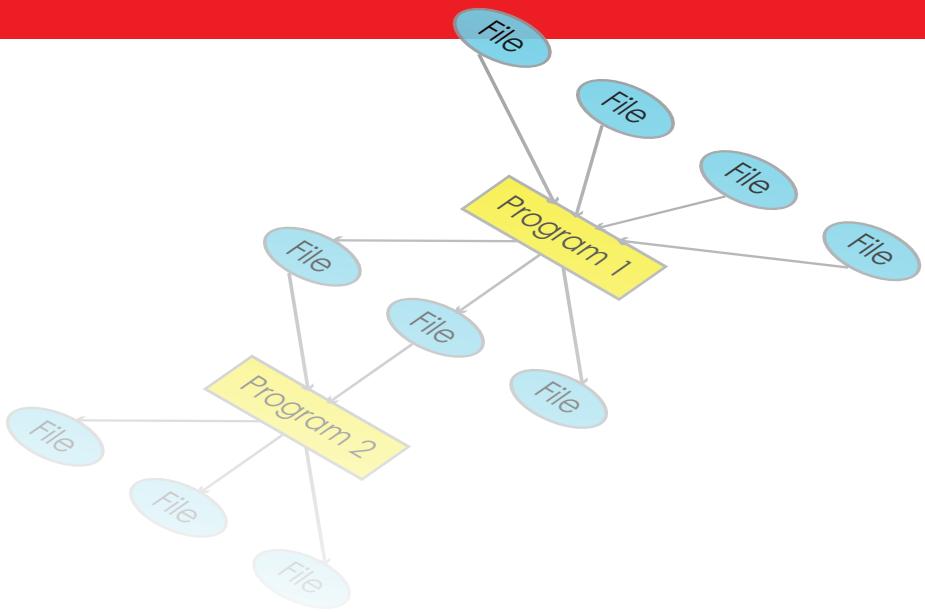
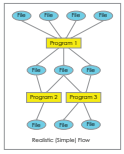
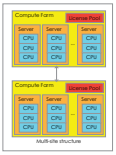


The Art of Flows

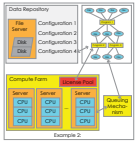
Runtime Design Automation, Inc.



What is a flow, and why is flow management important?



What are the most common pitfalls in flow management and how do you avoid them?



How can Runtime Design Automation's FlowTracer automate flow management?

Runtime DESIGN AUTOMATION

The art of flows

Design flows capture the details of how design gets done in the modern world of computer-aided design.

Managing the files, programs, and dependencies between them is critical to the success of any design-oriented activity, whether it be semiconductors, software, drug discovery, 3d modeling, or a host of other industries.

This volume explains the importance of the problem, and how best to solve it.

The insights in this book are based on over twenty years of experience in the definition, automation, and management of design processes across a range of different industries. The recommended techniques derive their robustness from a combination of graph theory and pragmatic methods for capturing and managing flow information.

With the rise of compute farms and global design communities, the design flow problem has expanded to encompass global wide-area networks, sophisticated resource management, and the mediation of software licenses across global infrastructure. Getting the overall execution environment right is a key part of deploying design tools effectively.

From this book you will learn the key issues, common pitfalls, and best practices for flow development and management, from small scale examples to the most complex cases.

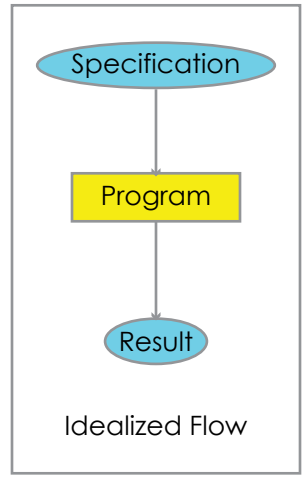
10 9 8 7 6 5 4 3 2 1

Contents

1 The importance of design flows

Wouldn't it be great if designing a chip, or developing software, just required us to run a single program, that used a single input file describing our specifications, and generated a single output file that contained the desired implementation?

Sadly this is not the case.



Idealized Flow

Designing anything complex requires us to create multiple input files, run multiple programs, and generate multiple outputs. This is true in semiconductors, graphics design, software development, 3-D modeling, drug discovery, scientific analysis, and in fact any domain where the problem is complex and the technology sophisticated.

To make life even more difficult, there are complex dependencies between the different files and programs, some of which may not be well understood or properly documented. This means that we need to keep track of the status of every part of the process, and make sure

everything is run in the right order, using the right versions of every file.

The design flow can be represented by a graph, where the nodes represent the files and programs, and the arcs represent dependencies. To be a little more precise, the files represent the state of the design, while the programs represent transformations that change the state of the design by acting upon their inputs and creating updated outputs.

The essence of a design flow is the definition of inputs, programs and outputs, and the dependencies between them.

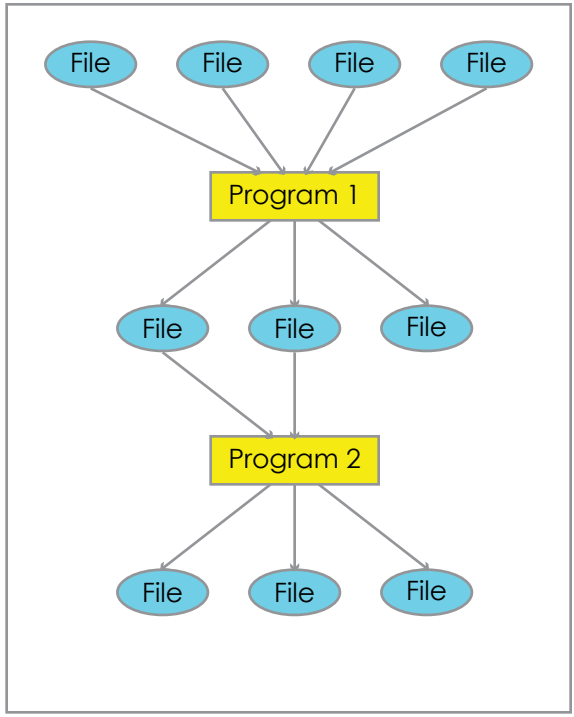
To be more specific, the files manipulated by the designer(s) represent the intelligence created by the human designer or the software tools. They are ultimately what is used to create the final product. Their consistency, accuracy and completeness are the ultimate measure of the state of the design: when everything is complete, consistent and correct (as defined by the specification and the designer), the work is done. Design flows capture the process of getting to completion, and the records of running flows provide insight into the state of the design at any point in time.

Why do design flows matter?

Design flows represent the collective knowledge of design within your organization. They are typically created

as methodologies by expert designers who understand technology and design criteria at a deep level. Modern design is a complex task, with many opportunities for error. The creation, management and deployment of design flows helps to ensure that all members of the design team are working in a consistent way, that exploits your proprietary design knowledge.

It's important to note that flows are usually not written for a single user or a single time. Flows are part of the intellectual capital of your organization, and as such they are a long-term asset. For this reason, documentation, maintenance, and enhancement of flows are key requirements. Even if they *are* written with a single user in mind, job changes mean there is real value in building flows in a way that supports re-use by other people. The importance of managing flows as an asset is even more clear with



global design teams, where the creator of a particular design methodology may not be available to help users in different time zones or geographies.

Design flows not only ensure that designers do the right things—in other words maintaining dependencies and execution order for correctness purposes—flows also support the interface between the designer-user and the compute environment. Design tools and computers are an expensive resource for any company, and their effective use can be a significant way to save money over the life of those resources. For this reason you should consider the interface between your design flow and the environment in which it runs to be a critical requirement for your design flow solution.

Last, but not least, in a meaningful way the state of the flow represents the state of the design. By observing the current state of execution of a flow, either in real time or via a batch report, the user or design manager can gain a precise understanding of the current state of work, the specific areas that need focus and attention, and the likely nature and location of bottlenecks or schedule issues. This ability to understand the state of the flow is absolutely critical to working effectively with a complex design methodology.

So there can be little doubt of the importance of flows. They are a key asset to any design-centric organization, and their effective use is a major contributor to produc-

tivity and accuracy in design. Let us now examine the key requirements for a flow management solution that delivers on the promise of flow management.

Critical flow requirements

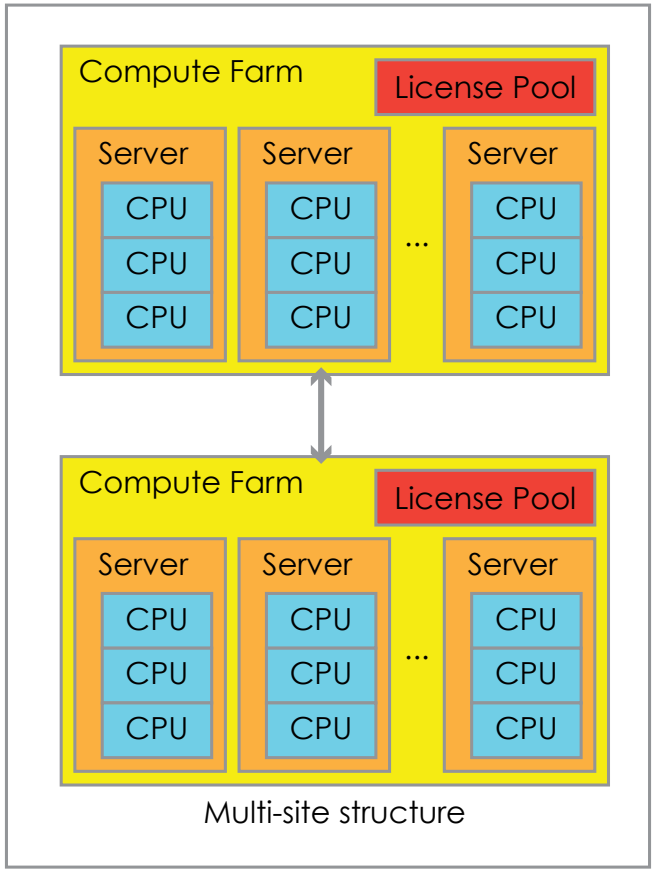
Even for a single user, dependency management and progress tracking are important, and there are very good reasons for building robust flows to improve documentation and adaptability to changing requirements. But the power of flows really expands when there are multiple users collaborating on a single design. There are several key issues:

- Deploying a flow effectively to multiple users
- Controlling access to design data
- Managing resources
- Taming complexity.

It's worth exploring each of these requirements in turn.

Flow deployment

In the semiconductor industry, the definition of a methodology can be extremely complex. There may be hundreds of tools, and tens of thousands of files produced and consumed. There are decision points, both manually and automatically assessed. There are interactive steps, and batch jobs. The time for a design step can be mea-



sured in days or weeks for large designs. In other domains the numbers may be smaller, but it is almost always

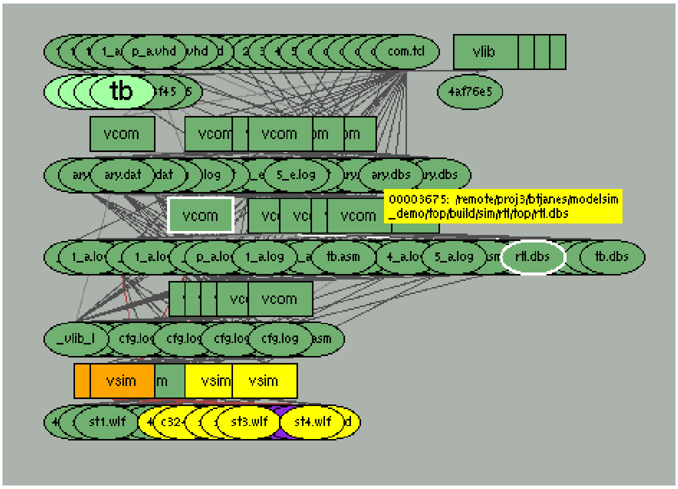
The implication of this complexity is that end users need help to be sure they are using the methodology correctly. This is not simply a matter of hiding all the complexity, and creating a "black box" within which everything magically happens. Designers are intelligent people, who need visibility into the processes they are operating, along with appropriate levels of flexibility so they can

deal with unexpected challenges or opportunities in the design.

Deploying a flow in a way that maximizes designer productivity therefore includes many elements, including:

- Education of users
- Documentation of the flow
- Visualization of progress and results
- The ability to explore and diagnose problems.

Providing an industrial strength solution to these challenges is a significant effort for any company, but an essential investment if design knowledge is to be effectively re-used.



FlowTracer Design Flow Visualization

Design data: repositories and workspaces

When multiple users are involved in the same design, contention for resources becomes an issue. The first manifestation of this is in the control of file access—ensuring not only that the correct files are used at each step of the flow, but also ensuring efficient execution based on file availability.

Configuration management helps make sure that critical files are properly versioned, and that you can't overwrite another person's work accidentally. Typically a *repository* is created in which master copies of all versions of the files in the design are stored. This may be in a single location, or it may be *mirrored* to multiple locations for performance reasons (in other words copies of the master repository are made that are kept synchronized by a background process).

Individual users can then *check out* a set of files to work on. The configuration management software typically copies the files from the repository to a *workspace* accessible by the user. This is usually simply a location in the file system accessible by the tools and the user. At the conclusion of a successful design task, the user can then *check in* the updated files, ensuring both that the work is saved, and that it can be made available to other users for subsequent work. In this process, the configuration management software keeps track of who has what files, so conflicts can be properly managed. This is a big help in managing the overall health of the design data.

Flows and configuration management

So configuration management is an essential element of the world of computer-aided design. But design flow dependencies are more complex in the presence of configuration management software, because time-stamps can actually move backwards. Here's how it happens.

Imagine checking in a file, only to decide that your previous version was better. You then check out the earlier version, which of course has an older timestamp than the newer version. Your dependency management solution must be able to tell that the file has changed, and be ready to re-run the related steps. This is difficult to achieve with a dependency management solution that relies on timestamps, because you have to do more than simply compare input and output timestamps: you have to actually know if the file is different from the one used in the last run (older *or* newer). The best way to do this is to keep a database of file information that is persistent from one run to the next: this way we can simply compare the timestamp of the file in the workspace to the timestamp we have stored from the last run.

Licence management

Software licence management is a challenge at many levels. The principle responsibility of the licence manager is to issue license keys on demand, subject to the constraints of the user's contract with the software provider. In recent years, licencing schemes have become com-

plex, with special capabilities for token-based licences, LAN and WAN licencing, and more. From the flow development perspective, the issue is that there is an interaction between licence availability, computing resources, and flow dependencies.

Licence servers allow for checkout of software capabilities, but when multiple users are working with limited numbers of licences, the dependency management solution must be intelligent about its ordering of jobs, based on licence availability. This is a key optimization that can make an enormous difference to throughput and design cycle time. In fact, the correct ordering of jobs where there are opportunities for parallel execution requires an understanding of dependencies, licence availability, compute resource availability, and issues of global distribution of tools, licenses and machines. We will discuss these issues next.

Global design teams

Increasingly, design is a global activity. In each of a number of locations you will find a group of designers, a set of software licences, and a pool of machines. The optimization problem involves serving the needs of all designers in a way that maximizes resource efficiency and minimizes design cycle time. Jobs need to be executed on machines with the appropriate architecture, operating system, processor complement, and disk and memory resources.

Among the special concerns in running flows across global networks, network reliability and latency are especially important. Ideally the flow will account systematically for any possible failure modes when computing in a global environment. This must be integrated with tools for managing all the compute resources: our next topic.

Resource management

Managing compute resources is an important task in any environment, but especially in the context of a design team. There is an optimum way to allocate jobs to machines, so that the big jobs go to the big machines, and the small jobs go to small machines. The flow management solution needs to understand resource availability, and to determine optimum execution order based on resource availability. In order to achieve this, the flow manager needs not only to execute jobs in the right order, it needs also to gain access to the most appropriate resources at run time. This requires *licence migration:* moving licences from one location to another, as well at potentially running some jobs locally, and others remotely.

This is a very complex problem, as any realistic environment will contain a significant number of different machine configurations, including single core, multicore, with different memory and disk configurations. The complexities of resource management have led to the development of a number of software systems dedicated to

the use of compute farms: Runtime Design Automation's *Network Computer* is one of them.

Visualization and flow status

At first glance it might seem that visualization is an optional feature. Surely it should be enough to know that the flow is correctly captured and executed? In fact, this is not the case.

We hold engineers responsible for executing a complex set of inter-related tasks, and they need to be able to understand the behavior of the system they are using. This is not only a matter of helping designers take responsibility for their work, but also of helping them to be productive. When something goes wrong in a flow, designers should not need to scan an arbitrarily complex set of log files in order to diagnose problems.

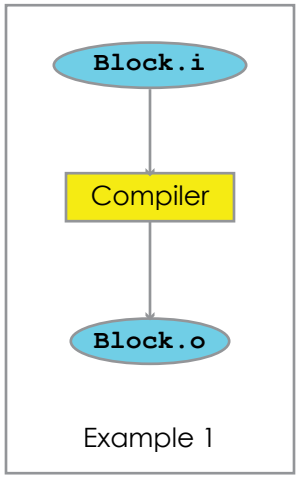
A graphical display that indicates exactly where the failure occurred is not just more productive, it is also likely to be more reliable. One reason for this is the sheer complexity of modern flows. In pathological cases, we have seen upwards of 15,000 files as inputs to a flow, and a similar number of output files. At this level of complexity, color-coding in the visualization can be an enormous help in sifting through the data and deciding how to fix a problem.

Beyond the practical issues of running the flows, communication between designers and managers is an issue of

real importance in the running of a business. Because of the complexity of design, there needs to be a consistent flow of factual and usable information about the status of the design between those doing the design and those responsible for business decisions concerning it. This is critical in order that the state of the schedule can be properly assessed and resources allocated appropriately in the interests of the most successful business outcome.

In the semiconductor industry, to take just one example, the cost of being late to market is measured often in the tens of millions of dollars of lost revenue, and there are often hard deadlines, like CES, for example, which can be directly linked to a year of revenue and the success or failure of a multi-million dollar product.

2 A simple example



Example 1

In the next three chapters we will look at three examples, at different levels of complexity. Each will clarify some of the key issues in the art of flows, starting with a simple, single-user example, moving to a more complex, single-user case, and finishing with a multi-site, multi-user example that is characteristic of real-world design flows.

There are many ways to implement a flow management system, but the most common are through scripting and/or the use of makefiles. In addition to these methods, we will show how Runtime Design Automation's FlowTracer handles the same issues.

It's useful to consider practical examples as we think about the art of flows. We'll start with a simple software development example: compiling a small program that takes a single input file and generates a single output file.

Problem definition

The illustration on the previous page shows the flow diagrammatically. There is a single input file, a single program that uses that file, and a single output file. Of course this is an artificially simple example, but our goal is to illustrate the principles as we move from the simplest case to the more complex.

Our goal is to set up an environment in which the file can be reliably compiled, and ideally to have some idea of what happened if it doesn't compile properly.

This is about as simple as you can get: we are assuming no configuration management, and only a single user working on the problem.

Scripted solution

This example is so basic that we could implement it directly at the command line. That will work so long as we know exactly what files and programs we are using, and our assumptions don't change. This also assumes two arguments: an input file and an output file with no command-line switches.

But we will assume for our current purposes that it's useful to create a script that sets up the environment and runs the compiler.

Here's a simple script to automate this flow:

```
#!/bin/csh -f
set in  = $1
set out = $2
eda_compiler $in $out
exit $status
```

As you can see, it simply passes its input arguments to the compiler program, and returns the exit status when its finished. No error checking, no awareness of anything except the name of the program, which it assumes is in the current path.

To be fair, even in this case, a smart programmer might well provide a bit more capability in a short script like this—on the next page you can see a more robust script for the same flow.

In this version you can see several of the key issues being addressed. Firstly we have an area where some variables are set to control which tools are to be executed. This is one approach to scripting where the writer sets up the execution conditions at the top, and the remainder of the script relies upon those settings. This has the benefit that you only need to edit the top of the script, but of course you do still have to make those edits whenever things change.

Secondly we have a section that looks for the input file, and lets you know if it's not there. This is a help, but it doesn't address the issue of whether or not the input file has changed since the last run. So this isn't real depen-

```
#!/bin/csh -f
set doCompile = 1
set rtn       = 0
set in        = $1
set out       = $2

if ( $doCopy ) then
    # Make sure the input exists
    # and the output is removed.
    if ( ! -e $in ) then
       echo "Input file $in is missing"
       exit 1
    endif
    if ( -e $out ) /bin/rm $out
    # Environment management.
    # and the LM_LICENSE_FILE
    setenv EDAROOT "/opt/edaVendor"
    setenv PATH    "$EDAROOT/ \bin:$PATH"
    setenv LM_LICENSE_FILE "3456@lichost"

    # Execute the command:
    echo "Starting now `date` on host `uname -n`"
    time eda_compiler $in $out >& compiler.log
    set rtn = $status
    echo "Done on `date`"

    # Check validity of result:
    if ( ! -e $out ) then
       echo "Output file $out is missing"
       exit 1
    endif
    grep 'Error:' compiler.log
    if ( $status == 0 ) then
       echo "Compilation errors."
       exit 1
    endif
endif
exit $rtn
```

dency management. We could at this stage also add a command to make a backup copy of the old output file, although this is not necessary if we can go back to a previous version of the input file and re-generate the output.

Thirdly we have some rudimentary licence management and environment setup. This section of the script sets up the variables for the licence file and the path to the compiler binary.

Finally, we run the command and check the validity of the output. This is the most useful aspect of the script: we can implement arbitrary checks in the script to help the user understand what went right or wrong.

The benefit of scripting is that it is at least theoretically capable of any computation. It's Turing-complete. But completeness and maintainability are two different things.

Because we implement checks and dependencies directly in the script, we have no separation of structure and implementation, which creates maintenance headaches.

One tactic some script writers use is to separate the environment setup into another file. This at least allows environment setup to be shared across multiple flows.

It would also be possible to create configuration files that hold the setup information captured at the beginning of

the script in some kind of ***rc** file, but now we would have to write parsers and processors to make use of that information, which also need to be maintained and tested as requirements evolve.

A final element to mention regarding this scripting example is the ability to create logfiles: by echoing status information to **stdout**, we can capture information that can subsequently be used to track down errors and to debug the flow.

Makefile solution

make works pretty well for simple situations like this. We can set up a single target to build the output file from the input file.

This is a minimalist version: there's only one explicit target, and an implicit target that expresses a dependency between the **.o** and **.i** files. The makefile looks like this:

```
set OUTPUTS = "Block.o"

all: $OUTPUTS

.o.i:
      eda_compiler $< $@
```

This simple makefile allows the user to run the compiler if the output file is older than the input file. The dependency tree is built at runtime, and the decision whether or not to run the command is made based on the time-

stamps of the input and output files at the time the command is run.

Because this is unrealistically simplistic, we have also created a more practical version. It's still a short file, but it's a little more robust and useful than the first version (although still probably insufficient for the real world: we'll get into more sophisticated approaches later).

```
set OUTPUTS = "Block.o"

all: $OUTPUTS

.o.i:
      echo "Starting on `date` on host \
        `uname -n`"; \
        /bin/rm -f $@;\
        time eda_compiler $< $@;\
        echo "Done on `date`"
```

Now we have defined the output file explicitly, and made the command running the program a bit more capable by echoing some runtime information for logging purposes. We also remove the previous output file (if it was there) prior to running the tool.

So here we have implemented some simple but real dependency management. The makefile, like the script, is capable of implementing arbitrary code (typically Bourne shell) so the makefile can become quite powerful. However, this power is orthogonal to the dependency management in the makefile structure, and so there is a danger that you end up with a significant part of the

makefile's intelligence implemented in scripting, which is not part of the dependency management provided by **`make`**.

It's worth noting, however, that some of the dependency information is implicit. For example, the intelligence to load any required header files is only implemented inside the program files. The makefile does not know about this, and so does not manage any of those dependencies. This means that if any of the header files change, **`make`** will not detect the change because it knows nothing about the header files, and will return a false positive. The makefile solution therefore fails to capture and document all the dependencies, and it also fails to provide any help when things go wrong.

FlowTracer solution

FlowTracer tackles the problem a bit differently. We can describe the dependencies declaratively, which means there is a minimum of interaction between the different statements in the flow description. This is done with FlowTracer's Flow Description Language (FDL), which is in turn based on Tcl. But while we can write down the dependencies and required actions in FDL, FlowTracer is also capable of watching the tools run, and deducing the dependencies by itself. This has the advantage that FlowTracer can respond more effectively to change, because the system will find dependencies even if we haven't documented them.

Here's a simple FDL description of our scenario:

```
set block "Block"
E BASE+EDAVENDOR
T vrt eda_compiler $block.i $block.o
I $block.i
O $block.o
```

This format may be new to you, so we'll go through it line by line.

The first line simply sets a variable that holds the name of the input and output files. This is straightforward enough.

The second line is a *directive,* responsible for setting up the environment (**E**) for the computation. FlowTracer allows environments to be captured once and re-used. In this case, we are setting up the environment from the catenation of two definitions: one is the base environment, and the other is specific to the compiler we are going to use.

The third line is a tool directive (**T**), which provides the command for running the tool, with two refinements. The first is the use of the vrt command, which invokes FlowTracer's run-time tracing (about which more later). The second is the substitution of variables to capture the names of the input and output files.

Following the **T** directive, the final two lines declare the one input (**I**) and one output (**O**) files for the job, without

excluding the possibility that more inputs and outputs may be discovered when the job is run (i.e. at runtime).

In order to execute this flow, FlowTracer builds the dependency tree based on the information in the FDL file. This is constructed and saved at build time—that is the time when the FDL is executed. Furthermore, the information about previous runs is persistent. This information includes the *start and finish time* of the job, the *name and timestamp* of all inputs and outputs. This overcomes the issue with reversing timestamps we described earlier, because we can tell if the file in the file system is the same or different from the one used previously: we don't simply rely on the delta between the timestamps of **`block.i`** and **`block.o`**.

FlowTracer will then run the job or jobs for which some inputs have changed since the most recent execution. This is the core implementation of dependency management.

In addition to simply running the flow correctly, FlowTracer does runtime tracing. This means FlowTracer is actually watching the file I/O of the tool, and tracking the real dependencies, which might be different from those documented in the flow description. This is a huge benefit, because it means that even if you don't know all the dependencies, FlowTracer will find them and manage them for you. In fact, when running a job, FlowTracer is able to detect if you have specified the dependencies

incorrectly, and will repair the dependencies for each executed job. This unique capability gives FlowTracer a level of robustness that is not achievable with other systems.

Another advantage of the FlowTracer approach is the availability of real-time visualization. This is especially important where the flow is complex. By using a color-coded graph of the flow that updates in real time, the user can clearly see what's going on, and can quickly click into the flow to diagnose and solve problems.

But as before, this is the simplistic version. We can do a better job with the following:

```
set block "Block"
E BASE+EDAVENDOR
J vrt eda_compiler $block.i $block.o
```

This assumes the existence of an *encapsulation,* which defines the inputs and outputs of the tool for FlowTracer's benefit. The advantage of this is further separation of the various re-usable elements of the script. Not only do we have our environment setup in a single location that can be used by multiple flows; we also have the descriptions of how to invoke our tools captured once so the same tool can be used in multiple flows. The **J** directive then causes FlowTracer to refer to the encapsulation for the required information.

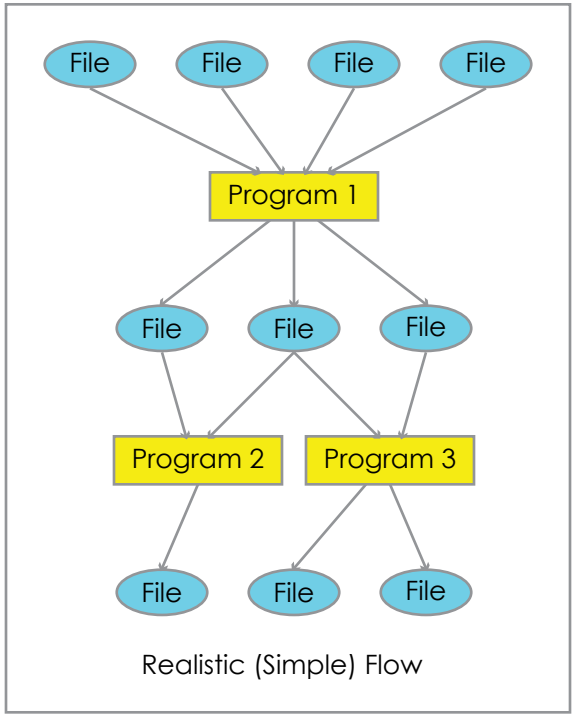
Thus FlowTracer allows the various elements of flow description (such as environment, tool encapsulation, resource requirements, command lines etc.) to be clearly documented in such a way that a change to one of the base elements will be properly propagated to all the locations where it is needed. Typically environments are set up based on the need for consistent run-time behavior; equally tool encapsulations allow common tool invocation protocols to be captured so the tool is always run correctly.

Finally, FlowTracer's ability to capture dependencies at runtime, and then to keep a persistent database of dependencies and flow status, make sure that the flow is correctly implemented.

In the next example we will explore some more elements of the Art of Flows: parallelism, configuration management and aspects of licence management.

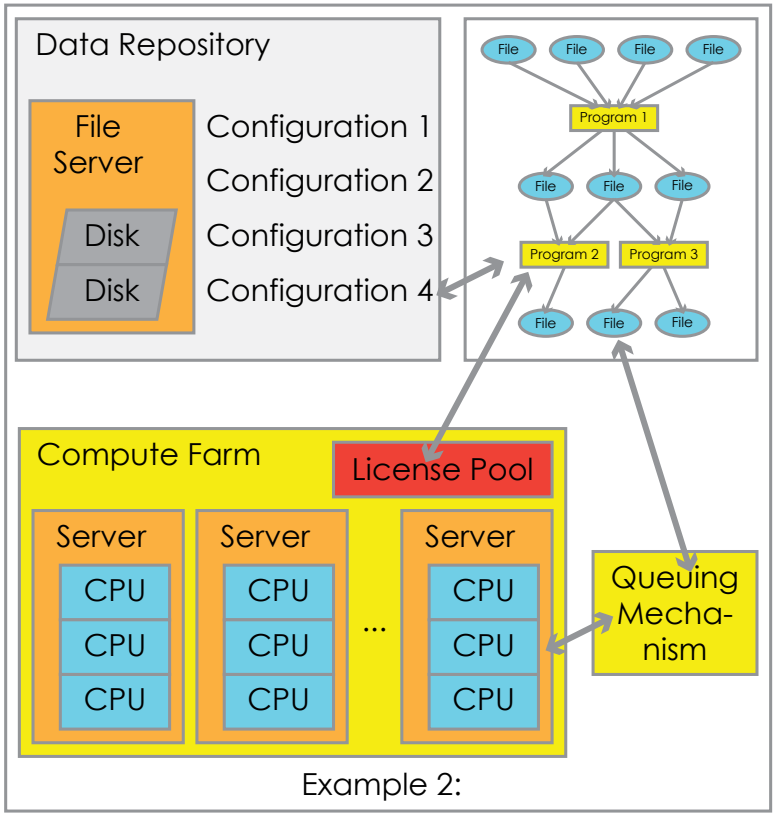
Runtime Tracing is the ability to track dependencies as the flow is running. It increases flow reliability by adapting to changes in tool behavior without requiring changes to the flow description.

3 A more realistic example



Realistic (Simple) Flow

Now we have seen the principles at work, it's time to look at a more complex example—one which addresses more of the technical realities of building flows for design or software development. In this chapter we'll consider not only the addition of multiple files, but also the management of multiple directories, configuration management, licensing, and the need to exploit parallelism to maximize throughput and minimize design cycle time.



Example 2:

To keep the example relatively manageable, we'll focus in this case on including all the key issues, but not on scaling up to the max. We'll keep that for our final example in the next chapter. So what you see here will illuminate the different issues, but will still not be as complex as a real-world flow.

The diagram only shows the core of the flow: dependencies between files and programs. We have three programs: the first will run any time its input files are different from those used to generate the current set of output files. The second program will run when the first

program's outputs are up to date, and if they are different from those used to generate the current set of output files from Program 2.

Parallelism

Notice that Program 2 does not depend on all the outputs from Program 1. The third program is similar to Program 2 in terms of dependencies; however we can also intuitively see that Program 2 and Program 3 could run in parallel if all the required resources are available.

Sequential program dependencies

There's a new element here: we are not only interested in the file I/O dependencies, we can see intuitively that there's no point in running Program 2 or Program 3 until Program 1 has done its work. This level of dependency needs to be modeled just like the file dependencies if we are to have a robust solution.

Licence management

There are other elements as well: let's imagine that Program 2 and Program 3 are instances of the same application, but that the application is licenced, and limited licences are available. So that means that unless two licences are available at runtime, Program 2 and Program 3 must in fact be run sequentially.

Resource optimization

We can add more complexity: Program 2 is a relatively small job that can be run in 4GB of memory, while Program 3 (even though it's the same application) needs 64GB to run efficiently. That means Program 2 and Program 3 should ideally be run on different machines so we don't waste resources on Program 2, or strangle Program 3.

Configuration management

Finally we will add the element of configuration management. We shall assume that there is a repository of design files that are versioned, and that the designer is responsible for checking out a valid configuration prior to running the flow, and then checking in the results if all goes well.

The challenge here is that the configuration involves all the files in the flow: both input files and output files. As we revert to older versions of these files, their timestamps will not properly capture the age of the files, so we need to be careful to model the dependencies with this in mind.

Putting it all together

So we have three programs, a bunch of files, software licences, dependencies, parallel and sequential execution, computing resource management and configuration management. The previous diagram only showed the file and program dependencies: let's see what the entire problem looks like diagramatically. We'll assume a

single compute farm and a single licence pool (reasonable simplifications for now, but many real-world situations will add WANs, multiple licence pools and more).

The final assumption in this example is the existence of three directories: one for the first set of input files; one for the intermediate files generated by Program 1, and one for the output files generated by Program 2 and Program 3. This is a simplification of a common situation: in real cases our files would exist in a complex file hierarchy that would be traversed by the tools, and that must also be traversed by the configuration management and flow tools in order to set up and manage execution.

Scripted solution

In order to script this flow, we need to build some infrastructure. It's just too complicated to build a single, *ad hoc* script. The elements we must manage are these:

1. Setting up paths for the tools
2. Setting up the licensing environment
3. Accessing the required data file configuration
4. Capturing the required switches, inputs and outputs for the tools
5. Capturing and acting upon exit status

6. Defining the computing resources (machines, memory, disk etc.)
7. Relating jobs to compute resources
8. Running the flow
9. Monitoring and managing outputs,

This is not a trivial set of tasks, but all of these elements must be managed robustly if we expect our flow to be reliable and easy to use. The purpose of capturing and managing all these issues is simply to help end users avoid mistakes that could be costly in terms of design time or correctness.

In fact we will have to provide all the same infrastructure for makefiles and for FlowTracer, but we'll go through it in a bit more detail as this is our first time through.

Paths and environment setup

It's good practice to separate the environment setup into its own file. That way we can more quickly re-target our flow to new environments. Here's our setup script:

```
environment script
```

Configuration management

We will assume that we don't have to directly implement configuration management in the flow script. This is consistent with common industry practice, where a set of configuration management commands are used independently of the flow to pull the desired file set from the repository in preparation for running the tools.

With this assumption, we will write the flow script to use timestamps as a way of understanding dependencies.

Tool encapsulation

For this example, we will write the command lines for each tool directly into the script file. This is commonly done, and it is reasonable if the focus is on minimizing the script development time, rather than on documentation and maintainability for re-use.

As we have said before, we regard the flows as an important corporate asset, so there's a strong case for building for scalability and maintainability, rather than just for implementation ease.

Computing resources & job mapping

How should we implement the definition of our compute farm and the mapping of jobs to other machines? We need some kind of database, recording the capabilities and availability of the different machines on the network. We also need a way to schedule jobs through such a sys-

tem. There are three systems out there in common use: LSF from Platform Computing, SGE from Sun Microsystems (soon to be Oracle), and Network Computer from Runtime Design Automation.

It really wouldn't make sense for a scriptwriter to create a solution from scratch, because this is a complex task involving networking, job monitoring, handshakes and locks, error recovery and a lot more. For the purpose of this example we'll use Runtime's Network Computer (NC).

Here's how it works:

```
simple NC example
```

The flow

Because we're using NC to manage jobs across the network and the compute farm, we can simply send the jobs to NC for execution, knowing that the resource management issues are addressed, and that all the complexities of networking errors, licence availability and so on are being handled by NC. Implementing these directly in a script is a very big task, and it doesn't make sense given the availability of industry-proven and cost-effective solutions.

Here's our script for the dependencies—this is the heart of our scripted solution"

<script implementing the core dependencies>

```
script implementing the core dependencies
```

Tracking outputs

Without building a custom reporting system, it's hard to do much more than writing a log file and grepping for errors. With a simple flow like this, such a solution might be good enough. But in a larger, more complex situation like chip design, this is not enough, and a lot of time can be wasted scouring log files looking for the source of a problem. Nevertheless, we'll focus for this example on a straightforward logging solution, because this is in fact what a lot of users suffer with.

```
output tracking script
```

Scripting summary

We can make our script arbitrarily complex, and ultimately we could handle all the issues if we really wanted to. This is not really the point, however, as we are more

interested in the most effective way to document and implement the flow. We have seen that in order to implement the basic elements of our more complex flow we need quite a number of separate scripts, with some dependencies and shared assumptions between them. Such a system is not easy to maintain. In addition, we didn't handle all the issues properly: for example making best use of the compute farm, or optimizing licence access based on dependencies.

Makefile solution

In the last section we went through all the elements in gory detail. We can leverage some of that for the makefile-based solution, because we still need the same kinds of things: environment setup, licensing, resource management etc.

In fact, most of the issues we identified in the previous section are not addressed in any meaningful way by the makefile, so here we can see that while superficially the makefile looks like a good solution, it really only handles the file and tool dependencies directly, and everything else has to be handled by scripting either inside the makefile or separately, if we handle it at all.

Again it's worth pointing out that this is one reasonable way to use makefiles for a problem like this: there are many different possibilities. Our goals are clarity, maintainability, and as far as possible, correctness.

Environment and setup

```
paths, environment variables, licensing
```

The makefile

Our makefile will include a richer set of targets than before.

```
dependencies, with a richer solution than before,
including parallelism, make clean, make all, and
assumptions about configurations.
```

Issues and optimizations

Our makefile-based solution has allowed us to capture the dependencies, with the assumption that the timestamps of the files are valid. That means that configuration management has been relegated to a separate task: it's not handled explicitly within the makefile. As we've observed, this is risky when you go back in time and access an earlier configuration. The makefile doesn't know which file you used last time, because it doesn't

maintain any state between runs. Every time the makefile runs, it traverses the dependencies described in the makefile, and checks the files based on those dependencies. We could implement something more complex by using checksums or some other kind of signature-based system, but the makefile won't give us any explicit help with that, so we would end up creating additional scripts, which is not really in the spirit of using the makefile as a solution.

The second issue that we didn't address adequately in the makefile was parallelism. This is because make does not understand the notion of a compute farm. The **`make -v`** switch simply passes multiple jobs to the operating system for scheduling. It makes no allowance for the resource requirements of any particular job.

The third issue with the makefile solution is that it doesn't track or optimize licence usage. It will certainly allow tools to ask for licences (although it doesn't provide any assistance in that regard), but if a licence is not available, the makefile will allow the job to block until a licence becomes available. This can be a big bottleneck in a complex flow. It would be much smarter if the makefile could look at compute resources, licence resources, and jobs in the queue, and find the best way to get it all done, consistent with the known dependencies. Unfortunately, a makefile is not capable of this level of optimization without a great deal of additional scripting.

FlowTracer solution

How much of the complexity of our problem can FlowTracer manage?

Let's go back to our original set of issues for this example, and work through them one by one.

Paths and environment setup

FlowTracer provides a mechanism for capturing and re-using environmental data. The idea is that once this is set up, it can be accessed from any flow description as a black box. The great benefit of this is that we can make changes to either the flows or the environment setup without affecting other elements of our system. Information separation is one of the basic principles of good software design.

Here's how we set up the environment for the tools and the licensing:

```
environment setup
```

FlowTracer understands licensing directly, and it can monitor and report on licence usage as it tracks jobs. This is an important optimization capability, because it allows users to understand where they might have too few or

(equally importantly) too many licences. This allows for maintenance and upgrading of the overall design environment over time as needs and workloads change.

Configuration management

Following the scripting and makefile examples, we will make the simplifying assumption that the configuration management takes place outside the system. In other words the user is responsible for making sure the right files are in place prior to running the flow.

Our job, therefore, is to make sure that the flow manager knows which files are current and which are not, so that the correct tools can be executed at runtime.

FlowTracer approaches this differently from other solutions, in that it maintains a persistent database about the flow, including the exact information regarding which files were used in previous runs. There is a bootstrapping phase, in which you run the flow once or twice to build up that database, but once it's current, it will correctly understand the status of the current files, and will therefore execute the dependencies correctly.

To be more specific, if FlowTracer notices that an input file is *different* from the version used to create a particular output file, it registers that as a change which invalidates the output. This change may be manifest as an earlier or a later timestamp. This is only possible because of the persistent database. A makefile-based or scripting

solution that does not have a persistent database cannot understand the idea of a changed file: it can only understand the concept of a newer file.

Tool encapsulation

FlowTracer offers a number of different ways to capture the behavior of a specific tool.

```
tool encapsulation for

Program 1,

Program 2, and

Program 3
```

Computing resources & job mapping

FlowTracer incorporates a powerful compute farm scheduling and dispatch capability, coupled with visualization and reporting so you can track what's going on (actually a superset of the NC capability we used in the scripting example). If we simply tell FlowTracer about the capabilities of the compute farm, and annotate each job with its resource requirements, FlowTracer will handle the rest.

```
Resource management information
```

The flow

The flow description language is again pretty simple: environment, tool, inputs and outputs. Here it is.

```
FDL for the 3 tools
```

The basic FDL is simple, but we've added an enhancement: we include information about the job's resource requirements: for example the fact that Program 1 runs on Windows, while Program 2 and Program 3 run on Linux, and that their resource needs differ. We could have put some of this information in the encapsulation, but you will recall that Program 2 and Program 3 are the same program, so we can use one encapsulation, and specify the differing resource requirements right here in the flow.

Tracking outputs

...

FlowTracer summary

The hard parts of making this work are not the dependencies. It's the integration of all the other elements, like licensing, parallelism, and resource management. Because FlowTracer is built from the ground up to handle these issues, it's a lot easier to solve them with FlowTracer than with either scripts or makefiles.

Not only did we handle all the issues directly, but our code size is smaller than in either of the other cases and it's also easier to read and maintain.

Finally, FlowTracer automatically provides visualization so we can track what's going on as we run the tools.

The one area in which we don't have any explicit integration (also true for the scripting and makefile solutions) is configuration management. However, because we have a persistent database, FlowTracer can correctly implement the dependencies in all cases, which is not true for makefiles or scripts.

4 Flows for the big league

Our final example is full scale. It's based on an ASIC implementation flow, with placement, routing, timing analysis, physical verification, and the back-end optimization required for a modern chip design.

We will be dealing with thousands of files, a range of tools from different vendors, a big compute farm, and jobs that can last days. We're going to pull out all the stops.

Issues of scale

In building anything of complexity, scale drives the requirements. How do we ensure manageability across hundreds of actions, spread across the globe, implemented by a large team of users working separately and together? Scalability is certainly about performance, but even more it is about robustness. We put a lot of effort into the development of the flows, in order that we can proceed with our design tasks confident in the knowledge that our decisions and constraints are being supported by the infrastructure.

Robustness, in turn, has many aspects. Among them are these:

1. Ability to address all the issues: licensing, distribution, global WANs, tool management, configuration management and so on.

2. Ability to test and validate the flow itself

3. Informative error messages and graceful failure modes when things go wrong

4. Visualization, reporting, and history management for auditing and effective management of the process

These are the goals and evaluation criteria for an effective flow management capability. We have explored their use with three different implementation strategies in previous chapters. You have seen that it is possible to solve these problems in a range of different ways, but that there are benefits to using a purpose-built solution like FlowTracer.

In this chapter we will look at how FlowTracer addresses a much larger example, with many tools, multiple queues, and other aspects of complexity. Think of this as a kind of template that you can use to implement FlowTracer on your own flows. We will break it down into sections as follows:

- Tool encapsulation
- Configuration and file management
- Computing resource management

- Licence management
- Flow description
- Automation
- Visualization
- Reporting

Because of the complexity of this example, it won't fit onto a single diagram. Instead we will illustrate the elements of each step as we go.

Tool encapsulation

Configuration and file management

Computing resource management

Licence management

Flow description

Automation

In previous sections we did not address this issue. Automation has to do with the addition of control structures and customized decision-making inside the flow. It can be important, but perhaps more significantly, simple automation can greatly improve the quality of the "OK/not OK" decision that is made after every tool is run. So far we have assumed that the successful creation of a new set of output files without errors means the step was successful. In fact we could easily use **`grep`** or other programming tools to implement more sophisticated validation of outputs. In this section we will show how this can be done with FlowTracer.

Visualization

This is one of the areas where FlowTracer delivers a big win. By showing the status of the flow as a graph, and updating the graph in real time as the tools run, the user has a powerful insight into the progress of the tasks and the satisfying of the dependencies.

Reporting

5 FlowTracer Notes

In this chapter we will summarize the most important capabilities of FlowTracer, and how to use them.

Scripting advantages

Scripting weaknesses

Makefiles

Makefile advantages

Makefile weaknesses

FlowTracer

Index

Runtime Design Automation, Inc. 3700 Augustine Drive, Suite 139, Santa Clara, CA, 95054.
http://www.runtime.com
408.492.0940