

Two Weeks to Tapeout: Do You Know Where Your Files Are?

Andrea Casotto
Runtime Design Automation
3155 Kearney Street, Suite 130
(510) 770-0300
info@rtda.com

Abstract

A technique called “runtime tracing” allows the design team to keep under control the thousands of files necessary to execute a large hardware or software project. This technique builds a complete graph of all the dependencies and supports automatic documentation of the design flow, network computing, and intelligent propagation of changes. The benefits of this technique are illustrated with data from real-world projects.

1. INTRODUCTION

The adoption of advanced Computer Aided Design (CAD) technology has transformed the design process for complex hardware and software systems into an elaborate collection of thousands and thousands of interdependent files. Some files are the creative product of designers, the others are the computed outputs of CAD tools. The complex dependencies among the files, if not managed, may cause uncertainty about the validity of the data. The pressure to maintain the competitive edge and the urgency expressed by impending deadlines call for a lean and safe management of the design files.

Runtime tracing is a new technique, developed at Runtime Design Automation, for the efficient management of the design flow. Runtime tracing automatically captures the detailed dependencies between the design files: it allows the design team to know exactly which files are in the design and how they are related to each other. If a last minute change is necessary, the team knows exactly how long it will take to propagate that change to all the dependent files. The design flow is executed accurately and in parallel on all the available computers in the network.

A brief mention of related work is presented in Section 3. The basics of runtime tracing are presented in Section 4. Section 5 through 8 talk about the main features of runtime tracing.

2. DEFINITIONS

We focus on the relationship between the tools and the files used in a design flow. A *tool* may be a CAD tool such as a static timing analyzer, a logic synthesizer, a netlist converter, or utilities like `cp`, `awk`, and `perl`. A *job* is a particular invocation of a tool, characterized by attributes such as its command line, its working directory, and environment. Each job is also characterized by a set of input and output files. A *flow* is a collection of jobs and their associated files. A *primary input* file is one which is not created by any job in the flow, for example, a Verilog file written by an engineer.

3. RELATED WORK

Most design flows are currently managed using either special purpose scripts or the UNIX utility `make`, neither of which can guarantee a complete representation of the dependencies, as explained in Section 6. A system that has some features of runtime tracing is ClearCase by Rational. By controlling access to the filesystem, ClearCase creates a “bill of materials” for each derived object in the design.

However, this approach does not work with the design files that reside on filesystems that are not managed by ClearCase. Runtime tracing is not limited in this way. ClearCase collects the information passively, while runtime tracing notifies the designers if they are about to use files that are outdated.

Many other practices are required to keep the design on the fast track:

- Use a modern revision control system; RCS or CVS which are in the public domain, DesignSync from Synchronicity, or SOS from ClioSoft, or ClearCase from Rational.
- Establish a well defined design discipline; typically each designer has his or her own workspace, in which files are checked out; other workspaces are used to integrate the work of the team so as to arrive at a consistent and complete set of deliverables.
- Use the latest and most powerful CAD tools.
- Use the latest and most powerful computers.
- Use network computing software; in-house solutions are still common, and commercial products such as Flowtracer/NC from Runtime Design Automation (<http://www.rtda.com>), and LSF from Platform Computing Corp. (<http://www.platform.com>) are also available.
- Get the smartest designers and give them incentives to work hard.

These practices are complementary to runtime tracing and, for the most part, well known and widely adopted.

4. CAPTURE ALL DEPENDENCIES

If a tool reads a file, the file is an input. If the tool writes a file, the file is an output (Figure 4.1).

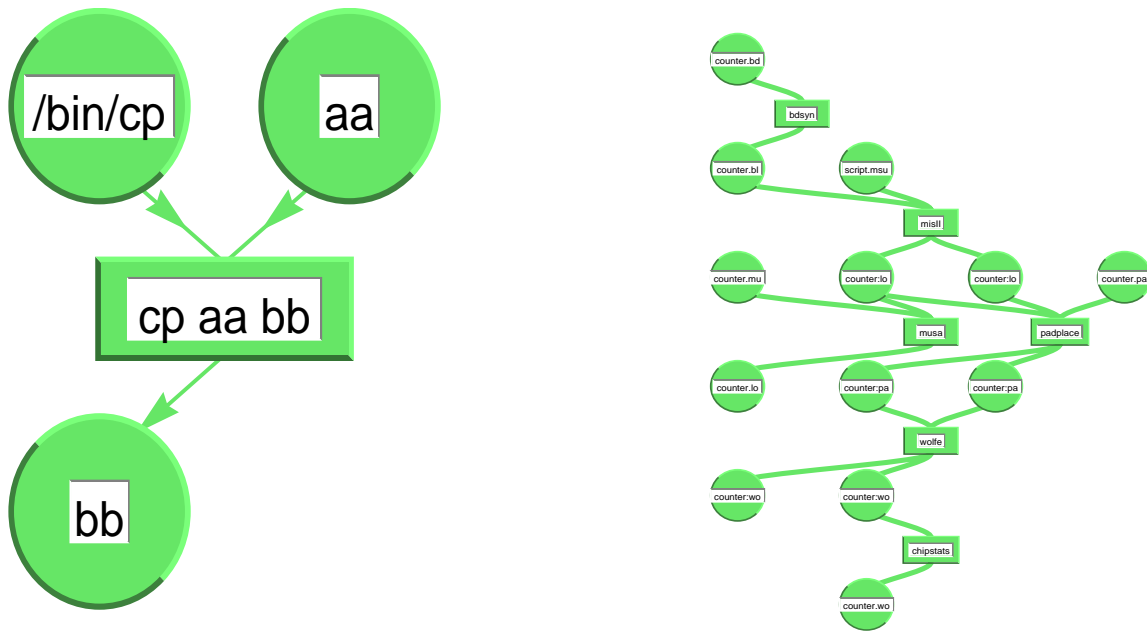


Figure 4.1 Left: The basic component of the dependency graph: circles represent files and rectangles represent jobs. Right: Example of a simple flow.

The output files depend on the input files, because, if the inputs change, the outputs must be recomputed to recover the consistency of the design. The detailed list of inputs and outputs is obviously known to the tool itself at runtime. The idea of runtime tracing is to extract such knowledge from the

tools at runtime, that is to allow the tools to communicate at runtime with the *runtime design management system* (RDMS). For each tool, three “integration” techniques are available (Figure 4.2):

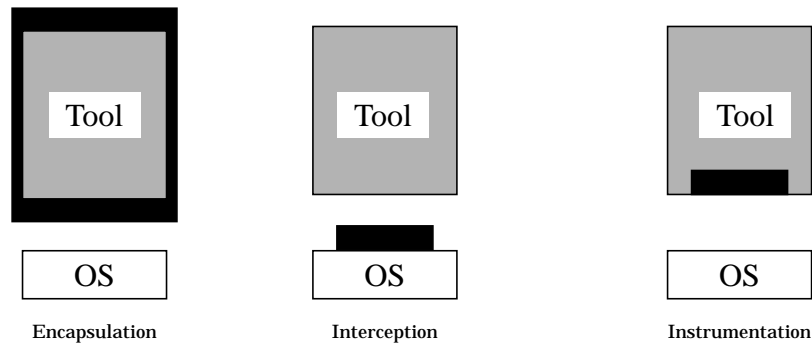


Figure 4.2 Graphical representation of the three integration techniques: the black areas represent the software modules responsible to send input/output messages to the runtime design management system.

- (1) **Runtime Encapsulation** consists of writing a script that computes the I/O list of the job at start time, communicates the list to the RDMS, and then executes the job. Encapsulation is powerful because it does not require any change to the tool. The encapsulation effort depends on the complexity of the tool.
- (2) **Runtime OS-level Interception** consists of trapping all calls to `open()` and to a few other OS procedures associated with file I/O. The trapping routines interpret the arguments to derive the I/O behavior of the calling tool. This technique relies on the dynamic linking technology offered by modern operating systems. This approach is easy, accurate, but it is not equally effective for all operating systems and may not work with a minority of tools that are statically linked.
- (3) **Source Instrumentation** consists of inserting special calls into the source code of the tool, so that the tool sends appropriate messages to the RDMS each time it is about to open a file. This is the most complete embodiment of runtime tracing and it opens the possibility of enhancing the behavior of the tool through its interaction with the RDMS, as explained in Section 8. Instrumentation is applicable to scripts as well as binary executables.

With any of these integration techniques, the tool, upon opening a file, sends a message to the RDMS to declare that said file is either an input or an output. For an input file, the RDMS checks whether the file is up-to-date, and reports the result to the tool. For an output file, the RDMS checks whether the tool is allowed to overwrite it. At the same time, the RDMS uses the information provided by the tools to build the dependency graph (Figure 4.1).

The net effect of runtime tracing is that the dependency graph is built and maintained as a side effect of executing the tools. Each time an integrated tool is executed, all its dependencies are recomputed. In this way, the dependencies are always complete, correct, and up-to-date, including the dynamic dependencies caused by changes in the input data. Dynamic dependencies occur commonly in design. Since the tools must be executed at least once in order to complete the design, we can say that the cost of building the dependency graph is zero.

Whether runtime tracing should be active all the time or only on demand is a policy issue, to be decided by the design team. It is our experience that designers prefer activation on demand.

5. FLOW DOCUMENTATION: WHERE ARE THE FILES?

The consistent use of integrated tools guarantees that the dependency graph completely and accurately

ly documents the design activity, in that it represents all the jobs that have been invoked and all the files used in the design. If a file is opened by an integrated tool, it appears in the dependency graph. Files that are created by the designers but never used by any tool do not appear in the dependency graph.

The files in the dependency graph are **valid** if they are either primary inputs or the outputs of successful jobs. If a file changes, all the dependent files and jobs are automatically labeled as **invalid** (Figure 5.1). The consistency of the design data can be revealed by the ratio of valid vs. invalid files in the graph.

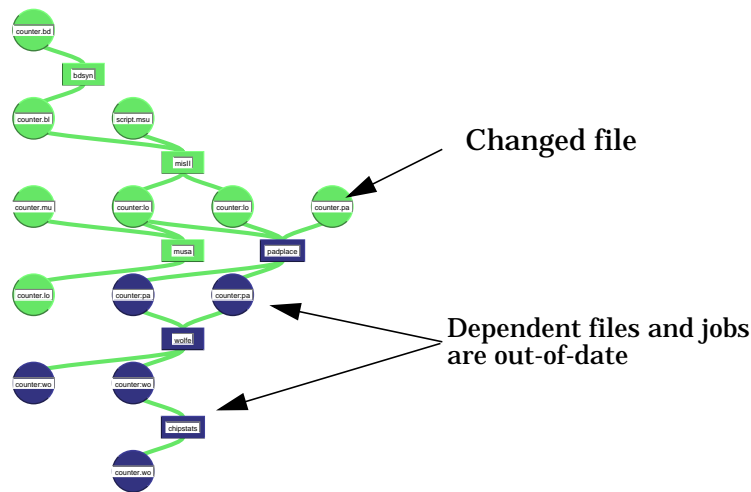


Figure 5.1 When a file changes, all the dependent files and jobs are labeled as invalid. In this case, the file *counter.pads* has changed.

With simple traversals of the graph, the RDMS can:

- Show the exact history of each file; how it was made, by whom, when, how long it took;
- Find out why a file is not up-to-date; is it because a tool failed, or because a transitive input has been changed?
- Compute the impact of a change to a particular file, that is the list of all the other files that are potentially affected by the change and the list of jobs that would need to be re-executed. By keeping a historical record of the duration of each job, the RDMS can give an estimate of how much work would be required to bring the design back to consistency.

6. FLOW AUTOMATION

The ability to follow designers in the creative exploration of the design space is a unique feature of runtime tracing. Though critical early in a project, the need to be creative in the execution of the design flow is not needed once the flow is in place. In the daily routine, the tools need to be executed using well defined methodologies, which have in the past often been implemented by means of shell scripts or of *makefiles*.

The rest of this section illustrates the difference between scripts, *makefiles* and runtime tracing by means of a concrete example: suppose that a designer needs to run a suite of hundreds of tests using a Verilog simulator; each test consists of two jobs: prepare and run; the command lines of the jobs differ only in the name of the test to be run.

A **script** like the one in Figure 6.1 can be used for this purpose. Scripts are easy to program and

```

foreach test ($LIST_OF_ALL_TESTS)
    prepareTest -com $test.vr
    runTest $test.vro > $test.log
end

```

Figure 6.1 This script can be used to execute a suite of tests in sequence. The script always runs all the tests and does not support the notion of dependencies.

offer convenient handling of tool sequences with powerful support for loops and conditionals. On the other hand, scripts offer no support for dependency management.

With a **makefile**, designers can express dependencies and rules, with limited ability to express loops and conditionals. When some input files are modified, **make** computes which tests need to be run based on the dependency rules described in the **makefile**. It is common for the rules in a **makefile** to be either incomplete or incorrect. For example, the **makefile** in Figure 6.2 only shows the dependencies for the files with suffixes “.vr” “.vro” and “.log,” while any real Verilog simulator certainly uses several more files to accomplish its job, occasionally because of file I/O encoded into the Verilog model itself. Because of these unrepresented dependencies, it is possible that changes to some design files will not be detected by **make**, which will then fail to run the appropriate tests. To be on the safe side, designers often clean-up their working directories before invoking **make**, forcing the execution of all the jobs and wasting CPU cycles, when running just a few jobs would be enough to regain complete design consistency.

```

all: $(LIST_OF_ALL_TESTS:.vr=.log)
.vr.vro:
    prepareTest -com $<
.vro.log:
    runTest $< $%

```

Figure 6.2 This makefile expresses explicitly, for each test case, the dependencies between the .vr, .vro, and .log file. On a test by test basis, there are other dependencies but they are difficult to express in a makefile.

Complex designs are often divided into subdirectories, with multiple **makefiles**, each one responsible for a small part of the dependencies. The result is the lack of a global representation of all the dependencies, which may lead to redundant or erroneous tool executions. When using the Flowtracer RDMS, the complete dependency graph is always available, though it may be created incrementally by multiple Flow Definition Files. This is in contrast with **make**, where putting all dependencies in a single **makefile** makes maintenance difficult.

With **runtime tracing**, designers need not be concerned with the details of the dependencies, since these are captured automatically at runtime. Rather, designers can concentrate on expressing their design intention.

Figure 6.3 shows a script, written in Tcl syntax (see <http://www.scriptics.com>), that describes the example design flow to an RDMS. The tokens **T**, **I**, and **O**, are procedures which are provided by Flowtracer. The dependencies described in the flow are the same “easy dependencies” as those described in the **makefile** of Figure 6.2. When executed, the Tcl script produces the flow shown on the left in Figure 6.4.

After the jobs have been executed, the flow is completed to include all the actual dependencies, as shown on the right in Figure 6.4.

```

foreach test $LIST_OF_ALL_TESTS {
  T prepareTest -com $test.vr
  I $test.vr
  O $test.vro
  T runTest $test.vro > $test.log
  I $test.vro
  O $test.log
}

```

Figure 6.3 A Tcl script to represent the methodology. The dependencies expressed here need not be complete, because they will be recomputed at runtime.

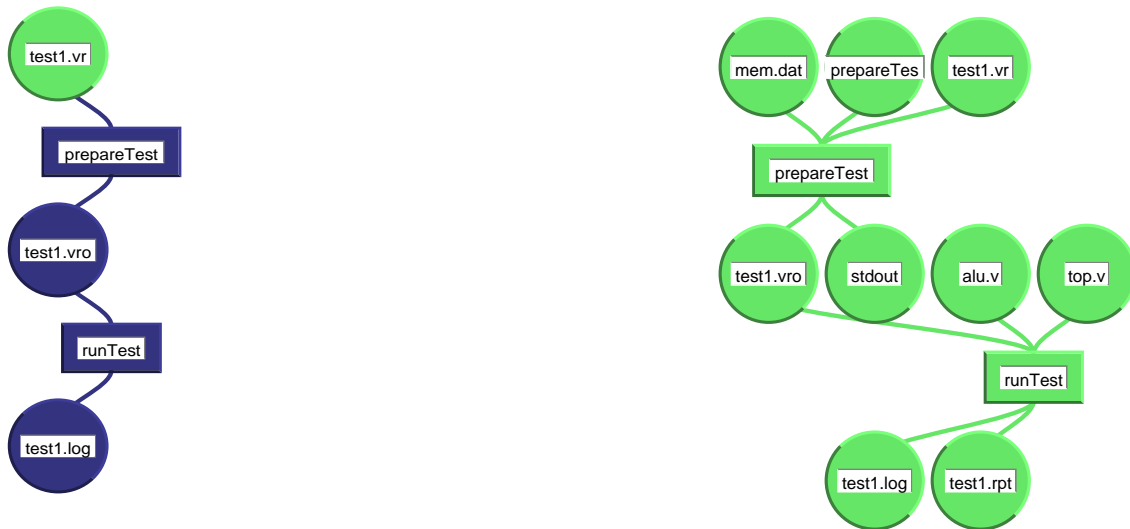


Figure 6.4 Left: the graph for a single test case as described in the Tcl script. Right: the graph for the same test case after the tools have been executed; several additional dependencies have been captured at runtime. This example, derived from a real flow, has been simplified for the benefit of this explanation.

Real example: A design team in Company A prepares millions of vectors to test its chips coming out of the fab. For some of the most complex chips, the flow to prepare the vectors consists of more than 35,000 files and more than 12,000 jobs. The flow, based on a combination of scripts and makefiles, used to take 17 hours. After reimplementing the flow using runtime tracing, it was discovered that many makefiles were executing the same jobs as some other makefile. Through detailed and automatic analysis of the dependencies, runtime tracing allowed the implementation of a flow in which those jobs are executed just once.

As an experiment, we modified a file called `constants.inc`, which would have triggered a complete 17 hour recomputation had scripts and makefiles been used. The runtime tracing system computed that, in reality, the amount of work to be done was only 4 hours. Using 6 computers, the flow was executed in a little over an hour: a gain of almost 17.

7. FULLY UTILIZE COMPUTING RESOURCES

One obvious way to speed up the design process is to make more computing resources available. Several commercial packages offer network computing capabilities and are effective in handling large numbers of independent jobs, but they offer only limited support for collections of jobs that are depen-

dent on one another, as in the case of complex design flows. Other products based on `make` address the problems of dependencies, although with all the limitations of `make`, but are weaker in terms of management of computing resources.

The dependency graph captured by runtime tracing is ideal for scheduling tool execution on a network of computers. For example:

- Jobs that are ready to execute are easily identifiable as those invalid jobs whose inputs are all valid;
- Parallelism is clearly represented;
- Historical information about the tool execution collected at runtime, such as tool duration and resources usage, can be used to guide the scheduling of jobs, so that longer lasting jobs are scheduled early;
- Impact analysis on each job allows the scheduling of jobs with large impact before jobs with lower impact;
- If a job fails, it is easy to identify all the dependent jobs that are no longer allowed to be executed, as they would be using invalid data.

8. ELIMINATE USELESS COMPUTATION WITH BARRIERS

Another time-saving and patented Flowtracer technology is the ability to propagate only the changes that are significant. This is a dependency management technique that is not based exclusively on the timestamps of files.

Suppose we have a VHDL based design and that everything, including simulation and synthesis, is up-to-date. Now a designer changes a comment in a VHDL file. We, intelligent humans, know that simulation and synthesis are not affected by the change in the comment*. Nevertheless, an automated design management system based on timestamp alone will detect the new timestamp in the modified file and will execute simulation and synthesis in order to guarantee the consistency of the design.

With runtime tracing, we introduce a new element in the dependency graph, the *barrier* (see Figure 8.1). The barrier is used to identify the selected files at which the propagation of changes can be safely

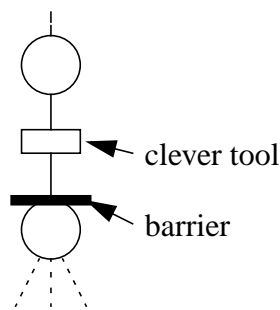


Figure 8.1 The barrier to change propagation is controlled by a tool capable of determining whether the changes to any of its inputs affect the output with the barrier.

stopped. The tool that outputs the file also controls the barrier. If the tool determines that the changes to its inputs do not have a significant effect on its output, the barrier is activated to signify that the

*.We ignore the fact that some CAD tools use pseudo-comments to control their behavior.

flow dependent on the file with the barrier is not affected.

At runtime, the tool sends a message to the RDMS to declare whether the barrier on its output file should be activated. The control of the barrier is given to the tool rather than to the RDMS, because the tool understands the meaning of the data, while the RDMS does not.

A simple tool that exhibits this behavior is Flowtracer's **clevercopy**. Before copying the input to the output, **clevercopy** checks whether the output is already identical to the input. In such a case, the changes that have caused **clevercopy** to be re-executed do not have any effect on the output; the copy operation itself is not necessary and the barrier on the output is activated to stop change propagation.

Clevercopy is a generic purpose tool that implements a weak notion of "significant change": in fact, any change is significant. More aggressive applications of barriers use tools that understand the semantics of the data. For example, there are EDA tools that can compare different versions of schematics and netlists and determine if they are significantly different.

9. CONCLUSION

Being prepared to manage the last two weeks in a design project means knowing everything about your design flow: which files are used, how they are generated, the impact of changing them. Runtime tracing is a technique that captures automatically the design flow and represents it in the form of a dependency graph. The graph can be used to guide the distribution of jobs across the machines in the network. By adding barriers to the flow, runtime tracing can be extended to intelligently block the propagation of changes that are not significant.

Important speedups in the execution of flows have been obtained by applying runtime tracing to complex designs with thousands of files. Runtime tracing has proved effective on designs from 4000 up to 500,000 files. We believe that its usefulness can only increase with the number of files to be managed.

BIOGRAPHY

Andrea Casotto completed his Ph.D. degree in Electrical Engineering in 1991 at the University of California, Berkeley, with a research on automated design management using traces. From December 1991, he worked at Siemens Corporate Research in Munich. In March 1995, he founded Runtime Design Automation of which he is president.

