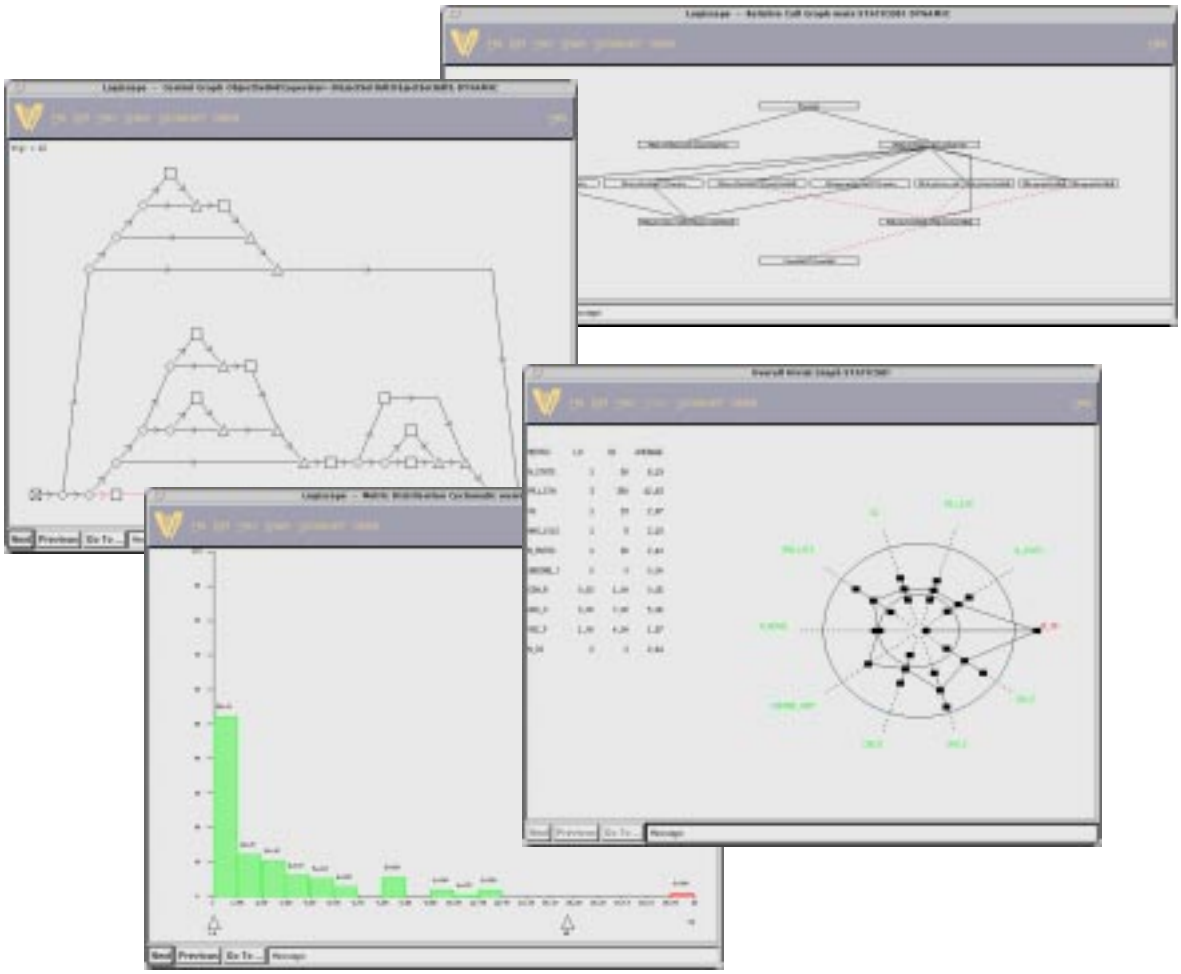
Logiscope/SRT Integration User's Guide

Arash Khodabandeh



ATLAS DAQ Technical Note 084

Document Version 1.9, February 1999

URL: `http://atddoc.cern.ch/Atlas/Notes/084/Welcome.html`
`http://atddoc.cern.ch/Atlas/Notes/084/usersguide.ps`
`http://atddoc.cern.ch/Atlas/Notes/084/usersguide.pdf`

Copyright CERN, Geneva 1997 - Copyright and any other appropriate legal protection of this documentation and associated computer program reserved in all countries of the world.

Organisations collaborating with CERN may receive this program and documentation freely and without charge.

CERN undertakes no obligation for the maintenance of this program, nor responsibility for its correctness, and accepts no liability whatsoever resulting from its use.

Program and documentation are provided solely for the use of the organisation to which they are distributed.

This program may not be copied or otherwise distributed without permission. This message must be retained on this and any other authorised copies.

The material cannot be sold. CERN should be given credit in all references.

This document has been prepared with Release 5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

Running text:	Palatino 10.5 pt on 13.5 pt line spacing
Chapter numbers and titles:	AvantGarde DemiBold 36 and 24 pt
Section headings	AvantGarde DemiBold 20 pt
Subsection and subsubsection headings:	Helvetica Bold 12 and 10 pt
Captions:	Helvetica 9 pt
Listings:	Courier Bold 9 pt

Use of any trademark in this document is not intended in any way to infringe on the rights of the trademark holder.

Preface

This document describes the Logiscope integration in the SRT based software development environment of ATLAS DAQ. It explains how the integration was implemented and gives some examples of usage. This document is not a replacement for Logiscope or SRT manuals.

How to Use This Document ii

How to Use This Document

You can use this manual differently depending on your role in the project.

- Chapter 1 gives a general overview of what Logiscope is and how it works.
- Chapter 2 gives an overview of how the integration is organized.
- Chapter 3 explains how the quality is measured, which quality models are available and how to switch from one to the other.
- Chapter 4 details some of the most common actions that one may want to perform with Logiscope under SRT.
- Appendix A lists the requirements for SRT in order to integrate Logiscope.
- Appendix B lists the known problems and workarounds if any.

As an End User of Logiscope/SRT

Start by Chapter 1 and Chapter 2 as introduction. They will help you to use the actions detailed in Chapter 4. Read Chapter 3 to see how to switch between quality models. Refert to Appendix B for a workaround to possible problems.

As the Project Quality Manager or the Project Manager

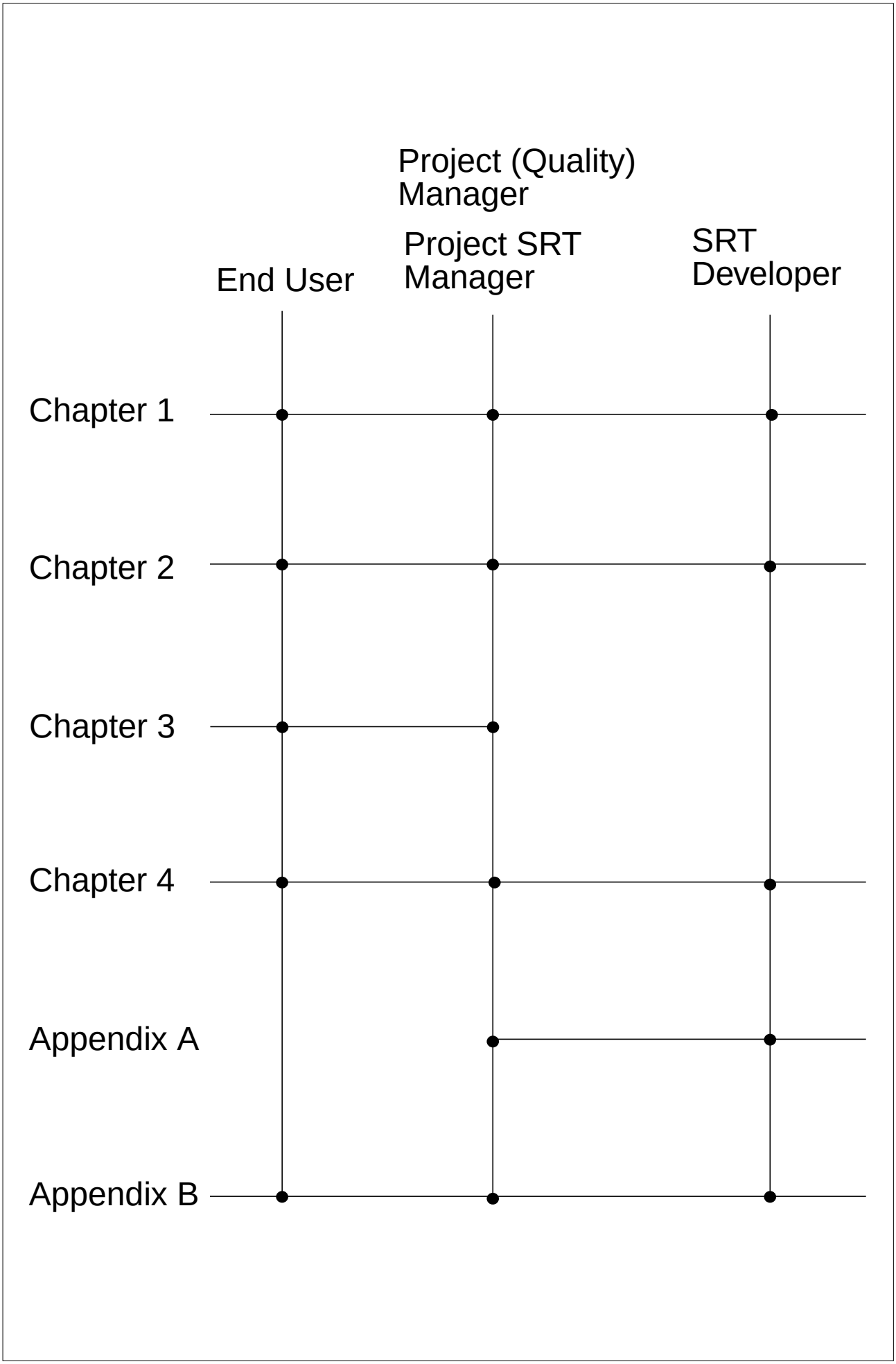
All chapters are relevant for you.

As the Project SRT Manager

As the person maintaining this Logiscope / SRT integration, all chapters are of interest to you.

As the SRT Developer

As the person implementing the requirements listed in Appendix A in SRT, read Chapter 1 and Chapter 2 to see what this is all about. The use cases in Chapter 4 can be turned into test cases. Appendix B lists outstanding problems some of which can be fixed in the inplementation.



A different path through the chapters can be followed depending on your role in the project.

Outline

		Preface	i
		Contents	vii
Chapter	1	Logiscope	1
Chapter	2	SRT Integration Strategy	9
Chapter	3	Quality Measurement	15
Chapter	4	Use Cases	23
Appendix	A	SRT Integration Requirements	35
Appendix	B	Known Problems	39
		References	43
		Index	45

Contents

Preface i
How to Use This Document ii
As an End User of Logiscope / SRT ii
As the Project Quality Manager or the Project Manager . . . ii
As the Project SRT Manager ii
As the SRT Developer ii

Outline v

Chapter 1
Logiscope 1
1.1 What is Logiscope 2
Logiscope CodeChecker 2
Logiscope TestChecker 2
1.2 Static Code Analysis Process 4
1.3 Dynamic Code Analysis Process 6
1.4 Logiscope at CERN 8

Chapter 2
SRT Integration Strategy 9
2.1 Definitions for a Makefile Target 10
2.2 Implementation per SRT Module 12
Analyzing Results for a Makefile Target 12
A Directory with Different Makefile Targets 12
A Directory with Sub-Directories 12
A Directory with Makefile Targets and Sub-Directories . . . 12

Chapter 3
Quality Measurement 15
3.1 What is a Quality Model? 16
3.2 Test Coverage Measurement 18
3.3 What Quality Model can be used 20

Chapter 4
Use Cases 23
4.1 Prepare for Analysis 24
Configure the Project for Analysis 24

Run the Analyzers . . . 24
Run the Dynamic Tests . . . 24
Post Analysis Phase . . . 25
4.2 View Static Results . . . 26
View Entire Project . . . 26
View a Directory . . . 26
View a Makefile Target . . . 26
4.3 Logiscope CodeChecker Actions . . . 27
4.4 Logiscope Static Viewer Actions . . . 28
Global Information . . . 28
Function Information . . . 28
Call Graph Information . . . 29
Workspace and Selection . . . 29
Save a Workspace . . . 30
Save a State . . . 30
4.5 View Dynamic Results . . . 31
View Entire Project . . . 31
View a Directory . . . 31
View a Makefile Target . . . 31
4.6 Logiscope TestChecker Actions . . . 33
4.7 Logiscope Dynamic Viewer Actions . . . 34
Global Information . . . 34
Function Information . . . 34
Test Workspace . . . 34

Appendix A
SRT Integration Requirements . . . 35
A.1 List of Requirements . . . 36
Static Result Generation per Makefile Target . . . 36
Static Result Generation per Directory . . . 36
Instrumented Target Generation . . . 37
Dynamic Result Generation per Makefile Target . . . 38
Dynamic Result Generation per Directory . . . 38

Appendix B
Known Problems . . . 39
B.1 List of Known Problems . . . 40
General Problems . . . 40
Problems Specific to Dynamic Analysis . . . 40

References . . . 43

Index . . . 45

Chapter 1

Logiscope

This chapter outlines the different components of Logiscope which have been integrated with SRT and the way they work.

1.1 What is Logiscope 2

1.2 Static Code Analysis Process 4

1.3 Dynamic Code Analysis Process 6

1.4 Logiscope at CERN 8

1.1 What is Logiscope

Logiscope [4] from Verilog [5] is a set of tools addressing code quality through source code metrics (CodeChecker*), test coverage (TestChecker), coding standards (RuleChecker) and resource usage analysis (ImpactChecker).

Logiscope CodeChecker and TestChecker have been integrated with SRT [2].

Logiscope CodeChecker

The role of Logiscope CodeChecker is to help the programmer to identify area of his/her code that do not follow the project quality model. The developer can then investigate and eventually take the necessary action. This type of analysis is called static analysis as it does not require the execution of the code.

Logiscope CodeChecker receives as input:

1. the source code on which it will measure a set of metrics,
2. the quality model according to which it will interpret and display the results.

Logiscope CodeChecker supports various programming languages (C, C++, Fortran...) and their dialects. This SRT integration concerns only C and C++.

Logiscope TestChecker

The role of Logiscope TestChecker is to analyze the test case traces and identify portions of codes that have not been executed during tests. The programmer can then design new tests to cover more of the code or to drop redundant tests that cover the same code. This type of analysis is called dynamic analysis as it requires the execution of the code.

Logiscope TestChecker receives as input:

1. the source code that it will instrument. This new instrumented code is then compiled and ran through the test suite instead of the original code.
2. the quality model according to which it will display the results.

Logiscope TestChecker supports the same set of languages than Logiscope CodeChecker. This SRT integration concerns only C and C++.

Note
When in this document a concept is explained referring to Logiscope, it should be understood that the concept is applicable to both to Logiscope CodeChecker and TestChecker.

* CodeChecker is also the name of a coding style/rule checker used in ATLAS. To prevent any confusion we will refer to Logiscope CodeChecker in the rest of this document.

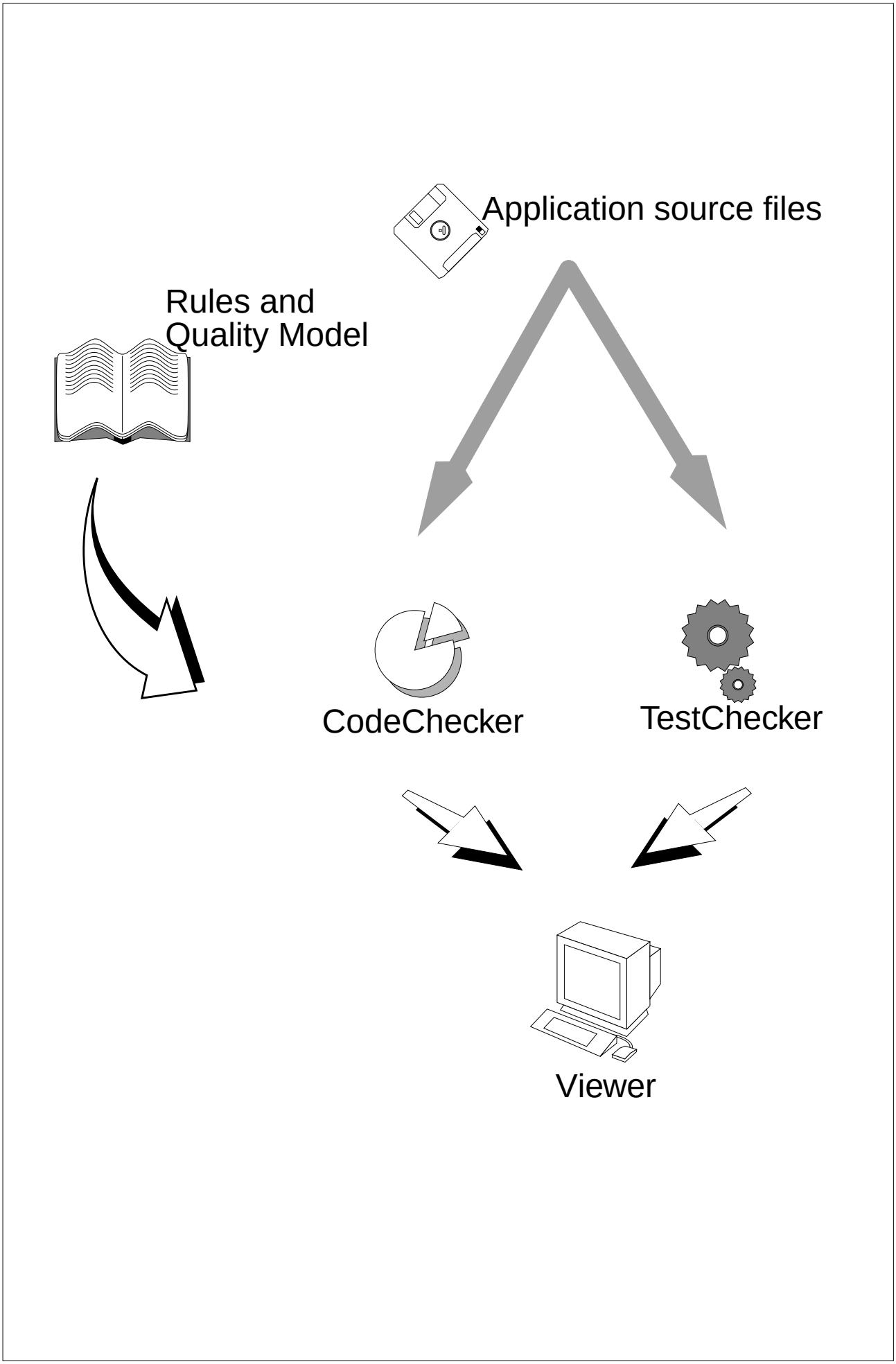


Illustration 1.1 Logiscope CodeChecker and TestChecker measure metrics on the source code and the test coverage traces and display the results according to the model.

1.2 Static Code Analysis Process

The static code analysis chain done by Logiscope CodeChecker is similar to the compilation chain:

1. Analysis

   At this stage Logiscope CodeChecker analyses individual source files. For each source file two separate data files are generated containing static information extracted from the source code.

2. Link

   At this stage Logiscope CodeChecker merges the different data files into archive files. Since there are two different types of data files, Logiscope CodeChecker generates two separate archive files.

3. Viewing

   The above archive files are loaded in various tools to be viewed.

The above static analysis chain is performed on the host where both the Logiscope tools and the source code are accessible. This means that no access to the target host (where the application should run) is required.

The Logiscope / SRT Integration automates steps 1 and 2.

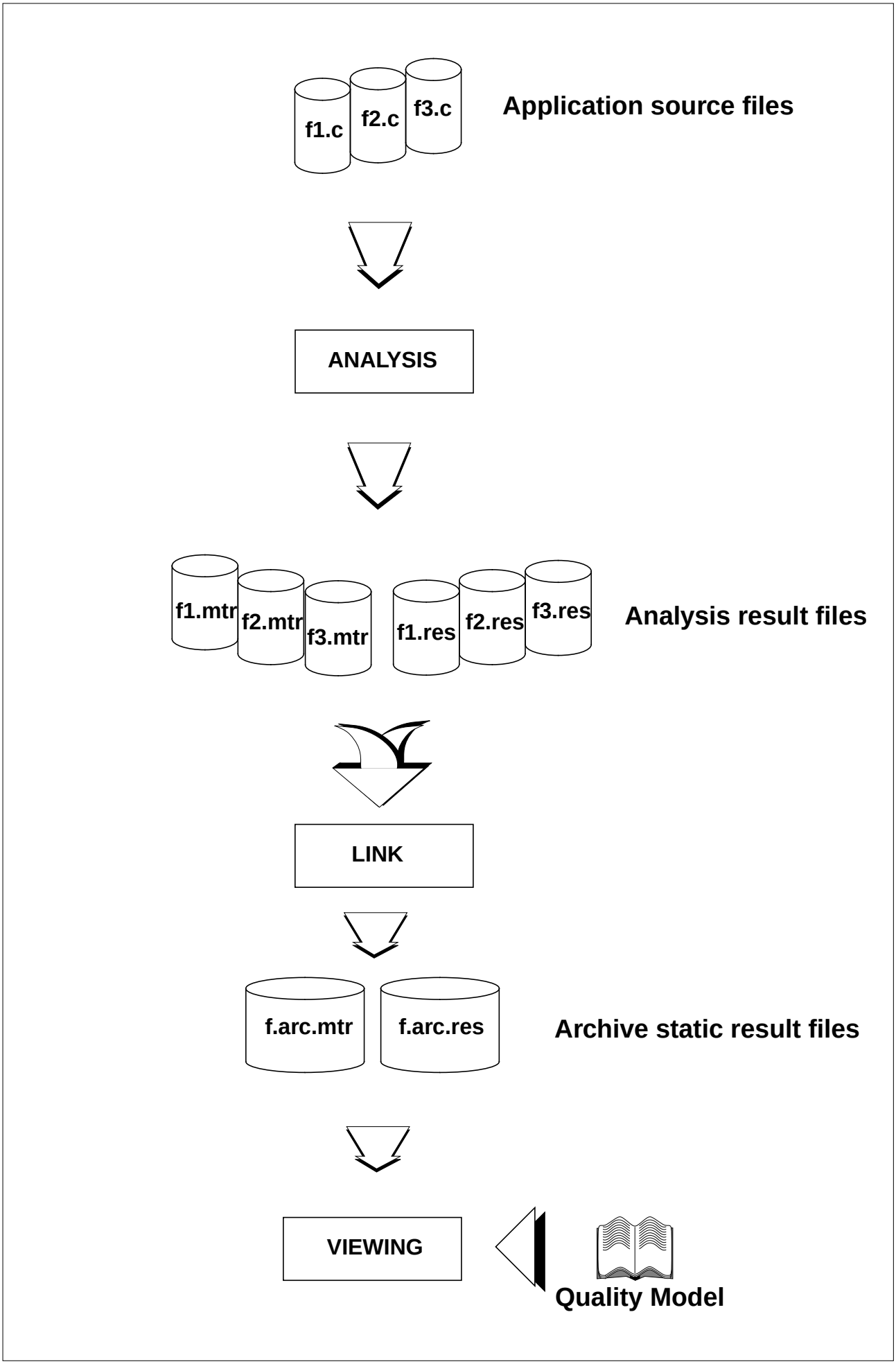


Illustration 1.2 The static analysis chain of Logiscope CodeChecker is similar to the compilation chain except that the output are two analysis archive files instead of the executable.

1.3 Dynamic Code Analysis Process

The dynamic analysis chain is as follows:

1. Instrumentation

   Logiscope TestChecker instruments every source file by placing special calls to Logiscope trace functions in well defined locations of the code. The effect of those calls is to generate formatted output to log (trace) which part of the code was executed and when.

   For each source file an instrumented version of it is generated.

2. Source transfer

   At this point, if the application has to run on a special target, the instrumented code should be transfer to the target host.

3. Compilation and linking

   The instrumented code is compiled and linked (with the Logiscope trace functions*) instead of the original code to obtain an instrumented executable.

4. Test execution

   The above instrumented executable is ran through the tests in place of the standard application. Each time the application is ran, an execution result file is generated.

5. Result transfer

   If the application was built and ran on special target, the generated execution result files should be transferred back to the analysis platform (where Logiscope is available).

6. Merging

   The different execution result files from different executions of the application are merged together in an archive execution trace file.

7. Viewing

   The above archive file can now be loaded in various tools to be viewed.

The instrumentation, merging and viewing of result files are performed on the host where both Logiscope and the source code are available. Only the compilation, link and test execution steps require access to the target host where the application should run.

Note
When performing this dynamic chain the static results are automatically generated. This means that if you run the dynamic analysis you do not need to run the static analysis separately. In fact the dynamic analysis requires the static data.

* The source code for the Logiscope trace functions is available so it can be ported to target hosts.

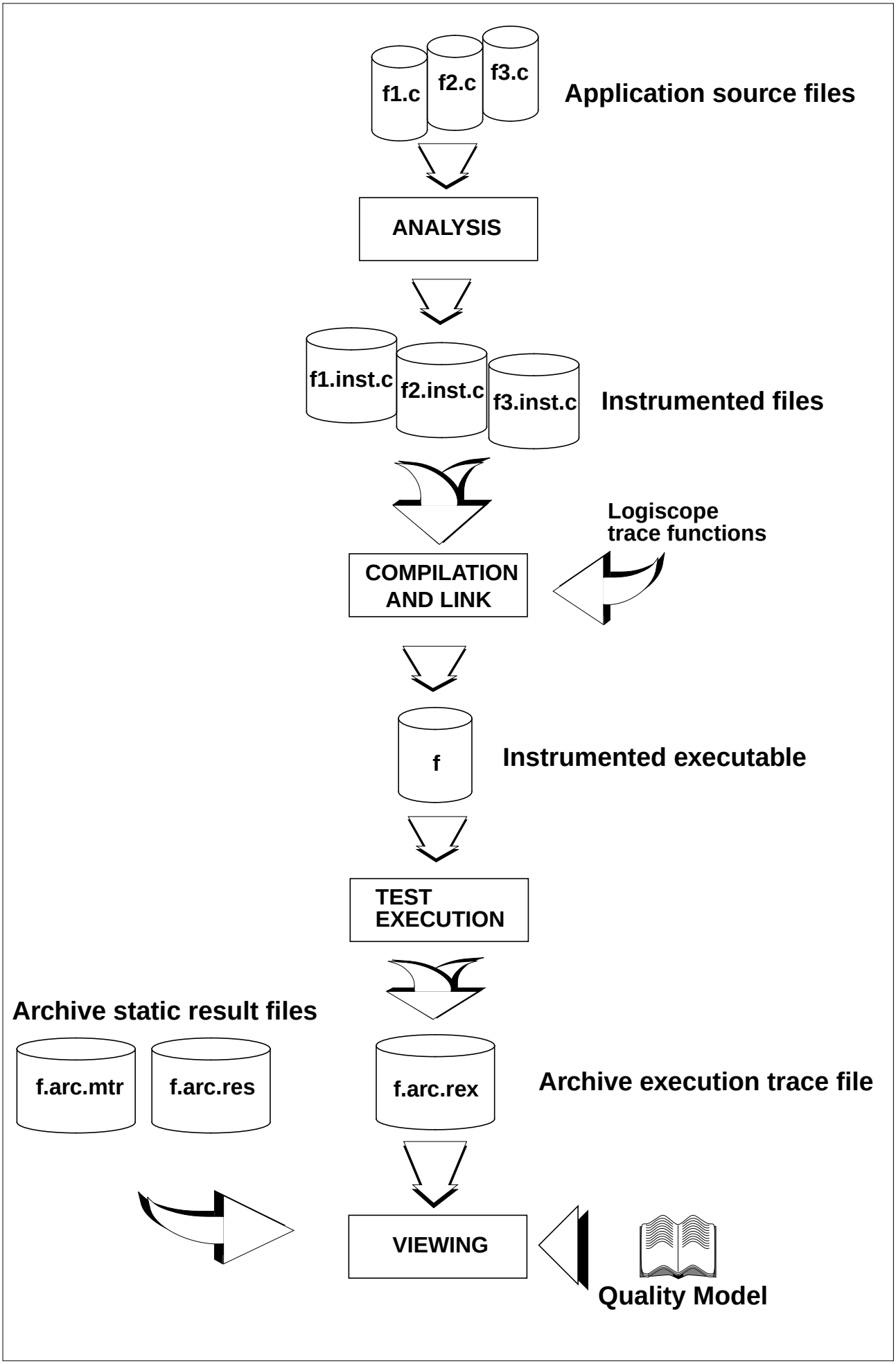


Illustration 1.3 For test coverage analysis, Logiscope TestChecker instruments the source files which are then compiled and run instead of the original application.

1.4 Logiscope at CERN

Logiscope is available through the Software Development Tools Service [3]. A number of licenses are purchased and can be used concurrently by a certain number of users.

Note
Logiscope uses floating licenses. To guarantee access to other users, please make sure you quit the application when you are not working with it.

In order to have access to Logiscope, you will need to update your environment. Open `http://www.cern.ch/PTTOOLS/Logiscope/How.html`, go to the page corresponding to the architecture of the computer where you will use Logiscope on and follow the instructions.

Chapter 2

SRT Integration Strategy

This chapter explains how the static and dynamic results are generated taking advantage of the SRT package structure in order to allow structured and focused analysis. It is recommended to read Chapter 1 for more information on what is Logiscope and how it works before starting with this chapter.

2.1 Definitions for a Makefile Target 10

2.2 Implementation per SRT Module 12

2.1 Definitions for a Makefile Target

For each makefile target, Logiscope generates the following files:

1-Static result files For each source code file two static result files are generated. They have the source file basename and the `.mtr` and `.res` extensions. They contain all the metrics values and structural information about the components contained in the corresponding source file.

2-Block count file This is an ASCII report generated for each source code file. It has the source file basename and the `.cb` extension. It is generated at the same time as the static result files.

3-Archive static result files In order to be analyzed in a more effective manner, a set of static result files can be merged together in archive static result files. Logiscope automatically merges all static result files of one target in archive static result files of that target. As we have two types of static result files, we also have two archive static result files for each target (library or executable). They have the target basename and the `.arc.res` and `.arc.mtr` extensions. They contain all the metrics values and structural information about the components contained in the source files of the corresponding target.

4-Instrumented source file For each source code and header file an instrumented version of it is generated. They have the source file basename and the `.inst.c` and `.inst.h` extensions. They contain the original source code plus special calls to Logiscope trace functions (see Section 1.3, "Dynamic Code Analysis Process").

5-Instrumented library When the instrumented source files are compiled and put together an instrumented library is produced. The instrumented library does not have any special extension.

6-Instrumented executable When the instrumented source files are compiled and linked an instrumented executable is produced. Depending on the Logiscope trace functions used the instrumented executable can be for batch (no special extension) mode or for interactive (executable basename with `.inter` extensions respectively).

7-Execution trace file Each time a batch instrumented executable is run it generates trace data in a separate dynamic trace file (executable basename with `.dyn` extension). The dynamic trace file is then converted into an execution trace file. It has the executable basename and the `.rex` extension. It contains information on which part of the code was executed in the corresponding run on the batch instrumented executable.

8-Archive execution trace file A set of execution trace files of a batch instrumented executable can be merged together in an archive executable trace file. It has the executable basename and the `.arc.rex` extension. It contains information on which part of the code was executed in all the runs of the corresponding batch instrumented executable.

9-Application file For each target an application file is generated. It lists all file names of the target and other target information. It will be the starting point of any analysis. It has the target basename and the `.app` extension.

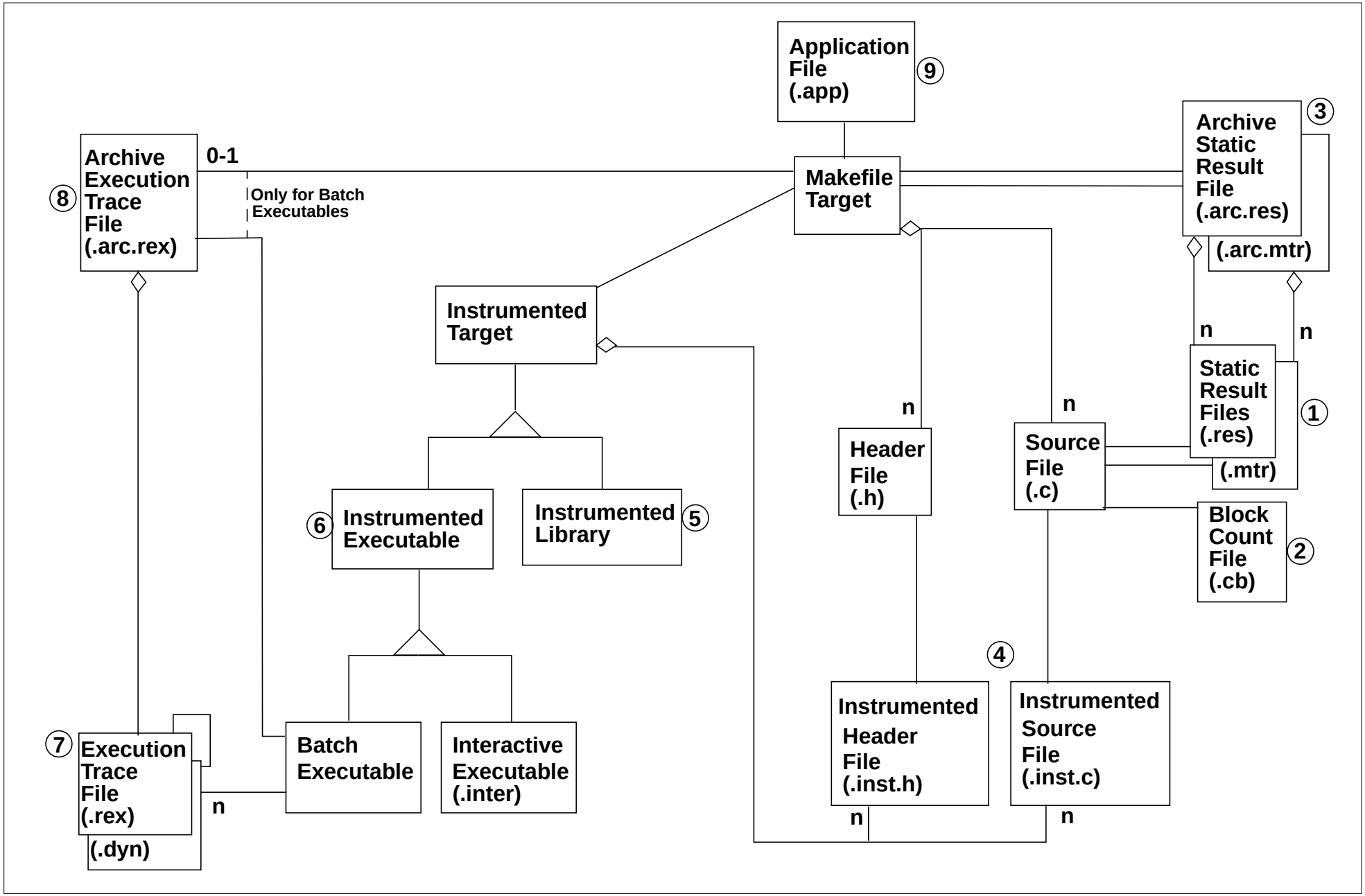


Illustration 2.1 Object diagram of the files generated by Logiscope per makefile target. The file extensions, when applicable, are shown in parenthesis.

2.2 Implementation per SRT Module

SRT is structured per module. Each module in SRT is organized as a directory tree. In each sub-directory of a module may contain one or more makefile targets (libraries or executables) and / or other sub-directories. The analysis reflects this file structure hierarchy, getting more specific lower down the tree.

Analyzing Results for a Makefile Target

As we have seen in Chapter 1, to view the static results for a target you need its two archive static result files. If the target is an executable and youe have performed the dynamic analysis chain, you would also need its archive execution trace file.

As we will see in Chapter 4, the easiest way to analyze the results of a target is to open the target application file with Logiscope.

A Directory with Different Makefile Targets

In the case of such a directory, the goal is to analyze all the targets it contains at once. In order to ease this analysis, all the application files of the targets of this directory are merged together in one application file. It has the directory name and the `.app` extension.

Similarly all archive static result files of the targets of this directory are merged together in archive static result files. As we have two types of archive static result files for each target, we also have two archive static result files for each directory. They have the directory name and the `.arc.res` and `.arc.mtr` extensions.

Similarly, if any of the targets contained in the directory are executables and the dynamic analysis chain is performed, you will also merge all the archive execution trace files of the executables of this directory together in an archive execution trace file. It has the directory name and the `.arc.rex` extension.

A Directory with Sub-Directories

In the case of such a directory, the goal is to analyze all the results of its sub-directories at once. The procedure used is similar to the one described above with the difference that we merge archive and application files of sub-directories instead of archive and application files of targets.

A Directory with Makefile Targets and Sub-Directories

This is a generalization of the above two cases. Here we use again the same process but we merge together the archive and application files of the targets **and** sub-directories contained in the directory.

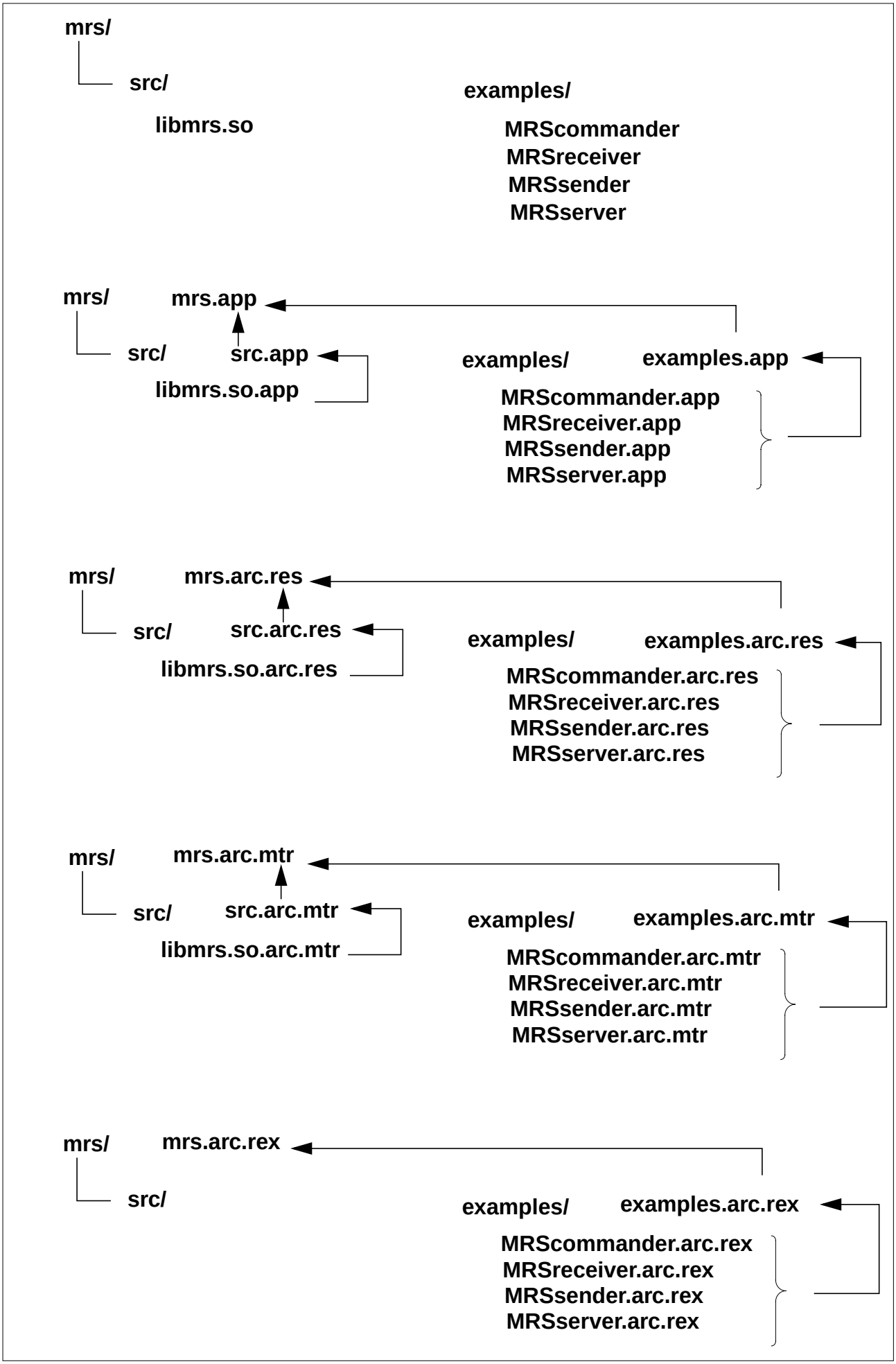


Illustration 2.2 The MRS module is composed of two sub-directories: src/ contains 1 library target; examples/ contains 4 executables target. The graphic illustrates how the merging of the different files as described in the previous page is performed for MRS. Note the fact that the libmrs.so target does not have an archive execution trace file and the implication this has on the merging.

Chapter 3

Quality Measurement

This chapter outlines how the quality is modeled and assessed and the different test coverage strategies.

3.1 What is a Quality Model? 16

3.2 Test Coverage Measurement 18

3.3 What Quality Model can be used 20

3.1 What is a Quality Model?

Logiscope has adopted the Boehm [6] and McCall [7] approaches to quality model. This approach defines software quality as a set of characteristics which:

- are important to the user: the quality factor,
- can be decided by the designer: the quality criteria,
- can be measured for verification: the quality metrics.

To be assessed the quality factor is decomposed in term of quality criteria. Each quality criteria is assigned a weight. In turn each quality criterion is decomposed in term of weighted quality metrics. The quality model is as ASCII file which describes the above structure and the respective limits.

The evaluation process has 3 steps:

1. All metrics are measured on the source code. Their values are compared to the limits defined in the quality model. They can be displayed using histograms, Kiviat graphs or simple tables.
2. Metrics are then combined together according to the quality model to deduce the quality criteria. Again the quality criteria values are compared to the limits defined in the quality model.
3. Finally, the quality factor is calculated from the quality criteria and the different application components are separated according to the quality model. The quality factor is usually displayed in terms of a pie chart, listing or table.

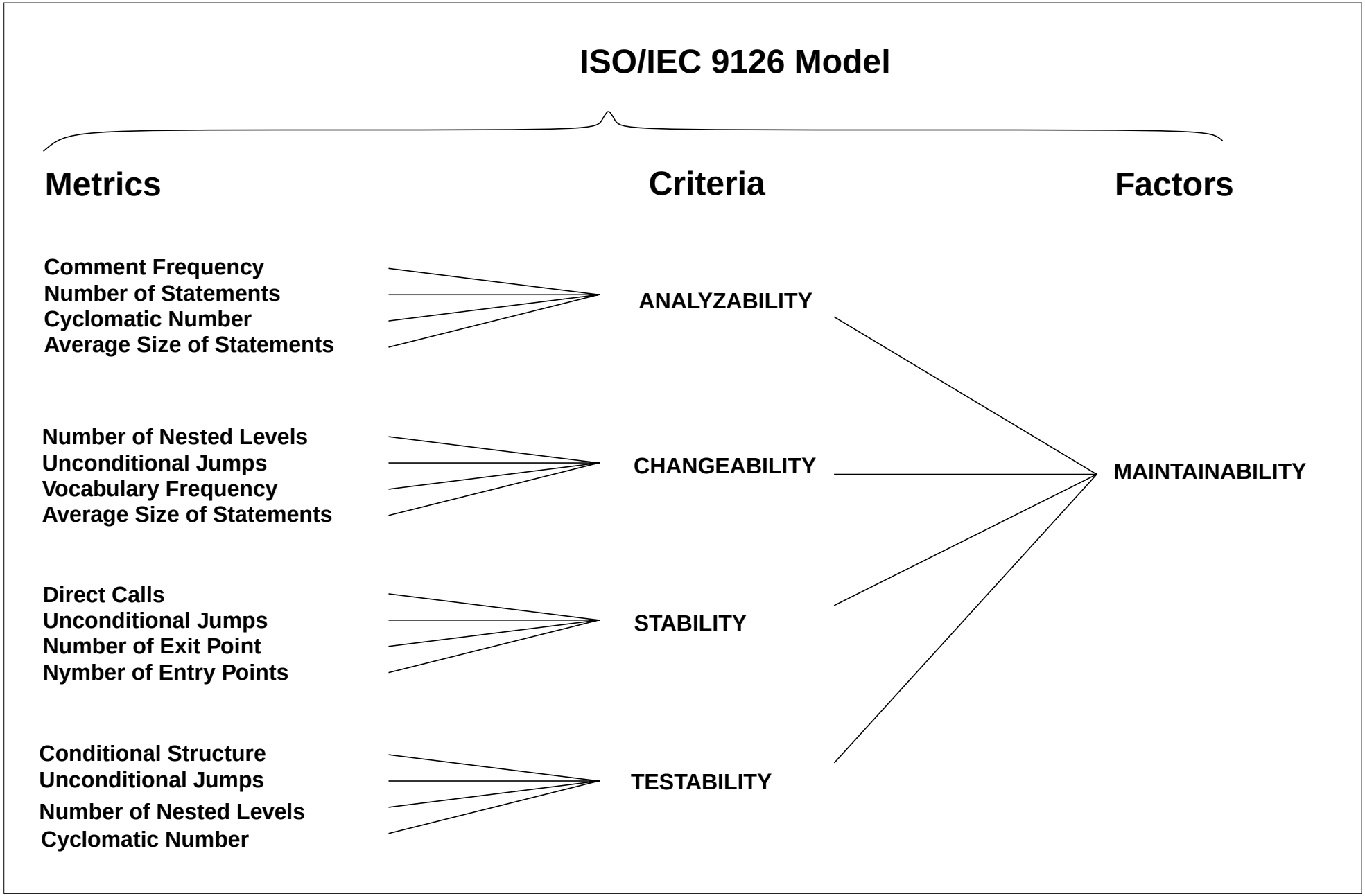


Illustration 3.1 The quality model implementing the ISO 9126 standard [8].

3.2 Test Coverage Measurement

Logiscope TestChecker addresses white box testing by highlighting areas of code which have not been executed during tests. The coverage of a code segment is addressed both at the call graph level (PPP) and the function control graph level (IB, DDP, LCSAJ).

Procedure to Procedure Path (PPP) This gives information on which function calls have not been executed. The result is displayed graphically on the application call graph where the un-executed function calls are highlighted.

Instruction Block (IB) An instruction block is a set of source code instructions without any branches. Instruction block coverage is more effective than line coverage as a line can contain more than one instruction block.

The IB addresses the individual function coverage. The result is displayed on the function control graph where the un-executed instruction block are highlighted.

Decision to Decision Path (DDP) A decision to decision path is a sequence of instructions containing no decision, starting at the entry of the function or at a decision (e.g. if, while), and ending at the exit of the function or the next decision.

As opposed to IB, DDP takes into account the invisible branches. For example the code:

```
If (a>b) then process(a);
```

has 1 IB, but 2 DDP (counting the hidden else).

Linear Code Sequence And Jump (LCSAJ) A linear code sequence and jump is a sequence of instructions containing no jump, starting at the entry of the function or an a target of a jump, and ending at the exit of the function or at a jump target.

LCSAJ can be viewed as an extension to DDP. It takes into account invisible branches ad well as combination of successive branches.

It requires more effort to obtain a higher coverage rate with LCSAJ than with DDP. In turn DDP coverage is more difficult to reach than IB coverage. On the other hand the completeness and effectiveness of the tests in higher in LCSAJ than in DDP than in IB. The decision on which coverage method to adopt depends on the importance of the component to test.

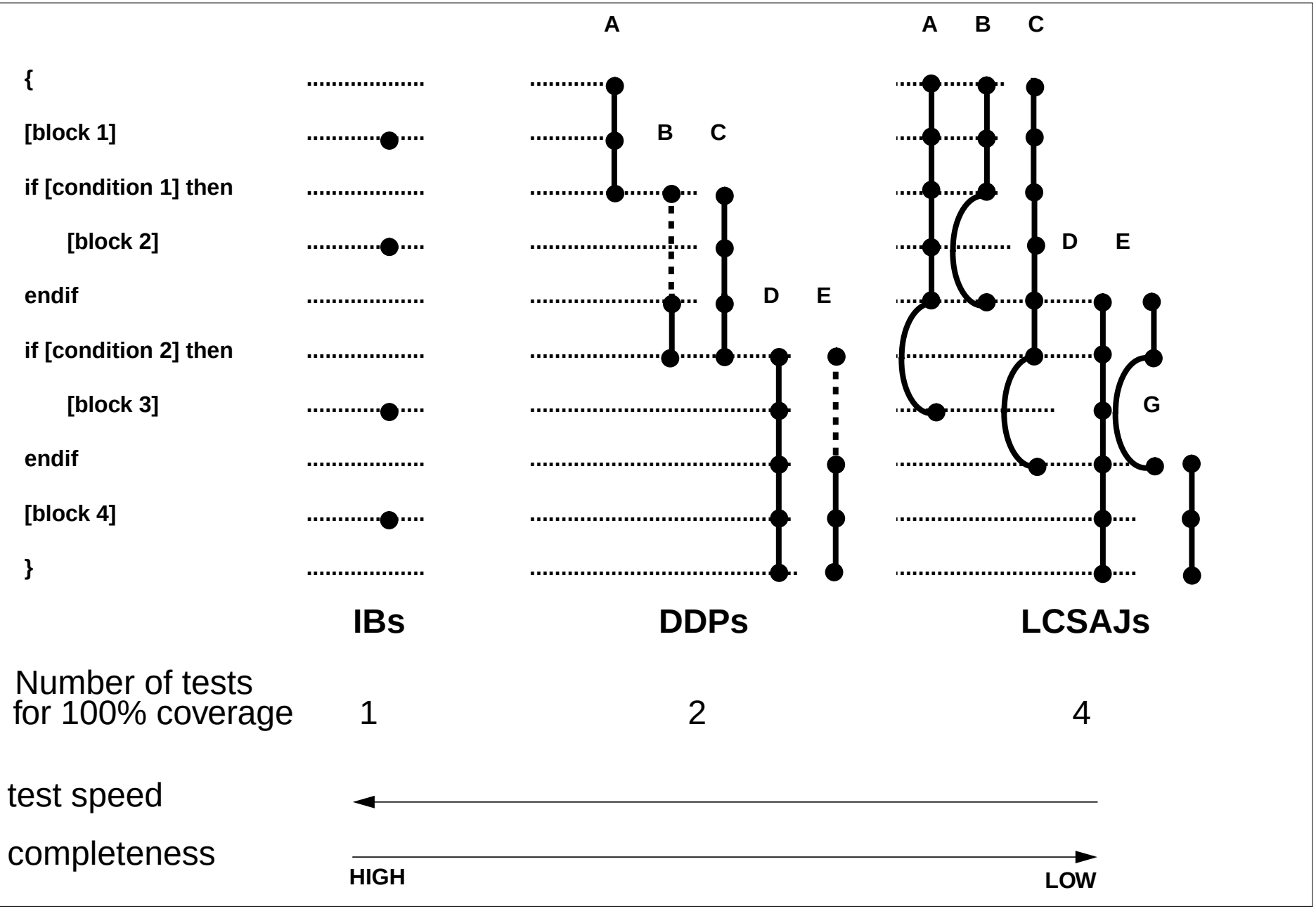


Illustration 3.2 The same algorithm will require more test cases to reach 100% coverage with LCSAJ than DDP or IB. On the other hand the tests are more complete and the component better tested.

3.3 What Quality Model can be used

There are various quality models provided with Logiscope*. The quality model depends on the language and the viewer:

- Logiscope C++ CodeChecker: command line `cppcchk`. It automatically loads the default model file (`.cpplogmodel`). This file contains 4 different quality models assessing function, class and application maintainability and class reusability.
- Logiscope C CodeChecker: command line `cchk`. It automatically loads the default model file (`.logmodel`). This file contains the translation of the Logiscope GUI default C quality model for Logiscope GUI.
- Logiscope GUI: command line `log_gui`. It is can also be started through the Logiscope C or C++ CodeCheckers. This viewer is language independent. It automatically loads the default model file (ref.dat). This file contains one model for C and another for C++. The C quality model gives the same results as the above Logiscope C CodeChecker quality model. But the C++ quality model is different from the above Logiscope C++ CodeChecker quality model.

One cannot load the quality model file of one viewer into another viewer. On the other hand it is possible to use different quality models in the same viewer. Changing the quality model means applying a different filter on the results gathered in the various archive files (`.arc.res`, `.arc.mtr`, `.arc.rex`). So when one changes the quality model or looks at the results in a different viewer which may be using a different quality model one gets different quality reports.

One may change the quality model without re-running the analysis. To change from one model to an other:

- Logiscope C and C++ CodeChecker: pop up the `Set Quality Model` dialog box with `Quality -> Set Model...`. Navigate to the directory containing the quality model you need to load and select the file. Click on `OK`.
- Logiscope GUI: pop up the `Load` dialog box with `File->Load...`. Check the `Editor Reference File` radio box. Navigate to the directory containing the quality model you need to load and select the file. Click on `Select`. Click on `OK`.

* The provided quality models are located under `/pttools/Logiscope/logiscope/ref/gb`.

Model file name	Language	Viewer	
ref.dat	C++	Logiscope GUI	The default model.
ref_iso9126.dat	C++	Logiscope GUI	
.cpplogmodel	C++	C++ CodeChecker	Close to ISO 9126 with OO metrics. Contains 4 different models. The default model.
ref.dat	C	Logiscope GUI	The default model.
ref_iso9126.dat	C	Logiscope GUI	
.logmodel	C	C CodeChecker	Same as ref.dat. The default model.

Illustration 3.3 Various models available.

Chapter 4

Use Cases

This chapter describes some of the common actions the user may want to perform. It is not a replacement of the manual which should be consulted for advanced actions.

The example used throughout this chapter is the MRS package [9] [10].

4.1 Prepare for Analysis 24
4.2 View Static Results 26
4.3 Logiscope CodeChecker Actions 27
4.4 Logiscope Static Viewer Actions. 28
4.5 View Dynamic Results 31
4.6 Logiscope TestChecker Actions 33
4.7 Logiscope Dynamic Viewer Actions 34

4.1 Prepare for Analysis

Configure the Project for Analysis

1. Setup SRT
2. Go to your working directory: `cd ~/work`
3. Check out MRS: `cvs co -d mrs backend/mrs`
4. `cd mrs`
5. Configure for Logiscope: `configure --with-tool=logiscope`
6. Go to the target host directory: `cd .srt/$SRT_TARGET`

Caution
Make sure you **do not** have write perminition to the SRT installation directories otherwise some Logiscope actions may generate files under that structure.

Run the Analyzers

If you want to perform static analysis only:

```
gmake logiscope-static
```

This generates the static result files, the archive static result files and the application files for each makefile target.

If you also plan to perform the dynamic analysis use this command instead of the previous:

```
gmake logiscope-dynamic
```

This generates the same files than the previous static command, plus the instrumented source files. It also compiles the instrumented libraries and executables. Both the interactive and batch mode instrumented executables are generated.

Run the Dynamic Tests

If you are performing dynamic analysis, run the various tests using the batch mode instrumented executables.

Caution 1

The various binaries should be started from their own subdirectory. (e.g. to start `MRSserver` you should first go to the `examples` subdirectory. This is to ensure: 1-you get the instrumented version and not the public version from the SRT installed directories, 2- the dynamic trace files are generated in the correct location.

Caution 2

In order to use the instrumented library update your `LD_LIBRARY_PATH` to point to local directories and not to the public version from the SRT installed directories. For example in this case you should point to `~/work/mrs/.srt/$SRT_TARGET/src` in order to use the instrumented `libmrs.so`.

Post Analysis Phase

If you are performing static analysis only run:

```
gmake logiscope-static-merge
```

This traverses the directory tree and generates the various archive static result files and application files for each directory.

If you are performing dynamic analysis:

```
gmake logiscope-dynamic-merge
```

This generates the same files than the previous static merge command, plus it converts all dynamic trace files to execution trace files and generates the various archive execution trace files (both for makefile targets and directories).

4.2 View Static Results

As we have seen in Chapter 2, the integration is such that one can analyze the results at different levels: the project, a sub directory, a makefile target or even individual files (we do not address this last possibility).

View Entire Project

The top level application file and archive files (`mrs.app/.arc.mtr/.arc.res`) contain all the results for the entire project. To view them start:

`cppcchk -app mrs.app`*.

View a Directory

If you are only interested in viewing the results of a sub directory instead of the entire project, use the application file and archive files of that directory. They contain all the results of the targets under that directory plus the results on any sub-directory.

Example 1 The `examples/` subdirectory contains 4 targets: `MRScommander`, `MRSreceiver`, `MRSsender` and `MRSserver`. By starting `cppcchk -app examples.app` you will get the results of all 4 targets displayed.

View a Makefile Target

If you are interested to view the results of a target independently of any other target and sub-directory, use the application file and archive files of that target.

Example 2 In the `examples/` subdirectory, the `MRSserver` target can be analyzed independently by starting `cppcchk -app MRSserver.app`.

Example 3 The `src/` subdirectory contains only one target and no subdirectory. So the archive files of the target contain exactly the same results as the archive files of the directory. In this case `cppcchk -app src.app` is equivalent to `cppcchk -app libmrs.app`.

* MRSserver is a C++ application. For a C project replace `cppcchk` by `cchk` in all the examples.

4.3 Logiscope CodeChecker Actions

When started, Logiscope CodeChecker displays the different categories of the quality factor. The number next to a category represents how many components are in that category. A component can be a function, a class or an application depending on the quality model.

1. Click on the triangle next to a quality factor name to expand it. This will display the list of the components in the category.
2. Double click on a component. This will display a new window with the values of the different quality criteria for that component.
3. Again click on the triangle next to a quality criteria. This will display the names and values of the metrics measured on the component. The metrics which values are outside the ranges defined in the quality model are highlighted in red.
4. Double click on a quality metric to expand the window. The right part of the window now displays the definition of the metric, an explanation of what it actually measures and some improvement hints.

Caution
If you identified a function to improve, use the hints as first indications. There is no general rule that can be applied in all cases.
It is too dangerous to concentrate at the individual metric level. Trying to have all metrics within the ranges defined in the quality model may cause you to loose the overall picture. Your goal is to improve the quality of your application and not of all individual functions.

5. You can switch to another quality model with the `View` menu.
6. You can load a different quality model file. See Section 3.3 for more details.
7. Start the Logiscope GUI by clicking on .
8. Exit the Logiscope CodeChecker with: `File->Exit`.

4.4 Logiscope Static Viewer Actions

When started, Logiscope GUI displays the list of all the functions in the application.

Global Information

The `Global` menu gives access to overall information about the set of components being studied.

1. Ask for the quality factor with `Global->Quality Report`. This displays a pie chart representing the different categories and the percentage of functions in each category.
2. You can view various textual information about the diagram with:

   `View->Textual Representation`: list of components in each category,

   `View->Info`: general information about the analyzed data,

   `Edit->Properties`: detailed information on each category.

   These menu items are normally accessible for the various diagrams.
3. The overall Kiviat graph (`Global->Overall Kiviat Graph`) shows in a synthetic way the metric situation. This diagram and its tabular representation (`View->Textual`) are useful tools to identify overall problems.
4. You can view the detailled distribution of the functions for each metric with: `Global->Metric Distribution... .`

   a. When the `SelectionDialog` window pops-up select a few or all the metrics. The first metric distribution graph is displayed.

   b. The same set of textual information used in 2 are applicable here.

   c. Use the `Next/Previous/Go To...` buttons to navigate to another metric distribution graph.
5. You can view the quality criteria distribution with: `Global->Criteria Distribution... .`

   The same tools as in 4 are accessible here as well.

Function Information

The `Component` menu gives access to information for every function.

1. `Component->Control Graph` displays the control graph of a function. A square is a set of instructions with no break. Other decisions and loops are modeled with diamonds, triangles and other symbols.
2. If the source file is displayed (`Edit->Source File`), clicking on any node on the control graph will indicate (with a `==>`) the corresponding source line.
3. `Component->Kiviat Graph` displays the same type of graph as the overall Kiviat graph but limited to only the current component.
4. `Component->Criteria Graph` displays the criteria for this component in

the form of a Kiviat graph as well.

Call Graph Information

1. `Global->Call Graph` displays the overall call graph. A node is a function (internal or external). The links are function calls. Recursions are shown in red and level skips in purple.
   a. In case the boxes are too small, the name and other information on a function can be displayed with: `Edit->Properties`.
   b. `Edit->Source File` pops-up the source browser for internal functions.
2. The graph can be modified using the `Graph` menu.

   For example, to remove external functions do: `Graph->Group On File...`, select `<Externals>` in the `File Select` window, click on `Select`. Click on the node `<External>` in the call graph, select `Graph->Remove Node`.
3. Specific call graph measurements are available for each function at this stage.
   a. Select a node on the call graph. This node will be the current root.

      From that point on the call graph measurements will apply to the relative call graph of the current root. You can display this relative call graph with: `Component->Relative Call Graph->Calling/Called.`
   b. `Component->Relative Call Measurements->Accessibility Table` and `Testability Table` help you measure the effort needed to reach a particular function from the current root.
   c. `Component->Relative Call Measurements->Call Kiviat Graph` and `Call Criteria Graph` will display in turn the call graph metrics and criteria for the current relative call graph.
4. If all your classes are only contained in one file, you can display the class call graph by: go back to the global call graph with `Global->Call Graph`, do `Graph->Group On File`, click on `All`. You can then group together the header and the source files. Click on `[messagetrue.h]` and `[messagetrue.cc]`, go to `Graph->Group`, type `messagetrue` in the text area and click `OK`.

Workspace and Selection

Example 1: To see where the functions rated as poor are located on the overall call graph:

1. Display the quality factor pie chart with `Global->Quality Report.`
2. Select the poor category by clicking on `REWRITE`.
3. Display the overall call graph with `Global->Call Graph`.
4. Select the functions rated as poor with `Edit->Select->From Previous View`.

Example 2: To identify a general problem in a set of functions, you can display the overall Kiviat graph for those functions only.

1. Display the quality factor pie chart with: `Global->Quality Report`.
2. Select the poor category by clicking on `REWRITE`.
3. Reduce the workspace to the poor functions selected with `Edit->Current Workspace->Set`. The pie chart now displays 100% `REWRITE`.
4. Give the workspace a name with `Edit->Component Workspaces`, type `poor` in the `Workspace Name` field, click `New`, select `poor` from the `Workspace List`, click on `List`, then on `Add Cur` and finally on `OK`.

From that point on all the global actions apply to the current workspace which consists of the poor category functions.

5. Display the overall Kiviat graph for the functions selected above with: `Global->Overall Kiviat Graph`.

To come back to the original workspace with all the functions go to `Edit->Current Workspace->Reset`.

To change to a different workspace go to `Edit->Component Workspaces`, select a workspace from the `Workspace List`, then click on `Set Cur`.

Save a Workspace

It is possible to save the current workspace (the one you are in) on disk with `Edit->Current Workspace->Save Cur`, enter a file name (`.ws`) in which to save the workspace information and click `OK`.

This is practically interesting when the workspace selected is large and built after a succession of selections.

It is possible to load a saved workspace from disk with `Edit->Current Workspace->Open Cur`, enter the file name of the workspace to load and click `OK`.

Save a State

It is possible to save the current state of the Logiscope GUI on disk with `File->State->Save`, enter a file name (`.st`) in which to save the state information and click `OK`.

This is extremely useful because when opening a saved state both the workspace in use and the view displayed are restored.

It is possible to restore a saved state of Logiscope GUI from disk with `File->State->Load`, enter the file name of the state to load and click `OK`.

Exit the Logiscope GUI with: `File->Exit`.

4.5 View Dynamic Results

As it was the case for static result analysis (c.f. Section 4.2) the dynamic results can be analyzed at different levels: the project, a sub directory, a makefile target or even individual files (we do not address this last possibility).

View Entire Project

The top level application file and archive files (`mrs.app/.arc.mtr/.arc.res/.arc.rex`) contain all the test coverage results for the entire project. To view then start:

`cpptchk -app mrs.app`*.

Caution
Always answer `no` when Logiscope TestChecker wants to build a binary. They have already been generated previously.

View a Directory

If you are only interested in viewing the dynamic results of a subdirectory instead of the entire project use the application file and archive files of that directory. They contain all the dynamic results of the targets under that directory plus the dynamic results on any subdirectory.

Example 1 The `examples/` subdirectory contains 4 targets: `MRScommander`, `MRSreceiver`, `MRSsender` and `MRSserver`. By starting `cpptchk -app examples.app` the test coverage results of all 4 targets are displayed.

View a Makefile Target

If you are interested in viewing the test coverage results of a target independently of any other target and subdirectory, use the application file and archive files of that target.

Example 2 In the `examples/` subdirectory, the `MRSserver` target can be analyzed independently by starting `cpptchk -app MRSserver.app`.

* MRSserver is a C++ application. For a C project replace `cpptchk` by `tchk` in all the examples.

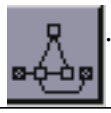
Note

The `src/` subdirectory contains only one target and no subdirectory. More over, the target under this directory is a library. So neither the target nor the directory have an archive execution trace file for analysis (c.f. Chapter 2). The dynamic results of libraries are stored in the dynamic results files of the executables they are linked with. In this case `examples/MRSserver.arc.rex` contains information on which part of `libmrs.so` was covered in the different executions on `MRSserver`. Consequantly `examples/examples.arc.rex` contains information on which part of `libmrs.so` was covered in the different executions of `MRSserver`, `MRSreceiver`, `MRScommander` and `MRSsender`.

4.6 Logiscope TestChecker Actions

When started, Logiscope TestChecker displays the different files that contain code. The number next to each file represents the DDP coverage ratio.

1. Click on the triangle next to a file name to expand it. This displays the list of the functions in the file with its DDP coverage ratio.
2. Double click on a function. This displays a new window with the DDP coverage break down per test case. It also displays the number of DDP still to be covered.
3. `Window->Test List` displays all the test cases ran on the application with the DDP coverage reached in each case.
4. Run a new test interactively:
   a. Go to `Window->DDP Spy`. The first bar shows the DDP coverage for all the tests. The second bar shows the DDP coverage for the last test only. The third bar shows the DDP coverage gain (i.e. the ratio of code covered in the last test compared to all the previous tests).
   b. Switch on the `Option->Keep LCSAJ` option.
   c. Go to `Execute->Start New Test...`, click on `Search...`, locate and select the interactive executable you want to test (`.inter`), fill the `Parameter` field if needed, click on `OK`.

      The `DDP Spy` window is updated dynamically to display the DDP coverage status and a new entry is added to the `Test List` window.
5. Start the Logiscope GUI by clicking on  .

Caution

If you have run new interactive tests, Logiscope will ask you if you want to save the new results. If you answer `no` all the interactive test data will be **lost**. If you answer yes the interactive test data will be merged with the archive execution trace file associated with the application file you loaded in Logiscope TestChecker.

6. Exit the Logiscope TestChecker with: `File->Exit`.

4.7 Logiscope Dynamic Viewer Actions

The Logiscope dynamic viewer interface is very similar to the Logiscope static viewer that we have see in Section 4.4. We will cover only the part specific to the dynamic analysis.

Global Information

The `Global` menu gives access to overall information about the set of components being tested.

1. `Global->Call Graph` displays the PPP coverage results as a global call graph similar to the one shown in Section 4.4. The black lines are the function calls that have been actually performed at least once. The red dotted lines are function calls that have not been performed in any of the test cases.
2. IB, DDP and LCSAJ coverage results are displayed with the corresponding `Global` menu items. Each of them shows the coverage distribution of the functions. The list of the components in each category can be obtained with `View->Textual Representation`.

Function Information

The `Component` menu gives access to information for every function.

1. The control graph displayed by `Component->Control Graph` is similar to the one shown in Section 4.4. Here the instructions that have not been executed in any of the test cases are highlighted with a dotted red line.

Test Workspace

So far we have looked at the cumulative coverage results of all the test cases. It is possible to analyze one or a limited set of test case results only.

1. Go back to the global call graph with: `Global->Call Graph`.
2. Display the list of test cases with `Edit->Test Workspace...`.
3. Select one or more of the test cases from the list, click `Apply`.

The call graph is updated and only the function calls actually performed in the selected test cases remain in black.

The same operation is possible with a function control graph.

Check Item
Check Section 4.4 as it describes general purpose actions which are also applicable here. (e.g. call / control graph operations, workspace creation, state save / restore.

Appendix A

SRT Integration Requirements

This chapter lists the requirements for the integration of Logiscope in SRT. This completes the description in Chapter 2.

A.1 List of Requirements 36

A.1 List of Requirements

Static Result Generation per Makefile Target

UR-1. Static result files, block count files, instrumented source files, archive static result files and application files must be generated by running the compilation and replacing:

- for a C++ application the compiler and linker by `log_c++`.
- for a C application the compiler and linker by `log_cc`.

UR-2 For each makefile target and source file `log_c++` and `log_cc` must be ran from the `$SRT_TARGET` sub-directory corresponding to the source directory.

UR-3 `log_c++` and `log_cc` must be passed as options and parameters:

- `-inst` (only if interested in dynamic results)
- `-ddef <path to the ATLAS DAQ definition file>`:

`log_c++_sun4os5_gnu27.def` for GNU 27 C++ on Solaris.

- `-dial <language dialect>`
- `-app <full directory path of the application file >/<target basename>.app`
- `-apc <target basename>.apc`
- `-header`
- `-cb`
- all the compilation/link options and parameters used to compile the source file.

UR-4 The ATLAS DAQ defintion file is an adaptation of the definition file delivered by Logiscope but customized to reflect the compiler and system setting in ATLAS DAQ.

UR-5 The source file name used in the above command must be given in full path.

Static Result Generation per Directory

UR-6 The archive static result files and application files for each directory must be generated with:

- `log_ld [<target basename>.arc.res]* [<sub-directory>/<sub-directory>.arc.res]* -o <directory>.arc.mtr`
- `log_ld [<target basename>.arc.mtr]* [<sub-directory>/<sub-directory>.arc.mtr]* -o <directory>.arc.mtr`
- `log_app_ld -o <directory>.app [<target basename>.app]*`

`[<sub-directory>/<sub-directory>.app]*`

UR-7 The above steps must be performed for each directory recursively expect for the top `$SRT_TARGET` directory. In that case the results must be generated with:

- `log_ld [<target basename>.arc.res]* [<sub-directory>/<sub-directory>.arc.res]* -o <CVS module name>.arc.res`
- `log_ld [<target basename>.arc.mtr]* [<sub-directory>/<sub-directory>.arc.mtr]* -o <CVS module name>.arc.res`
- `log_app_ld -o <CVS module name>.app [<target basename>.app]* [<sub-directory>/<sub-directory>.app]*`

UR-8 `log_ld` and `log_app_ld` must be run from the corresponding directory.

Instrumented Target Generation

UR-9 For each C++ instrumented executable a separate Logiscope batch trace function file must be used which:

- uses the ATLAS DAQ `vlgins_lcsaj.C` as template
- replaces the `_ATD_BIN_TARGET_ATD_` string by `<target basename>Result`.

UR-10 For each C instrumented executable a separate Logiscope batch mode trace function file must be used which:

- uses the ATLAS DAQ `vlgins_lcsaj.c` as template
- replaces the `_ATD_BIN_TARGET_ATD_` string by `<target basename>Result`.

UR-11 The ATLAS DAQ `vlgins_lcsaj.C` and `vlgins_lcsaj.c` are copies of the files delivered with Logiscope, but where:

- `MAX_FILE_ESSAI` is increased from 10 to 1000
- `FICNOM1` and `FICNOM2` file basenames was replaced from `result` to `_ATD_BIN_TARGET_ATD_`.

UR-12 Instrumented libraries must be called the same way as non instrumented libraries.

UR-13 For each executable two instrumented executables must be generated:

- the batch executable called the same way as the non instrumented executable.
- the interactive executable called the same way as the non instrumented executable plus the `.inter` extension.

UR-14 The compilation of instrumented targets must be the same way as non instrumented targets plus:

- an additional include flag to the logiscope directory containing the

`vlgins.h` include file.

- use the instrumented source files in place of the original source files.

UR-15 Batch executables must be linked with their Logiscope batch trace function file (prepared as explained in UR-9 and UR-10).

UR-16 C++ Interactive executables must be linked with the Logiscope interactive trace function file called `vlgins_log2.C`.

UR-17 C Interactive executables must be linked with the Logiscope interactive trace function file called `vlgins_log2.c`.

Dynamic Result Generation per Makefile Target

UR-18 Every batch executable must be run from its subdirectory.

UR-19 Every dynamic trace file must be converted to execution trace file with:

- `log_format -t “<dynamic trace file basename>” -x <dynamic trace file basename>.dyn -f <dynamic trace file basename>.rex`

UR-20 For each batch executable the archive execution trace file must be generated with:

- `log_arc_dyn -o <target basename>.arc.rex -m [<target execution trace file>]*`

Dynamic Result Generation per Directory

UR-21 The archive execution trace files for each directory must be generated with:

- `log_arc_dyn -o <directory>.arc.rex -m [<target basename>.arc.rex]* [<sub-directory>/<sub-directory>.arc.rex]*`

UR-22 The above steps must be performed for each directory recursively expect for the top `$SRT_TARGET` directory. In that case the results must be generated with:

- `log_arc_dyn -o <CVS module name>.arc.rex -m [<target basename>.arc.rex]* [<sub-directory>/<sub-directory>.arc.rex]*`

UR-23 `log_arc_dyn` must be run from the corresponding directory.

Appendix B

Known Problems

This chapter lists the known problems with the integration. For each problem a description and a workaround if any is provided.

B.1 List of Known Problems 40

B.1 List of Known Problems

General Problems

KP-1 Top level archive files names

Description: all top level archive files and application file (c.f. Chapter 2) are named with the `$SRT_TARGET` basename instead of the package name.

Example: the top level archive files of MRS are called `g++-2.7.2.arc.mtr/res/rex` instead of `mrs.arc.mtr/res/rex` and the application file is called `g++-2.7.2.app` instead of `mrs.app`.

Workaround: Use the new file name for the time being.

Action: Lassi is waiting a fix to the make fragment by Igor to make it work properly.

KP-2 Problem in include statement

Description: On some packages the include statement looks as if some strings have been concatenated and the `-I` dropped.

Symptom: Logiscope complains not finding include files.

Example: running gmake on Run Control. See with Bob and Igor for more details.

Action: Correction in SRT.

Problems Specific to Dynamic Analysis

KP-3 `gmake logiscope-dynamic` does not behave as `gmake`

Description: the default tag is not `logiscope-dynamic`

Symptom: running `gmake logiscope-dynamic` does not produce the instrumented executables for batch mode but only for interactive.

Workaround: use `gmake` instead of `gmake logiscope-dynamic` waiting for the correction in SRT.

Action: Correction in SRT.

KP-4 Two header files with the same name

Description: If the package contains a header file which has the same name that an external package header file that one of its files includes, then the dynamic analysis of Logiscope is confused.

Example: MRS and Tools.h++ have both a header file called `message.h`. They are included with `#include “mrs/message.h”` and `#include “rw/message.h”`.

Symptom: At the instrumentation stage (`gmake logiscope-dynamic`) complains that it can not access the external package instrumented header file to which it does not have write access anyway. In the example above it tries to access `rw/message.inst.h` under the SRT installed

area which is write protected.

Origin: This seams to be a bug in Logiscope which has been reported to Verilog.

Workaround: It is best to rename the local package header file to something specific to the package. In the example above it could be `mrsmessage.h`.

Action: Waiting for a fix by Verilog (call # 13186).

KP-5 Problems in the ILU C++ stubber generated header files

Description: The instrumented files do not compile.

Symptom: The C++ compiler complains about the syntax of the instrumented version of the ILU generated header file. Syntax error du to a missing include statement in the instrumented version, but present in the original version.

Example: In MRS, the ILU generated file `src/mrs/mrs.hh` contains the following include statement `#include <ilu/ilu.hh>` which is by mistake removed by the Logiscope instrumentation process from `src/mrs/mrs.inst.hh`.

Origin: This seams to be a bug in Logiscope which has been reported to Verilog.

Workaround: Modify your `GNUmakefile.in` in order to remove the condition directives around the disappearing include statement in the ILU generated header file. In the example above the following lines have been modified `in src/GNUmakefile.in`:

```
mrs/mrs.hh: mrs.cc FORCE
     sed '/^#ifndef.*ilu/d' mrs/mrs.hh > ./junk1
     sed '/^#endif.*ilu/d' ./junk1 > ./junk
     rm -f mrs/mrs.hh
     rm ./junk1
     mv ./junk mrs/mrs.hh
```

Action 1: Waiting for a fix by Verilog (call # 13185).

Action 2: Configure SRT to ignore the generated files.

References

1 **ATLAS DAQ Software Development Environment.**
`http://atddoc.cern.ch/Atlas/DaqSoft/sde/Welcome.html`

2 **SRT - Software Release Tool**
`http://wwwcn.cern.ch/~lat/exports/srt/manual.ps.gz`

3 **SDT - Software Development Tools Service**
`http://www.cern.ch/PTTOOLS/`

4 **Logiscope at CERN**
`http://www.cern.ch/PTTOOLS/Logiscope`

5 **Verilog**
`http://www.verilogusa.com/`

6 **Characteristics of Software Quality,**
TRW, 1975, B.W.Boehm

7 **Factor in Software Quality**
General Electric n.77C1502, June 1997, J.A.McCall

8 **Information technology - Software product evaluation - Quality characteristics and guidelines for their use,**
ISO / IEC 9126, 1991-12-15

9 **Design of the Message Reporting System for the ATLAS DAQ Prototypr-1,**
D.Burckhart et al., ATLAS DAQ Technical Note 032,
http: / / atddoc.cern.ch / Atlas / Notes / 032 / Note032-1.html

10 **Users Guide and Implementation of the Message Reporting System for the ATLAS DAQ Prototype-1,**
D.Burckhart et al., ATLAS DAQ Technical Note 059,
http: / / atddoc.cern.ch / Atlas / Notes / 059 / Note059-1.html

Index