Visual Design Flows for Faster Debug and Time to Market

FlowTracer™

White Paper
Introduction

As System-on-Chip (SoC) designs have increased in complexity in terms of size, power and performance requirements, and the inclusion of IPs, an increasing challenge has been the managing of files generated during the development process. Today’s complex design flows consist of several compute intensive steps including static timing analysis, design verification, Design Rule Checks (DRC), Layout vs. Schematic (LVS) checks, to name a few. EDA tools have to manage the interactions and dependencies between the thousands of files generated throughout the semiconductor design process.

Integrated circuit design has evolved from compiling code on mini-computers, to running a few scripts across a few workstations, to running tens of thousands of jobs distributed across thousands of High Performance Computing (HPC) clusters. EDA tools have also evolved in parallel to accommodate the added complexity.

Until recently, most semiconductor designs involved the use of sets of scripts, most commonly in “make” form, typically defined by the CAD or Design Automation (DA) functions. With the growth in design complexity and time to market pressures, it has been necessary for design teams to adopt and migrate to visual flow-based design methodologies such as Runtime’s FlowTracer to manage their complex SoC design flows and interactively control their environment. At first glance, it may appear that FlowTracer functions as a visually-enriched version of “make”. However, it is far from the truth as there are an abundant of significant benefits that are discussed in depth below.

This whitepaper discusses the advantages of migrating from script-based designs to visual flow-based ones to meet the needs of today’s complex designs.

“make” Scripts

Introduced in the 70’s, “make” was defined to identify dependencies and improve software compilation speed by tracking work that had already been completed. “make” was a key component of many development processes including EDA flows. However, its capabilities and shortcomings are as follows:

*Dependencies:* “make” is rule-based. It builds dependencies based on a system of explicit (system-based) and implicit (user created) rules. One key rule is that “make” uses a file’s timestamp to determine which files to compile or process. It looks across the dependency tree, locates all the dependent files and checks the timestamps to see what has changed, and rebuilds just what is needed without wasting time rebuilding other files. When input files are modified, “make” computes the tests that need to be run based on the dependency rules that are described in the “makefiles”. However, since the user has to manually specify dependencies, errors can occur and the source may not be recompiled. A conservative approach is often re-compiling code more often than necessary, wasting expensive compute resources.

*Sequential Operation vs. Concurrency:* “make” operates from a sequential context. This means all files must be entered in the precise order of execution since “make” by itself has no capability for branching or identifying parallel operations. Sequential programming results in extremely big text-based “makefiles”, increasing the opportunity for error and resulting in scaling problems for large jobs. Another key shortcoming of “make” is that it operates with
Transient states; once an operation has been executed the ability to revert to the previous state is lost, which has a huge impact on debug. This works for small projects, but becomes less manageable and can be unacceptable for larger projects.

Text-based Inputs: No visibility. Complex designs are often divided into subdirectories, with multiple “makefiles”, each one responsible for a small part of the dependencies. In addition, lack of support for parallel execution and limited capability for running regressions, loops and conditionals means that the number and size of makefiles for a complex design can become unsustainable quickly. Furthermore, debugging becomes excessively more difficult without a visual representation of design steps and their dependencies.

‘make’s’ shortcomings have been well-documented and presented over the last 20 years including:

*Recursive Make considered harmful*

“For large UNIX projects, the traditional method of building the project is to use recursive make. On some projects, this results in build times which are unacceptably large, when all you want to do is change one file ... This paper explores a number of problems regarding the use of recursive make, and shows that they are all symptoms of the same problem. These problems include recursive makes which take “forever” to work out that they need to do nothing, recursive makes which do too much, or too little, recursive makes which are overly sensitive to changes in the source code and require constant makefile intervention to keep them working.”

*Debugging make: Tips and tricks to get make working for you, not against you*

“To debug make, you have to be able to read a makefile... One of make’s key features is dependency management: make attempts to rebuild only what it has to when a program is updated. “Your goal in debugging a makefile is to figure out what make is trying to build, and what commands it thinks will build it. If make is picking the right commands, and they’re failing, you may be done debugging make -- or you may not.”

In summary, “make” has failed to scale to meet today’s design needs. Key shortcomings are the size of the files created, the debugging functionality and the lack of resource awareness.

**FlowTracer**

Runtime’s FlowTracer is designed as a mission critical dependency management platform for complex design flows. FlowTracer features “Runtime Tracing” to quickly analyze flows, unravel dependencies and identify the inherent parallelism built into today’s complex designs. It manages design flows in entirety, providing powerful visualization, debugging, and interactive capabilities.

*Dependencies: FlowTracer uses a database approach rather than rule-driven which is much more powerful for enabling a superior user experience. It uses a Runtime Tracing algorithm to build a complete and correct dependency job graph for any design flow where the*
dependencies are verified at run time. If the inputs change, the outputs are recomputed to recover the consistency of the design. FlowTracer keeps track of changes, not rules.

**Flow-based Operation:** FlowTracer technology has the ability to propagate only the changes that are significant. With FlowTracer, design flows can be created incrementally - and by concurrent programmers - as inputs do not need to be entered in any specific order. It identifies both parallel operation and branching, allowing for creation of more compact programs. It also knows which jobs to execute on appropriate computing resources (local vs distributed, memory requirements, etc.). By being resource-aware, FlowTracer can forecast the compute time needed and other design impacts if a section of design needs to be re-done.

**A Visual Flow:** FlowTracer’s FlowGraph allows its users to explore design flows in a simple and readable format through an interactive intuitive interface. It separates graphing from execution and automatically captures the dependency graph and performs optimum execution/re-execution of the visuals as the design files change. A standardized design view enables full collaboration 24/7 of design teams dispersed around the globe.

**Collaboration:** FlowTracer is designed for multi-site sharing and collaboration. Once a flow is built, it can be easily deployed and collaboratively debugged via web interface. With “make” the designer has to create a specific target file with examples that another design team member then has to use to recreate the problem. In FlowTracer a remote engineer can simply connect with the session, help to debug it, and fix the problem in real-time.

**Cross-Platform Solution:** FlowTracer is a cross-platform solution. It can take inputs from common scripting languages (“makefiles”, shell scripts, etc.) as well as FlowTracer’s native scripting language FDL (Flow Description Language), allowing for gradual migration of design flows.

**Table: Comparing “make” to FlowTracer**

<table>
<thead>
<tr>
<th>Feature Summary</th>
<th>“make”</th>
<th>FlowTracer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Systems Aware</td>
<td>No (developer must insert “sleep” commands)</td>
<td>Yes (protects the network from glitches)</td>
</tr>
<tr>
<td>GUI</td>
<td>No</td>
<td>Yes (colored graph allows View, Up-Cone, Down-Cone)</td>
</tr>
<tr>
<td>Collaboration, Web-enabled</td>
<td>No</td>
<td>Yes: Share flows through browser</td>
</tr>
<tr>
<td>File Reversion</td>
<td>No (rule driven, based on timestamp)</td>
<td>Yes (stores all changes and timestamps)</td>
</tr>
<tr>
<td>Scalability</td>
<td>No (large “makefiles” are complex and recursion loops are harmful)</td>
<td>Yes: Scales to 10’s of millions of files</td>
</tr>
<tr>
<td>Acceleration</td>
<td>Limited (Use make –j4 or –j8 commands)</td>
<td>Yes (1000s of jobs running simultaneously)</td>
</tr>
<tr>
<td>Resource Aware</td>
<td>No</td>
<td>Yes (Distributed Operation)</td>
</tr>
</tbody>
</table>
FlowTracer Overview

Almost any language can be used for building flows: Tcl, C-shell (tcsh), perl, python and “make”. FlowTracer leverages Flow Description Language (FDL) in order to construct flows. FlowTracer draws and orders the FlowGraph showing Nodes, Jobs, Inputs and Outputs. Files represent design data and Jobs represent actions. These actions can be simple Linux commands such as “cp” (copy), or complex commands such as “dc_shell” (Synopsys Synthesis). Circles represent files, rectangles represent jobs. The following example highlights the execution of a flow in FlowTracer.

1) Starting Point

*Figure 1: Dependency Graph*

All jobs and files start at an invalid state (Purple).

2) Executing a Graph

FlowTracer parses the tool’s command-line and input files, then it executes the graph and reports job status through color-coding.

*Figure 2: First Job Queued*

FlowTracer orders the jobs intelligently. FlowTracer determines that input file a is valid (Green), and identifies that cp1 is queue to run (Orange) before cp2 (Cyan)
**Figure 3 First Job Runs**

The first job (cp1) is running (Yellow). Once it completes it will turn Green if it is successful, or Red if it fails.

**Figure 4 Second Job Queued**

The first job (cp1) has completed successfully (all the files and jobs are marked as valid, Green) and the second job (cp2) is queued (changes from Cyan to Orange).

**Figure 5 Second Job Completes with Errors**

In this example the second job (cp2) did not complete successfully and is shown as Red and the output, “c” is shown as Purple (invalid).
3) Flow Analysis & Debug

By clicking on a failed job (Red), users can quickly identify the cause of failure and it allows them to review the error log file.

Figure 6 illustrates how FlowTracer GUI and debug features can be used to quickly diagnose the root cause of failure. By clicking on the failed job (red) one can analyze the cause of failure. Once the problem is fixed, the job can be re-submitted from the point of failure.

4) Abstract Hierarchical Complexity

In reality design flows are far more complicated than illustrated in the simple examples above. FlowTracer can scale to monitor and manage tens of thousands of jobs and demonstrated to scale to more than 1.5 million instances in a single project. Once a flow is built, it is easily deployed and debugged via FlowTracer’s web interface making it easier for cross site collaboration, debug and flow sharing.

5) Flow Viewing Options

FlowTracer’s GUI provides a wide array of methods for viewing and organizing flows for the end users. Examples include: FlowGraph View (Vertical & Horizontal), Grid View and Hierarchical Set View.

Figure 7 illustrated FlowTracer’s FlowGraph View. FlowGraph view is quite similar to the diagrams shown in the example above. In this view jobs are shown as rectangles, files as ellipses, and dependencies as lines. It allows for flows to accommodate user preferences.
**Grid View** (Figure 8) is used when the project becomes more complex and needs another level of hierarchical abstraction. Jobs consume a much smaller real estate on the screen allowing a large number of jobs to be viewed while allowing failing jobs to be detected easily.

![FlowTracer Grid View](image1.png)

**Hierarchical Set View** (Figure 9) allows users to group their jobs by logical groupings to accommodate a more structured view of what they are running. For example, standard cell characterization jobs can be organized and viewed by groups of cells of the same type (NAND, NOR, INVERTER, etc.).

![Hierarchical Sets](image2.png)

**Migrating Scripts-Based Flows to FlowTracer**

Many organizations today are dependent on large volumes of scripts or “makefiles” that have either been accumulated over a number of designs or developed in conjunctions with the deployment of new EDA tools and flows. Although FlowTracer makes a compelling case over most script-based solutions by delivering greater efficiency and productivity through smarter dependency awareness and visualizing flows for easier debug, the task of converting entire libraries of scripts that an organization has built over a long period time to FlowTracer’s native FDL may appear to be a daunting task. However, this is not the case, by design.

FlowTracer’s cross-platform nature was created with the consideration of coexistence of scripts, “makefiles” and FDL, enabling organizations to migrate their existing scripts in a gradual manner. FlowTracer also features “vovmake” as a mechanism to translate “makefiles” to FDL to ease flow migration. This utility provides a vehicle to migrate most “flat makefiles” from “make” to FDL. More
complex “makefiles” such as nested or self-modifying “makefiles” require additional efforts in breaking down the flow prior to migrating to FDL.
FlowTracer also allows for the running of a combination of FDL alongside with existing scripts. A recommended rule of thumb for migration from “make” to FDL is to adopt a gradual migration of scripts that would gain the greatest benefits from FlowTracer’s dependency awareness, visualization and ease of debug. Some examples include, IPQA, cell library characterization, and verification testing.

Summary

Much of the underlying design flow management infrastructure for semiconductors today are still heavily dependent on the use of scripts such as “makefiles”. Script-based solutions face significant disadvantages in keeping up with the rapid increase in design complexity of today’s SoCs.

FlowTracer’s “Runtime Tracing” technology quickly analyzes flows, unravels dependencies and identifies the inherent parallelism built into today’s complex designs. It manages design flows in entirety, providing powerful visualization, debugging, and control capabilities. Designed as a cross-platforms solution, it allows for coexistence of scripts and FDL accommodating a gradual migration path for complex flows. FlowTracer’s GUI allows for designers to quickly detect and debug flow and design defects. FlowTracer can maximize the use of data center resources as it unravels flow dependencies and accommodates parallel execution of design flows. FlowTracer’s dependency-aware architecture combined with its visual interface allow designers to make incremental corrections to their flow and run only the portions of the design flow that are affected by that change saving both time and resources.