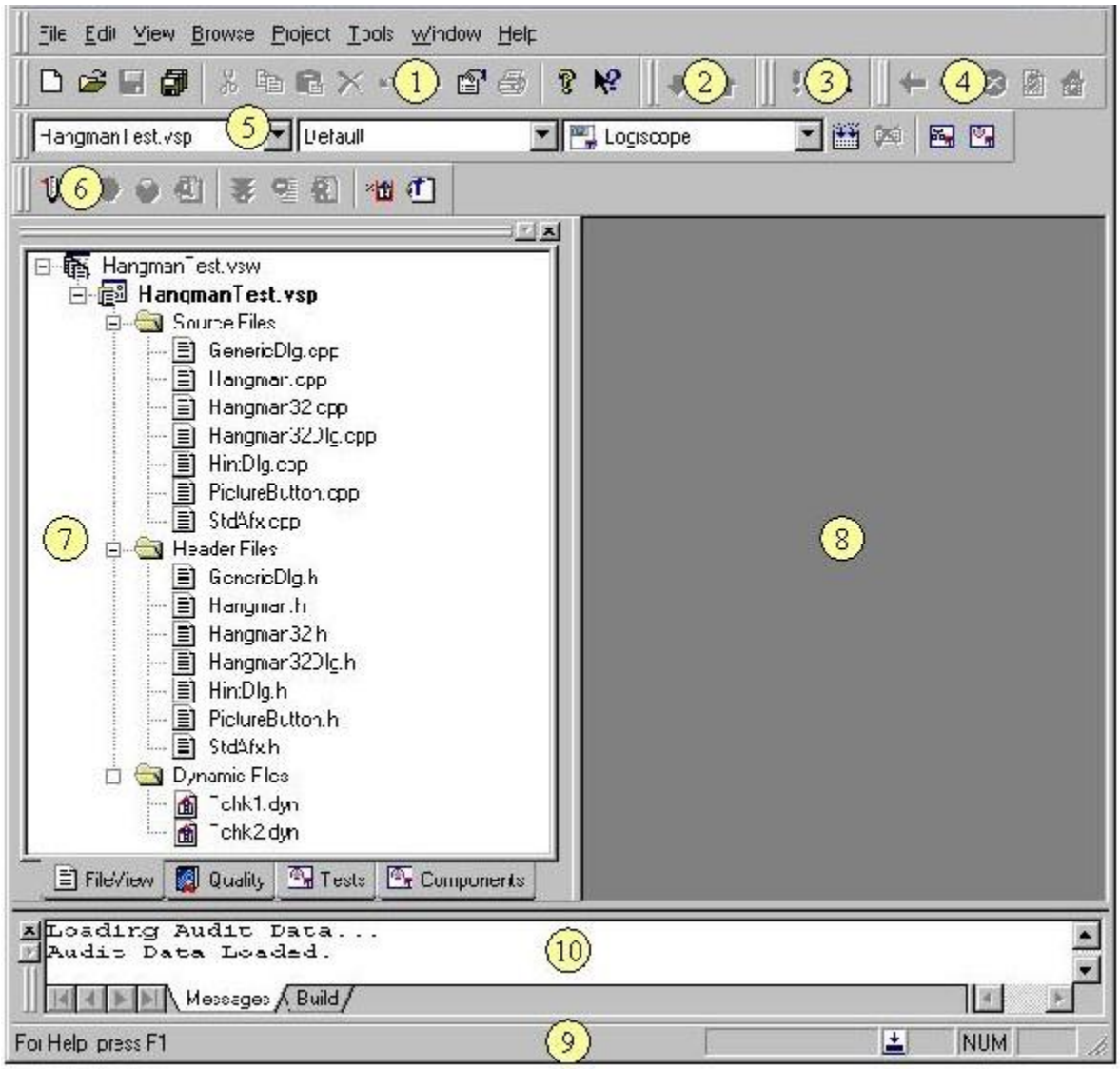

17. Click **Finish** to create your first C++ *TestChecker* project.

The **Studio** main window is now updated and contains the workspace view of your project (see next section).

Two files has been created by this process and are of the form: <ProjectName>**.ttp** for project and <ProjectName>**.ttw** for associated workspace. They are both located in the folder specified as the Logiscope Repository.

# 3.4 Introducing Logiscope Studio

Once the Logiscope *TestChecker* project has been created, the Logiscope **Studio** main window looks as below:




It contains the following components:

1 **Tool Bar**

Provides shortcuts for most commonly used commands of File and Edit menus.

2 **Browse Bar**

 Provides shortcuts for **Browse** menu commands.

3 **Tcl Script Bar**

 The script wizard: Logiscope internal data navigator.

4 **UML Browser Bar**

 Allows navigation in HTML documents and internal data.

5 **Project Bar**


Build the project and start the *Viewer* and *TestChecker* tools.

6 **Logiscope TestChecker Bar**

 Allows to display key *TestChecker* results.

- Test Coverage,
- HTML Test Coverage Report

7 **Workspace View**
Displays a specific view related to the project: header files, the quality model file and source files.
With a double-click on any file, the original one is displayed in the Result Pane.

8 **Result Pane**
Used to display various windows.

9 **Status Bar**
Indicators when building and idle . The status bar also shows short definitions corresponding to toolbars described above.

10 **Output Window**
Displays project messages as the first tab is created; also shows errors messages, warnings or results.

You can use the **View** command to customize toolbars.

# 3.5 Building the Instrumented Executable

Your project is ready to be built. Building consists in:

- instrumenting the code using the instrumentation method selected for the project,
- generating (i.e. compiling and linking) the instrumented executable.

When building the instrumented executable, in order the makefile works, the original sources files are temporarily replaced by the instrumented ones and then restored.

First of all, you must adapt the Make Command to your compilation environment:

1. Open a text editor and load either the file **makefile** file or the file **makefile.vc** if you intend to compile the code using a Microsoft Visual compiler.
   It starts by the following lines:

```
# makefile for ATM c++ exampleV1.5
LOGISCOPE_INSTALL = ../../../..
```

2. Adapt the value of the variable `LOGISCOPE_INSTALL` to correspond to the path of the Logiscope installation directory.
3. **On Windows only:** open a text editor and load the **MakeLog.bat** file located in the directory where the *ATM* source files are: e.g. **C:\ATM**.
   It contains the two following commands:

```
call "c:\program files\microsoft visual studio\vc98\bin\vcvars32.bat"
nmake /A /F makefile.vc lgatm
```

4. Update the path specified in the `call` command to correspond to the installation directory of the compiler to be used.

Once the Make command has been adapted:

5. Select the **Project-Build** command or click the  icon. A new tab is added in the **Output** window and will contain code instrumentation and generation messages:

```
Instrumenting: ../ATM/account.cc...
Instrumenting: ../ATM/atm.cc...
Instrumenting: ../ATM/bank.cc...
...
gcc -c -I../../../../instr/include ./account.cc -o Objects/account.o ...
gcc -c -I../../../../instr/include ./atm.cc -o Objects/atm.o ...
gcc -c -I../../../../instr/include ./bank.cc -o Objects/bank.o ...
...
lgatm Objects/account.o Objects/atm.o  ...

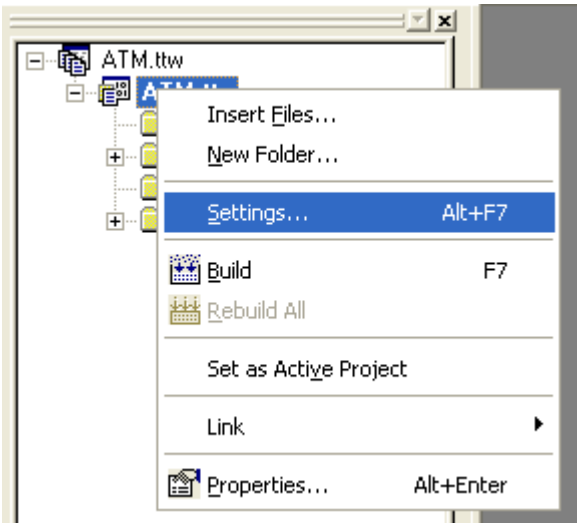
Build finished
```

After building the project, the **Message** tab of the **Output** window will contain final results.

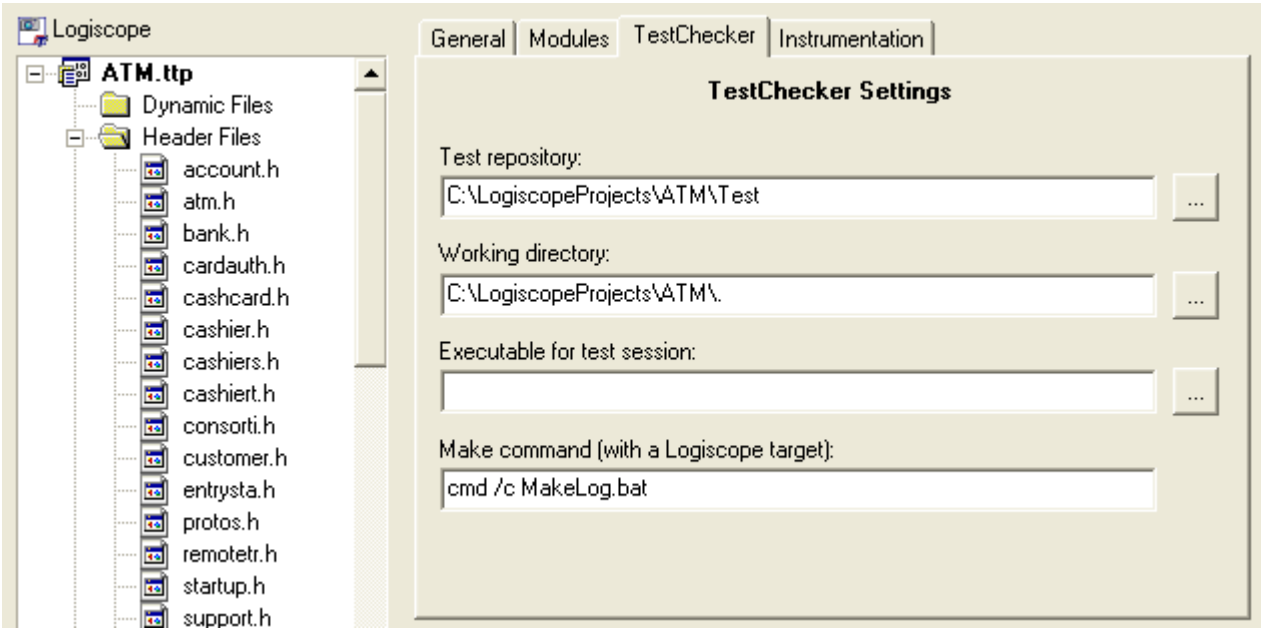
The project is built. Otherwise this window will display error messages.

You will now end up *TestChecker* settings specifications.

6. Select **Project-Settings** option or run the <Alt+F7> command to specify the executable file. Or, use a shortcut: right click on the Project filename as follows:




7. Select the **TestChecker** tab.




8. You can now specify the **Executable for test session:** i.e.the command to launch the instrumented executable:

**On Windows:** Click on the ... button.

A browse window appears. Select the *ATM.exe* executable file that has been generated in the **Objects** sub-directory of the Working Directory when running the Build command.

**On Unix:** Type *xterm -e lgatm*

If you command name has blank spaces put it between double-quotes (“). The space is interpreted as the separator between the command and its parameters.

9. Click **OK** to take changes into account.
10. Save your project with **File-Save Workspace**.

The instrumented executable generation is complete, your project is ready to be tested. For this you are going to use the Logiscope **TestChecker** tool. Move to the next chapter.

# 3.6 Updating the alias file

The alias file can be used to inform the parser about special macros.

## 3.6.1 Syntax of the file

// Introduces a line of comments

<macro-name>|<replacement>

<macro-name>()|<replacement>

<macro-name>(<param>[,<param>]*)|  <replacement>

where:

<macro-name> is the name of the macro to replace

<param> is either ## for "normal parameter" or $$ for "special parameter" (see below)

<replacement> is one of the following:

| | | |
|---|---|---|
| { | : | the macro is to be considered as an opening curly bracket |
| } | : | the macro is to be considered as a closing curly bracket |
| function{ | : | the macro id to be considered as a function definition containing the first opening bracket, the $$ parameter will indicate the position of the name of the function. Other parameters will be ## |
| function | : | same as function{ but not containing the first opening bracket |
| ; | : | the macro is to be considered as a semicolon. |
| for | : | the macro is to be considered as a "for" instruction, the $$ parameter indicates the position of the loop condition |
| while | : | the macro is to be considered as a "while" instruction, the $$ parameter indicates the position of the loop condition |

if : the macro is to be considered as a "if" instruction, the $$ parameter indicates the position of the condition

switch{ : the macro is to be considered as a "switch" instruction including the opening curly bracket. The $$ parameter indicates the position of the "expression" of the switch.

switch : same as the previous one but not including the opening curly bracket.

case: : the macro is to be considered as a "case" instruction including the colon symbol. The $$ parameter indicates the position of the condition.

case : same as the previous one, but not including the colon symbol.

default: : the macro is to be considered as a default instruction including the colon symbol.

default : same as the previous one, but not including the colon symbol

catch{ : the macro is to be considered as a "catch" instruction including the opening curly bracket. The $$ parameter indicates the position of the catch expression.

catch : same as the previous one, but not including the opening curly bracket.

## 3.6.2 Example

Source file:

```
#define DECLARE(x,y,z) void x(y,z)

#define FOR(x,y,z) for (x;z;y)

DECLARE(f,int argc, char **argv)
{
    A a;
    FOR(int x=0, x++, x<10) {
a.print();
    }
}
```

Analyzing this code without a correct alias file will provide the following output:

Output without correct alias file:

```
/* file begin */
#include "log_inst.h"

#define DECLARE(x,y,z) void x(y,z)

#define FOR(x,y,z) for (x;z;y)
```

```
DECLARE(f,int argc, char **argv)
{
/* function begin */
char *vlg_funcname = "::DECLARE::5"; <== The function should be named
"f"
VLG_DDP1(vlg_funcname, "05/11/04-17:40:04");
{

    A a;
    FOR(int x=0, x++, x<10) { <== the for condition has not been
detected
    a.print();
    }
}/* function end */
}


/* file end */
```

If we add the following lines in the alias file (log_inst.al):

DECLARE($$,##,##)|function

FOR(##,##,$$) |for

Output with a correct alias file:

```
/* file begin */
#include "log_inst.h"

#define DECLARE(x,y,z) void x(y,z)

#define FOR(x,y,z) for (x;z;y)
DECLARE(f,int argc, char **argv)
{
/* function begin */
char *vlg_funcname = "::f::5"; <== The name of the function is correct
VLG_DDP1(vlg_funcname, "05/11/04-17:40:04");
{

    A a;
    FOR(int x=0, x++,VLG_COND(vlg_funcname, ( x<10) ? 1 : 0, 2, 3)) {
<== for condition has been detected
    a.print();
    }
}/* function end */
}
```

```
/* file end */
```

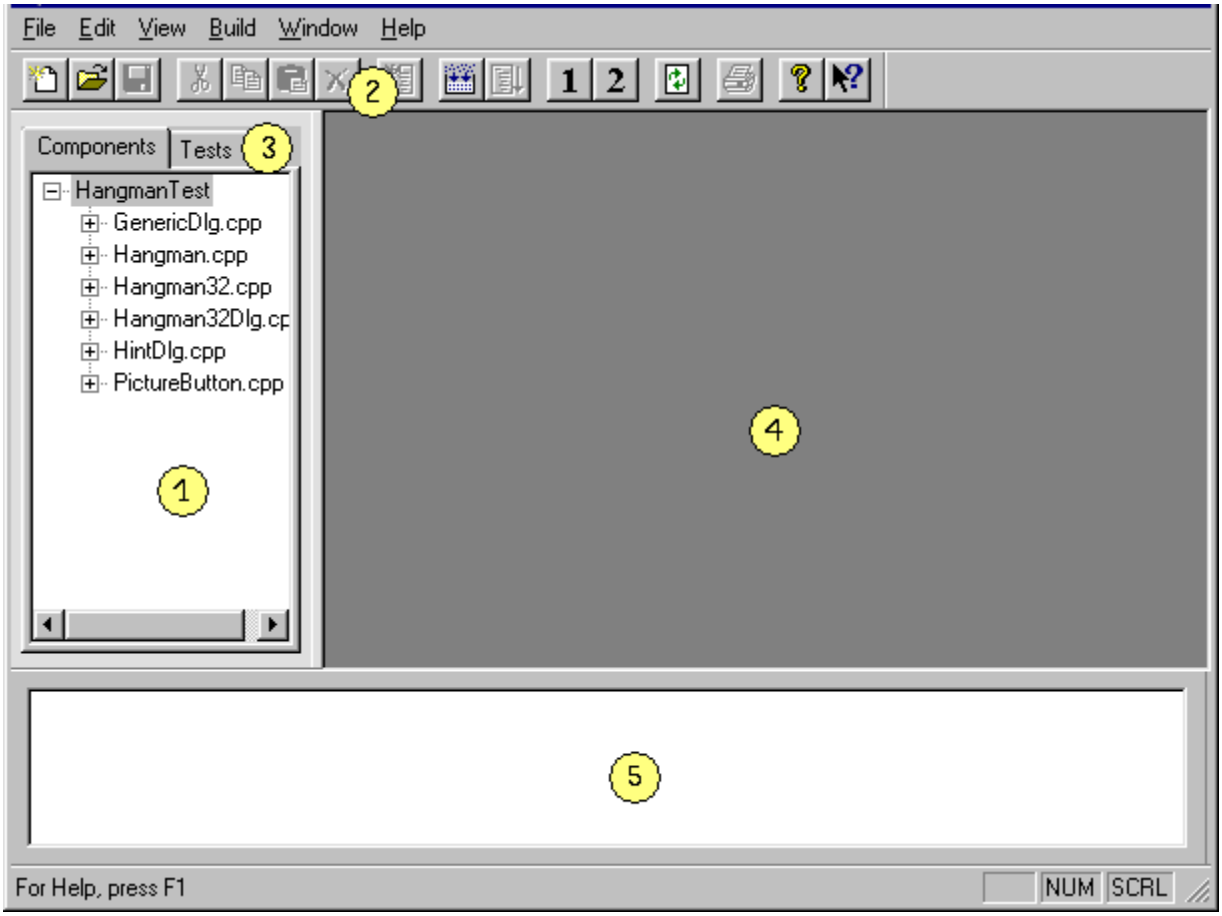
# Chapter 4

# *Testing on a Host Machine*

## 4.1 The Logiscope TestChecker Window

1. In Logiscope **Studio**, open the **Project** menu and select the **Start TestChecker** option or click the  icon in the Logiscope toolbar.

The Logiscope **TestChecker** tool opens up and looks like this:




1. **Project window**
   The Components pane displays the list of analyzed files. If you select and expand one

of these items, corresponding source file components (functions or methods) appear. The Tests pane shows the list of Logiscope test result files. At this point, nothing can be displayed because no test has been executed yet.

2. **Tool bar**
   Provides shortcuts to most commonly used Logiscope **TestChecker** commands.
3. **Project window tabs**
   Used to switch between the Test pane and the Component pane.
4. **Result pane**
   This pane will be used later to display *TestChecker* results. Component or Test pane will also be used to navigate from one result set to the other.
5. **Messages window**
   Displays error and warning messages

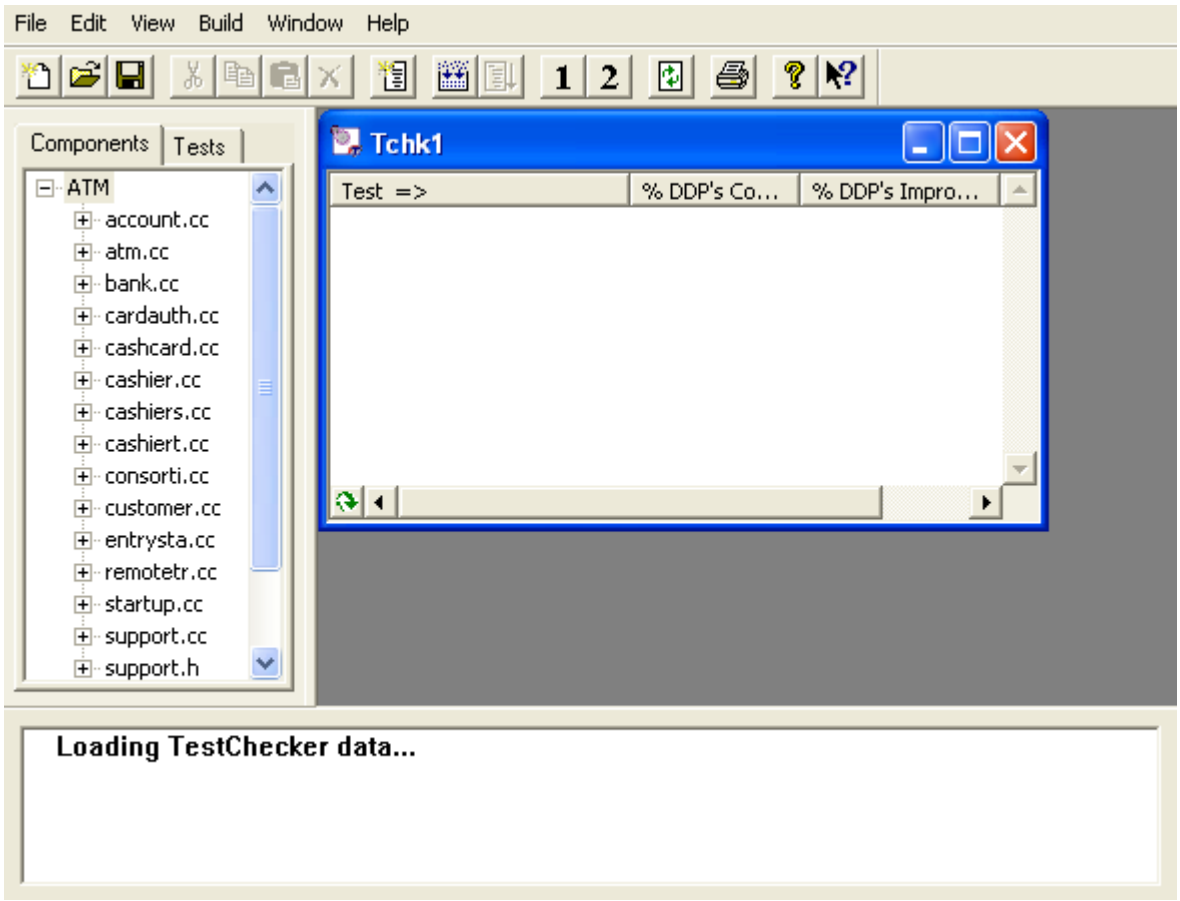
# 4.2 Creating and Running Your First Test

Tests are stored in test suites. You can create as many test suites as you need. They allow you to handle tests according to how your testing process is organized. Before running a test, you must create a test suite. This test suite will contain test coverage results. Of course, if a test suite has already been created it can be reused.

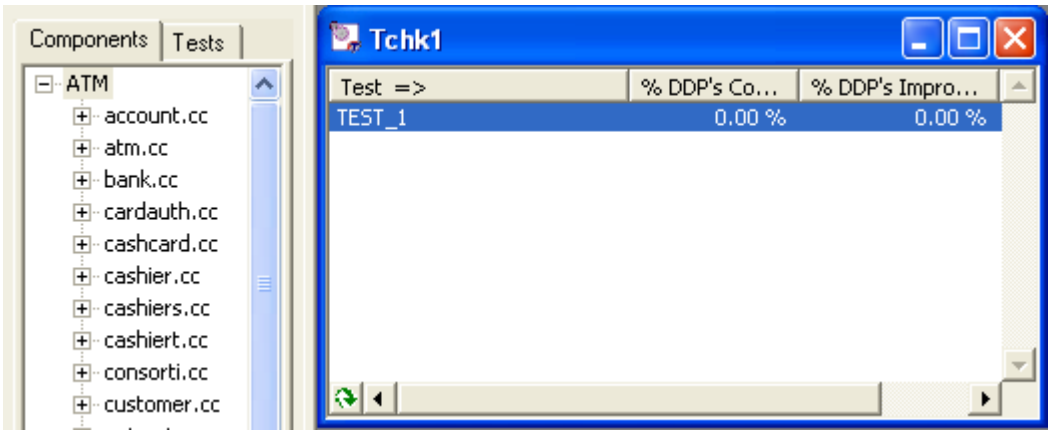
## 4.2.1 Starting the Test

1. Select the Test pane of the Project window by clicking on the **Tests** tab. You are ready to create a test suite.

2. Select the **File-New** command or use the  toolbar icon to create the test suite.
   A test suite window is displayed with the name *Tchk1*, as shown in the illustration below.

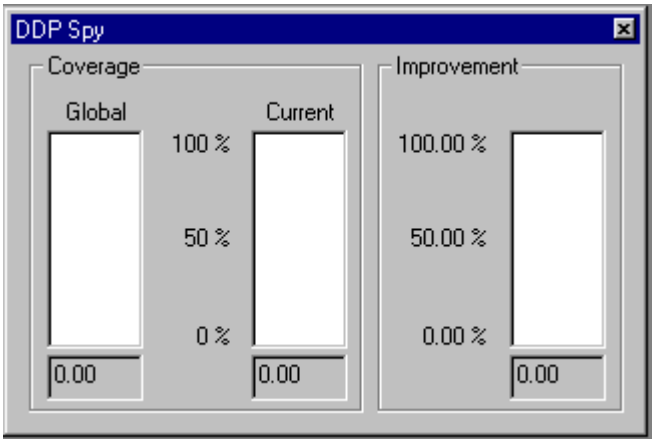



3. Select the test suite you just created by clicking on it and select the **Edit-New Test** command or click the  icon.
   This action creates a new test in the current test suite. The default name for this new test is *TEST_1*.




## 4.2.2 Viewing Coverage While Testing is in Progress

1. Select the **View-DDP Spy** command.The DDP Spy window appears.

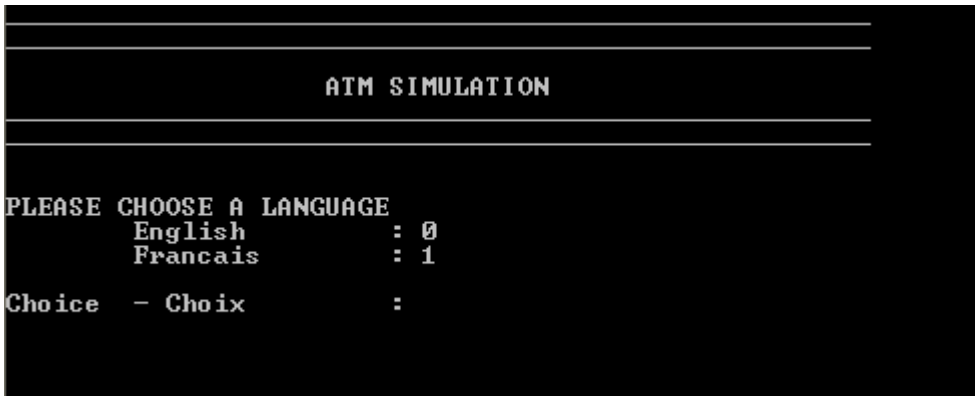



It will display the progress of code coverage during testing:

- the Global bar shows the cumulated coverage for all tests,
- the Current bar shows the coverage for the current test,
- the Improvement bar shows the global coverage improvement secured through the execution of the current test.

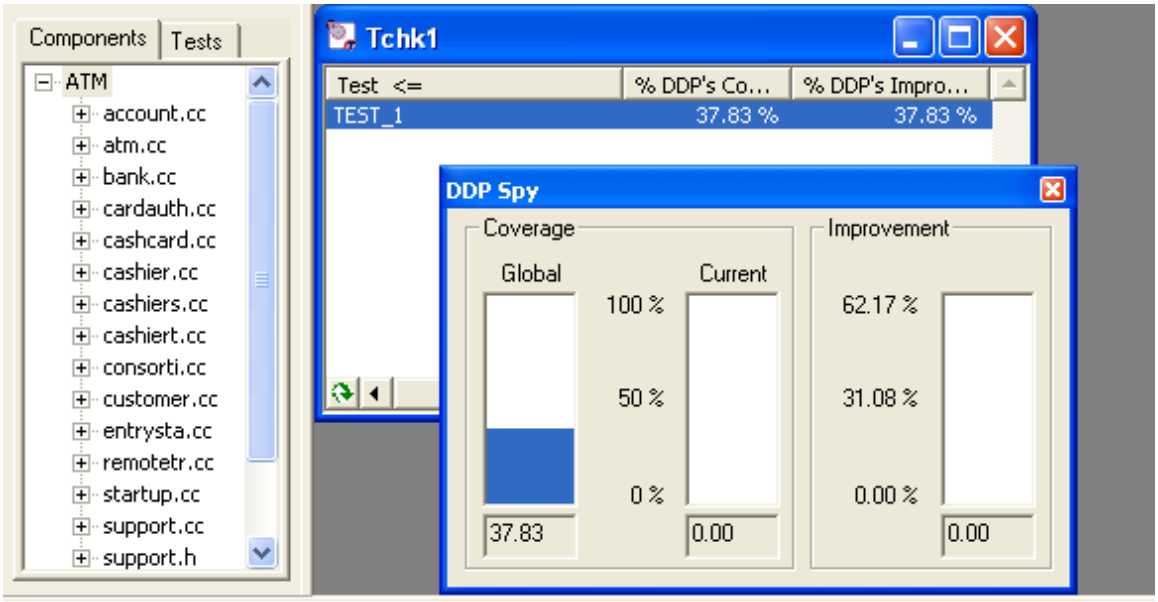
2. Click on *TEST_1*. This is the test you are going to run.

3. Press the <F5> key or click the  icon.
The test begins: a window appears in which you simulate a withdrawing operation.




As you execute / test the ATM program, you can see coverage rates increase in the **DDP Spy** window, but *TEST_1* is the only test executed for the moment so three indicators display identical values.

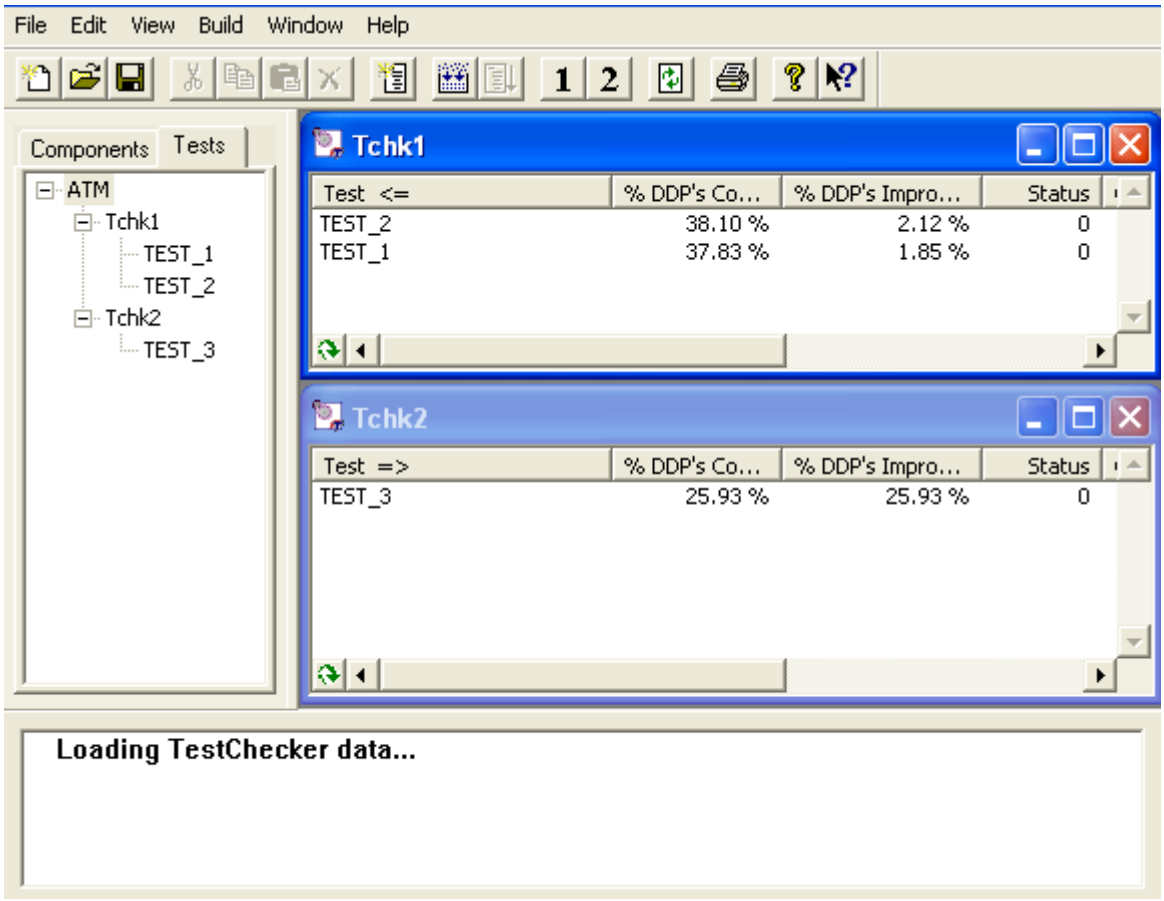
At the end of the test execution, the **DDP Spy** window displays the total coverage of the *TEST_1*.



### 4.2.3 Creating and Running More Tests

1. Create another test *TEST_2* in the *Tchk1* test suite and run it as indicated before.
2. Create another test suite, *Tchk2* for instance.
3. Similarly create and run another test *TEST_3* in *Tchk2*.

At this point, your *TestChecker* window should look approximately like the illustration below, although test coverage figures may somewhat differ.



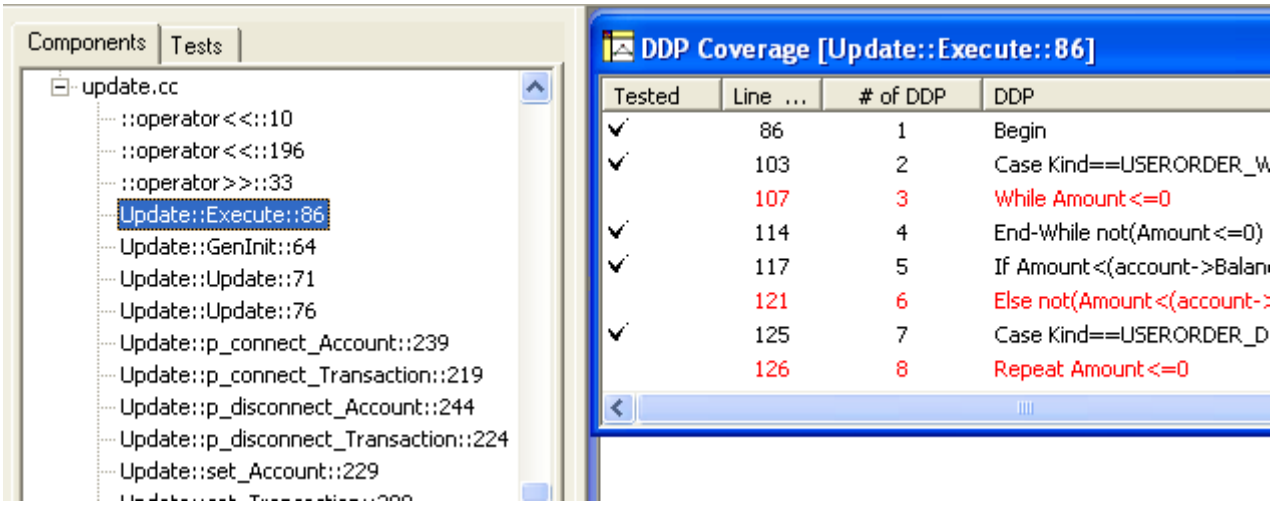

# 4.3 Displaying Tested and Untested DDPs

Logiscope **TestChecker** can display tested and untested Decision-to-Decision Paths (DDPs) for a function or a method. This will help you design complementary tests to achieve a better overall test coverage of the program.

To display the DDP Coverage window for a component:

1. Select a component, either in a Component window or in the Components pane of the Project window.
2. Select the **View-DDP Coverage** command.

The DDP Coverage window appears and shows tested and untested DDPs for the function or method currently selected.




Unreached DDPs are displayed in red. Covered DDPs are displayed in your standard font color.

*Note: Tested and untested DDPs color display depend on your computer settings.*

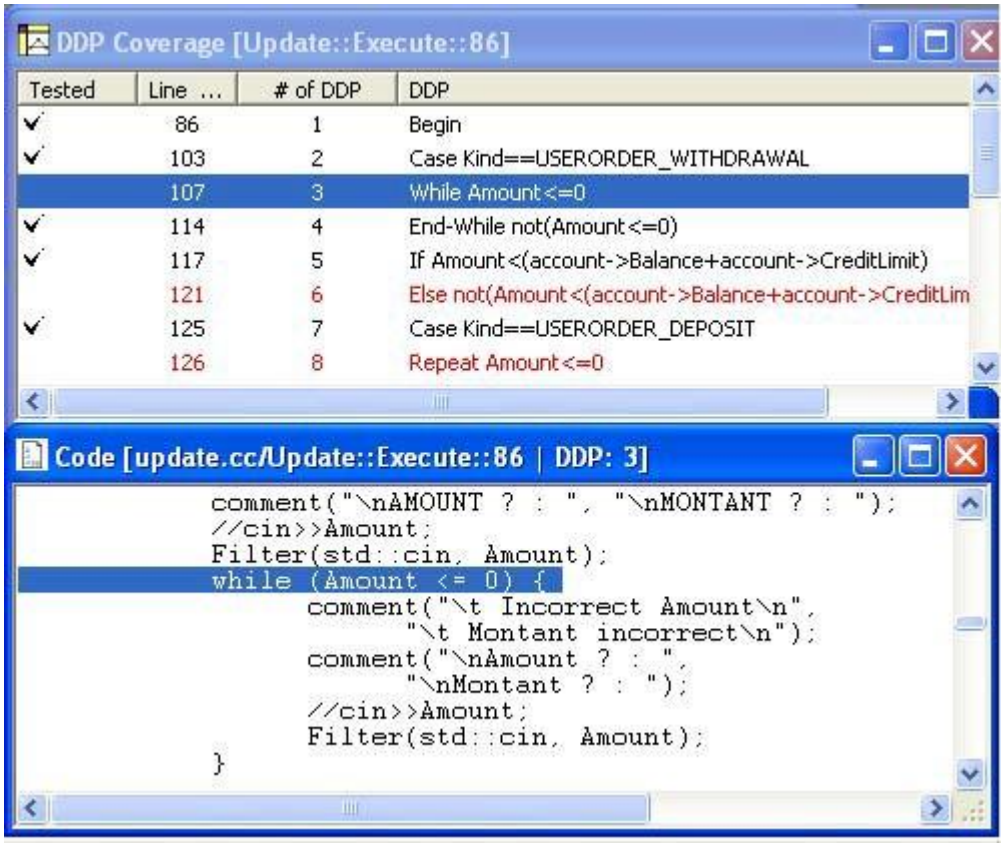
The above illustration shows that at line 107 the required condition to execute the untested DDP: `While Amount <= 0` must be true.

You just have to design a new test case in order to cover this DDP and thus increase the test coverage.

# 4.4 Displaying the Source Code

Logiscope **TestChecker** can also display the source code related to a function/method or to a DDP.
To display the Source Code window:

1. Select a Component in a Component window or in the Components pane, or a DDP in a DDP Coverage window.
2. Select the **View - Source Code** command.
3. The Code window pops up and displays the source code of the function/method or decision selected: e.g. *Update::Execute* in the file update.cc.




4. Select another component: the Source Code window is updated accordingly.
5. In a DDP Coverage window, select a decision line: here again, the Source Code displays appropriate lines of code.

# 4.5 Saving and closing a Project

To save the current project:

1. Select **File-Save All** command. Save test sessions under **Tchk1.dyn** and **Tchk2.dyn.**
2. When *TestChecker* is closed, a warning message pops up from Logiscope **Studio** asking for reloading the current modified workspace. Click **Yes** to reload.

   The **Studio** main view is updated taking into account test sessions done previously. These are stored in the **Dynamic Files** folder in the tree structure of the project.
3. In **TestChecker** window, select the **File - Save Project** command.

To close the current project:

4. Select the **File-Close Project** command.
5. Select **File-Exit** to quit Logiscope **TestChecker** tool.

Chapter 5

# *Analyzing Test Coverage from Logiscope Studio and Viewer*

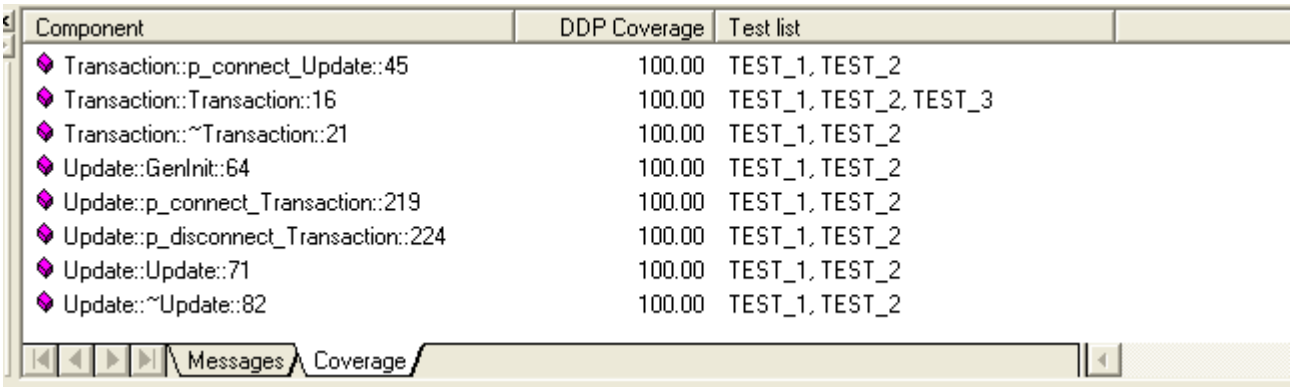
At the end of the previous chapter, you have executed several test cases on the ATM program.

Logiscope **Studio** and **Viewer** help you reviewing the progress of the testing process using test coverage results.

## 5.1 Test Coverage Analysis Using Logiscope *Studio*

### 5.1.1 Test Coverage

1. Select **Browse-Test-Component Coverage** menu or click the  icon.
   A new tab is added to the **Output** window containing the list of tested components with the associated DDP (Decision-to-Decision Path) coverage rate, as well as the whole test set to which each component belongs.

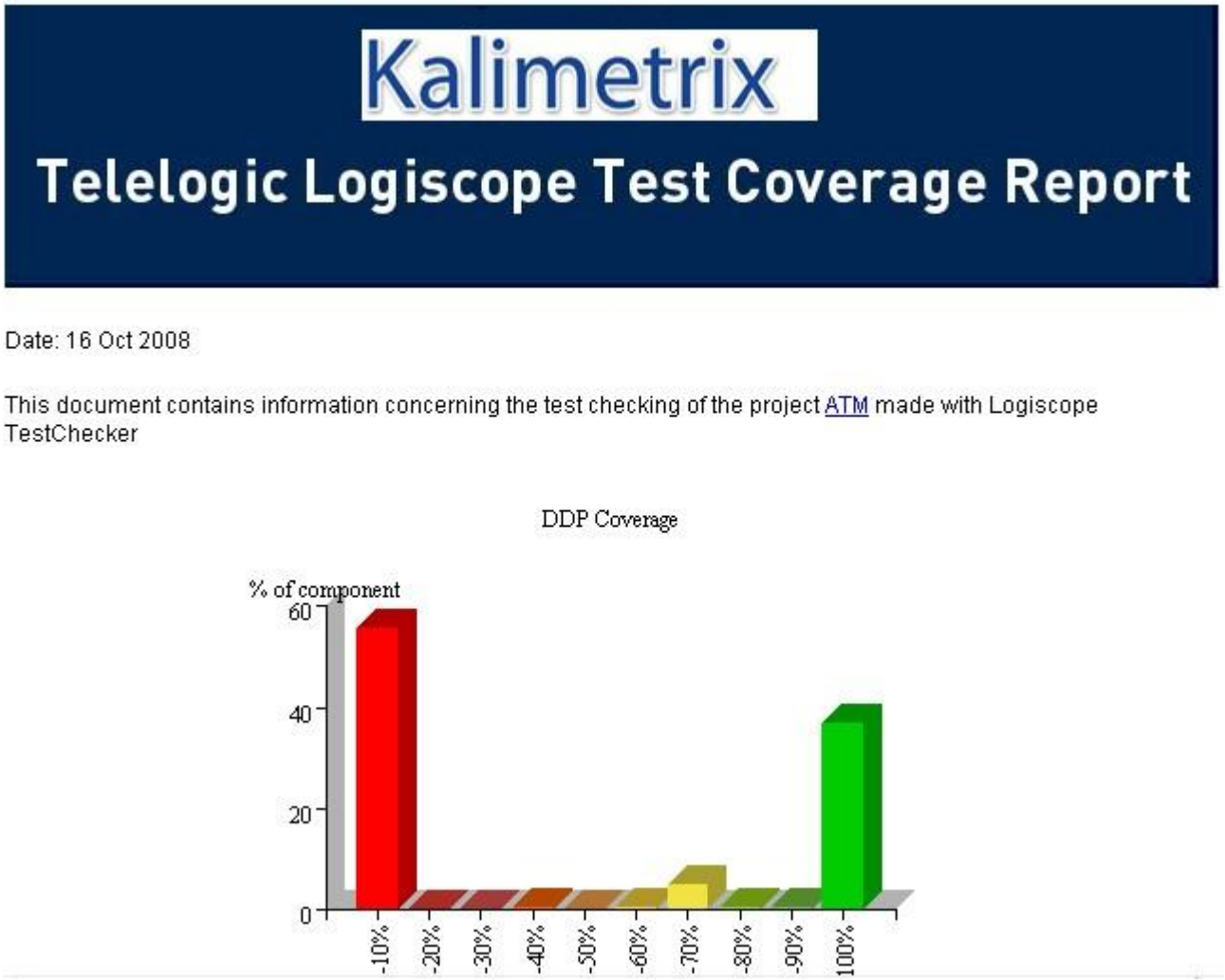



If you double click on the component, the corresponding source code appears.

You can rank components according to their coverage rate by clicking on the **DDP Coverage** column. By default, they are sorted alphabetically.

## 5.1.2 Test Report

1. Select **Browse-Test-Test Report** menu or click the  icon. An HTML window is displayed containing a synthesis of your application test coverage.




You can use the **HTML Browser Toolbar** to navigate back and forth within the Test Report. It is possible to save it using the command **File-Save As...**

*Note: Histograms shown in the following chapters are not available on UNIX. They are replaced by tables. If you want to generate a report with histograms, change HTML reports option in **Tools-Options...** command (you will see them in your favorite Internet navigator).*

2. Click on the **60-70%** yellow bar. The list of components whose DDP coverage is between 60% and 70% is displayed.

## DPP coverage between 60% and 70%

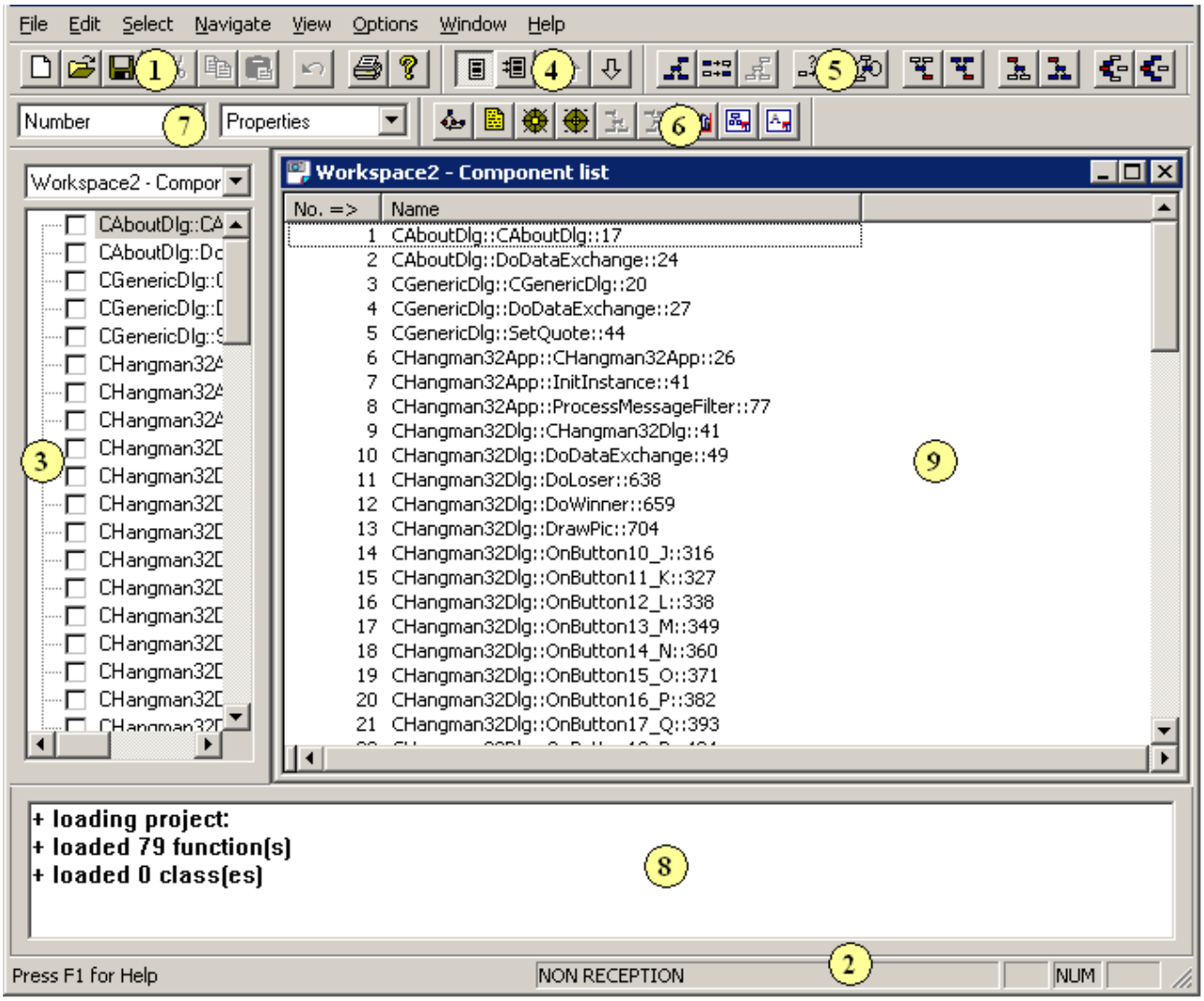
| Name | Ddp | Tests |
|---|---|---|
| CardAuthorization::p_disconnect_RemoteTransaction::126 | 60.00% | TEST_1, TEST_2 |
| PolyText::GetText::117 | 60.00% | TEST_1, TEST_2, TEST_3 |
| ::operator<<::196 | 66.67% | TEST_2 |
| ::operator>>::33 | 66.67% | TEST_1, TEST_2 |
| Account::unset_Customer::127 | 66.67% | TEST_1, TEST_2, TEST_3 |
| Bank::unset_Consortium::105 | 66.67% | TEST_1, TEST_2, TEST_3 |
| CardAuthorization::unset_Bank::65 | 66.67% | TEST_1, TEST_2, TEST_3 |
| CardAuthorization::unset_Customer::139 | 66.67% | TEST_1, TEST_2, TEST_3 |
| CashCard::unset_CardAuthorization::44 | 66.67% | TEST_1, TEST_2, TEST_3 |
| Text::SetStr::36 | 66.67% | TEST_1, TEST_2, TEST_3 |
| Update::unset_Transaction::214 | 66.67% | TEST_1, TEST_2 |

3. Select the **Home** icon  in the HTML toolbar to go back to the Report first page.
4. Click on the ATMTest hypertext link. The list of project source files appears.
5. Select **File-Close** to close the Test Report.
6. Select **File-Close Workspace** to save all project modifications before lanching Logiscope Viewer and get more results on test coverage

# 5.2 Test Coverage Analysis Using Logiscope *Viewer*

Open Logiscope *Viewer* from the **Project-Start Viewer** menu of Logiscope **Studio** or click the  icon..

The Logiscope **Viewer** main window looks as follows:



This window contains the following elements:

1 **Toolbar**
Provides shortcuts for most commonly used commands of **File** and **Edit** menus.

2 **Status Bar**
Indicates the status (RECEPTION, NON RECEPTION) of the active window displayed in the Result Pane.

3 **Control Palette: Workspace1-Component list Window**
Displays a view of the components after loading a Logiscope project. Select or deselect the one you want to explore.

4 **Navigation bar**
Provides shortcuts for the commands of the **Navigate** menu.

5 **Selection Bar**
Provides shortcuts for the most commonly used commands of the **Select** menu.

6 **Component windows bar**
Allows to display graphical results: control graph, source code, metric and criteria Kiviat graph, and to go to the Application window.

7 **Selector Bar**
Uses the selectors to choose and display additional information in the active Domain window. The content of this bar depends on the view being displayed in the active Domain window.

8 **Messages window**
Displays error messages or indications on the loading project.

9 **Result Pane**
Used to display window command results.

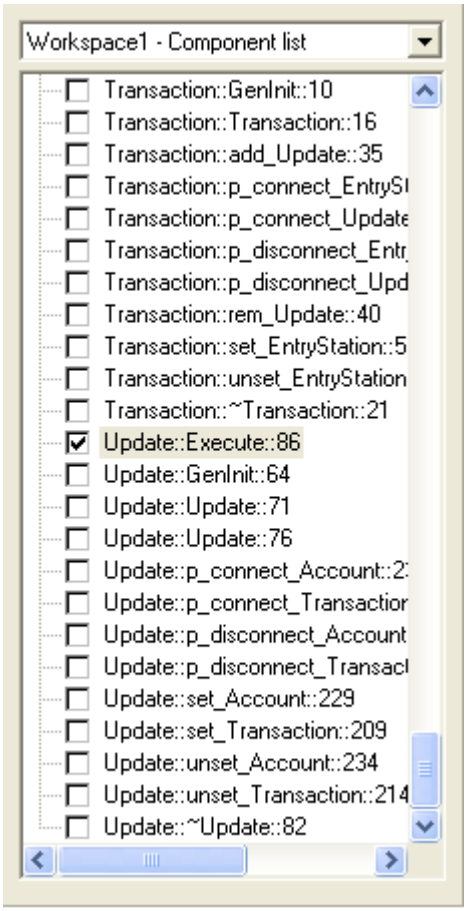
Use the **View** menu to show or hide some items described above and to customize the *Viewer* main window as you wish.

## 5.2.1 Selecting/Deselecting a Function

All functions and methods defined in the program are listed in **Workspace1-Component list** window as well as in the **Control Palette** .

You can use indifferently either the **Control Palette** or the **Workspace1-Component list** window to select or deselect functions.

1. In the **Control Palette,** click the function **Update::Execute::86**.



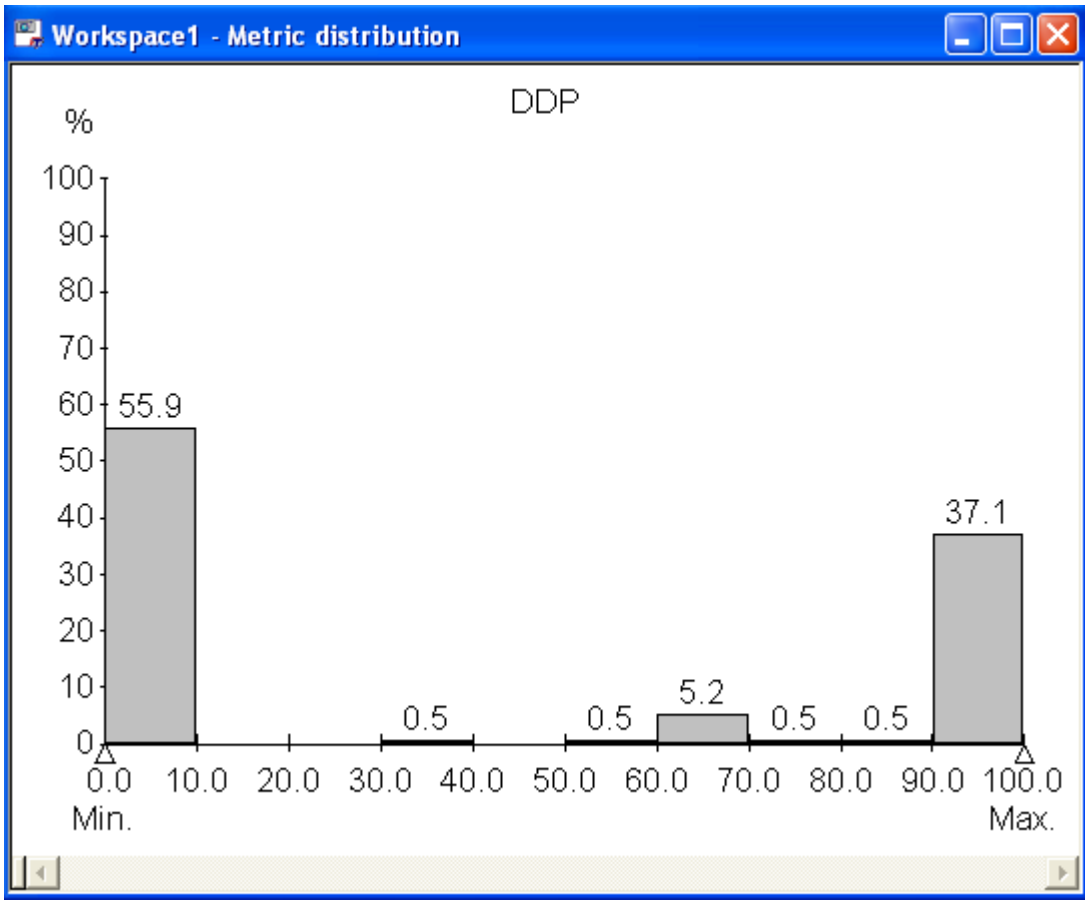

The selection has been propagated to the **Workspace1-Component list** window.

You can select a particular function in every **Domain** window displayed in the **Result Pane**.

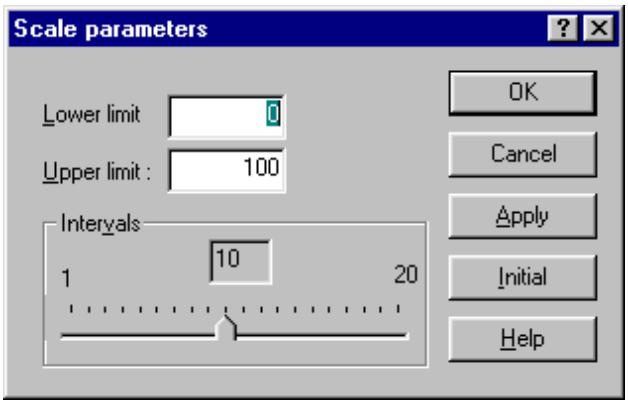
## 5.2.2 Viewing Test Coverage Results

### Decision to Decision Path Coverage

1. Select **View-DDP Coverage Distribution**. The component list window is updated.
2. Select **Options-Scale** to change the display format.




You can parametrize Scale Display using **Options-Scale Parameters...** command.



Let us go back to the **Workspace1- Metric distribution** window.

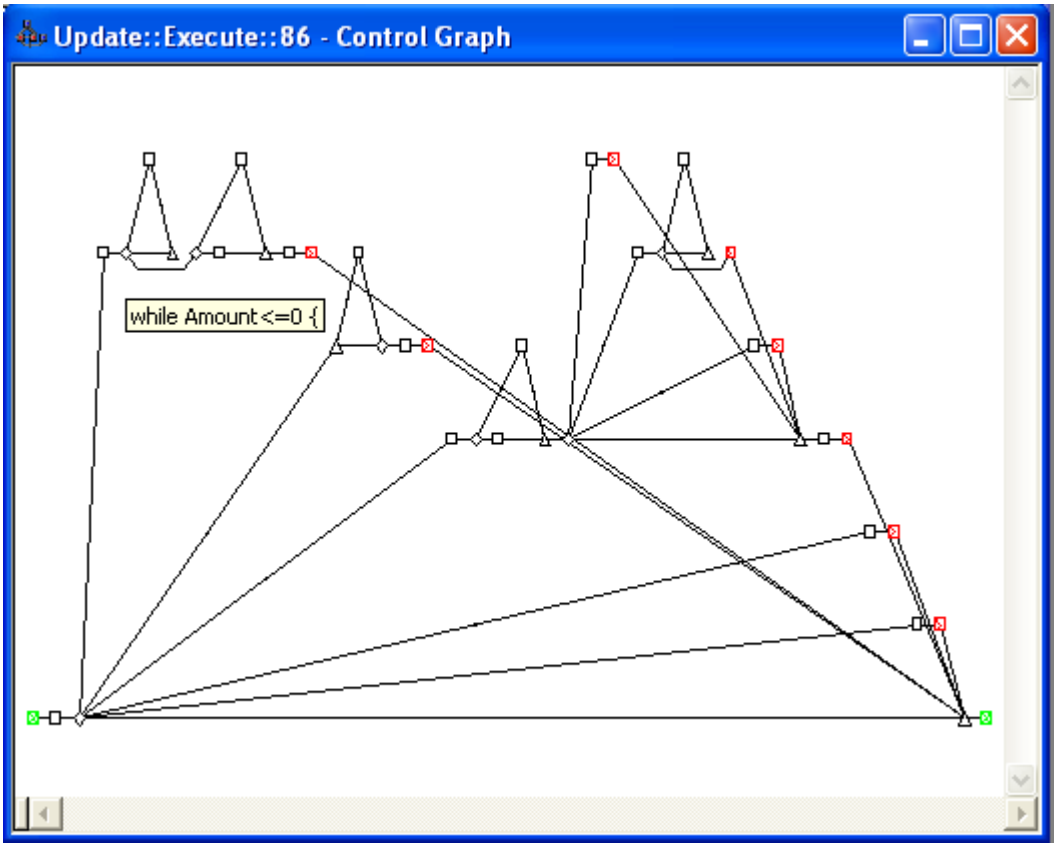
If you select a bar, it becomes blue and all components in this category are check-marked in the workspace view as well as selected in the **Control Palette**.

This distribution is the same as the one in the Test Report representation (see previous chapter).

3. Select **View-Component List** to make the list of all project components appears.

4. Select the component *Update::Execute::86* .

5. Select **Window-Control** or click the  icon. The corresponding control graph is displayed in a new window.
   The control graph is the graphical representation of the selected function with featuring geometric symbols (nodes) linked by arrows (edges). It represents the logical structure of the function.
   If you place the cursor over the first diamond-shaped node the pseudo-code linked with it is displayed as shown below:

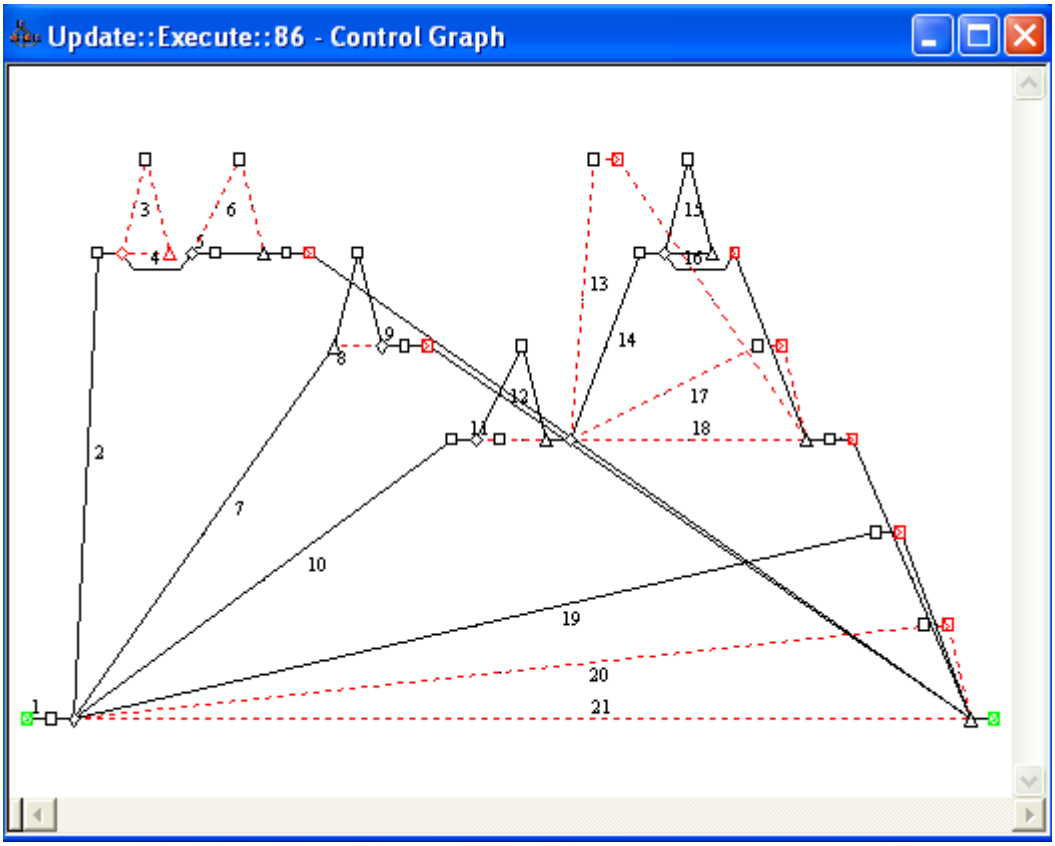



For more details on the control graph representation, please refer to the *IBM Rational Logiscope Basic Concepts* manual.

6. Select **Options-DDP Numbers** to add DDPs numbers to the graph.

7. Select **Options-Coverage** to display covered and not covered paths.

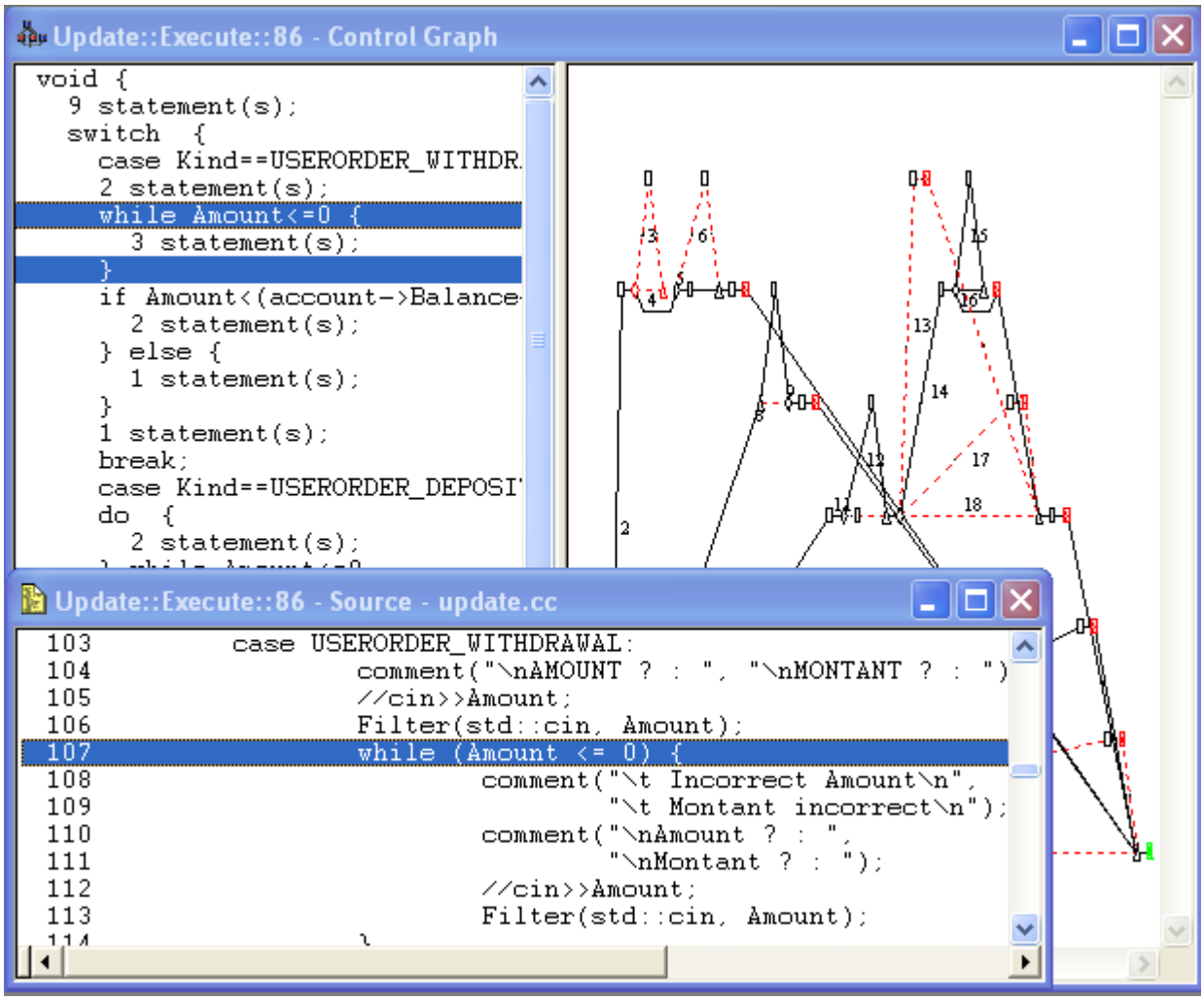
Now the control graph looks like this:




Covered paths are represented by black solid lines and not covered paths by red dotted lines.

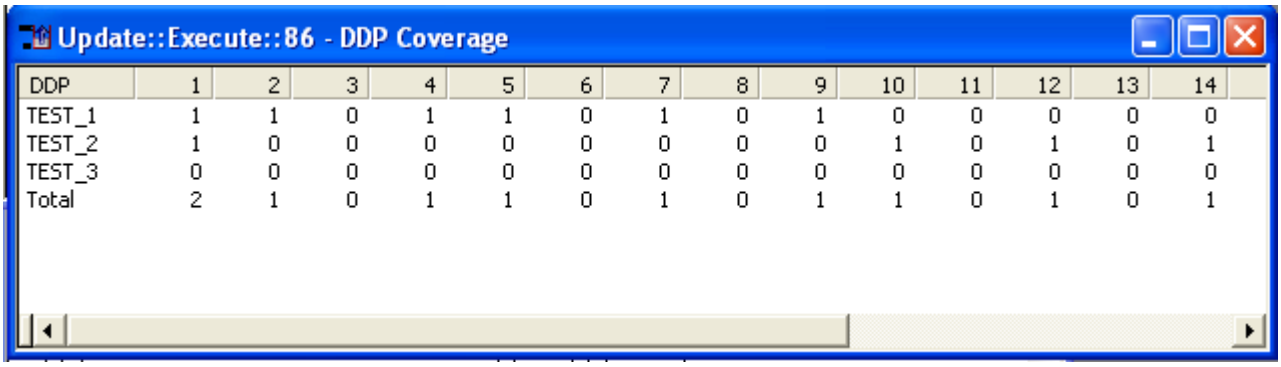
For instance, the DDP #3 has not yet been exercised by the executed test cases. you can used many way to understand what decision shall be satisfied to test this untested path:

- place the cursor over the corresponding diamond-shaped node as shown previously,
- select the DDP starting node and display the entire pseudo-code at the same time as the control graph thanks to **Window-Split** command,
- use the **Window-Source** command to display the source code of the function in a new window and then, select the DDP starting node to highlight the corresponding code.




These representations can be configured with the **File-Preferences** menu.

8. Select **Window-Coverage** or click the  icon**.** Rearrange both windows to compare all results.

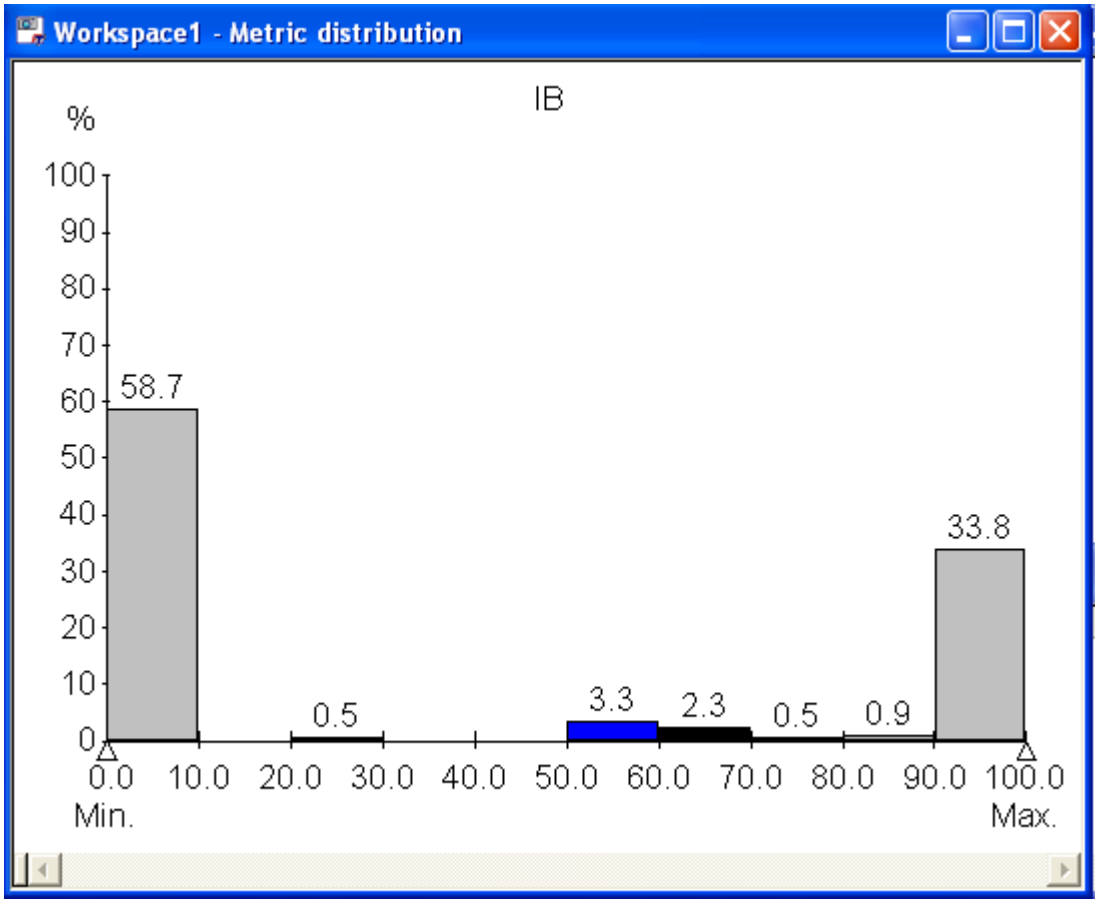



| DDP | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TEST_1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| TEST_2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| TEST_3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 2 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

In the **DDP Coverage** window, you can see the coverage of each test case and for each DDP of the function: value is 1 for covered DDP and 0 for uncovered. The Total line indicates how many times the DDP has been exercised by the test cases.

### Test Coverage: Instructions Block Coverage

1. Select the **Workspace1- Metric distribution** window.
1.
2. Select **View-IB Coverage Distribution.**
   The Workspace view is updated




You can repeat the steps of the previous paragraph about control graph replacing DDP by IB.

## 5.2.3 Ending *Viewer* and *Studio* Sessions

1. In Logiscope *Viewer,* select **File-Exit** to end the Logiscope **Viewer** session.
2. In Logiscope *Studio,* select **File-Exit** to end the Logiscope **Studio** session.

# Chapter 6

# *Building a C Instrumented Code for MC/DC Analysis*

## 6.1 Before you start

Along with this chapter, you are provided with a program written in C language, an implementation of the *Mastermind* game. The program has been carefully designed for you to use all features of Logiscope *TestChecker*.

Source files of this program are stored in the directory **<InstallationDir>\samples\Tchk\C\Mstrmind.**

As a precaution to keep original files safe, it is highly recommended that you copy this subdirectory into a working directory of your own: e.g. **C:\Mstrmind**. on Windows, $HOME/**Mstrmind** on UNIX.

In addition, you will create Logiscope projects and associated repositories: i.e. sets of files containing internal data used by Logiscope. It is recommended to a create a dedicated directory to store these data: e.g. a folder named **LogiscopeProjects**.

## 6.2 Creating a *TestChecker* Project

First, you shall define a Logiscope *TestChecker* project which mainly consists in:

- the list of source files to be first instrumented and then being tested for test coverage analysis,
- applicable source code instrumentation options according to the compilation environment,
- the special traces that will be generated by the Logiscope libraries during the execution of the test cases on the instrumented application.

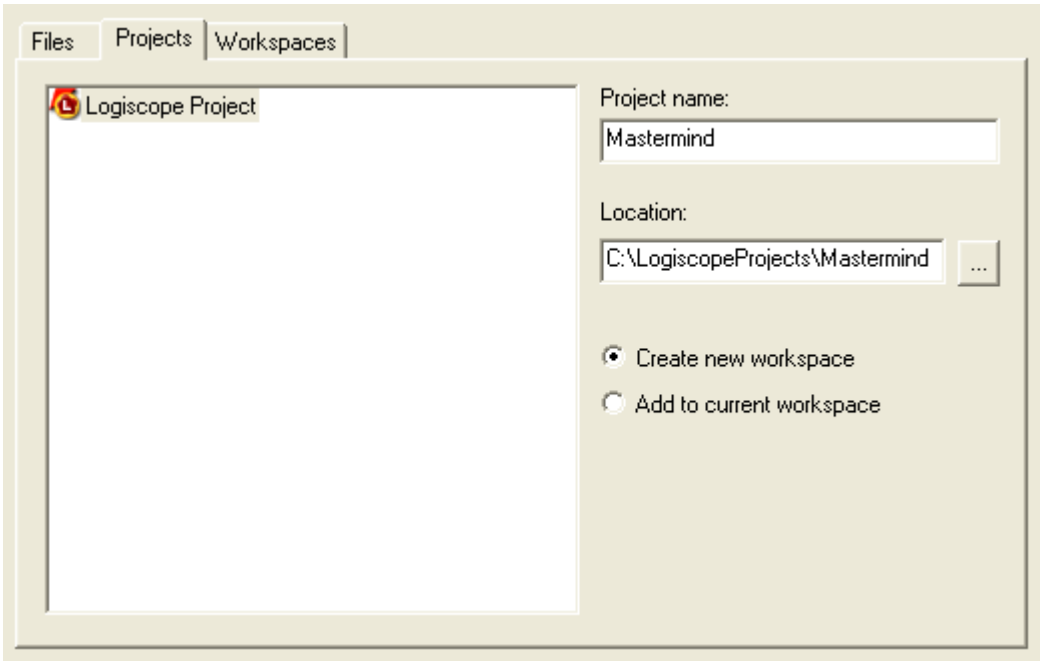
1. Open a Logiscope **Studio** session (see section 3.2).
2. In the **File** menu, select the **New**... command or click the  icon.

The **New** Logiscope Projects dialog box appears.

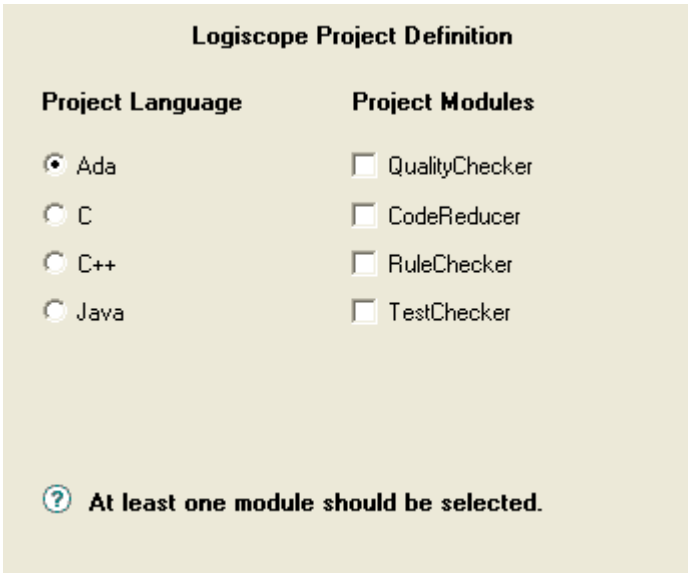
3. In the **Project name:** pane, enter the name for the new Logiscope project to be created. In the context of the guided tour, this simply can be the name of the application under test: e.g. Mastermind.
   The information provided in this pane will be then refer as the <ProjectName>.

4. Then select its **Location:** i.e. the directory where the Logiscope project (i.e. a "**.ttp**" file) and the associated Logiscope repository will be created; the Logiscope repository is a folder in which Logiscope internal analysis result files are generated.
   The information provided there will be then refer as the <LogiscopeRepository>.

Note: By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.




5. Click **OK** to access to the **Logiscope Project Definition** first window.



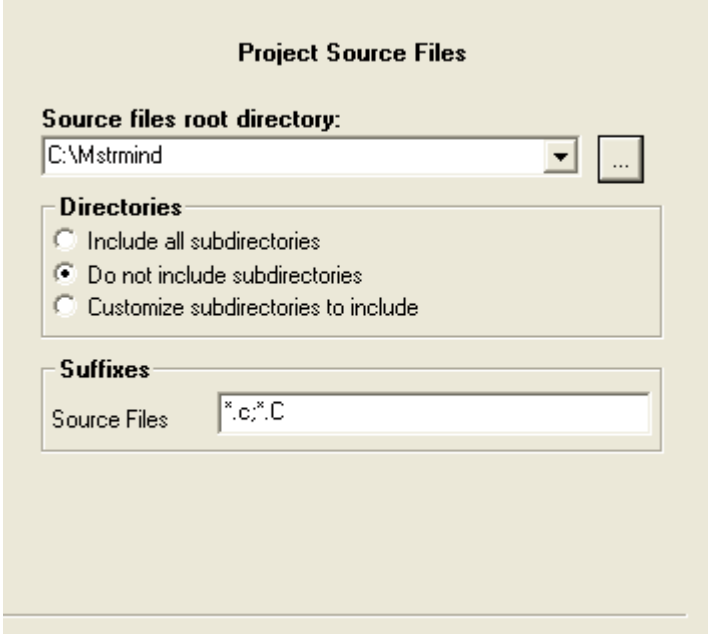

6. Select the **Project Language:** i.e. the programming language in which are written the source code files to be analysed.
   For the *Mastermind* project, select **C**.

   Note: Only one language can be selected. If your application contains source code files written in several languages, you should create several distinct Logiscope projects: one for each language.

7. Select the **Project Modules:** i.e. the verification modules to be activated on the source files of the project .
   For this guided tour, select **TestChecker**.

8. Click the **Next** button to continue the creation.

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.




**Source files root directory** shall specify the location directory of the source files to be analyzed.

9. Browse to select the directory where the *Mastermind* sample source files are located: i.e. in the **samples/Tchk/C/Mstrmind** folder of the Logiscope installation directory or in the directory where the source files have been copied as recommended: e.g. **C:/ Mstrmind.**

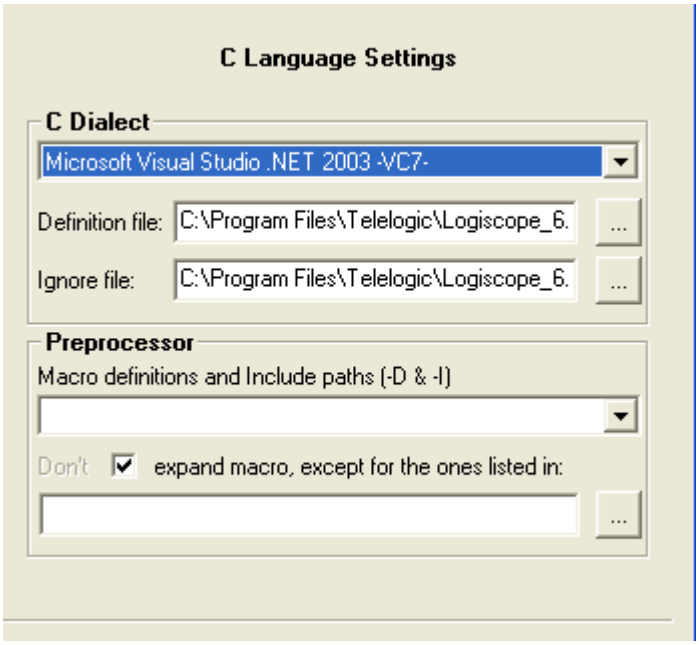
The **Directories** choice allows to select the list of repertories covering the application source files.

- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

**Suffixes** choices allow to specify applicable source file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

10. Click the **Next** button.

The **C Language Settings** dialog box  allows setting up C source code parsing options:

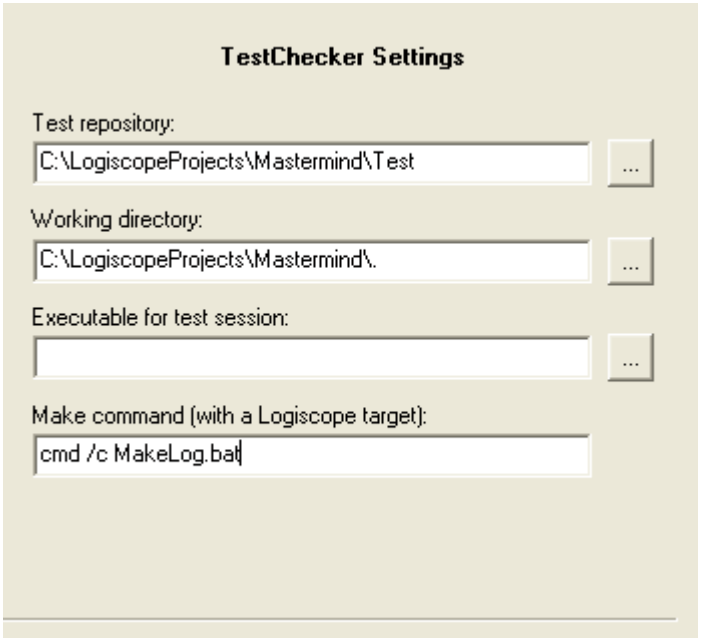



The default values are here appropriate for the context of the *Mastermind* example.

For more details on these options, please refer to the chapter *Parsing Options* in the *Kalimetrix Logiscope RuleChecker & QualityChecker C Reference Manual.*

11. lick the **Next** button.

The **TestChecker Settings** dialog box is now displayed. It allows to specify some of the key settings of a Logiscope *TestChecker* project.




12. The **Test repository:** is the directory in which the traces files generated when executing the instrumented executable will be saved.
    Keep the default location i.e. a Test folder to be created in the Logiscope repository specified in the **New** Logiscope Projects dialog box (see Item 3.).

13. The **Working directory**: is the directory where the make file can be found and where the executable will be generated (unless otherwise specified by the make file).

14. The **Executable for a test session:** shall specify the instrumented executable.
    In this context, the executable is not yet generated and will be chosen later.

15. The **Make command** file shall contains the command to build the instrumented executable. Type the following command:
    **on UNIX: make lgmstrmind**
    **on Windows, cmd /c MakeLog.bat**

    *Note: According to your DOS version, use the equivalent of the 'cmd' command.*

The Make command will launch the make file in which a Logiscope target has been defined, to compile and link-edit together the instrumented source files and the instrumentation library file located in **<InstallationDir>/instr/src/vlgtchk.c**.
In the next section "Building the Instrumented Executable", you will be prompted to edit and modify the make file specified in this pane to adapt it to your compilation environment.

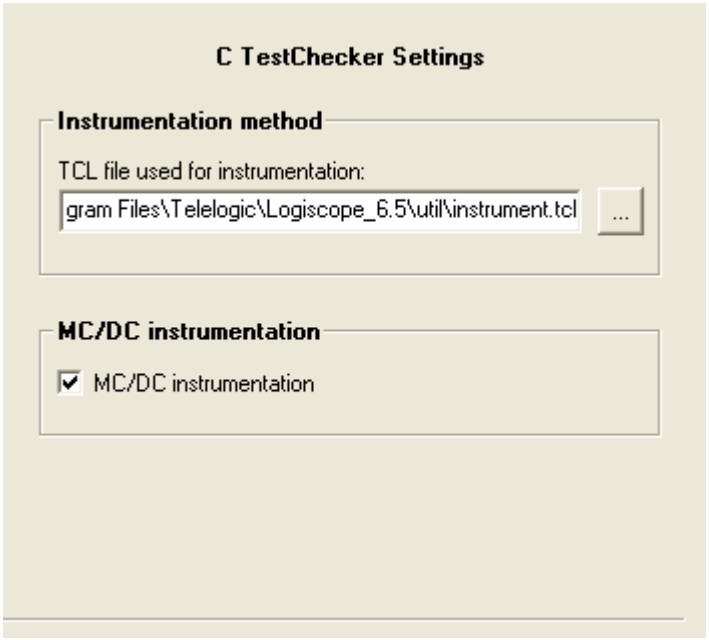
16.Click **Next**.

The following wizard box will allow you to complete the *TestChecker* project specification with some specifics of the C language.

17.In the **Instrumentation method** part, you can choose the way of instrumenting the source code. In the context, keep the default instrumentation model provided with the product.

18.As an example for this session, check the **MC/DC instrumentation** option to benefit from the MC/DC advantages .

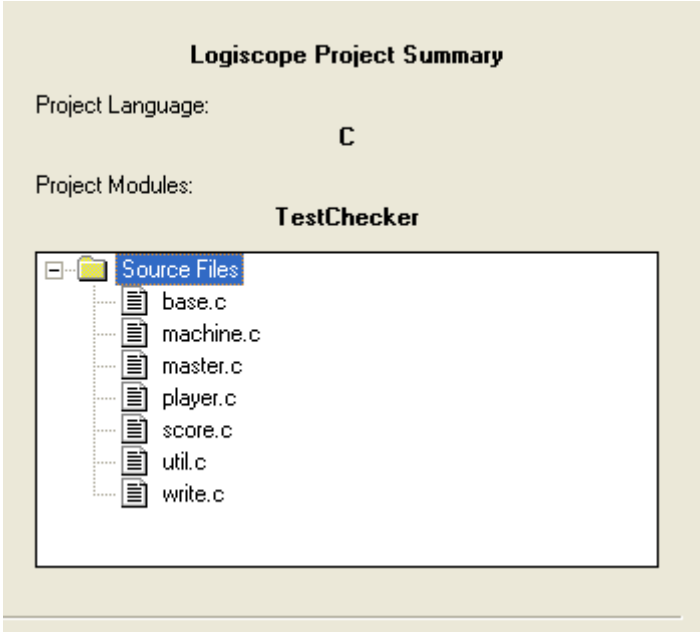
.




For more details on MC/DC, please refer to Chapter 2.

19.Click **Next**.

The last wizard window is displayed. You can check if all files are correct by expanding folders:




20. Click **Finish** to create your first C *TestChecker* project.

The Logiscope **Studio** main window is now updated and contains the workspace view of your project (see next section).

Two files has been created by this process and are of the form: <ProjectName>**.ttp** for project and <ProjectName>**.ttw** for associated workspace. They are both located in the folder specified as the Logiscope Repository.

# 6.3 Building an Instrumented Executable

Your project is ready to be built. Building consists in:

- instrumenting the code using the instrumentation method selected for the project,
- generating (i.e. compiling and linking) the instrumented executable.

When building the instrumented executable, in order the makefile works, the original sources files are temporarly replaced by the instrumented ones and then restored.

First of all, you must adapt the Make Command to your compilation environment:

1. Open a text editor and load either the file **makefile** file or the file **makefile.vc** if you intend to compile the code using a Microsoft Visual compiler.
   It starts by the following lines:

```
    # makefile for Mastermind C example
LOGISCOPE_INSTALL = ..\..\..\..
```

```
OUTDIR = Objects
INDIR = ..\Mstrmind
...
```

2. Adapt the value of the variable `LOGISCOPE_INSTALL` to correspond to the path of the Logiscope installation directory.
3. **On Windows only:** open a text editor and load the **MakeLog.bat** file located in the directory where the *Mastermind* source files are: e.g. **C:\Mstrmind**.
   It contains the following lines:

```
set VC8=C:\Program Files\Microsoft Visual Studio 8\Vc\bin\vcvars32.bat
set VC7=C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin\vcvars32.bat
set VC6=C:\program files\microsoft visual studio\vc98\bin\vcvars32.bat
```

4. If you intend to compile the code using a Microsoft Visual compiler, set the appropriate path to the installation directory of the compiler to be used.

Once the Make command has been adapted:

5. Select the **Project-Build** command or click the  icon. A new tab is added in the **Output** window and will contain code instrumentation and generation messages:

```
Analyzing: ../Mstrmind/base.c...
log_cc : using default options file : ANSI.def

Analyzing: ../Mstrmind/machine.c...
log_cc : using default options file : ANSI.def

Analyzing: ../Mstrmind/master.c...
log_cc : using default options file : ANSI.def
...
```

After building the project, the **Message** tab of the **Output** window will contain final results.

The project is built. Otherwise this window will display error messages.

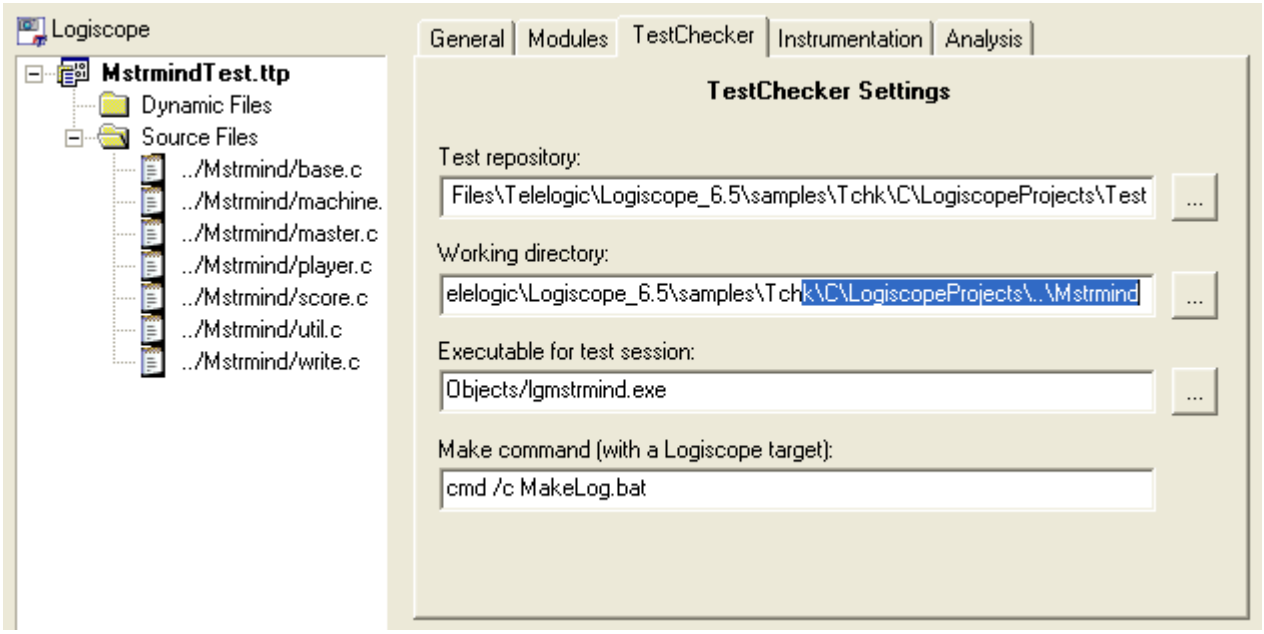
You will now end up *TestChecker* settings specifications.

6. Select **Project-Settings** option or run the <Alt+F7> command to specify the executable file.
7. Select the **TestChecker** tab.
8. You can now specify the **Executable for test session:** i.e.the command to launch the instrumented executable:

   **On Windows:** *Objects/lgmstrmind.exe*

   **On Unix:** *xterm -e lgmstrmind*

   If you command name has blank spaces put it between double-quotes (“). The space is interpreted as the separator between the command and its parameters.



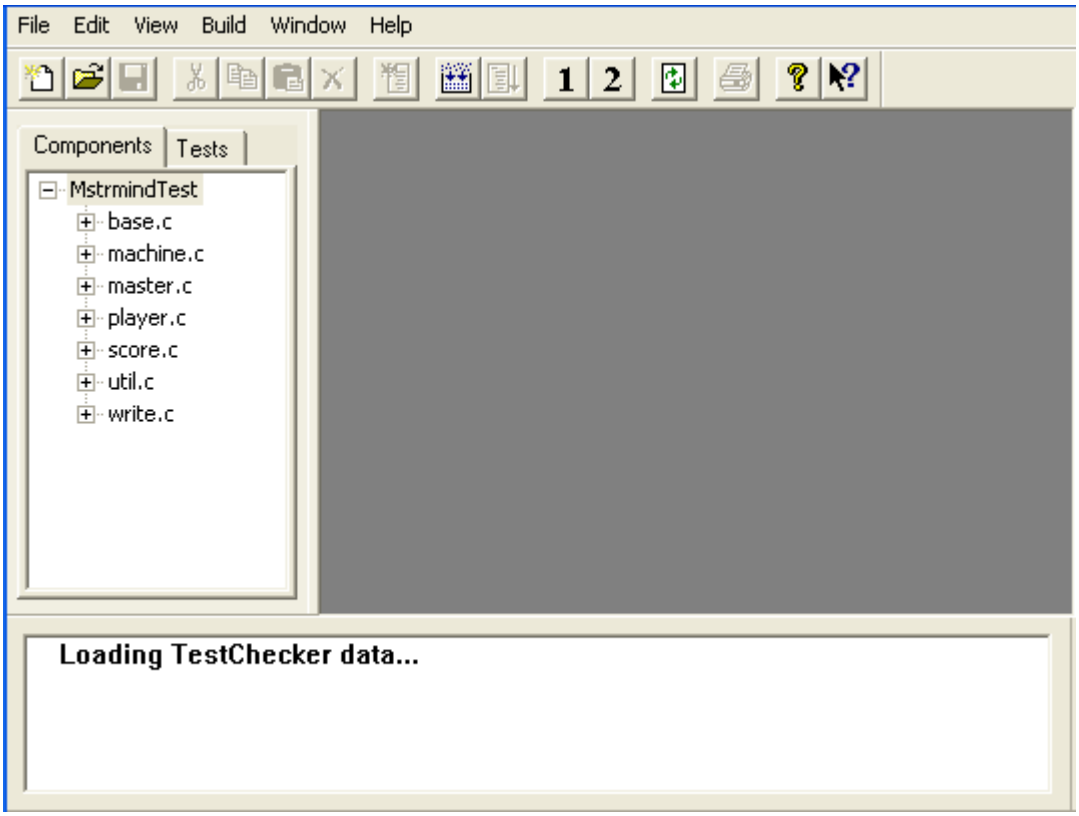

9. Click **OK** to confirm.

10. Do not forget to save your project!
    Select **File-Save Workspace** to save it.

The instrumented executable generation is complete, your project is ready to be tested. For this you are going to use the Logiscope **TestChecker** tool.

# 6.4 Testing the Instrumented Executable

1. Select **Project-Start TestChecker** to load the *Mastermind* project in Logiscope **TestChecker**. The main window looks as follows:




## 6.4.1 Starting the Test

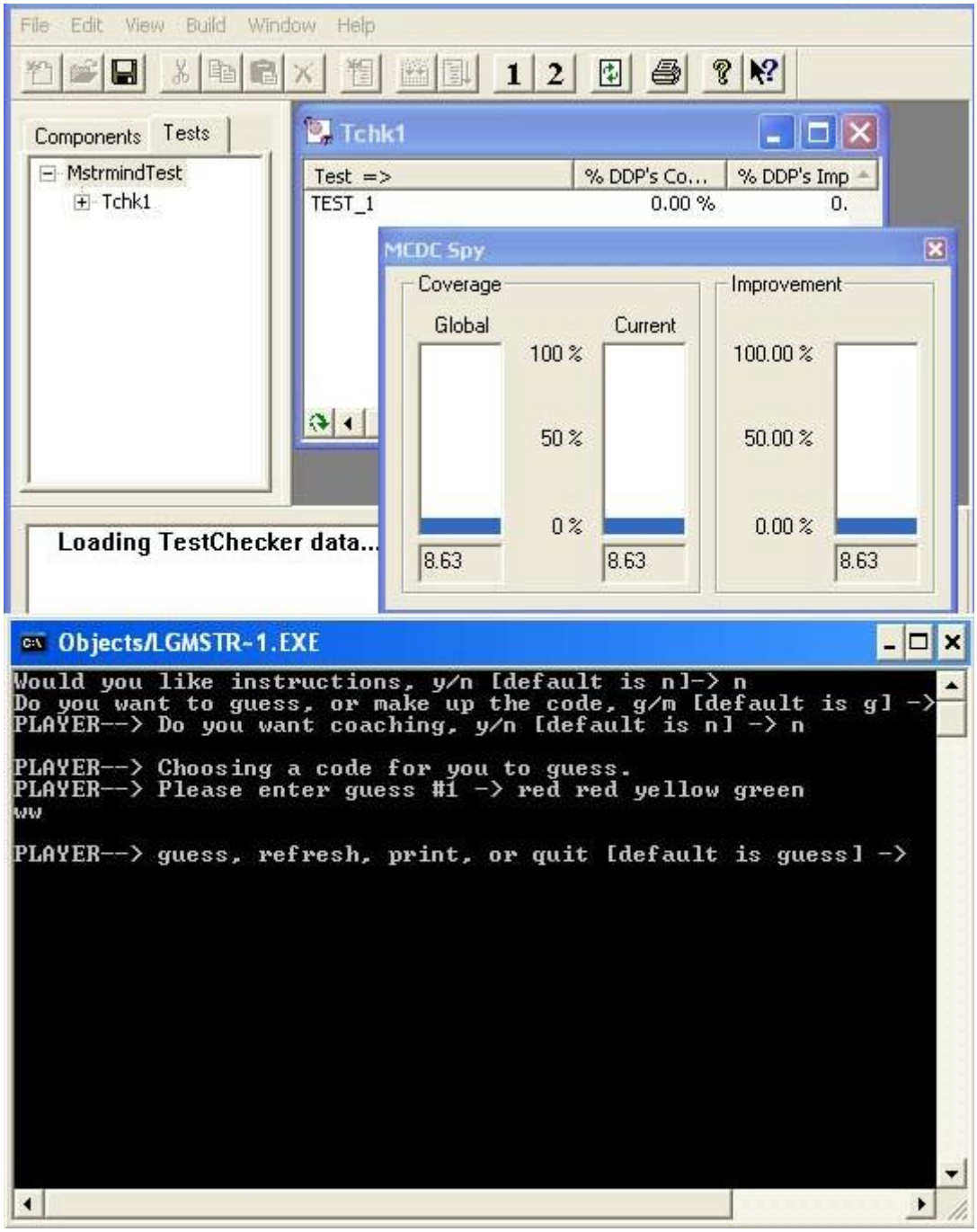
1. Select the Test pane of the Project window by clicking on the **Tests** tab. You are ready to create a test suite.
2. Select the **File-New** command or use the  toolbar icon to create the test suite.
   A test suite window is displayed with the name *Tchk1*.
3. Select the test suite you just created by clicking on it and select the **Edit-New Test** command or click the  icon.
   This action creates a new test in the current test suite. The default name for this new test is *TEST_1*. This is the test you are going to run.

## 6.4.2 Viewing MC/DC While Testing

1. Select the **View-MCDC Spy** command.The **MCDC Spy** window appears.
   It will display the progress of code coverage during testing:
   - the Global bar shows the cumulated coverage for all tests,

- the Current bar shows the coverage for the current test,
- the Improvement bar shows the global coverage improvement secured through the execution of the current test

2. Press the <F5> key or click the  icon.
   The test begins: a window appears in which you start playing mastermind.
   Your screen will look like this:



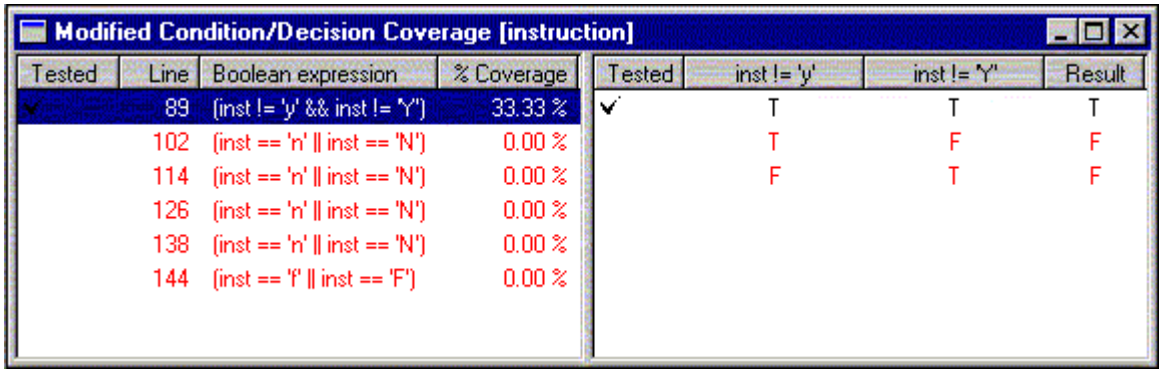
# 6.5 Refining Modified Conditions

Logiscope *TestChecker* can refine further conditions of a decision coverage.

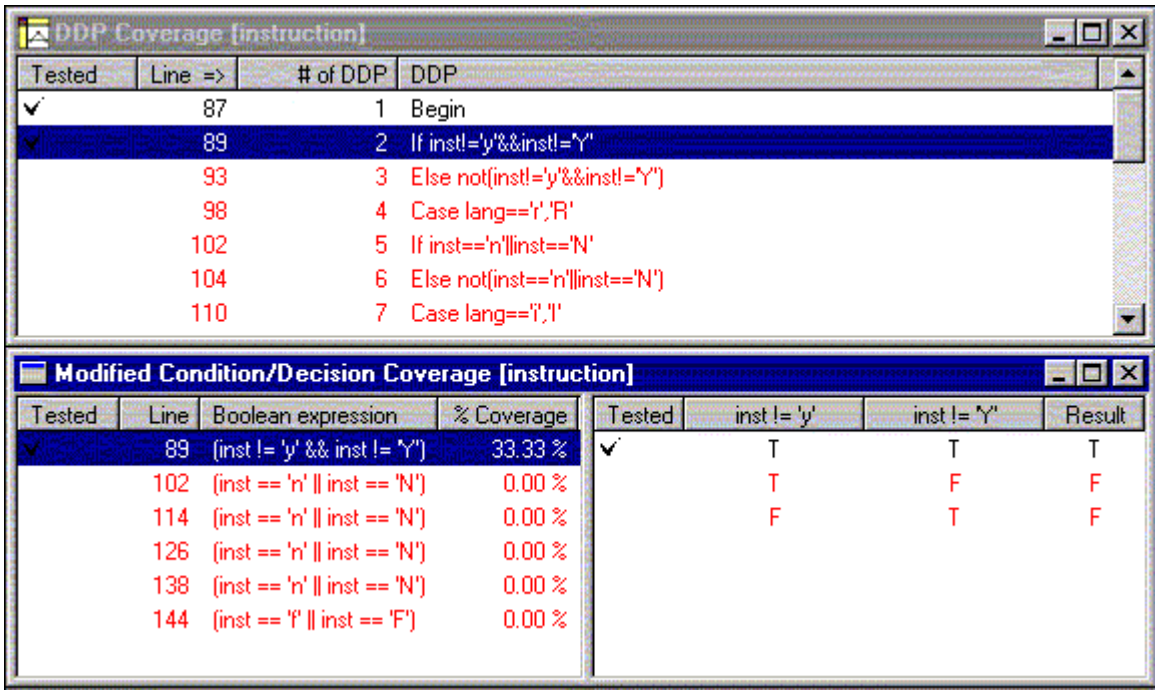
To do this:

1. Select a component or function in a Component window or in the Components pane of the Project window for example machine.c/instruction.
2. Select the **View-MC/DC** command.

The Modified Condition/Decision Coverage window pops up and displays the modified condition/decision coverage (MC/DC). This window lists boolean expressions of the selected component and its MC/DC coverage.




3. Select a boolean expression in the left part of the window. In the right part of the window appears the decomposition of the selected boolean expression.
4. Select the **View – DDP Coverage** command.



5. Select another boolean expression: the DDP Coverage window shows the DDP associated to the selected expression.

6. Select another component: the Modified Condition/Decision Coverage and DDP Coverage windows are updated accordingly.

# Chapter 7

# *Testing on a Target Machine*

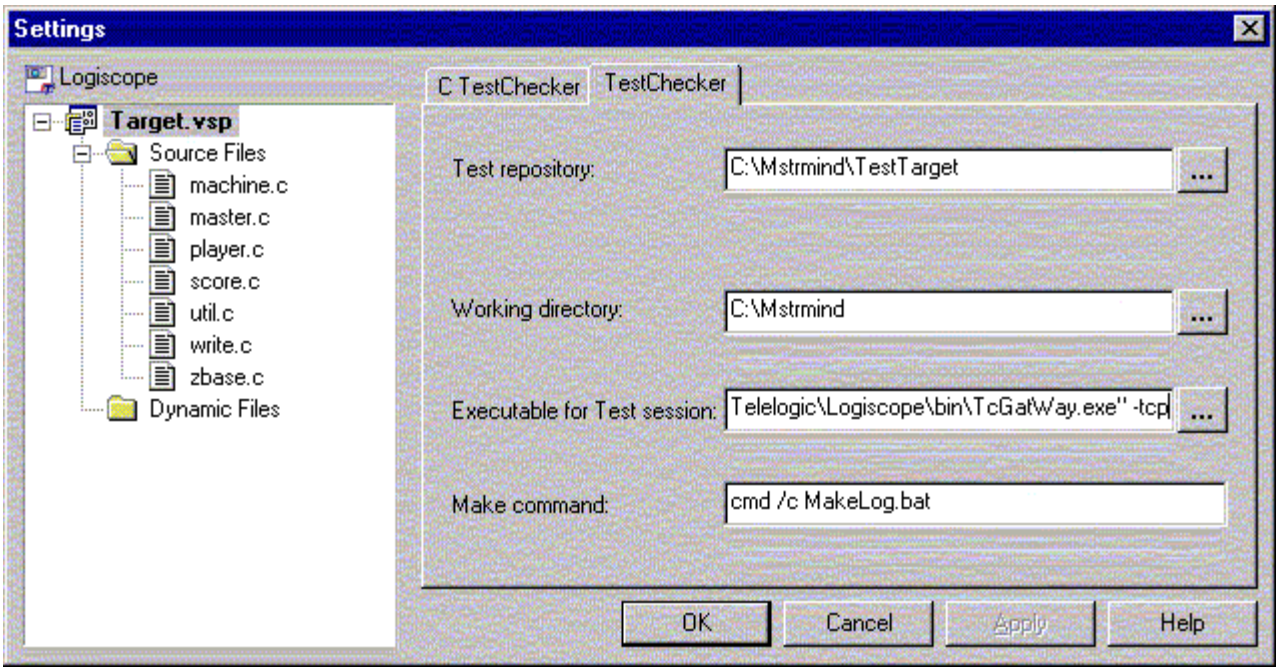
## 7.1 Preliminaries

The Logiscope product supplies as an example a way to test on the following Real Time Operating System (RTOS) targets: VxWorks and PSOS. For other targets, please contact your Kalimetrix for customization.

When testing on a target machine, *TestChecker* runs on the host machine and your instrumented application runs on the target. Both processes communicate by a communication program named *TestChecker* Gateway.

The C project construction used in this case is the same as before but it is now named *Target*. The only difference is the executable command used to start your tests.

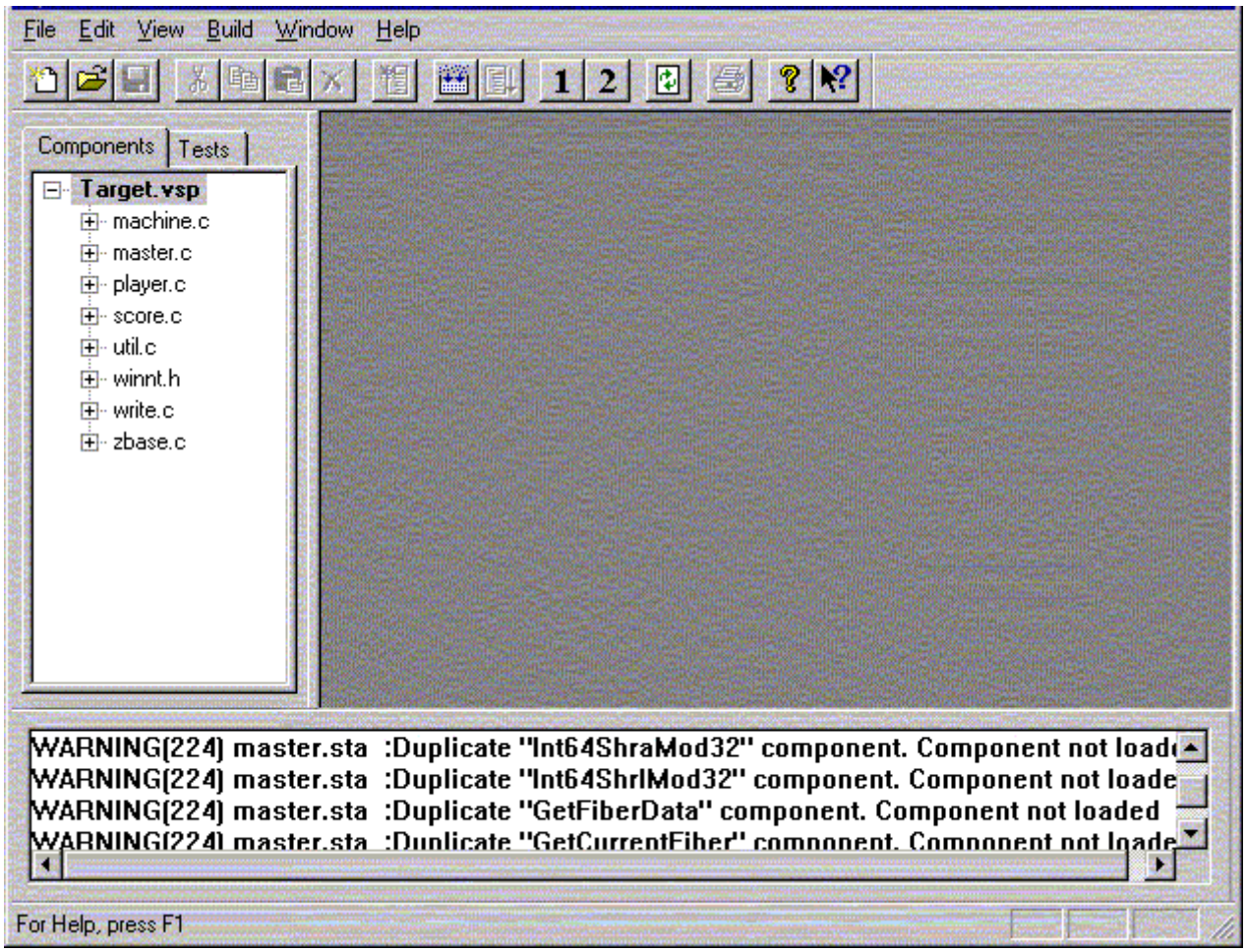
1. In Logiscope *Studio* after creating and building your project, open the Settings window:




*Settings window for target testing*

The communication-handling program used for these tests is **TcGatWay.exe** located in the <*log_install_dir*>**\bin** directory. It takes as parameter **-***port_name*, where port_name is an available communication device (**-tcp** or **-serial COM2** for example) for communicating with the target machine on which the instrumented application will be executed. Please refer to *TestChecker* on-line Help for information on the TcGatWay options.

2. Save your project and load Logiscope *TestChecker.*
3. Select **File-Open** command and open the *Mastermind* Logiscope project.
   At startup, the Logiscope **TestChecker** main window is as follows:




For more information on main window fields, see Chapter 4, Testing on a Host Machine.

Then change the settings to allow the communication with **TcGatWay**:

1. Select **Build-Settings**.
2. On the **Test** tab, check **Use standard communication pipe** option.
3. Click **OK** to keep the changes.

# 7.2 Creating and Running Your First Test

Tests are stored in test suites. You can create as many test suites as you need. This allows you to handle tests according to how your testing process is organized. Before running a test, you must create a test suite. This test suite will contain the test coverage results. Of course, if a test suite has already been created it can be reused.

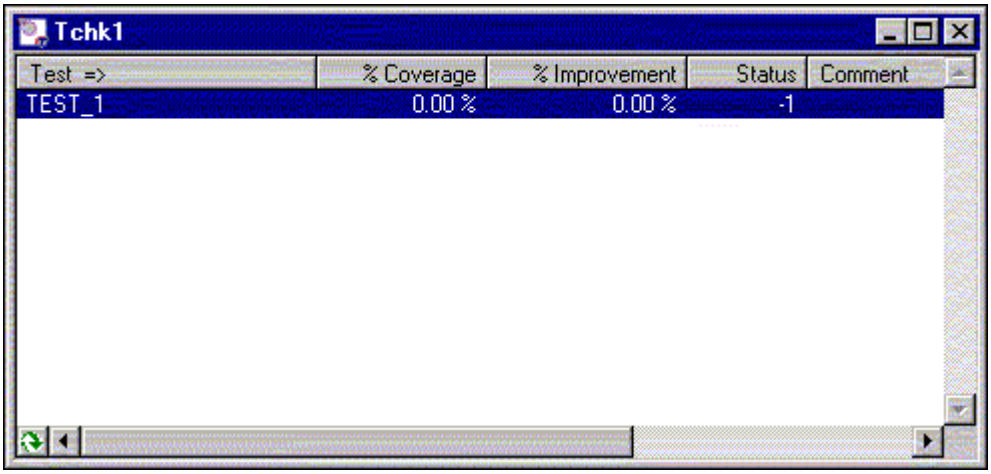
## 7.2.1 Starting the Test

1. Select the Tests pane of the Project Window by clicking on the Test tab. You are ready to create a test suite.

2. Select the **File - New** command, or use the  toolbar icon to create the test suite. A test suite window is displayed with the name *Tchk1*, as shown in the illustration below.

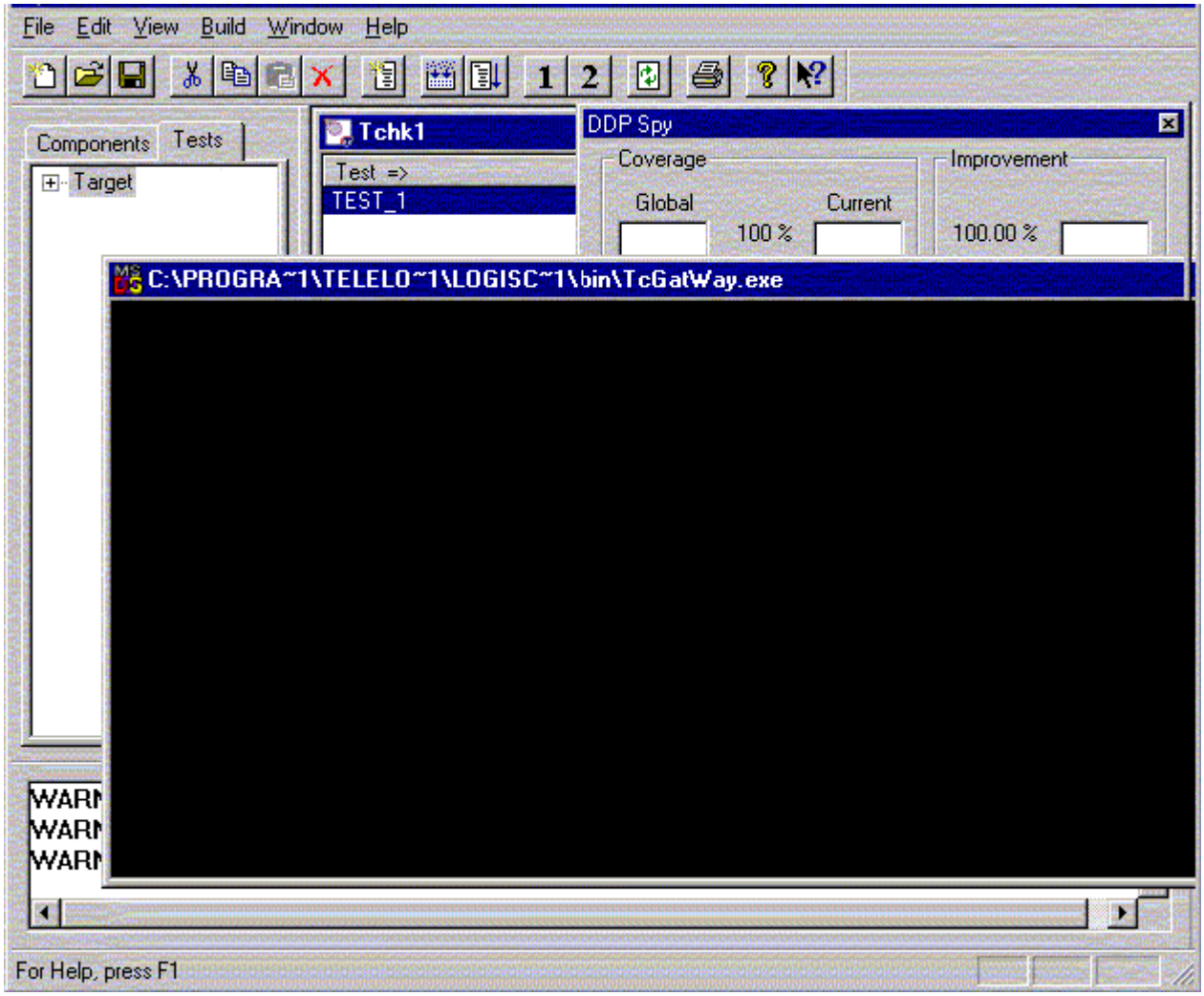

3. Select the test suite you have just created by clicking on it,

4. Select the **Edit - New Test** command or click the  icon.
This action creates a new test in the current test suite. By default, this test is named *TEST_1*.




## 7.2.2 Viewing Coverage Rates While Testing is in Progress

1. Select the **View - DDP Spy** command.
   The DDP Spy appears. This window will display the progress of code coverage during testing:
   - the Global gauge shows the cumulated coverage for all tests,
   - the Current gauge shows the coverage for the current test,
   - the Improvement gauge shows the global coverage improvement secured through the execution of the current test.
2. Click on *TEST_1*. This is the test you are going to run.
3. Press the F5 key or click the  icon.
   **TcGatWay** starts up: an empty MS-DOS window appears. The Logiscope *TestChecker* gateway is waiting for information from the instrumented binary.
4. Run the *Mstrmind.exe* instrumented application on the target machine and follow instructions to play the mastermind game.




As you play, you can see coverage rates increase in the DDP Spy window, but *TEST_1* being the only test that has been executed for the moment, three progress indicators display identical values.

# Chapter 8

# *Creating and Testing Ada Instrumented Code*

## 8.1 Before you start

Along with this chapter, you are provided with a program written in Ada language, an implementation of the *One Armed Bandit* game. The program has been carefully designed for you to use all features of Logiscope *TestChecker*.

Source files of this program are stored in the directory **<InstallationDir>\samples\Tchk\Ada\OneArmedBandit.**

As a precaution to keep original files safe, it is highly recommended that you copy this subdirectory into a working directory of your own: e.g. **C:\OneArmedBandit**. on Windows, **$HOME/OneArmedBandit** on UNIX.

In addition, you will create Logiscope projects and associated repositories: i.e. sets of files containing internal data used by Logiscope. It is recommended to a create a dedicated directory to store these data: e.g. a folder named **LogiscopeProjects**.

## 8.2 Creating an Ada *TestChecker* Project

First, you shall define a Logiscope *TestChecker* project which mainly consists in:

- the list of source files to be first instrumented and then being tested for test coverage analysis,
- applicable source code instrumentation options according to the compilation environment,
- the special traces that will be generated by the Logiscope libraries during the execution of the test cases on the instrumented application.

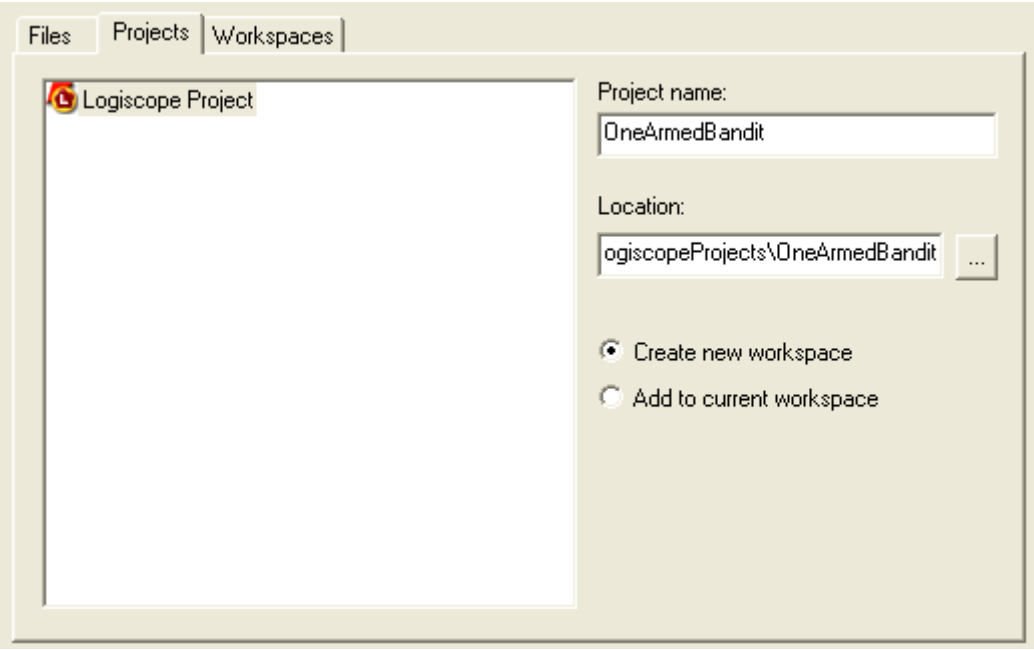
1. Open a Logiscope **Studio** session (see §3.1).
2. In the **File** menu, select the **New**... command or click the  icon.

The **New** Logiscope Projects dialog box appears.

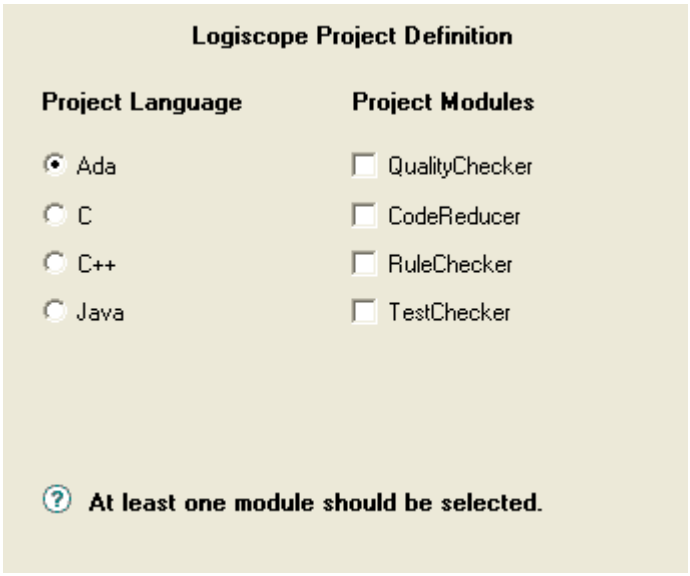
3. In the **Project name:** pane, enter the name for the new Logiscope project to be created. In the context of the guided tour, this simply can be the name of the application under test: e.g. *OneArmedBandit*.
   The information provided in this pane will be then refer as the <ProjectName>.

4. Then select its **Location:** i.e. the directory where the Logiscope project (i.e. a "**.ttp**" file) and the associated Logiscope repository will be created; the Logiscope repository is a folder in which Logiscope internal analysis result files are generated.
   The information provided there will be then refer as the <LogiscopeRepository>.

Note: By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.




5. Click **OK** to access to the **Logiscope Project Definition** first window.



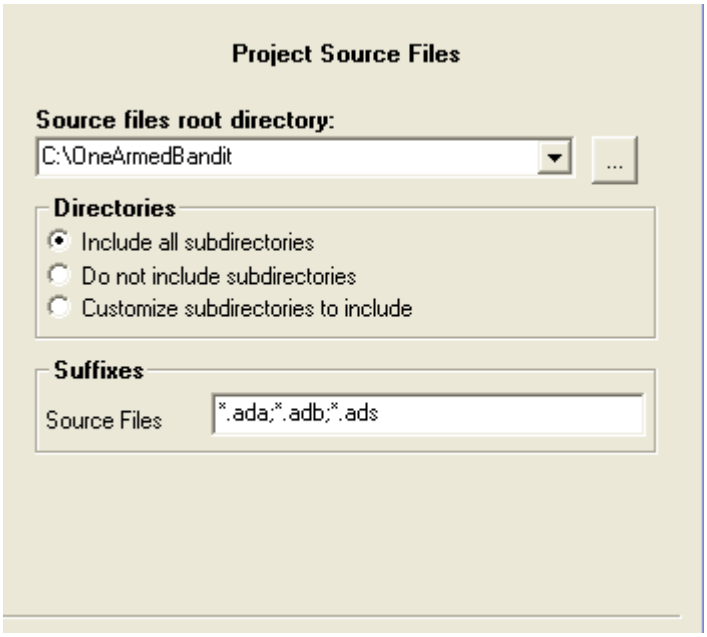

6. Select the **Project Language:** i.e. the programming language in which are written the source code files to be analysed.
   For the OneArmedBandit project, select **Ada**.

   Note: Only one language can be selected. If your application contains source code files written in several languages, you should create several distinct Logiscope projects: one for each language.

7. Select the **Project Modules:** i.e. the verification modules to be activated on the source files of the project .
   For this guided tour, select **TestChecker**.

8. Click the **Next** button to continue the creation.

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.




9. **Source files root directory** shall specify the location directory of the source files to be analyzed.
   Browse to select the directory where the OneArmedBandit sample source files are located: i.e. in the **samples/Tchk/Ada/OneArmedBandit** folder of the Logiscope installation directory or in the directory where the source files have been copied as recommended in section 1.2: e.g. **C:/OneArmedBandit.**

The **Directories** choice allows to select the list of repertories covering the application source files.

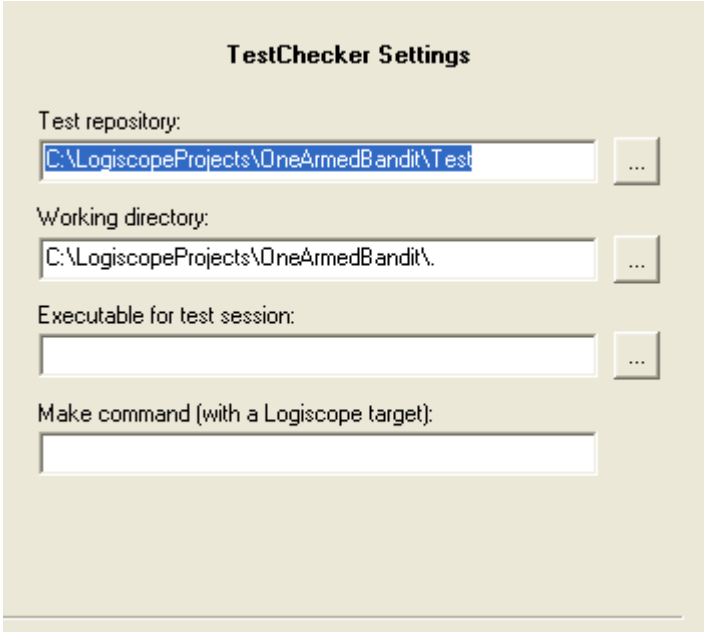
- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

**Suffixes** choices allow to specify applicable source file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

10. In this case, you are going to include two file types only: ***.adb** and ***.ads**. So, you can delete the ***.ada** extension.

11. lick the **Next** button.

The **TestChecker Settings** dialog box is now displayed. It allows to specify some of the key settings of a Logiscope *TestChecker* project.




12. The **Test repository:** is the directory in which instrumented code files are generated . Keep the default location i.e. a Test folder to be created in the Logiscope repository specified in the **New** Logiscope Projects dialog box (see Item 3.).

13.The **Working directory**: is the directory where:

- the make file can be found
- the executable will be generated (unless otherwise specified by the make file) ,
- the traces files will be saved.

14.The **Executable for a test session:** shall specify the instrumented executable. In this context, the executable is not yet generated and will be chosen later.

15. The **Make command** file shall contain the command to build the instrumented executable. Type the following command:

**On UNIX: MakeLogAda.**

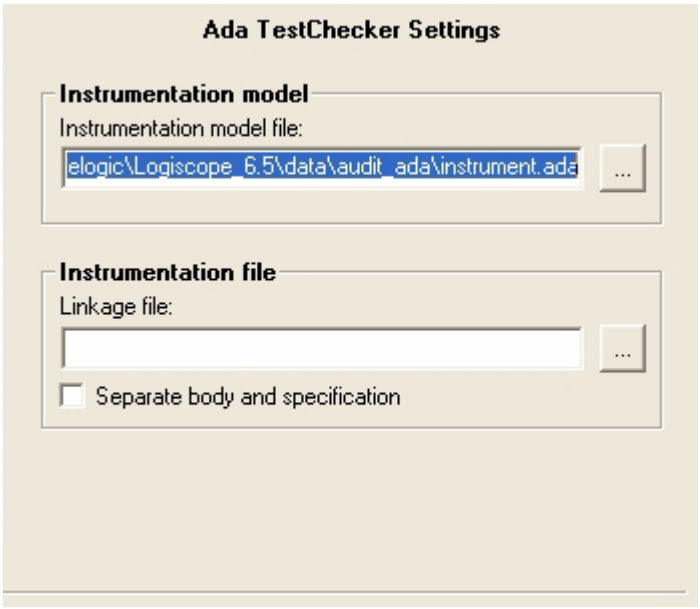
**On Windows**: **cmd /c MakeLog.bat**.

*Note: According to your DOS version, use the equivalent of the "cmd' command.*

In the next section, you will be prompted to edit and modify the make file specified in this pane to adapt it to your compilation environment.

16.Click **Next**.

The following wizard box will allow you to complete the *TestChecker* project specification with some specifics of the Ada language.




The**Instrumentation model** is an Ada file to be used as a template to generate the instrumentation file when building the instrumented executable (see next section).

A default instrumentation model file named **instrument.ada** is provided in the **\data\audit_ada** folder in the Logiscope installation directory. With this default instrumentation model, the test coverage information is produced in a file named **instrum.dyn** in the Working directory, which has a specific format.

In order not to write the test coverage information in such a file, or to modify the format of this file, the default file can be modified and use as a new instrumentation model.

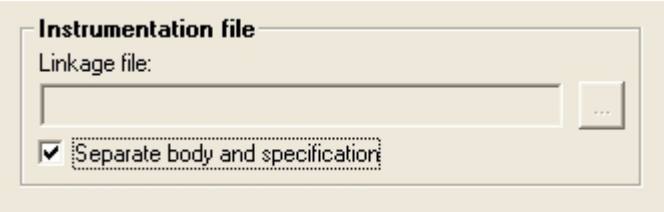
17.In this context, keep the default instrumentation model.

The **Instrumentation file** file is generated from the instrumentation template during the building of the instrumented executable (see next section). Its purpose is to produce the test coverage information during an execution of the instrumented executable. This file must be compiled and linked with the instrumented application.It can consist:

- either in a single file also named **instrument.ada** as the instrumentation template,
- or in two separate files named **audit_instrum.ads** for the specification and **audit_instrum.adb** for the body depending if the **Separate body and specification** option is checked .

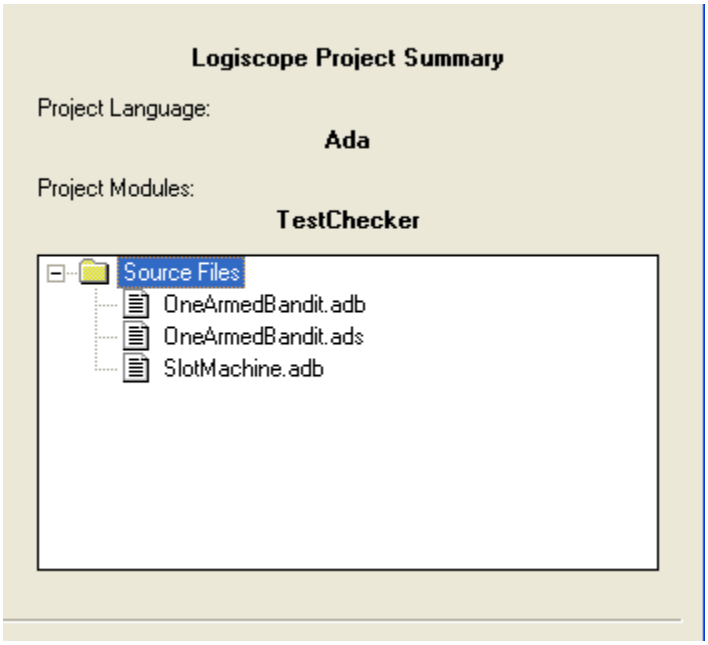
In some context, you can use the Linkage file pane to specify either the name of the file to be generated or even an existing file.

18. In your case, as you have separate sources (**.ads** and **.adb** source files), check the **Separate body and specification** option.




19.Click **Next**.

The last wizard window is displayed. You can check if all files are correct by expanding folders.




20. Click **Finish** to create your first Ada *TestChecker* project.

The **Studio** main window is now updated and contains the workspace view of your project (see next section).

Two files has been created by this process and are of the form: <ProjectName>**.ttp** for project and <ProjectName>**.ttw** for associated workspace. They are both located in the folder specified as the Logiscope Repository.

# 8.3 Inserting Pragmas for the Probes

By default, during the execution, traces are written in a file called *coverage file*. In order not to undermine time behaviour during execution, the coverage file is not written at each trace, but before the end of execution, or at special defined steps.

This may be realized either directly inside the source code (before the instrumentation), or inside the instrumented files. Inside the source code, insert the following statement:

```
pragma Audit_Instrum;
```

This statement can be added at several places before the possible exits or at a place representing a partial execution which is of interest.

This pragma has no effect when the source code is compiled normally. When this code is instrumented before compilation, all occurrences of this pragma are transformed into the following statement:

```
Audit_Instrum.Audit_Stop;
```

which by default performs the writing of the coverage file.

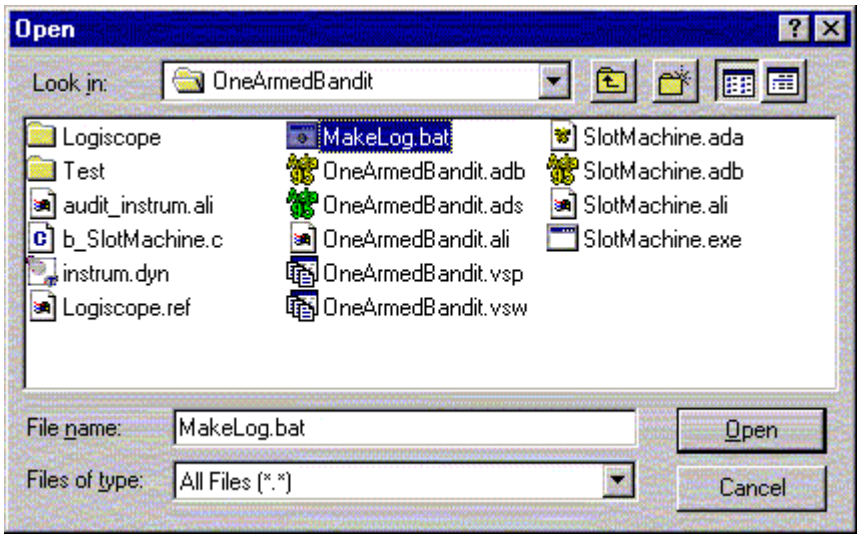
The implementation of the **Audit_Instrum.Audit_Stop** procedure can be changed using the instrumentation template: e.g. to write the test coverage results not into a file, change the format of the traces.

An alternative to inserting the **Audit_Instrum** pragma in a source file is to insert the call to **Audit_Instrum.Audit_Stop** into the corresponding instrumented file or even into a non instrumented file, e.g. a main test program. In the last case, a `with Audit_Instrum;` statement shall also be inserted.

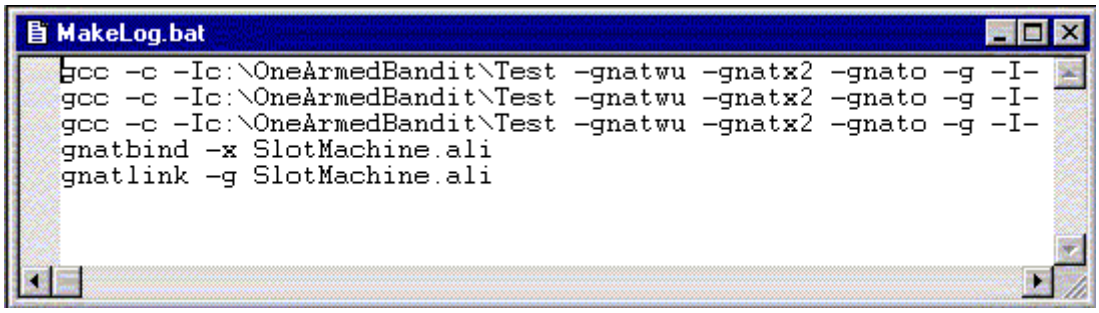
# 8.4 Building the Instrumented Executable

In this part, you will define necessary commands for this program compilation and link. This information is contained in **MakeLog.bat** file for Windows and in **MakeLogAda** for UNIX.

1. Select **File-Open** command. Change the file type in order to show the **All files (*.*)** option.




2. Select **MakeLog.bat** file and click **Open** to edit it.

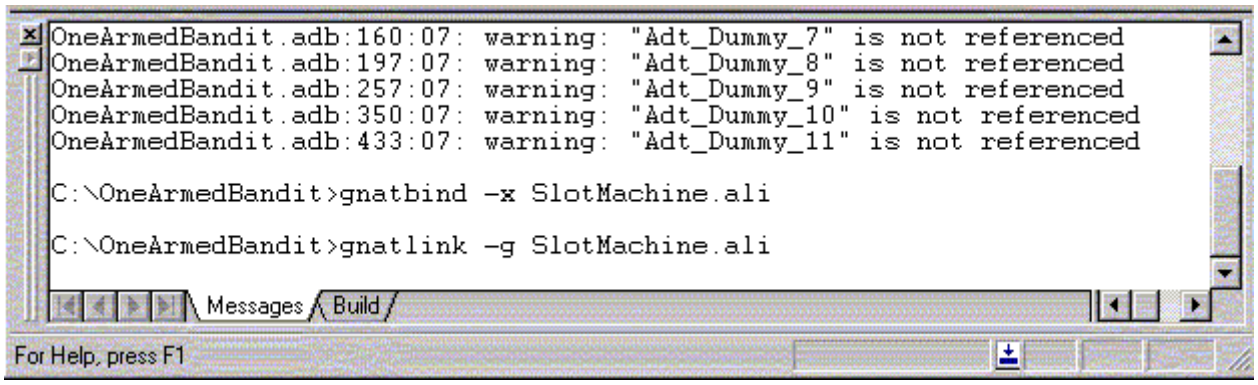



These commands have been written for a Gnat compiler. If you have another compiler, you shall replace them with the right commands. Note that you have to compile the **audit_instrum.adb** file (coming from the instrumentation) with your program.

3. If you have changed some commands, save this file with the **File-Save** command confirming the update or click on the  icon.

Your project is now ready to be built.

4. Select **Project-Build** to launch the generation of the instrumented executable.



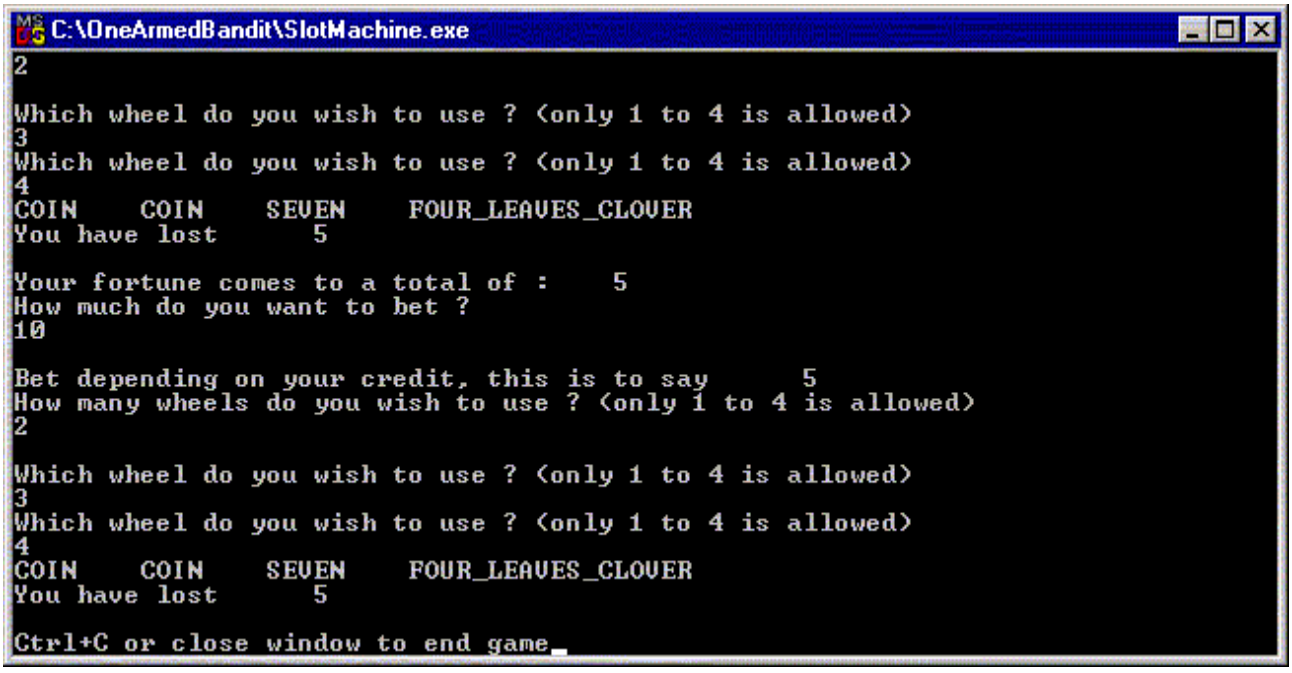

Results files of the binary generation are generated at the same level as the source files.

# 8.5 Testing the Instrumented Executable

In this section, you are going to proceed differently, executing directly the instrumented binary outside the Logiscope tools.

For Logiscope Ada *TestChecker* projects there is only one way to proceed: first execute outside *TestChecker* and then add generated execution trace file (*.dyn*) to the project.

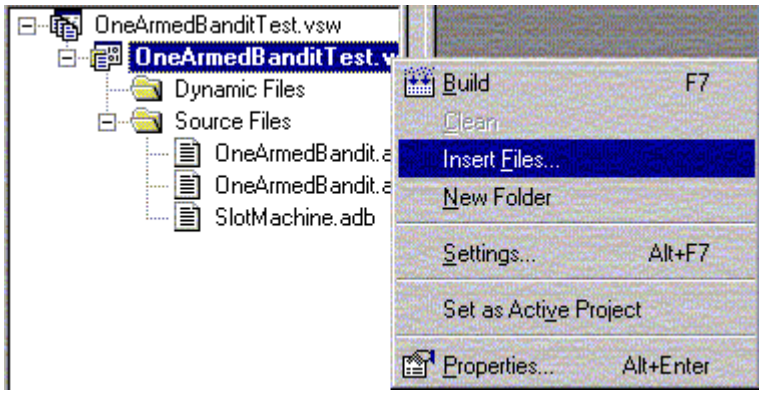
1. Double click on the **SlotMachine.exe** file. A DOS window opens up and you can start playing. Good luck !!



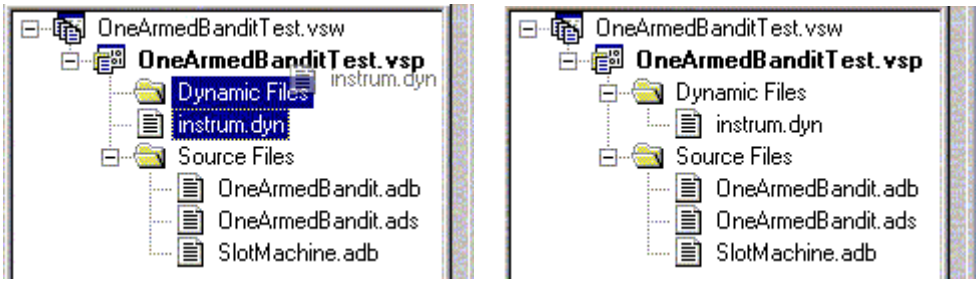
2. To end the test session, run **<Ctrl+C>** command. A test coverage file called **instrum.dyn** is then generated in the Working directory **.**

*Note: If you make others tests, results will be appended to this file.*

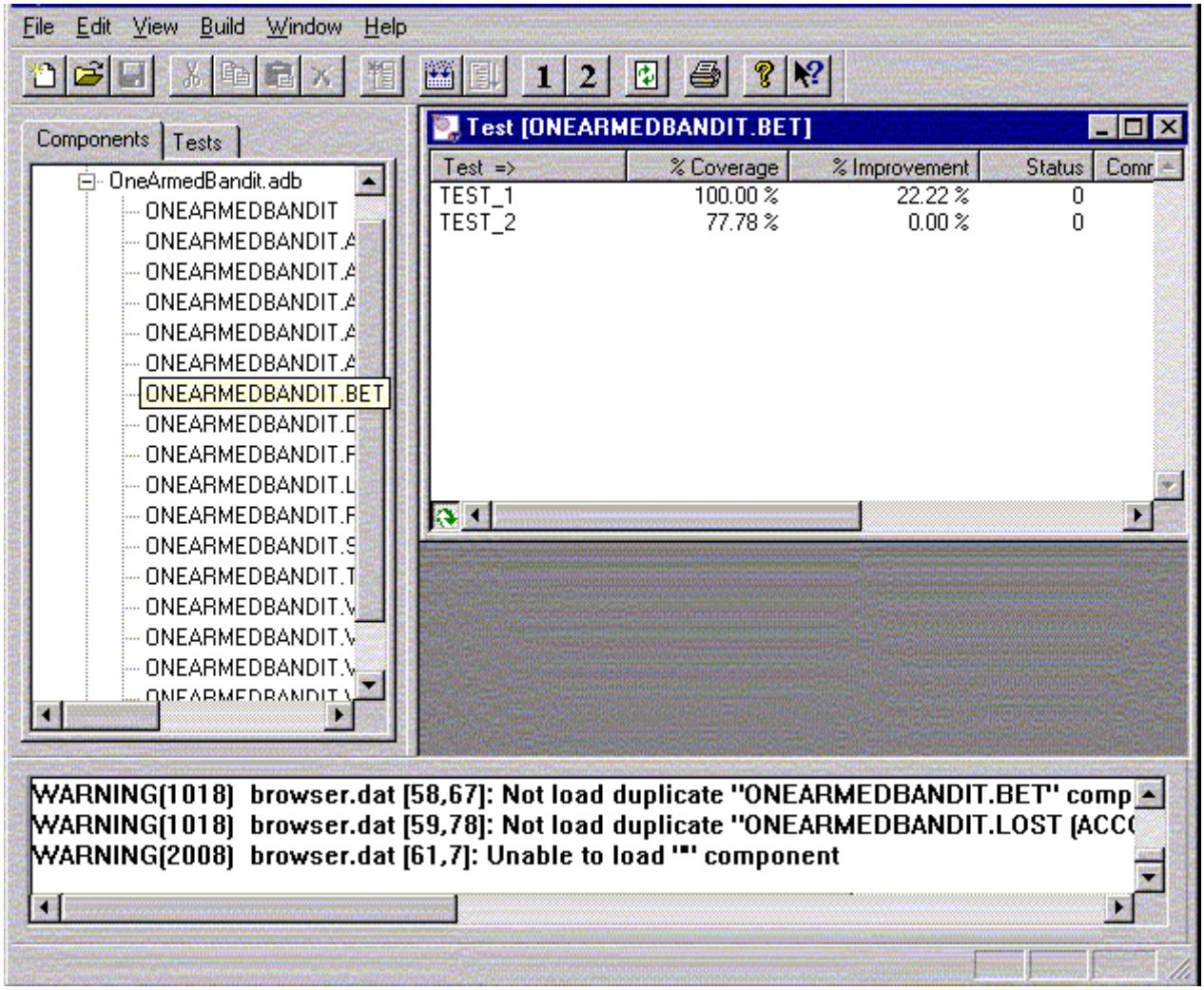
3. This file should be added to your project to take into account tests results. To do this, go back to Logiscope **Studio**, right click on **OneArmedBanditTest.ttp** in the Workspace view.




4. The **Open** window is displayed. Select **instrum.dyn** file and click **OK** to insert it.
5. Select the newly added file and move it up to **Dynamic Files** folder as shown below:

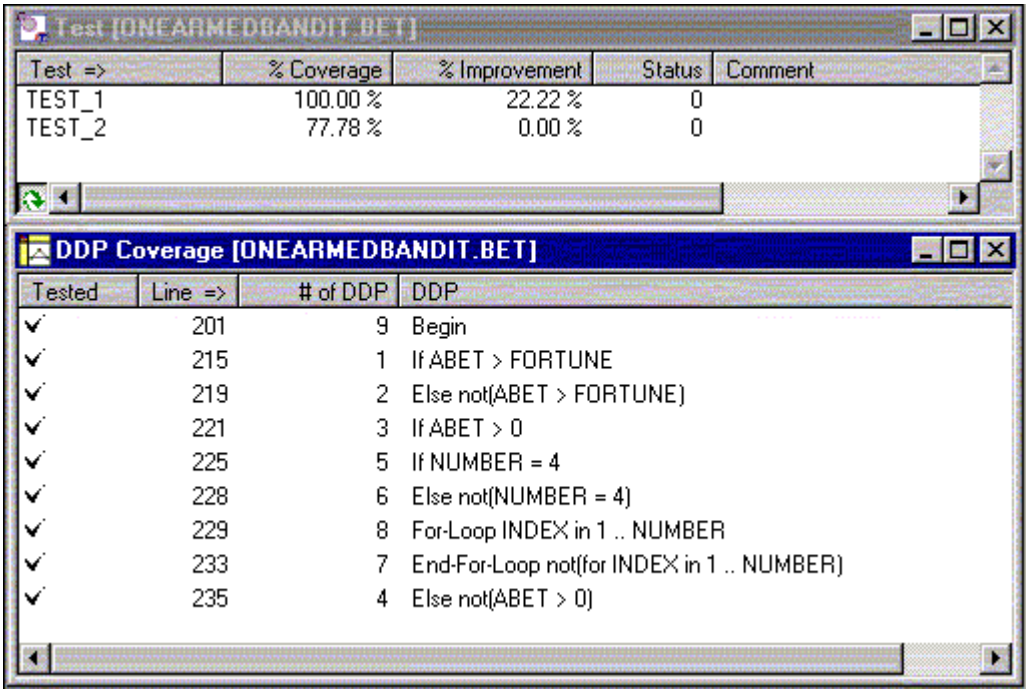



6. Save your project with **File-Save Workspace** command.
7. Open the Logiscope *TestChecker* as you know how.
8. Double click on the **ONEARMEDBANDIT.BET** component:



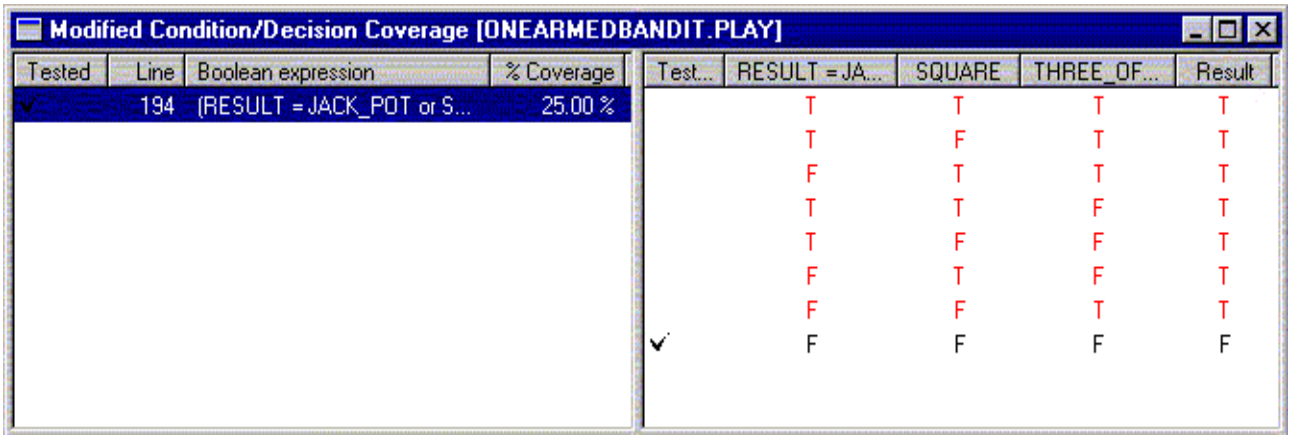

*TestChecker main view*

9. Select **View-DDP Coverage**.




*ONEARMEDBANDIT.BET DDP Coverage*

10. Select **View-MCDC** to visualize modified conditions:




*ONEARMEDBANDIT.BET MC/DC*

As shown in Chapter 5, you can use the *Studio* and the *Viewer* to see tests coverage results.

# 8.6 Customizing the Instrumentation Primitives

The instrumentation model must contain several Ada subprograms which will be called by the instrumented executable. These subprograms (called instrumentation primitives) are aimed at detecting the passing through the different branches of the control graph, the subprogram calls, and the MCDCs, and at producing the test coverage information. The calls to these primitives are automatically inserted in the instrumented application during the instrumentation.

To write a particular instrumentation template, the following primitives shall be implemented:

- `procedure Audit_Init_Application (Appli : String ; Data : String ; Max_Func : Natural);`

  This procedure must be called at the beginning of the application. Its parameters are the following:

  Appli: name of the application,

  Data: directory containing the results of the application analysis (control graph, call graph, ...),

  Max_Func: maximum number attributed to the subprograms in the application call graph (>=0).

- `procedure Audit_Init_Function (Func_Id : Positive ; Func_Name: String ; Func_Date : String ; Nb_Bran : Natural ; Nb_Calls : Natural ; Nb_Mcdcs : Natural);`

  This procedure must be called at the beginning of the application for each instrumented subprogram and for each external subprogram called in the application. Its parameters are the following:

  Func_Id: identifier of the subprogram; it must be unique in the instrumentation file (>0),

  Func_Name: full name of the subprogram (prefixed with the possible package(s) and containing the name of its possible parameters),

  Func_Date: time of the last analysis of the subprogram,

  Nb_Bran: number of branches in the control graph of the function (>=0),

  Nb_Calls: number of (distinct) subprogram calls inside the subprogram (>=0),

  Nb_Mcdcs: number of MCDCs inside the subprogram (>=0).

- `function Audit_Start_Func (Func_Id : Positive ; Vect_Size : Natural) return Boolean;`

This function is called at the beginning of each instrumented subprogram. Its parameters are the following:

Func_Id: identifier of the subprogram (>0),

Vect_Size: maximum number of single conditions in the MCDCs of the subprogram (>=0).

Its return value is not significant.

- `procedure Audit_Set_Branch (Func_Id : Positive ; Bran_Id : Natural);`

This procedure is called at the beginning of each branch of the control graph. Its parameters are the following:

Func_Id: identifier of the current subprogram (>0),

Bran_Id: number of the corresponding branch (>=0); 0 corresponds to the main branch, before any control structure; numbers greater than 0 are the same as in the control graph files.

- ```
  procedure Audit_Set_Call (Func_Id: Positive ;
  Called_Func_Id : Positive);
  function Audit_Set_Call (Func_Id: Positive ;
  Called_Func_Id : Positive) return Boolean;
  ```

This procedure or this function is called just before a subprogram call. Its parameters are the following:

Func_Id: identifier of the current subprogram (the calling one) (>0).

Called_Func_Id: identifier of the called subprogram (>0).

The return value of the function is not significant.

Because of certain restrictions of the Ada language, it is impossible to call this primitive at the exact location of the call. Therefore, its execution does not fully prove that the corresponding call has taken place.

- `function Audit_Set_Mcdc (Func_Id : Positive ; Mcdc_Id : Positive ; Nb_Cond : Positive ; Exp : Boolean) return Boolean;`

This function is called for each MCDC. Its parameters are the following:

Func_Id: identifier of the current subprogram (>0),

Mcdc_Id: identifier of the MCDC (>0).

Nb_Cond: number of single conditions in the MCDC (>0).

Exp: boolean result of the MCDC (which is returned by this function).
  ```

- `function Audit_Set_Sgl_Cond (Func_Id : Positive ; Mcdc_Id : Positive ; Index : Positive ; Exp : Boolean) return Boolean;`

This function is called for each single condition of a MCDC. Its parameters are the following:

Func_Id: identifier of the current subprogram (>0),

Mcdc_Id: identifier of the current MCDC (>0).

Index: number of the condition in the MCDC (>0).

Exp: boolean value of the single condition (which is returned by this function).

- `function Audit_Set_Bool_Exp (Func_Id : Positive ; Bran_T, Bran_F : Natural ; Exp : Boolean) return Boolean;`

This function is called for each boolean expression in an `if` or `exit when` structure. Its parameters are the following:

Func_Id: identifier of the current subprogram (>0),

Bran_T: number of the control graph branch corresponding to the case where the expression is true (>0).

Bran_F: number of the control graph branch corresponding to the case where the expression is false (>0).

Exp: result of the boolean expression (which is returned by this function).

- `procedure Audit_Stop;`

This procedure is aimed at producing the test coverage information.

In order to automatically include this procedure in the instrumented code, just add the pragma **Audit_Instrum** in the source code. Each such pragma will be replaced by a call to **Audit_Stop** during the instrumentation.

- `procedure Audit_Start;`

This procedure is called at the beginning of the application.

During the instrumentation, a call to **Audit_Init_Application** will be automatically inserted at its beginning, and for each instrumented subprogram, a call to **Audit_Init_Function** will be inserted at its end.

# Chapter 9

# *Building and Testing Java Instrumented Code*

## 9.1 Before you start

Along with this chapter, you are provided with a program written in Java language, an implementation of the *Mine Finder* game. The program has been carefully designed for you to use all features of Logiscope *TestChecker*.

Source files of this program are stored in the directory: **<InstallationDir>\samples\Tchk\Java\JMineFinder**.

As a precaution to keep original files safe, it is highly recommended that you copy this subdirectory into a working directory of your own: e.g. **C:\JMineFinder**. on Windows, **$HOME/JMineFinder** on UNIX.

In addition, you will create Logiscope projects and associated repositories: i.e. sets of files containing internal data used by Logiscope. It is recommended to a create a dedicated directory to store these data: e.g. a folder named **LogiscopeProjects**.

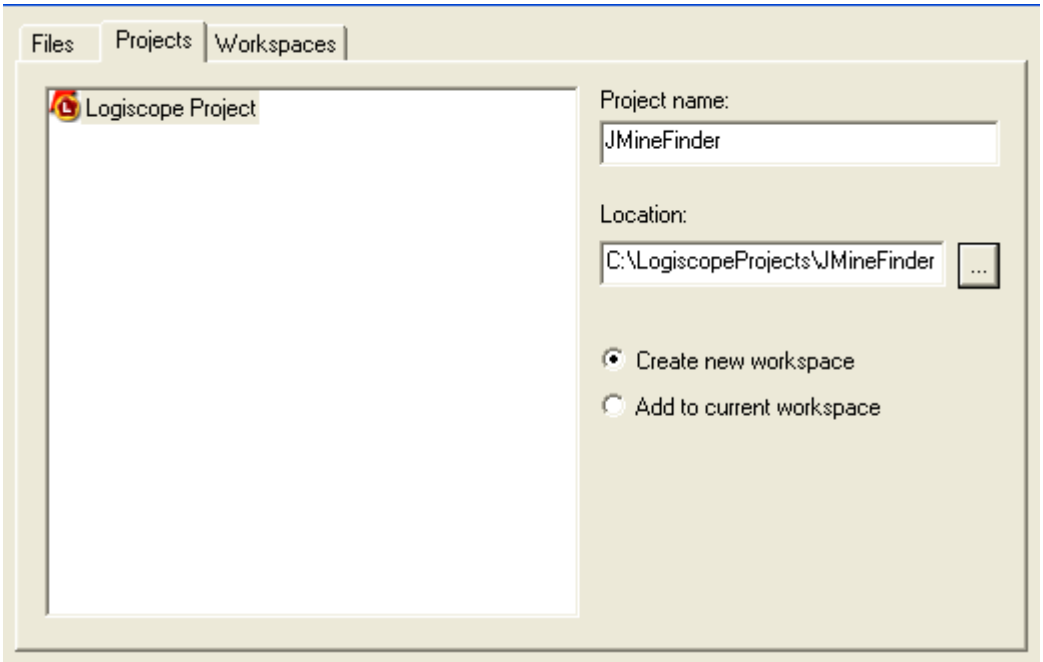
## 9.2 Creating a Java TestChecker Project

First, you shall define a Logiscope *TestChecker* project which mainly consists in:

- the list of source files to be first instrumented and then being tested for test coverage analysis,
- applicable source code instrumentation options according to the compilation environment,
- the special traces that will be generated by the Logiscope libraries during the execution of the test cases on the instrumented application.

1. Open a Logiscope **Studio** session (see section 4.1).
2. In the **File** menu, select the **New**... command or click the  icon.

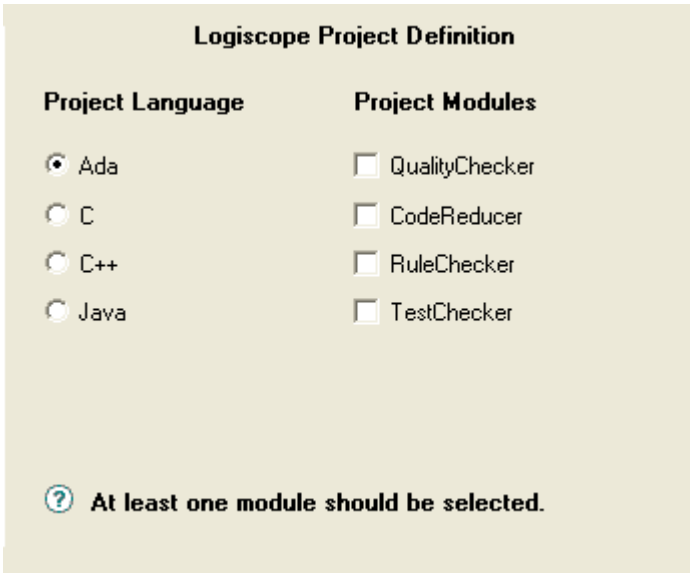
3. In the **Project name:** pane, enter the name for the new Logiscope project to be created. In the context of the guided tour, this simply can be the name of the application under test: e.g. JMineFinder.
   The information provided in this pane will be then refer as the <ProjectName>.




4. Then select its **Location:** i.e. the directory where the Logiscope project (i.e. a "**.ttp**" file) and the associated Logiscope repository will be created; the Logiscope repository is a folder in which Logiscope internal analysis result files are generated.
   The information provided there will be then refer as the <LogiscopeRepository>.

Note: By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.

5. Click **OK** to access to the **Logiscope Project Definition** first window.



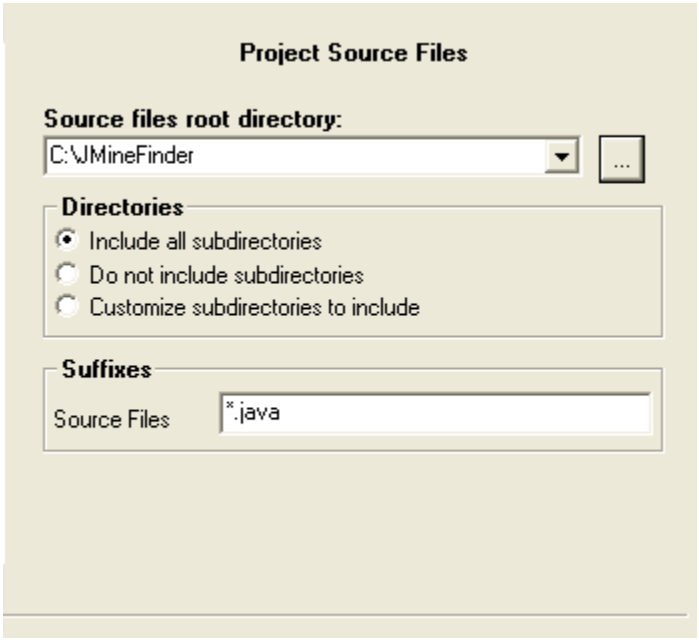

6. Select the **Project Language:** i.e. the programming language in which are written the source code files to be analysed.
   For the JMineFinder project, select **Java**.

   Note: Only one language can be selected. If your application contains source code files written in several languages, you should create several distinct Logiscope projects: one for each language.

7. Select the **Project Modules:** i.e. the verification modules to be activated on the source files of the project .
   For this guided tour, select **TestChecker**.

8. Click the **Next** button to continue the creation.

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.




9. **Source files root directory** shall specify the location directory of the source files to be analyzed.
   Browse to select the directory where the OneArmedBandit sample source files are located: i.e. in the **samples/Tchk/Java/JMineFinder** folder of the Logiscope installation directory or in the directory where the source files have been copied as recommended: e.g. **C:/JMineFinder.**

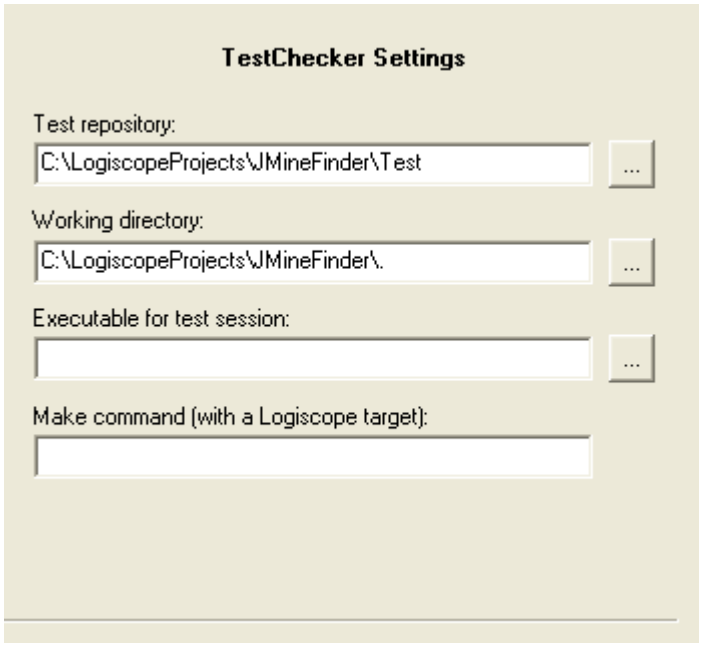
The **Directories** choice allows to select the list of repertories covering the application source files.

- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

**Suffixes** choices allow to specify applicable source file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

10. Click the **Next** button.

The **TestChecker Settings** dialog box is now displayed. It allows to specify some of the key settings of a Logiscope *TestChecker* project.




11. The **Test repository:** is the directory in which traces files generated when executing the instrumented executable will be saved.
    Keep the default location i.e. a Test folder to be created in the Logiscope repository specified in the **New** Logiscope Projects dialog box (see item 4.).

12. The **Working directory**: is the directory where the make file can be found and where the executable will be generated (unless otherwise specified by the make file).

13. The **Executable for a test session:** shall specify the instrumented executable.
    In this context, the executable is not yet generated and will be chosen later.

14. The **Make command** shall contains the command to build the instrumented executable. Type:

    **On UNIX: MakeLogJava.**

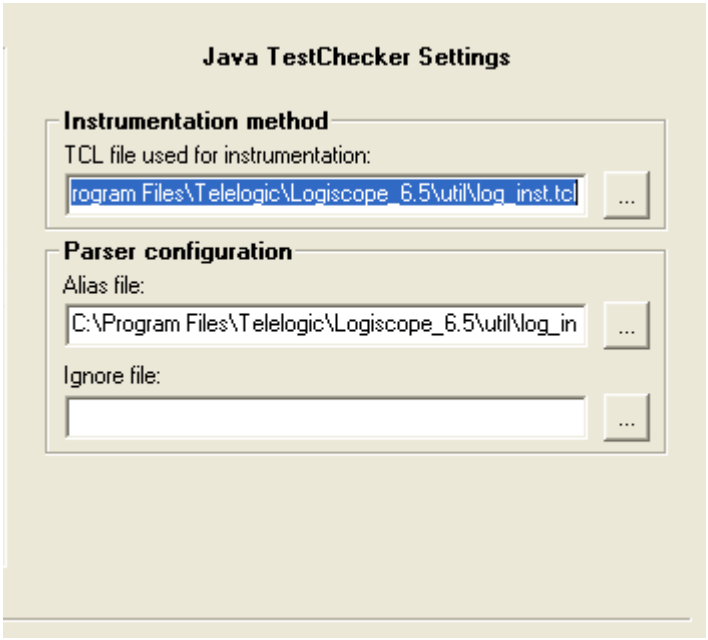
    **On Windows: cmd /c MakeLog.bat**.

    *Note: According to your DOS version, use the equivalent of the "cmd' command.*

    In the next section, you will be prompted to edit and modify the Make command specified in this pane to adapt it to your compilation environment.

15. Click **Next**.

The following wizard box will allow you to complete the project specification with some specifics of the Java language.



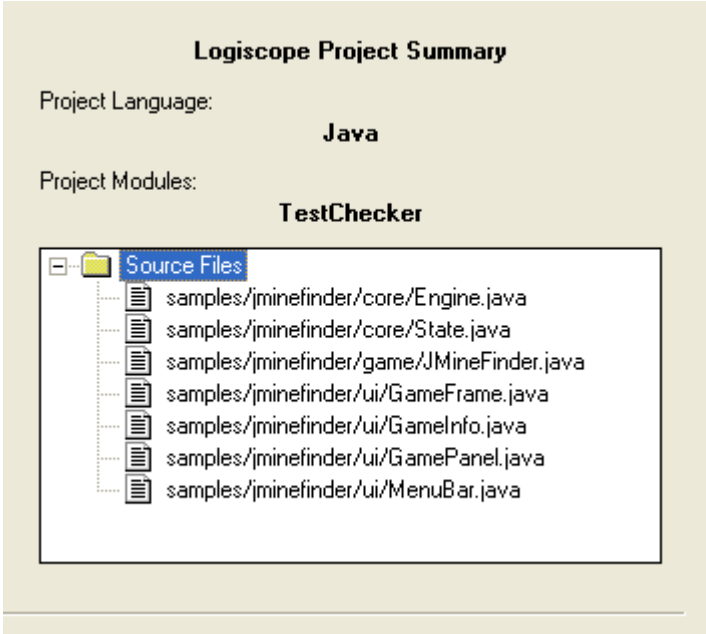

16.In this dialog box, you can choose the Tcl instrumentation file between two files:

- **log_inst.tcl** using the **VlgInstrument.java** instrumentation library for Java applet and,
- **log_inst_jvt.tcl** using the**VlgTrace.java** instrumentation library for Java application.

Note: The**VlgInstrument.java** file contains the socket declaration and related instrumentation functions. If your system does not support sockets, you can use the **VlgTrace.java** file saving results in trace files ( with the extension "**.trc**" files) which can be loaded in **TestChecker**).
The socket declaration must contain the real name of the target machine and the chosen port number.
For more details, see the readme.txt file in the **<InstallationDir>\instr\jv** folder.

17.Click **Next**.

The last wizard window is displayed. You can check if all files are correct by expanding folders.




18. Click **Finish** to create your first Java *TestChecker* project.

The **Studio** main window is now updated and contains the workspace view of your project (see next section).

Two files has been created by this process and are of the form: <ProjectName>**.ttp** for project and <ProjectName>**.ttw** for associated workspace. They are both located in the folder specified as the Logiscope Repository.

# 9.3 Building the Instrumented Executable

In this part, you will define necessary commands for this program compilation and link. This information is contained in **MakeLog.bat** file for Windows and in **MakeLogJava** for UNIX. This file has been specified in the **Make command** pane when creating the project.

1. Edit the **MakeLog.bat** file located in the **JMineFinder** folder.

The content of the file starts with the following lines:

```
@echo off
set JDK=C:/Program Files/Java/jdk1.5.0_10/bin
set LOGISCOPE_INSTALL=../../../..
...
```

2. If necessary, modify the content of the Make file to adapt it to your environment: e.g.
   - `JDK`: the path to access to the Java compiler,
   - `LOGISCOPE_INSTALL` the directory where Logiscope is installed.
3. And save it.

Note that you have to compile the **VlgInstrument.java** file with your program.

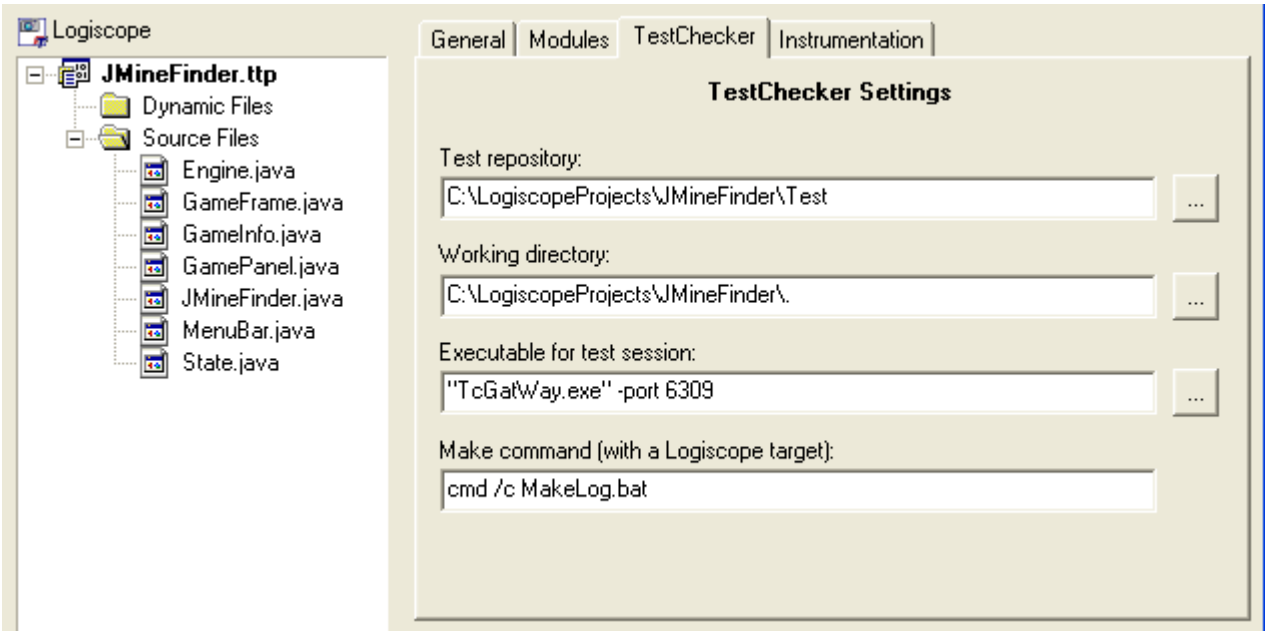
Your project is now ready to be built.

4. Select **Project-Build** to launch the construction of the executable.

```
Building TestChecker Data for Project JMineFinder.ttp...
Instrumenting: C:\jminefinder\core\Engine.java...
Instrumenting: C:\jminefinder\core\State.java...
...
*** Instrumented Java classes compilation ***
...
C:\JMineFinder>"C:/Program Files/Java/jdk1.5.0_10/bin/javac" /jminefinder/core/
Engine.java
...
C:\Program Files\...\Logiscope_6.6\samples\Tchk\Java\JMineFinder>exit 0
Build finished.
```

5. Open **Project-Settings** window to specify the execution command.

6. In the **Executable for test session** field, enter the **TcGatWay.exe** path located in the \bin install directory) between “ “ and specify **-port 6309** as parameter (or another value if you have changed the port number).



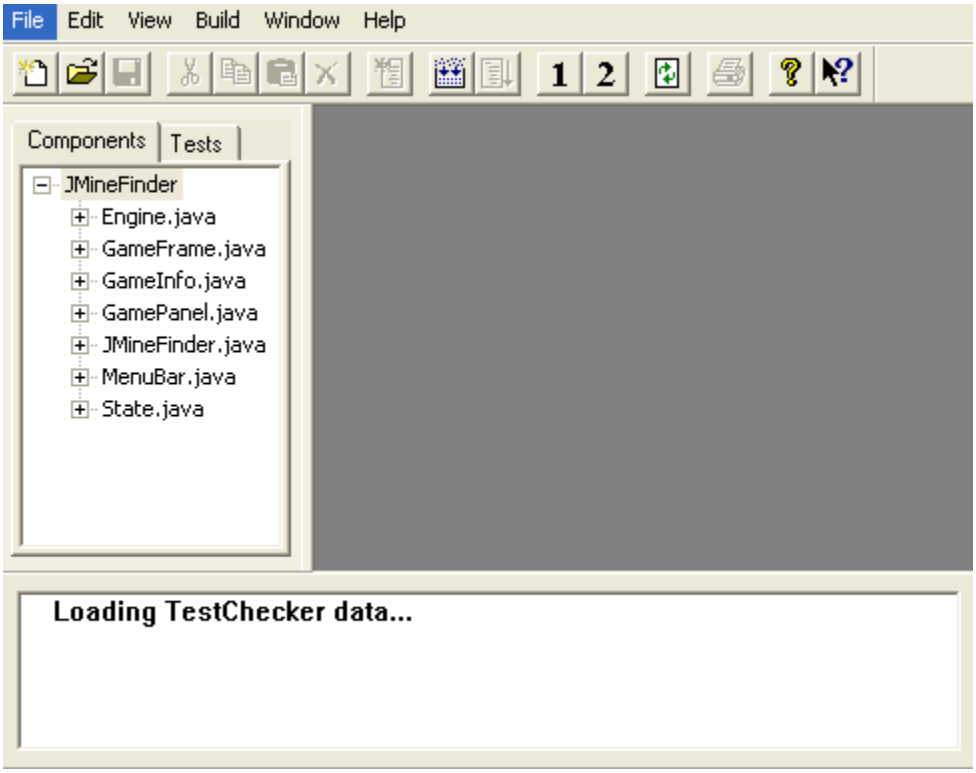

7. Save your new *TestChecker* project settings.

You are now ready to start testing the newly created instrumentation executable.You will use Logiscope **TestChecker** to manage the test execution.

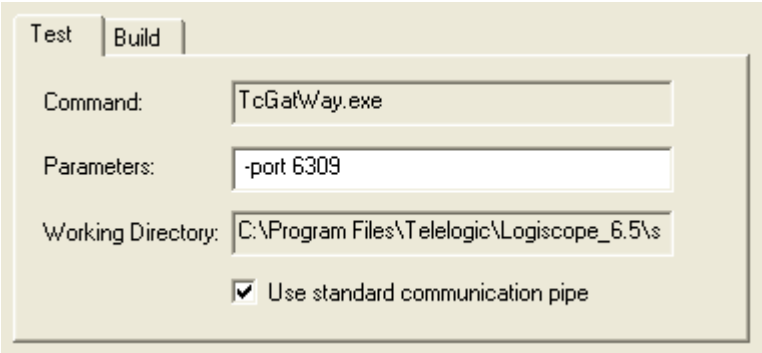
# 9.4 Testing the Instrumented Executable

## 9.4.1 Settings

1. In Logiscope **Studio** main window, open the **Project** menu and select the **Start TestChecker** option or click the  icon in the Logiscope toolbar.




2. Select **Build-Settings**.
3. Check **Use standard communication pipe** option.



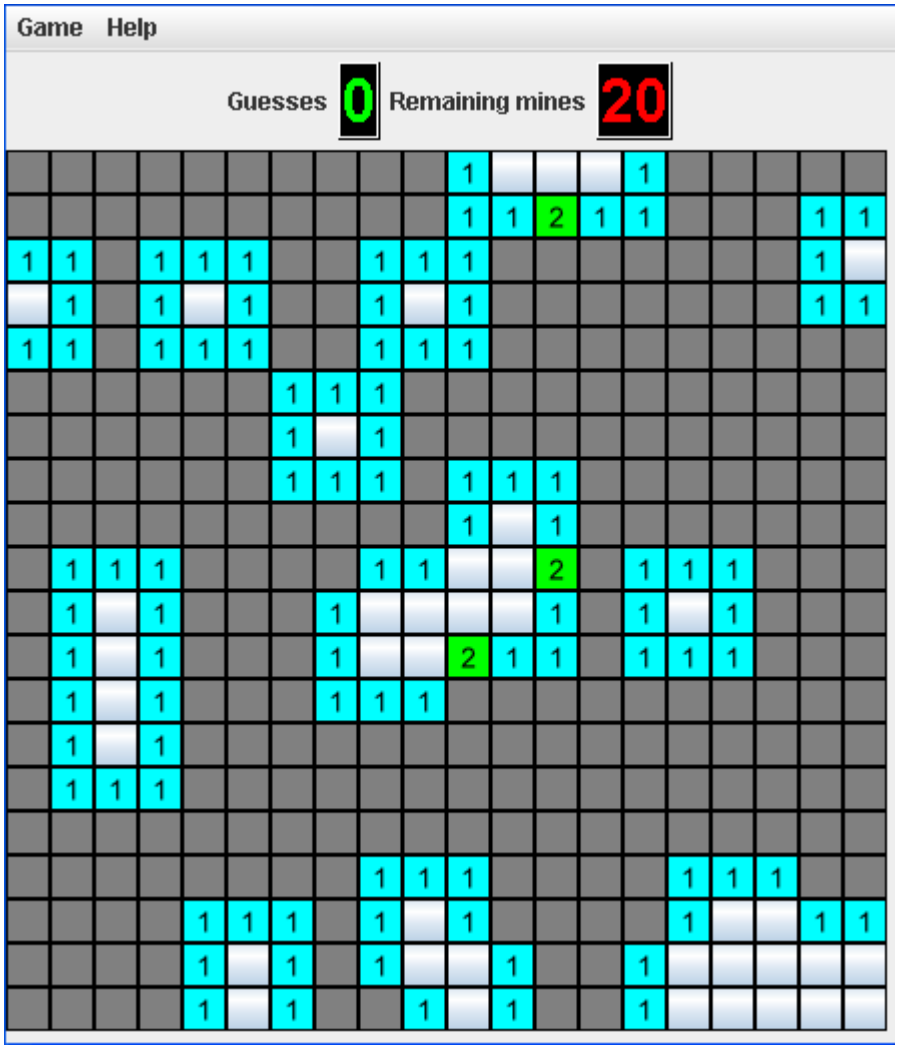

4. Click **OK** to keep the changes.

## 9.4.2 New Test

1. Click **File-New** command to create a default test suite called Tchk1 in the Test panel.
2. Select **Edit-New Test** to create a test called TEST_1 under Tchk1.
3. Select **Build-Go** to start a test session.
   The TcGatWay window pops up.
   Do not close it; it will close automatically when the applet exits.
4. Go to the **JMineFinder** directory and launch the game: i.e.

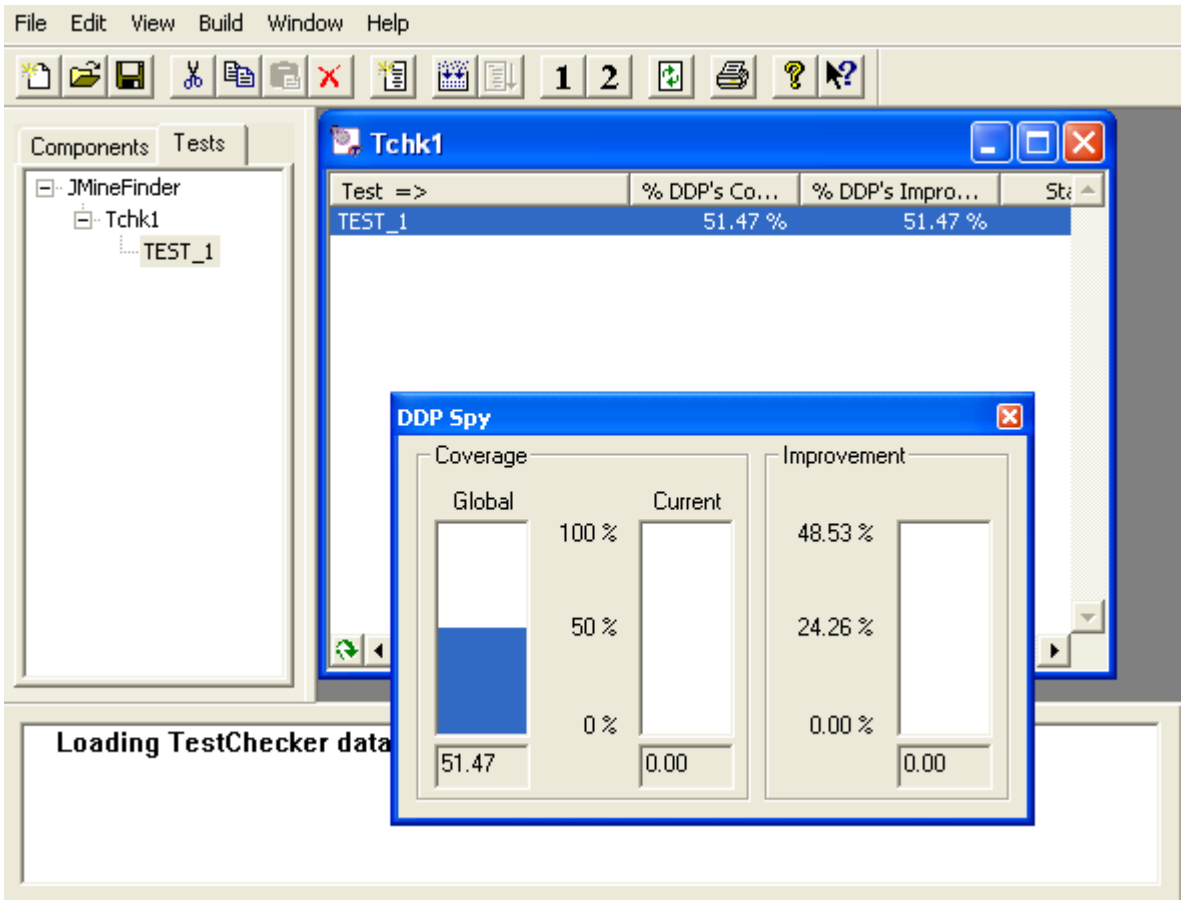
- **on Windows**: launch the script **run.bat**;
- **on UNIX:** launch the script **run.sh** .

5. Make sure you have the **DDP Spy** window opened in *TestChecker* to see the increase in code coverage.
6. Let's play.




7. To stop testing, just close the JMineFinder window.

Test coverage results are ready to be consulted.



# Chapter 10

# *Command Line Mode*

## 10.1 Logiscope create

Logiscope projects: i.e. "**.ttp"** file are usually built using Logiscope **Studio** as described in previous chapter.

The logiscope **create** tool builds Logiscope projects from a standalone command line or within makefiles (replacing the compiler command) .

### 10.1.1 Command Line Mode

When started from a standard command line, The **create** tool creates a new project file with the information provided on the command line.

For a complete description of the command line options, please refer to the Command Line Options paragraph.

When used in this mode, there are two different ways for providing the files to be included into the project:

#### Automatic search

This is the default mode where the tool automatically searches the files in the directories. Key options having effect on this modes are:

**`-root <root_dir>`** : the root directory where the tool will start the search for source files. This option is not mandatory, and if omitted the default is to start the search in the current directory.

**`-recurse`** : if present indicates to the tool that the search for source files has to be recursive, meaning that the tool will also search the subdirectories of the root directory.

#### File list

In this mode, the tool will look for the –**`list`** option which has to be followed by a file name. This provided file contains a list of files to be included into the project. The file shall contain one filename per line.

**Example**: Assuming a file named `filelist.lst` containing the 3 following lines:

```
/users/logiscope/samples/C/mstrmind/master.c
/users/logiscope/samples/C/mstrmind/player.c
/users/logiscope/samples/C/mstrmind/machine.c
```

Using the command line:

```
create aProject.ttp –test –lang c –list filelist.lst
```

will create a new Logiscope C project file named aProject.ttp containing 3 files: master.c, player.c and machine.c on which the *TestChecker* module will be activated.

## 10.1.2 Makefile mode

When launched from makefiles, **create** is designed to intercept the command line usually passed to the compiler and uses the arguments to build the Logiscope project.

The project makefiles must be modified in order to launch **create** instead of the compiler. In this mode, the name of the project file (".ttp" file) has to be an absolute path, otherwise the process will stop.

When used inside a Makefile, **create** uses the same options as in command line mode, except for:

-root, -recurse, -list : which are not available in this mode

-- : which introduces the compiler command.

The following lines can be introduced in a Makefile to build a Logiscope C project file :

```
CREATE=create /users/projects/myProject.ttp –test –lang c
CC=$(CREATE) -- gcc
CPP=$(CC) -E
...
```

In this mode, the project file building process is as follows:

1. **create** is invoked for each file by the make utility, instead of the compiler.

2. When **create** is invoked for a file it adds the file to the project, with appropriate preprocessor options if any, then Create starts the normal compilation command which will ensure that the normal build process will continue.

3. At the end of the make process, the Logiscope project is completed and can be used either using Logiscope **Studio** or with the **batch** tool (see next section).

***Note:*** *Before executing the makefile, first clean the environment in order to force a full rebuild and to ensure that the* ***create*** *will catch all files.*

# 10.1.3 Options

## Logiscope Ada TestChecker Project Options

```
create -test -lang ada
    <ttp_file>                : Logiscope project file (".ttp" extension).
    [-root <directory>]       : where <directory> is the starting point
                                of the source search. Default is the
                                current directory. This option is exclusive
                                with -list option.
    [-recurse]                : if present the source search is done
                                recursively in subfolders.
    [-list <list_file>]       : where <list_file> is the name of a file
                                containing the list of filenames to add to
                                the project (one file per line).
                                This option is exclusive with -root option.
    [-repository <directory>]: where <directory> is the name of the
                                directory where Logiscope derived files
                                will be stored.
    [-source <suffixes>]      : where <suffixes> is the list of accepted
                                suffixes for source files (e.g. "*.ada").
    [-test_dir <directory>]   : where <directory> is the name of the
                                directory where Logiscope test information
                                will be stored.
    [-working_dir <directory>]: where <directory> is the name of the
                                directory to go in before starting the
                                instrumented binary.
    [-make <cmd>]             : where <cmd> is the name of the command to
                                build the instrumented binary.
    [-exec <cmd>]             : where <cmd> is the name of the command
                                to execute the instrumented binary.
    [-link <file>]            : where <file> is the name of the single
                                instrumentation file to generate in the
                                target directory.
    [-sep]                    : causes the generation of 2 instrumentation
                                files:
                                 - audit_instrum.ads for the specification,
                                 - audit_instrum.adb for the body.
    [-model <file>]           : where <file> is the template to be used to
                                generate the instrumentation file(s).
```

## Logiscope C TestChecker Project Options

```
create –test –lang c
    <ttp_file>                 : Logiscope project file (".ttp" extension)
    [-root <directory>]        : where <directory> is the starting point
                                 of the source search. Default is the
                                 current directory. This option is exclusive
                                 with -list option.
    [-recurse]                 : if present the source search is done
                                 recursively in subfolders.
    [-list <list_file>]        : where <list_file> is the name of a file
                                 containing the list of filenames to add to
                                 the project (one file per line).
                                 This option is exclusive with -root option.
    [-repository <directory>] : where <directory> is the name of the
                                 directory where Logiscope derived files
                                 will be stored.
    [-source <suffixes>]       : where <suffixes> is the list of accepted
                                 suffixes for source files (e.g. "*.c").
    [-test_dir <directory>]    : where <directory> is the name of the
                                 directory where Logiscope test information
                                 will be stored.
    [-working_dir <directory>]: where <directory> is the name of the
                                 directory to go in before starting the
                                 instrumented binary.
    [-make <cmd>]              : where <cmd> is the name of the command to
                                 build the instrumented binary.
    [-exec <cmd>]              : where <cmd> is the name of the command
                                 to execute the instrumented binary.
    [-mcdc]                    : if present sources are instrumented with
                                 multiple decision/condition coverage
                                 ativated.
    [-tcl <tcl_file>]          : where <tcl_file> is the name of the TCL
                                 script used for instrumentation.
                                Default is <install_dir>/util/instrument.tcl
    [-dial <dialect_name>]     : where <dialect_name> is one of the
                                 available C dialects.
    [-def <definition_file>]  : where <definition_file> is a .def file
                                 containing include paths and macro
                                 definitions.
    [-ign <ignore_file>]       : where <ignore_file> is a .ign file
                                 containing specification of C code to
                                 ignore.
    [-I<include_path>]*        : same syntax as a compiler. To be used only
                                 To be used only if option -- is not used.
    [-D<macro_name>]*          : same syntax as a compiler.
                                 To be used only if option -- is not used
    [-U<macro_name>]*          : same syntax as a compiler. To be used only
                                 To be used only if option -- is not used.
    [-mode=exp|noexp]*          : to specify the mode of macros preprocessing.
```

Default is `exp`: macros are expanded. .

`[-mac <macro_file>]` : where <macro_file> is a text file specifying a list of macros statements to be or not to be expanded according to the value of the -mode option..

`[--]` : when used in a makefile, this option introduces the compilation command with its arguments.

## Logiscope C++ TestChecker Project Options

```
create –test –lang c++
    <ttp_file>                : Logiscope project file (".ttp" extension).
    [-root <directory>]       : where <directory> is the starting point
                                of the source search. Default is the
                                current directory. This option is exclusive
                                with -list option.
    [-recurse]                : if present the source search is done
                                recursively in subfolders.
    [-list <list_file>]       : where <list_file> is the name of a file
                                containing the list of filenames to add to
                                the project (one file per line).
                                This option is exclusive with -root option.
    [-repository <directory>]: where <directory> is the name of the
                                directory where Logiscope derived files
                                will be stored.
    [-source <suffixes>]      : where <suffixes> is the list of accepted
                               suffixes for source files (e.g. "*.c; *.cpp")
    [-test_dir <directory>]   : where <directory> is the name of the
                                directory where Logiscope test information
                                will be stored.
    [-working_dir <directory>]: where <directory> is the name of the
                                directory to go in before starting the
                                instrumented binary.
    [-make <cmd>]             : where <cmd> is the name of the command to
                                build the instrumented binary.
    [-exec <cmd>]             : where <cmd> is the name of the command
                                to execute the instrumented binary.
    [-tcl <tcl_file>]         : where <tcl_file> is the name of the TCL
                                script used for instrumentation.
                                Defaut is <install_dir>/util/instrument.tcl
    [-alias <alias_file>]     : where <alias_file> is the name of an alias
                                file (.al file).
                                Default is <install_dir>/util/log_inst.al
    [-ign <ignore_file>]      : where <ignore_file> is a .ign file
                                containing specification of code to ignore.
                                Default is <install_dir>/util/log_inst.ign
    [-memory <file>]          :  where <file> is the name of the c++ file in
                                which coverage information is collected
```

```
                                during execution. Then, at the end of the
                                execution, coverage information is flushed.
    [--]                      : when used in a makefile, this option
                                introduces the compilation command with
                                its arguments.
```

## Logiscope Java TestChecker Project Options

```
create –test –lang java
    <ttp_file>                : Logiscope project file (".ttp" extension)
    [-root <directory>]       : where <directory> is the starting point
                                of the source search. Default is the
                                current directory. This option is
                                exclusive with -list option.
    [-recurse]                : if present the source search is done
                                recursively in subfolders.
    [-list <list_file>]       : where <list_file> is the name of a file
                                containing the list of filenames to add to
                                the project (one file per line).
                               This option is exclusive with -root option.
    [-repository <directory>] : where <directory> is the name of the
                                directory where Logiscope derived files
                                will be stored.
    [-source <suffixes>]      : where <suffixes> is the list of accepted
                                suffixes for source files (e.g. "*.java").
    [-test_dir <directory>]   : where <directory> is the name of the
                                directory where Logiscope test information
                                will be stored.
    [-working_dir <directory>]: where <directory> is the name of the
                                directory to go in before starting the
                                instrumented binary.
    [-make <cmd>]             : where <cmd> is the name of the command to
                                build the instrumented binary.
    [-exec <cmd>]             : where <cmd> is the name of the command
                                to execute the instrumented binary.
    [-tcl <tcl_file>]         : where <tcl_file> is the name of the TCL
                                script used for instrumentation. Default is
                                 <install_dir>/util/instrument.tcl
    [-alias <alias_file>]     : where <alias_file> is the name of an alias
                                file (.al file). Default is
                                <install_dir>/util/log_inst.al
    [-ign <ignore_file>]      : where <ignore_file> is a .ign file
                                containing specification of code to
                                ignore. Default is
                                <install_dir>/util/log_inst.ign
```

# 10.2 Logiscope batch

Logiscope **batch** is a tool designed to work with Logiscope in command line to:

- instrument the source code files specified in a Logiscope project: i.e. ".ttp" file,
- generate reports in HTML and/or CSV format automatically.

Note that before using **batch**, a Logiscope project shall have been created:

- using Logiscope **Studio**, refer to Section 1,
- or using Logiscope **create**, refer to the previous section.

Once the Logiscope project is created, **batch** is ready to use.

## 10.2.1Options

The **batch** command line options are the following:

`batch`

| | |
|---|---|
| `<ttp_file>` | the Logiscope TestChecker project file (with ".ttp" extension). |
| `[-dyn <dynamicfile>]` | where `<dynamicfile>` is the name of the dynamic file i.e. the file containing the execution traces generated when executing the instrumented binary.<br>In case several dynamic files have been gererated, they shall first be merged using the lgdynld tool (see next section), |
| `[-tcl <tcl_file>]` | name of a **Tcl** script to be used to generate the reports instead of the default **Tcl** scripts. |
| `[-o <output_directory>]` | directory where the all reports are generated. |
| `[-nobuild]` | generate reports without rebuilding the project. The project must have been built at least once previously. |
| `[-clean]` | before starting the build, the Logiscope build mechanism removes all intermediate files and empties the import project folder when the external violation importation mechanism is activated. |
| `[-addin <addin> options]` | where `addin` nis the name of the addin to be activated and `options` the associated options generating the reports. |

| | |
|---|---|
| `[-table]` | generate tables in predefined html reports instead of slices or charts. By default, slices or charts are generated (depending on the project type).<br>This option is available only on Windows as on Unix there are no slices or charts, only tables are generated. |
| `[-noframe]` | generate reports with no left frame. |
| `[-v]` | display the version of the **batch** tool. |
| `[-h]` | display help and options for **batch**. |
| `[-err <log_err_folder>]` | directory where troubleshooting files **batch.err** and **batch.out** should be put. By default, messages are directed to standard output and error. |

## 10.2.2Examples of Use

Considering a Logiscope C *TestChecker* project **LogProj.ttp** as an example:

1 Produce an instrumented binary by typing on a command line or in a script:

```
batch LogProj.ttp
```

2 Execute the instrumented binary in order to produce one or more dynamic result files.

3 Merge the dynamic files, using the **lgdynld** command (see next section) in order to obtain a single dynamic file named **LogProj.dyn**.

4 Generate a test coverage report using the default Logiscope Tcl script **TestReport.tcl**. by typing on a command line or in a script:

```
batch LogProj.ttp -dyn LogProj.dyn
```

To read the report into an HTML browser, just open the **LogProjtest.html** file generated in the **<LogProj>/Logiscope/report** directory.

# 10.3 Logiscope lgdynld

**lgdynld** is a tool designed to merge dynamic coverage files into one file.

## 10.3.1Options

`lgdynld`

| | |
|---|---|
| `[<ttpfile>]` | Logiscope project. |
| `-dyn <dynfilelist>` | `dynfilelist` is a text file containing the list of the dynamic coverage files (one file per line) to be merged. |
| `[-skip]` | to merge or not with dynamic coverage files already present in the **.ttp** file. |
| `<outputfile>` | name of the resulting merged dynamic coverage file. This file must have a .**dyn** extension. |

## 10.3.2Examples of Use

There are two main ways to use **lgdynld**. One allows checking consistency of the dynamic coverage files with the results of static analysis, the other one without checking.

### Without Consistency Checking

```
lgdynld -dyn dynfilelist output.dyn
```

This call will merge the dynamic coverage files found in `dynfilelist` into **output.dyn** file.

**WARNING: this call makes no consistency check, the results of the static analysis should be the same for all dynamic coverage files to be merged in order to ensure the accuracy of the resulting output.dyn file.**

### With Consistency Checking

```
lgdynld project.ttp -dyn dynfilelist output.dyn
```

This call will merge the dynamic coverage files found in the `ttp` file and in the `dynfilelist` into **output.dyn** file. The consistency with the project file is secured. Anyway, the resulting dynamic coverage file is not loaded in the `ttp` file at the end of the execution. This can be done through Logiscope *Studio* or Logiscope Batch.

```
lgdynld project.ttp -dyn dynfilelist -skip output.dyn
```

This call using `-skip` option has the same behavior as the previous one except that the dynamic coverage files found in the `ttp` will not be merged into **output.dyn** file.

## 10.3.3Merging .trc Files

**lgdynld** also allows to merge raw trace files (**.trc**) with dynamic coverage files (**.dyn**) and then generates a **.dyn** file.

Example:

```
lgdynld project.ttp –dyn trcfilelist output.dyn
```

where `trcfilelist` may contain **.trc** files or **.dyn** files, and `project.ttp` is optional.

# Notices



The licensed program described in this document and all licensed material available for it are provided by Kalimetrix under terms of the Kalimetrix Customer Agreement, Kalimetrix International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-Kalimetrix products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Kalimetrix has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Kalimetrix products. Questions on the capabilities of non-Kalimetrix products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

## Copyright license

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Kalimetrix, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Kalimetrix, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
© (your company name) (year). Portions of this code are derived from Kalimetrix Corp. Sample Programs. © Copyright Kalimetrix Corp. _enter the year or years_.

## Trademarks

Kalimetrix, the Kalimetrix logo, Kalimetrix.com are trademarks or registered trademarks of Kalimetrix, registered in many jurisdictions worldwide. Other product and services names might be trademarks of Kalimetrix or other companies.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.
AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.
Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.
Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
Pentium is a trademark of Intel Corporation.
ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S Patent and Trademark Office.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other company, product or service names may be trademarks or service marks of others.