

ALTAIR

ONLY FORWARD

Altair FlowTracer 2025.1.2

Developer Guide

Contents

Altair FlowTracer Developer Guide	5
FlowTracer Roles.....	6
Environment Management.....	8
Parameterized Environments.....	9
Composite Environments.....	11
Environment Examples.....	11
Refresh Environments.....	12
Develop Environments.....	12
Environment Management: Commands.....	16
Standard Environments.....	18
Environment Debugging.....	20
Manage Umask.....	21
Environment Management: Limits.....	22
Execute Jobs From the Command Line.....	26
Command Line Representation.....	28
Exit Status.....	28
Stdout and Stderr.....	31
Handling Pipes and Redirection with VOV Wrappers.....	32
Conflict Detection.....	33
Firing Jobs.....	34
Command Interception.....	35
Flowbuilding.....	36
Building the Flow with vovbuild.....	37
Flow Description Language (FDL).....	38
Flow Language Procedures.....	39
FDL Procedures Reference.....	42
J versus T.....	60
Build Flows with vovbuild.....	60
Conflicts in vovbuild.....	62
Use O_CONFLICT to Choose Behavior in Case of Output Conflict.....	62
vovbuild Argument Parsing.....	63
Task Oriented Flows.....	64
Dependencies in FDL.....	66
Forcing Dependencies.....	67
vtk_transition_add.....	68
Difference Between E and ves.....	69
Linear Coding in FDL.....	69
Set Creation in FDL.....	69
Conditional Flows.....	74
Decisions in Flows with IFJOB.....	76
Parse Configuration Files.....	77

RECONCILE_WITH_FILE_SYSTEM.....	78
Create Directories in FDL.....	78
File Generation in FDL.....	79
Generate FDL Using the Instrumentation Library.....	79
Run Jobs in an xterm.....	80
FDL and make Comparison.....	82
Run Interactive Jobs with vxt.....	83
Tcl Interface to Various Set Operations.....	84
Tool Integration.....	89
Wrappers.....	90
Track the Origins of Dependencies.....	91
Integration by Interception.....	92
Library-based Interception.....	93
Integration by Instrumentation.....	93
VOV Instrumentation Library (VIL).....	94
Instrumentation Procedure.....	96
VILtools.....	97
Integration by Encapsulation.....	100
Encapsulation Procedures.....	101
Capsule Post-Processing.....	105
Capsule On-the-Fly.....	105
Capsules for VHDL and Verilog Tools.....	106
Integrate Difficult Tools.....	107
Start a Subflow Before a Job is Completely Done with vovfileready.....	107
Runtime Change Propagation Control.....	110
Triggers on Files.....	112
Barriers to Change Propagation.....	112
vovbarrier.....	114
clevercopy / cleverrename.....	115
Flow Library.....	117
Add a Flow.....	117
Flow Library Packager vovflowcompiler.....	118
Flow Library Procedures.....	119
Files, Equivalences, Exclusions.....	121
Files and File Names.....	121
Canonical and Logical File Names.....	121
Databases.....	122
The LINK Database.....	124
The JOBSTATUS Database.....	126
Define Equivalences for File Names.....	127
Equivalence Cache.....	129
Use vovequiv to Check Equivalences.....	130
Taskers: Mixed Windows NT and UNIX Environment.....	130
Exclude Files From the Graph.....	131
Change the exclude.tcl File.....	134
Automatic Zipping and Unzipping Files.....	135
Makefile Conversion.....	137

Use Makefiles.....	140
Configure vovmake.....	142
Makefile to FDL Utility.....	144
Advanced Tasker Topics.....	146
Hardware Resources.....	146
HOST_OF_jobId and HOST_OF_ANTECEDENT.....	150
Resource Management.....	151
Resource Mapping.....	151
Resource Reservation.....	153
Delete Resources.....	155
Resource Daemon Configuration.....	155
vovresourcemgr.....	156
Policies and Resources.....	158
Add Resources.....	159
CGI Interface.....	161
Write CGI Scripts.....	161
VOV CGI Procedures.....	162
Procedure Descriptions.....	167
Events.....	173
Miscellaneous.....	175
Controlling Jobs on Tasker Host.....	175
Customize the Browser Interface.....	176
TclXML Parser.....	177
Create New Tcl Commands.....	184
Set Names.....	185
vovexport.....	185
GUI_Label Property.....	186
Limits for Objects and Strings.....	187
Upgrading to 2019.9 and Beyond.....	190
Legal Notices	193
Intellectual Property Rights Notice.....	194
Technical Support.....	200
Index	201

Altair FlowTracer Developer Guide

This chapter covers the following:

- [FlowTracer Roles](#) (p. 6)
- [Environment Management](#) (p. 8)
- [Execute Jobs From the Command Line](#) (p. 26)
- [Flowbuilding](#) (p. 36)
- [Flow Description Language \(FDL\)](#) (p. 38)
- [Tool Integration](#) (p. 89)
- [Integrate Difficult Tools](#) (p. 107)
- [Runtime Change Propagation Control](#) (p. 110)
- [Flow Library](#) (p. 117)
- [Files, Equivalences, Exclusions](#) (p. 121)
- [Makefile Conversion](#) (p. 137)
- [Advanced Tasker Topics](#) (p. 146)
- [Resource Management](#) (p. 151)
- [CGI Interface](#) (p. 161)
- [Events](#) (p. 173)
- [Miscellaneous](#) (p. 175)

FlowTracer Roles

When using FlowTracer in large deployments, three main roles emerge: Administrator, Developer and User. In smaller organizations, a given person may have multiple roles, but the duties of each role still exist.

The FlowTracer Administrator will install and configure the product and help other users to access it. FlowTracer Administrator enables effective use of FlowTracer:

- Installing FlowTracer
- Installing needed licensing
- Configuring FlowTracer for the whole company
- Managing local files
- Establishing security rules
- Defining user roles
- Helping users set up command line environment

A FlowTracer Developer will create and maintain the job definitions and dependency rules for different applications that form the basis for a flow that is registered with FlowTracer to manage.

FlowTracer Developers use the features that define and manage flows:

- Makefiles
- Tcl Scripts
- Flow Description Language (FDL)
- Enabling runtime tracing
 - Encapsulation
 - Instrumentation
 - Interception
- vtk_* Application Programming Interfaces (APIs)
- Command Line Interface (CLI) programs that manage a flow

A FlowTracer User is the recipient of flows pre-developed by a developer. The user relies on the standardization and accuracy provided by FlowTracer to increase their productivity. Typically, they will:

- Execute flows that have been prepared by somebody else.
- Change input files and ask FlowTracer to run the dependent jobs that generate new outputs (this is called "retracing").
- When using the FlowTracer console, the user watches the job nodes changing from purple to yellow to green. When the whole set of nodes is green, the full set of jobs has run and the results can be checked.
- If a job node turns red (FAILED), the user needs to investigate the failure: is it a real failure caused by a problem in the input data, or is it caused by external causes such as disk full, missing licenses, bad machines, dependency violations, or whatever else?
- If the jobs appear to be stuck on some status (think of "color"), the user investigate the cause.

This manual is for topics related to the usage of FlowTracer with pre-existing flows.

Environment Management

To ensure the correct and repeatable behavior of the tools, the environment must be controlled. This chapter explains how VOV supports multiple reusable environments.

Environments Overview

When using the UNIX shell, it is standard procedure to establish a working environment by setting environment variables within the shell login script, such as the `.profile` or `.cshrc` file.

The Altair Accelerator products are built to establish control and direction from the values of environment variables. The set of such controls is long enough that it makes it hard to manage all the values. A technique is supported to help manage the complexity so that a person does not use a single monolithic login script to set every possible environment variable.

This technique is to partition the set of control environment variable into working groups that are appropriate for use in certain situations and by certain activities.

Each group is given a name, and tools are provided to modify the environment to add or delete environment variables by group name.

The technique of using FlowTracer tools to set a particular working environment by name is used instead of doing a native UNIX sourcing of a variety of different scripts as needed, or of one large script sourced at login.

Definitions and Benefits

In both UNIX and Windows platforms, there is a mechanism that allows processes to communicate information to their subprocesses environment variables. For example, in a shell, the following command can be used set the value of the environment variable VOVDIR.

```
% setenv VOVDIR /remote/VOV/<version>/<platform>
```

All jobs started by the shell inherit the environment. This enables finding the root directory of the VOV installation by looking up the value of the variable VOVDIR. The behavior of many programs is affected by such environment variables.

Environment indicates the collection of all the environment variables. A single environment that can execute all tools required in a project is desirable but not always feasible. For example, if a project requires tools from many vendors, the PATH variable may become too long. In other cases, there may be conflicting requirements for the value of some variables (for example, LM_LICENSE_FILE). for these reasons, multiple environments are a valuable asset.

To address these issues, some users have developed a more or less unstructured collection of setup files, scattered around the file system, leaving it up to the individual designers to remember to use those files when needed. Others have developed a system in which the designers, instead of invoking the tools directly, invoke specialized wrappers. Wrappers set up the appropriate environment for the tool on the fly before invoking the actual tool.

If you have scattered setup scripts, the VOV environment management facilities offers a way to organize them under a logical framework. If you have wrappers, you can keep using them as a complement to the VOV environment management facilities.

VOV environment facilities let you:

- Precisely control the environment used by each job in your flow
- Load each environment on demand rather than use an overloaded environment
- Simplify the shell startup script (such as `~/.cshrc`)
- Create many small environments that are optimized and easy to maintain
- Share the environments across multiple shells (ksh, tcsh, ...)
- Share the environments among the project team members
- Share the environments among multiple projects

Environment Basics

Each VOV environment has a name. This name is an alphanumeric string, usually in uppercase. Underscores are also allowed. For example, an environment could be named `TEX` for running Latex, and another environment named `SYNOPSIS` for running the Synopsys tools.

The name of the current environment is represented by the environment variable `VOV_ENV`. If this variable is not set, VOV assumes that the name is `DEFAULT`.

Environment definitions are found in the directory `$VOVDIR/local/environments`. The optional variable `VOV_ENV_DIR` can be used to identify other directories where additional environments can be found. The value of this variable is a list of directories separated by colons ":" on UNIX systems and by semicolons ";" on Windows.

The environment files may be written in C-shell, in Bourne Shell, or Tcl. Regardless of the syntax used to describe it, any environment can be used in any shell; an environment written in C-shell can be used, even if ksh is being used.

VOV stores the name of the environment that is used to execute each job. Before re-executing a job, VOV switches to the appropriate environment. The switch of environment is actually performed on the taskers. Taskers cache environments, resulting in instantaneous switches between environments.

Parameterized Environments

An environment definition may accept parameters. This is useful, for example, to select different versions of some tool.

Parameters are passed to the script either one of syntaxes shown below. The older syntax uses parentheses, which need quoting when used from the shell

In this syntax, parentheses are used:

```
environmentName (parameter1[,parameter2]...)
```

In this 'comma' example, parentheses are not used:

```
environmentName,parameter1[,parameter2]
```

The parameters are a comma-separated list of tokens that are placed in parentheses after the environment name or after the first comma. No spaces are allowed. Arguments cannot contain

commas, spaces, quotes, or other special characters. Proper quoting must be used when switching to a parameterized environment from the command line.

Examples

These examples show how to add a parameterized environment. The two pairs of lines have the same effect.

```
% ves '+D(DISPLAY=tahoe:0.0) '  
% ves '+D,DISPLAY=tahoe:0.0 '  
% ves '+SYNOPSIS(1998.08) '  
% ves '+SYNOPSIS,1998.08 '
```

From inside the environment script, the parameters are accessible by means of the variable '\$argv' in C-shell or the list \$argv in Tcl. Parameters are passed to both the `start.tcl` script, and the `end.tcl` script.

For example, this is the definition of the standard environment `D`:

```
# This is D.start.tcl  
# An environment to define variables.  
# Usage: ves +D,VAR1=value,VAR2=value,...  
foreach arg $argv {  
    if [regexp {[^=]+)=(.*)} $arg all var value] {  
        setenv $var $value  
        lappend env(D_env_vars) $var  
    }  
}
```

Multiple environment variables can be set while launching a job using `D` using parentheses, like this:

```
% nc run -e "D(VOV_LIMIT_maxproc=8192,VOV_LIMIT_openfiles=8192)" env
```

Multiple environment variables can also be set for individual jobs by using the comma notation without parentheses and without quotes.

```
% nc run -e D,VOV_LIMIT_maxproc=8192,VOV_LIMIT_openfiles=8192 env
```

Inside a jobclass definition file, a parametrized environment can be specified like this:

```
set VOV_JOB_DESC(env) "SNAPSHOT+D,VOV_LIMIT_maxproc=8192,VOV_LIMIT_openfiles=8192"
```

Curly braces are also supported in the use of environment variables. This permits the use of commas, among other special characters. For example:

```
nc run -e 'D(FOO={value,with,commas},BAR=normal_value)' ...  
  
and  
  
ves 'D(FOO={bar,baz}) '
```

Another example:

```
set VOV_JOB_DESC(env) "SNAPSHOT  
+D,VOV_LIMIT_maxproc=8192,MY_CSV_VAR={a,b,c},VOV_LIMIT_openfiles=8192"
```

Composite Environments

Complex environments can be built via *composition*; use the operator "+" with `ves`.

For example, if you there are two environments `E1` and `E2`, they can be combined by switching to the environment `E1+E2` as shown below:

```
% ves E1+E2
```

Order of Environment Components

The order of the environment components in environments is significant, because some environment definitions can be destructive, while others may be in conflict with each other.

For example, the environment `BASE` sets the variable `PATH` to a well-defined list of directories, ignoring any previous value. For this variable, the environments `BASE` and `E1+BASE` are identical, because it is completely determined by the `BASE` environment. Note that in general, the environment `BASE+E1` is a true composite environment (assuming that `E1` is not destructive).

For example, to use tools from Synopsys and Virage, they can be run in the combined environment `SYNOPSYS+VIRAGE` as shown below:

```
mars chip@mercury BASE src/vhdl > ves SYNOPSYS+VIRAGE
mars chip@mercury SYNOPSYS+VIRAGE src/vhdl >
```

Environment Examples

The following is an example of a start script for the environment named `SYNOPSYS`. Typically, this script is stored in: `$VOVDIR/local/environments/SYNOPSYS.start.csh`:

```
#####
# Typical Synopsys Environment: SYNOPSYS.start.csh
#####

setenv SYNOPSYS      /home/eda/synopsys/synopsys3.0
setenv SIM_ARCH      sparc
setenv LD_LIBRARY_PATH `vovenv APPEND -: $SYNOPSYS/$SIM_ARCH/sim/lib
                        $LD_LIBRARY_PATH`
setenv MANPATH        `vovenv APPEND -: $SYNOPSYS/doc/sim/man  $MANPATH`

set path = `vovenv APPEND $SYNOPSYS/$SIM_ARCH/sim/bin  $path`
set path = `vovenv APPEND $SYNOPSYS/$SIM_ARCH/motif/bin $path`
set path = `vovenv APPEND $SYNOPSYS/$SIM_ARCH/syn/bin  $path`
set path = `vovenv APPEND $SYNOPSYS/$SIM_ARCH/sge/bin  $path`
```

The following is another example of building a composite environment:

```
# -- A combined environment: call it COMBI.start.csh
# -- Get Synopsys and EPIC tools together.
source $VOVDIR/etc/std.vov.aliases

# Show two different ways to use ves.
```

```
ves BASE+SYNOPSIS  
ves +EPIC
```

Refresh Environments

To save time while switching environments, taskers cache up to eight environments, which makes switching instantaneous. The cached environments can be refreshed either through the GUI by clicking **Project > Taskers control > Refresh environments**, or by entering:

```
% vovtaskermgr refresh
```

Refreshing is necessary if you modify an environment after the taskers have cached it. Restarting the taskers achieves the same result at a slightly higher cost.



CAUTION: On Windows NT, refresh does not work. You must restart the taskers instead.

Develop Environments

Each environment can be described with Tcl, C-shell, or Bourne-shell scripts, which allows the re-use of existing scripts. The Tcl syntax is recommended; the resulting environment can be used on both UNIX and Windows systems, and Tcl supports aliases.

When necessary, the environment definition, written in any of the above languages, is automatically converted to use with Bourne-shell and the derivatives of Korn-shell and bash, C-shell and derivatives, Tcl scripts, and DOS prompt.

Each environment is described by the files shown in the following table. A minimal description of an environment consists of the `start*` script and the DOC file.

The `.end*` script is usually needed only for environments that are used with `ves` from the command line. The vovtasker binary caches eight recently used environments. When all eight cache slots are full and a new one is needed, the least-recently used environment is discarded without calling any of the `.end*` scripts.

Some C-shell implementations have small limits on some important variables, such as the length of the `path`. If environments are needed that exceed those limits and `tcsh` is on the hosts, the `.tcsh` script suffix can be used.




Note: A vovtasker does not execute jobs using any shell. Instead, a vovtasker uses the `execve()` system call. The shell implied by the environment script suffix is only used to compute the environment.

Suffix	Language	Description
start.csh	C-shell	Initialization scripts, executed before entering the environment. If multiple scripts exist for the same environment, VOV will prefer in the following order, Tcl, C-Shell, Bourne-Shell, tcsh.
start.sh	Bourne-shell	
start.ksh	Korn-shell	
start.tcl	Tcl	
start.tcsh	tcsh	
end.csh	C-shell	Termination scripts, read when exiting the environment. See comment above about choice of language
end.sh	Bourne-shell	
end.tcl	Tcl	
end.tcsh	tcsh	
pre.tcl	Tcl	Executed before the execution of the job.
post.tcl	Tcl	Executed after the execution of the job.

General Rules

A good environment definition is minimal, incremental and reversible.

 **Note:** These general rules are recommendations; they are not requirements. VOV works well with environments that are not minimal, incremental, or reversible.


- **Minimal:** The definition adds only the minimum number of variables necessary to correctly execute a certain class of tools.
- **Incremental:** It builds upon the original environment.
- **Reversible:** It is possible to restore the original environment.

Rules to Write Environments in c-shell

In this example, an environment is created. The environment is named `MYENV`, which contains the directory `/usr/local/bin` in the path. The start script for this environment is `$VOVDIR/local/environments/MYENV.start.csh`.

In C-shell, either the shell variable `path` or the environment variable `PATH` can be set. An example follows:

```
# -- This is MYENV.start.csh
set path = ( /usr/local/bin $path )
```

 **Note:** This solution has a disadvantage. Switching to the `MYENV` environment, the resulting `PATH` may contain duplicates of `/usr/local/bin`. In the long run, it is possible for the `PATH` variable to exceed its maximum allowed length (about 1kB), which can be imposed by some implementations of `csh`.

A better solution avoids duplicates. For this purpose, use `vovenv`, which is a script to manipulate environment variables. The usage for `vovenv` is:

```
vovenv OPERATION [-colon] wordlist
```


Operation	Description
DELETE	Deletes word from list.
APPEND	Adds the word at the end of the list.
PREPEND	Adds the word at the beginning of the list.

If `-colon` is used, the list is assumed to be colon-separated, as for the environment variable `PATH`. Otherwise, it is a space-separated list such as the C-shell variable `path`; Instead of `-colon`, `-:` can be written.

For example:

```
# -- This is a better MYENV.start.csh
set path = `vovenv PREPEND /usr/local/bin $path`
```

In the `MYENV.end.csh` file, revert the changes made by the start script with the operation `DELETE` of `vovenv` as shown below.

 **Note:** This practice should be applied to all the environment variables that describe lists of files or directories, such as `PATH`, `MANPATH`, `LD_LIBRARY_PATH` and `LM_LICENSE_FILE`.

For example:

```
# -- This is MYENV.end.csh
set path = `vovenv DELETE /usr/local/bin $path`
```

Rules to Write Environments in Tcl

If you are familiar with Tcl, consider writing the environment definitions in this language. The advantage is the portability between UNIX and Windows.

 **Note:** Tcl must be used to describe environments for Windows.

To write an environment in Tcl, it is important to remember that all environment variables are available through the associative array `env()`. For example, the value of the variable `VOVDIR` is accessible as `$env(VOVDIR)`. You also need to become familiar with the following Tcl procedures supplied by VOV:

```
setenv name value
unsetenv name ...
```


```
vovenv name separator op arg ...  
alias name words ...
```

These procedures look similar to their C-shell equivalent. In fact, they are Tcl procedures that are defined in `$VOVDIR/tcl/vtcl/vovenvutils.tcl`.

The procedures `setenv` and `unsetenv` behave as their C-shell counterparts. The procedure `vovenv` has the same functionality as the shell utility `vovenv`, but with a different syntax. Refer to the VOV/Tcl book for more information about these procedures.

Error handling: if errors are detected while processing of the environment definition, do not call `exit`. Instead, use the call `error`.

The environment `MYENV` that was described in the previous section can be described with the Tcl syntax as shown below.

 **Note:** Because the colon ":" is used as a path separator, this example only works for UNIX. (The example shown after this works with Windows.)

```
# This is MYENV.start.tcl  
vovenv PATH : PREPEND /usr/local/bin
```

```
# This is MYENV.end.tcl  
vovenv PATH : DELETE /usr/local/bin
```

To have an environment that also works on Windows the following form can be used:

```
# This is MYENV.start.tcl  
if { $::tcl_platform(platform) eq "windows" } {  
    # Quote ; because it is the command separator in Tcl.  
    vovenv PATH ";" PREPEND c:/local/bin  
} else {  
    vovenv PATH : PREPEND /usr/local/bin  
}
```

For Windows environments, care must be taken in dealing with case insensitivity and with the confusion between backward and forward slashes. The variables `Temp` and `TEMP` are indistinguishable in Windows, because they differ only in case. In Tcl, however, `env(Temp)` and `env(TEMP)` are distinct and only one of the two can be used. If the value of an environment variable is needed, first call the procedure `nt_preprocess_env` to create an upper-case only version of the variable:

```
set tmpdir $env(TEMP) ;# May not work  
  
nt_preprocess_env  
set tmpdir $env(TEMP) ;# Guaranteed to work.
```

Another useful procedure is `nt_slashes`, which is used to convert the direction of slashes in file names. Example:

```
nt_preprocess_env  
set tmpdir [nt_slashes $env(TEMP)]
```

Support for Aliases

Some customers desire the ability to define aliases in environments. Aliases are useful shorthands and reduce typing. They are useful only in command shells. Aliases are not used when taskers execute jobs.

To define an alias, you have to describe an environment using Tcl syntax. Aliases defined in the environment become available to the following shells: C-shell, Tcsh, Korn-shell. They are not available in Bourne-shell or in DOS.

The synopsis to define an alias is:

```
alias NAME WORD ....
```

For example: define an alias called 'l' for 'ls -sF':

```
# At the end of $VOVDIR/local/environments/BASE.start.tcl
alias lll ls -sF
```

```
% ves BASE% alias lll
ls -sF
```

Pre and post Conditions

As part of environment definition, you can prepare two scripts, called `NameOfEnv.pre.tcl` and `NameOfEnv.post.tcl`, which can be used to take care of pre- and post-conditions on a job by job basis.

Environment Management: Commands

The commands described in this section are used to manage **named environments** and can also be used independently of VOV.

Command	Description
vel	List all available environments.
ves	Switch between environments.
vec	Create an environment variable for each LOGICAL name used in the <code>equiv.tcl</code> file (works only if connected to a VOV project)

vel: List Environments

To list all available environments use `vel` (Vov Environment List). Here is an example:

```
% vel
vel: message: Environment directories:
1 /release/VOV/latest/sun5/local/environments
1 * tcl BASE          UNIX utilities, X11, and VOV.
1 . tcl D             Define variables: Usage: ves "+D(V1=value1,...)"
```

```
1 . tcl DEFAULT      Just a name for whatever you already have.
1 . tcl DOT           Append "." to the path variable
1 . tcl GNU           gcc, g++, emacs1 . tcl OCTTOOLS Octtools 5.1
1 . csh CSHRC         Source the ~/.cshrc file.
1 . csh LIBTECH       Library Technologies, Inc.
1 . csh SOS           SOS from ClioSoft (Revision control manager)
1 . csh SPICE         The analog simulator SPICE.
1 . csh TRAINING      Used for VOV Basic Training
1 . tcl MSDEV         Windows: VisualC++ development.
```

First, `vel` shows a list of directories where environments can be found. Then for each environment, the command shows:

- The ordinal number of the directory where the environment definition resides
- An asterisc to identify the current environment (BASE in the example) or a dot for all other environments
- A label to identify the syntax used to describe the environment, which is either Tcl or csh
- The name of the environment
- A short description of the environment

ves: Switch Environment

To switch to an environment use `ves` (Vov Environment Switch) as in the following examples:

```
% ves BASE
% ves SPICE
% ves +TRAINING
% ves BASE+TRAINING
% ves 'BASE+D(DISPLAY=tahoe:0.0) '
% ves BASE+D,DISPLAY=tahoe:0.0
```

The first two examples are simple switches from the current environment to the specified environment. Each switch involves two steps:

1. Exiting the current environment
2. Entering the new environment

The third and fourth examples illustrate the use of combined environments. If you prefix the environment name with a "+", VOV enters the new environment without exiting from the current one.

The fifth example shows the use of [Parameterized Environments](#). Proper quotation must be used because parameter passing requires special characters, the parentheses, which can be consumed by the shell if not protected.

The sixth example show the same parameterized environment specified using the "comma syntax" instead of the parentheses. This syntax is easier to use because it does not require quoting of the environment specification.

A side effect of switching environment is that the [prompt](#) is changed, unless the variable `VOV_USE_VEP` is set to 0.

Using a Tcl file as Environment

You can also use `ves` with a Tcl file as argument. A typical example can be the `setup.tcl` file in the SWD directory of a project

```
% ves proj.swd/setup.tcl
```

vec: Promote Equivalences to Environment Variables (UNIX only)

The [equivalence file](#) may establish an equivalence between a logical name and a physical path and it is not necessary that the logical name be the value of a predefined environment variable.

For example, the equivalence file may define:

```
vtk_equivalence WORK1 /remote/projects/work1
```

so that the files under `/remote/projects/work1/*` will be referred to with the name `${WORK1}/*`.

When working from the command line, it may be convenient to define the variable `WORK1`. This can be done automatically with the command `vec`.

Example:

```
% echo $WORK1
WORK1: Undefined variable.
% vec
% echo $WORK1
(remote/projects/work1)
```

Standard Environments

Most VOV installations offer at least the following environments:

Environment Name	Description
BASE	A minimal environment for the basic operating system tools, X11, and VOV. This environment typically resets the PATH variable. Use this environment to cut the size of the PATH variable.
CLEAN	A destructive environment that eliminates most environment variable except a few essential ones. Normally used in conjunction with some other environment, as in CLEAN+BASE.
CSHRC	The environment obtained by sourcing the <code>~/.cshrc</code> file. Use this with caution, since many <code>~/.cshrc</code> files fail to setup the environment properly, or have undesired side effects.
D	A parameterized environment used to define environment variables. See also the environment 'U' to unset variables. Examples: <div> <pre>% ves 'BASE+D(DISPLAY=tahoe:0.0)'</pre> </div>

Environment Name	Description
	<pre>% ves BASE+D,DISPLAY=tahoe:0.0</pre>
DOT	An environment that explicitly adds "." to the search path. Please remember that having a "." in the PATH is actually a security risk, so avoid using this environment if possible.
DEFAULT	An inactive environment. The use of this environment is discouraged, because it implies giving up on environment control.
M	<p>Interface to the "modules" package. This environment requires customization. The list of modules to load is passed as a comma separated list of module names. Examples:</p> <pre>BASE+M(cadence/5.1) BASE+M(cadence/5.1,module1,module2)</pre>
SNAPSHOT & SNAPPROP	<p>Use "snapshot" environments captured into a file or into a property using the utility <code>vovenvcapture</code>. When saved into a file, the environment can be used on a job by using the SNAPSHOT environment which takes the file name as a parameter, i.e. the job environment specification is <code>SNAPSHOT(name_of_the_file)</code>. If the environment snapshot is saved as a property, the environment can be used by specifying SNAPPROP environment, using the job environment specification <code>SNAPPROP(vov_id)</code></p> <p>When <code>vovproject start</code> is executed, it automatically saves a snapshot of the server's environment to a file inside the server working directory.</p> <pre>vovenvcapture: Usage Message DESCRIPTION: Capture the SNAPSHOT environment into a file or add it as a property to an object. USAGE: % vovenvcapture -help % vovenvcapture -autofile % vovenvcapture -file <filename> % vovenvcapture -id <id> NORMAL FLAGS: -help This message. -id Id of the object to add a property containing the the environment environment. Use the property with SNAPPROP(<id>) -file File to write the environment to. Use the file with the environment SNAPSHOT(<filename>) -autofile Automatically choose the file to write the environment to. The file will be located in a subdirectory of the SWD.</pre>

Environment Name	Description
	<pre>EXAMPLES: % vovenvcapture -help % vovenvcapture -autofile % vovenvcapture -file /Users/cadmgr/snapshots/fileABC % vovenvcapture -id 1</pre>
U	<p>A parameterized environment used to undefine environment variables. See also the environment 'D' to set variables. Examples:</p> <pre>% ves 'BASE+U(DISPLAY,SHELL) ' % ves BASE+U,DISPLAY,SHELL</pre>


Environment Debugging

A faulty or incomplete environment definition can cause problems with running jobs. Example: A job succeeds when executed directly from the command line but fails when executed by the taskers.

The utility `taskerdebug` can be used to debug the environments used by the `taskerdebug`. This utility prints all environment variables, aliases and equivalences into the file that is given as its first argument. In the following example, the environment named `BASE` is debugged, and `base.out` is the output file:

```
% ves BASE
% vov taskerdebug base.out
```

The command can now be retraced on selected taskers and check the file `base.out` for clues about the problem with the environment. If necessary, use the resource mechanism to direct the job to the desired tasker.

 **Note:** The `taskerdebug` command is implemented as a `csh` script on UNIX, and a `.bat` script for Windows. The command `machinfo` can also be used, which is implemented in Tcl. Implemented in Tcl allows using this command in either UNIX or Windows. This may be preferred, as the `machinfo` output provides more information than the command `taskerdebug` on Windows.

Environment Checking

To verify if an environment is good, use `vovenvcheck`, which checks that all variables set in the `start.*` file are properly unset in the `end.*` file. Example:


```
% vovenvcheck env_name
```

Manage Umask

The umask feature is used on UNIX to set the permissions on new files and directories. VOV supports umask with the environment variable `VOV_UMASK`. This variable is checked by the wrappers (vw, vov, etc.). When set, the wrapper adjusts the umask accordingly.

The environment variable `VOV_UMASK` is automatically set to the value of the umask in the submit environment when using an environment snapshot in Accelerator.

If using a named environment, `VOV_UMASK` may need to be set separately. To do so, add `D(VOV_UMASK=value)` to the environment specification.

 **Note:** The logfile of the job is created by the vovtasker, and its mode is controlled by `VOV_UMASK`. However, the date-stamped directory `YYYYMMDD` under `vnc_logs` is created at job submit time by the `nc run` command; the file permission (mode) of the logfile is controlled by the umask in the submit shell.

Examples with VOV_UMASK

This section provides examples of using `VOV_UMASK`.

In the following example, the umask is set in the current shell to a value that is different from the one in the `~/.cshrc` file. The 077 umask removes all permissions from group and others.

```
% umask 077
% umask
77
% grep umask ~/.cshrc
umask 022
% nc run -v 0 -r unix -wl csh -c umask
----STARTING ON some-host-name----
22
----END OF LOG----
----EXIT STATUS 0----
```

The next example shows the umask value that is set in the `~/.cshrc` file, since the csh ran that file when it started. This value overrides the value that is captured in the environment snapshot.

```
% nc run -v 0 -r unix -wl csh -fc umask
----STARTING ON some-host-name----
77
----END OF LOG----
----EXIT STATUS 0----
```

The following example shows the umask value that is set in the current shell, captured in the snapshot environment and reproduced in by `vw2`. That shell does not run the `~/.cshrc` file because of the `-f` option. The snapshot file contains the following:

```
...
VOV_UMASK='077'
export VOV_UMASK
...
```

The next example shows a umask value that is different from the current shell, that was most likely inherited from the one set up in the startup script of the owner of Accelerator. It could also be set by VOV_UMASK in the startup script for the BASE environment. The BASE environment shipped by Altair does not set VOV_UMASK.

```
% nc run -v 0 -e BASE -r unix -wl csh -fc umask
----STARTING ON some-host-name----
22
----END OF LOG----
----EXIT STATUS 0----
```

The following example shows the umask value that is set by the VOV_UMASK environment variable, which is different from both the current shell, and the shell startup file.

```
% nc run -v 0 -e 'BASE+D(VOV_UMASK=055)' -r unix -wl csh -fc umask
----STARTING ON some-host-name----
55
----END OF LOG----
----EXIT STATUS 0----
```

The following example creates the logfile named by the command, and possibly the 20060713 subdirectory of vnc_logs, if it did not exist. The subdirectory is mode 0700, since it was initialized from the umask in the current shell.

The logfile itself, 162014.8525, has mode 0620 because it was initialized from the OR of umask 055 and 666 in the wrapper vw2.

```
% nc run -e 'BASE+D(VOV_UMASK=055)' -r unix -wl date
Resources= unix CPUS/1
Env       = BASE+D(VOV_UMASK=055)
Command   = vw date
Logfile    = vnc_logs/20240713/162014.8525
JobId      = 00424353
----STARTING ON some-host-name----
Thu Jul 13 15:57:56 PDT 2024
----END OF LOG----
----EXIT STATUS 0----
% ls -ld vnc_logs/20240713
drwx----- 2 cadmgr rtda 4096 Jul 13 15:57 vnc_logs/20240713
% ls -l vnc_logs/20060713
-rw--w---- 2 cadmgr rtda 29 Jul 13 15:57 vnc_logs/20240713/162014.8525
```

Environment Management: Limits

In addition to environment variables, hard and soft limits can affect tool behavior. Hard and soft limits are set in the shell and are imposed by the operating system. Both UNIX and Windows provide a mechanism for processes to communicate information to their subprocesses via environment variables.

VOV uses special environment variables to communicate limit information. The environment variables are named VOV_LIMIT_<name>. name is the name of the limit, such as VOV_LIMIT_cputime.

A complete list of VOV_LIMIT_<name> environment variables include:

Variable	Value Type
VOV_LIMIT_coredumpsize	ByteSpec
VOV_LIMIT_cputime	VOV TimeSpec
VOV_LIMIT_datasize	ByteSpec
VOV_LIMIT_filesize	ByteSpec
VOV_LIMIT_maxproc	Ignore any unit
VOV_LIMIT_memorylocked	ByteSpec
VOV_LIMIT_openfiles	Ignore any unit
VOV_LIMIT_stacksize	ByteSpec
VOV_LIMIT_vmemoryuse	ByteSpec

The variables are interpreted in the wrapper program `vw2` which uses the C-language `getrlimit()` / `setrlimit()` system calls to set the limits for the child process when the job runs.

The value of a variable can be either of the following (depending on the value type):

- A ByteSpec, which is a sequence of digits followed by an optional unit indicator letter, e.g. 5M
- A TimeSpec, which is a sequence of integers and unit indicators, e.g. 3m, or 4h30m
- A sequence of digits without a unit indicator, e.g. 1000 (applicable to all value types)
- The string value 'unlimited' (applicable to all value types)

The recognized unit indicators (case insensitive) for the ByteSpec format are:

- K, kilobytes (same as no unit indicator)
- M, Megabytes (multiply by 1024)
- G, Gigabytes (multiply by 1024*1024)
- T, Terabytes (multiply by 1024*1024*1024)

The recognized unit indicators (case insensitive) for the TimeSpec format are:

- s, Seconds (same as no unit indicator)
- m, Minutes (multiply by 60)
- h, Hours (multiply by 60*60)
- d, Days (multiply by 60*60*24)
- w, Weeks (multiply by 60*60*24*7)
- y, Years (multiply by 60*60*24*365)

In Accelerator, the limit can be set at submission time. In the following example, a limit of 20 seconds of CPU is set for a job.

```
% nc run -e 'BASE+D(VOV_LIMIT_cputime=20)' .... shortjob.csh
% nc run -e 'BASE+D(VOV_LIMIT_vmemoryuse=5G)' .... bigjob.csh
% nc run -e 'BASE+D(VOV_LIMIT_vmemoryuse=5G)' .... bigjob.csh
```

In the following example, the command lines provide the roughly equivalent limit of 5GB for a job.

```
% nc run -e 'BASE+D(VOV_LIMIT_vmemoryuse=5000M)' .... bigjob.csh
% nc run -e 'BASE+D(VOV_LIMIT_vmemoryuse=5G)' .... bigjob.csh
% nc run -e 'BASE+D(VOV_LIMIT_vmemoryuse=5000000k)' .... bigjob.csh
```

The multipliers used with the memory specifications for limits are as follows: 'k' (KiB, 2^{10} , kibibytes) 'M' (MiB, 2^{20} , mebibytes) 'G' (GiB, 2^{30} , gibibytes) and 'T' (TiB, 2^{40} , tebibytes). The multipliers are case-insensitive.

If an integer has no unit specification, kilobytes are the units used. For example, 1024K is the same as 1024. In addition, using `unlimited` as a value is acceptable.

The following two examples show how to find the current limits.

For `csh/tcsh`:

```
% limit
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    unlimited
coredumpsize 0 kbytes
vmemoryuse   unlimited
descriptors  1024
memorylocked unlimited
maxproc      2048
openfiles    1024
```

For `sh/bash`:

```
bash-2.05$ ulimit -a
core file size (blocks)      unlimited
data seg size (kbytes)       unlimited
file size (blocks)           unlimited
max locked memory (kbytes)    unlimited
max memory size (kbytes)      unlimited
open files                   1024
pipe size (512 bytes)        8
stack size (kbytes)           unlimited
cpu time (seconds)            unlimited
max user processes            5119
virtual memory (kbytes)       unlimited

## Use option -H to get the hard-limits
bash-2.05$ ulimit -a -H
...
```

In most cases, everything in the shell startup file can be set as unlimited. This setup gives the tools the greatest possibility of a successful run. It is extremely rare for this method to not work.

When an environment snapshot is used when submitting jobs to Accelerator, the limits in the submission environment are captured automatically.

If using a named environment, the following Tcl code in the `ENV.start.tcl` script can be used to set the variables for limits and umask. This is the same as what is done in the environment snapshot.

```
if { [info command vtk_umask_get] != {} } {  
    setenv VOV_UMASK [vtk_umask_get]  
    catch {  
        vtk_limits_get limit  
        foreach n [array names limit] {  
            setenv VOV_LIMIT_$n $limit($n)  
        }  
    }  
}
```

Tcl-language API

VOV provides two API procedures to get and set the limits, `vtk_limits_get` and `vtk_limits_set`. Both procedures take a single array parameter, which contains the limits, keyed by name.

Because different platforms have different limits available, the VTK procedures support only a common subset of limits.

The following names are supported:

VTK Limit Procedure Name	Description
stacksize	Size of process stack segment, bytes
datasize	Size of process data segment, bytes
cputime	Maximum process CPU time, seconds
filesize	Maximum file size, bytes
coredumpsize	Maximum core dump file size, bytes

In the following example, Tcl code is used to eliminate `core` files by setting the `coredump` size limit to zero:

```
catch {  
    vtk_limits_get L ;          # get existing limits into array L  
    set L(coredumpsize) 0 ;    # set coredump limit  
    foreach n [array names L] {  
        setenv VOV_LIMIT_$n $limit($n) ; # propagate limits to envVars  
    }  
}
```

Execute Jobs From the Command Line

Execute Tools

You can build a design flow by executing tools, provided you activate runtime tracing for each tool in the flow. This is done by executing each tool from within a [VOV Wrapper](#). The examples in this page use the most sophisticated wrapper, called `vw2`.

At runtime, each tool connects with the server and exchanges information about its execution environment and its I/Os (input/output).

First, make sure that you are in the proper [environment](#) and in the proper working directory. Use `vel` to [list the available environments](#) and `ves` to switch to the proper environment. For example, if a tool should be executed in the BASE environment, type:

```
% ves BASE
```

To copy file `aa` to `bb` without FlowTracer, you will probably use the tool `cp` and the command:

```
% cp aa bb
```

With FlowTracer, do the same, except with the `vw2` prefix:

```
% vw2 cp aa bb
```

The same is true for all other tools. Examples:

```
% vw2 diff aa bb
% vw2 gcc -c -I../include hello.c
% ves SYNOPSIS
% vw2 dc_shell -f alu.scr
```

It up to you to decide which commands belong to the design flow and which commands do not. Sample commands that typically do not belong in the flow are:

```
% ls
% vi hello.c
% mail boss@work.org < todo.txt
```

You can verify that the tool has been added to the trace with `vsx`. This utility shows inputs and outputs of selected nodes in the trace. In the following example, the argument `!"` refers to the most recent job in the working directory:

```
% vw2 cp aa bb
% vsx !
00234069 VALID      ${PROJECT_DIR}/txt/aa
>>>> Node 00234051  VALID  vw2 cp aa bb
00234076 VALID      ${PROJECT_DIR}/txt/bb
```

Pipes and Redirection

To redirect stdout to a file, use the redirection option ``>'`. From the command line, you must quote or escape ``>'` so that `vw2`, not the shell, will do the actual redirection.

Here are two equivalent way of escaping the redirection option '>':

```
% vw2 cat aa bb \> cc
% vw2 cat aa bb ">" cc
```

Unlike the shell, `vw2` requires that the redirection options is separated by spaces from the other command line arguments.

Similarly, to use pipes, you can use the symbol `|` as in:

```
% vw2 cat aa bb \| grep ciao \> cc
```

Only stdout can be piped and/or redirected at this time.

Tools That Need No Wrapper

The tool integrator chooses whether, for a specific tool, runtime tracing should always be on or if it should be activated on demand. In the first case, any invocation of the tool will be traced, whether the wrapper is used or not. The tool `clevercopy` is such an example. Therefore, the following two commands are usually the same because `clevercopy` is instrumented. With some slow NFS servers, you may need to wrap `clevercopy` with a VOV statement.

```
% clevercopy aa bb
% vov clevercopy aa bb
```

Troubleshooting and Verbosity

In versions earlier than 2016.09u9, setting `VOV_VW_VERBOSE` to a nonzero value would cause the `vw/vov` wrapper to create a file called `.vw2_pid_%d_verbose.out` that received verbose messages from the wrapper. Using the `-v` option would add some messages, but not as many as using `VOV_VW_VERBOSE` because many of the messages would get piped to `/dev/null`. In addition, `VOV_VW_LOGNAME` could be used to modify the name of the file for the verbose output.

In 2016.09u9 and later, setting `VOV_VW_VERBOSE` still creates the file and `VOV_VW_LOGNAME` can still be used to change the name, but now, using `-v` also does the same thing. Setting `VOV_VW_VERBOSE=3` in the environment is equivalent to using `-v -v -v` as options to `vw/vov`. In addition, if `VOV_VW_VERBOSE` is 3 or greater, or 3 or more `-v` arguments are passed, an additional file called `.vw2_pid_%d_verbose_env.out` gets written with the full contents of the environment and equivalences used for the job. The `VOV_VW_ENV_LOGNAME` environment variable can be used to change the name of this file.

Since a job can potentially run on different operating systems, the equivalences may vary depending on which machine the job was run on. The environment, however, should be the same on different machines.

Also in this Section

Command Line Representation

The command line stored by VOV is a **space-separated list of tokens**. In order to allow arguments to have spaces and other special characters, the following rules apply:

- If an argument contains a space, the whole argument must be double-quoted:

```
% vw2 echo "Ciao bello"
```

- If an argument contains a double quote, the quote must be escaped with a "\"

```
% vw2 echo Ciao\"bello
% vw2 echo 'This is a quote: \"'
% vw2 cc -DHEADER=\"hello.h\" -c hello.c
```

- If an argument contains a backslash "\" it must be escaped with another backslash:

```
% vw2 echo "this is a backslash \\"
```

Exit Status

The exit status is an important piece of information to decide whether a job has succeeded or not. Normally, an exit status of 0 (zero) is used to signify the successful execution of a tool.

The following table describes some of the exit statuses used by VOV.

Status	Possible Meaning
0	Successful termination.
1	Generic failure. This status is commonly used to indicate failures.
2	A program printed the usage message and exited.
4	Command not found.
5	A program exited because it received a signal.
6	A wrapper (e.g. <code>vw</code> or <code>vrt</code>) cannot redirect stdout or stderr.
7	An interactive job (see option <code>-I</code> of <code>nc run</code>) did not complete (was descheduled or forgotten).
8	The vovtasker encountered an error while managing the job (please report this condition to https://www.pbsworks.com/ContactSupport.aspx)

Status	Possible Meaning
33	<ul style="list-style-type: none"> The vovserver cannot bind the main socket. (OLD. New value is 76) A VOV client cannot connect to the server. (OLD. New value is 75)
72	A job sent to Accelerator with an indirect tasker failed.
73	A job was lost in Accelerator.
74	A capsule has failed. You will probably have to run the job from the terminal to get the complete description of the error.
75	A VOV client cannot connect to the server.
76	The vovserver cannot bind the main socket.
77	VOV Channel error. Occurs when the tool exits without calling <code>VovEnd()</code> . See VOV Instrumentation Library (VIL) for more details.
78	vovtasker could not redirect stdout.
79	vovtasker could not find executable.
91-93	Problems in collecting status of job.
100-120	Problems with the VOV Instrumentation Library (VIL) .
126	No longer used. See 136.
136	<p>Typically a deep error in the execution of a job. This was previously exit 126, but has been changed because <code>/bin/sh</code> also uses this code.</p> <ul style="list-style-type: none"> The vovtasker is unable to perform an <code>execve()</code> of the job.
137	Typically a deep error in the execution of a piped command as a child of vovserver. Rarely seen (was previously 127).
138	Typically a deep error in the execution of <code>utcc_system()</code> , an internal utility (was previously 127).
201-215	These codes are used to force automatic rescheduling of jobs. A job that exits with one of these codes will be rescheduled immediately with Priority given by code - 200. Thus a job that exits with status 201 will be rescheduled with priority low (1) while a job that exits with status 215 will be rescheduled with top priority. These codes are typically used by jobs that fail because of a spurious license problem. They allow the tool developer to signal VOV that the

Status	Possible Meaning
	failure is probably due to a missing license and that it will probably not occur again if the job is resubmitted.
216	Used to reschedule a multiphase job. See .
240	Job could not be dispatched to selected best tasker.
249	Channel has died (a VIL problem) (same as -7)
251	Tool had an output conflict (same as -5)
252	Tool had an input conflict (same as -4)
253	Tool had a cycle conflict (same as -3)

Legal Exit Status

A property of each job is the list of exit status values that are accepted as meaning success. By default, only the exit status 0 (zero) is accepted. To change the list of accepted exit status values:

From VIL

Use the procedure `VovCorrectExitStatus(char* list)` as in:

```
/* This is a fragment of C or C++ code. */
VovCorrectExitStatus( "0 2 12" );
```

With VIL-Tools

Use `VovCorrectExitStatus`, as in

```
#!/bin/csh -f
# This is an instrumented shell script.
...
VovCorrectExitStatus "0 2 12"
...
```

From Flow.tcl

Use the procedure `L`, as in:

```
# This is a fragment of Flow.tcl
E BASE
R sun5
# Like E and R, the procedure L has effect on all
# subsequent jobs.
L "0 2 12"
J vw cp aa bb
```

Exit Status Fields

The fields associated with the exit status are:

Field	Description
EXIT_STATUS	The actual exit status
OK_STATUS	(old) The list of allowed exit status values (same as LEGALEXIT)
LEGALEXIT	(new) The list of allowed exit status values (same as OK_STATUS)

Stdout and Stderr

The output channels known as stdout and stderr are used by many tools to inform the user about the progress of execution and abnormal conditions. This information must be preserved, because it is often crucial to diagnose the cause of a tool failure. The VOV wrappers do this automatically by capturing both stdout and stderr into files.

On UNIX, the name of the stdout file has the following structure: `.stdout_cmd_id_unique_id` where `cmd_id` consists of the first few non-blank characters of the command line (excluding the wrapper) and `unique_id` is a nine digit number computed by hashing the entire command line.

The name for stderr is similarly computed as `.stderr_cmd_id_unique_id`

Example:

```
% echo "Ciao bello" > dddd ; # No wrapper, not part of flow.
% vw2 cat dddd
Ciao bello
% vsx !
00234120 VALID ${SEV_PROJECT_DIR}/txt/dddd
>>>> Node 00234141 VALID vw2 cat dddd
00234127 VALID ${SEV_PROJECT_DIR}/txt/.stdout_catdd_013671688
```

On Windows NT, the names have the following structure: `./vovstd/out_cmd_id_unique_id.txt` `./vovstd/err_cmd_id_unique_id.txt`

If the files are empty upon termination of a successful job, they are forgotten from the trace. You can change this behavior by proper [encapsulation](#).

Change the Name and Location with VOV_STDOUT_SPEC

By setting the variable `VOV_STDOUT_SPEC`, you can redirect stdout and stderr to a different file name. The value is a computed by substituting the the substrings `@OUT@` and `@UNIQUE@` and `@ID@`.

Examples:

```
setenv VOV_STDOUT_SPEC .std@OUT@.@UNIQUE@
setenv VOV_STDOUT_SPEC .std@OUT@.@ID@
```

Cleaning up Stdout and Stderr Files

Over time, it is possible for many stdout and stderr files to accumulate in a directory. An efficient way to cleanup a directory is to use `vovcleandir`, as in this example:

```
% vovcleandir .  
vovcleandir: message: Wait while I ask the VOV server about 2 files ...  
vovcleandir: message: The directory is clean.  
% vovcleandir -all  
...output omitted...
```

Handling Pipes and Redirection with VOV Wrappers

To use pipes or redirection on your command lines, apply these two methods:

- Use either the wrapper `vov` or `vw2`
- Hide the pipes and redirections in a script, instrument the script with VILtools, and then execute the script

For example, suppose that you need to execute the following command in your design flow:

```
% sed 's/high/low/g' FILEIN | awk '{print "LINE", $0}' > OUT
```

which replaces all occurrences of "high" with "low", prints each line of the resulting file with the prefix "LINE", and saves the result in the file OUT. This command can be added to the graph with the following modification:

```
% vw2 sed 's/high/low/g' FILEIN \ | awk '{print "LINE",$0}' \> OUT
```

Note the use of calling the program `vw2` to run the shell command. The escaping of the symbols `|` and `>` is necessary so that the `vw2` program can evaluate them and not the shell. Only stdout is piped and redirected in the present implementation. All tools in the pipeline must return 0 (zero) before the job can succeed. If any of the tools returns something else, `vw2` reports the non-zero status it returns.

The second method listed above is based on a simple shell script:

```
#!/bin/csh -f  
# This is script highToLow.  
VovInput  $1 || exit 1  
VovOutput $2 || exit 1  
  
cat $1 | tr '[A-Z]' '[a-z]' > $2  
exit $status
```

To execute the script without adding any nodes to the graph, use:

```
% highToLow fileA fileB
```

To have this activity create nodes in the graph, use:

```
% vov highToLow fileA fileB
```

Conflict Detection

When executing the tools from the command line, it is possible to violate some of the dependency constraints. Without FlowTracer, conflicts go undetected, causing loss of data, tool failures, wasted time.

FlowTracer detects three types of conflicts:

- Input Conflict
- Output Conflict
- Cycle Conflict

Input Conflict

An **input conflict** occurs whenever a job uses an input that is not VALID. The designer who has invoked the job is notified of the problem and given a chance to avoid it.

Here is an example where the user is trying to copy file 'bb' which is not up to date:

```
VOV: ATTENTION!   Input conflict detected! VOV: ATTENTION!
VOV: ATTENTION!   Input conflict detected! VOV: ATTENTION!

----- User Decision Required -----
INPUT CONFLICT for tool
vw cp bb cc
(directory ${HOME}/tmp)

The tool needs
FILE:${HOME}/tmp/bb
which is currently INVALID

1 --      CONTINUE
2 --      STOP ASKING
3 -- (*)  ABORT
Please choose (1--3) >>>>
```

Output Conflict

An **output conflict** occurs when a job declares as output a file that is already the output of another job. Normally, a file can be the output of only one job.

The designer is asked to confirm the decision to forget the old job and replace it with the current one. In the following example, the designer has previously run `vw cp bb cc` and is currently running `vw cp aa cc` which changes the way of creating `cc`. FlowTracer prompts the user with this dialog:

```
VOV: ATTENTION!   Output conflict detected! VOV: ATTENTION!
VOV: ATTENTION!   Output conflict detected! VOV: ATTENTION!

----- User Decision Required -----
OUTPUT CONFLICT caused by data item
FILE:${HOME}/tmp/cc
Command lines are different.
Common part is 6 characters long.
'bb cc' != 'aa cc'.
^      ^
^      ^

1 --      CONTINUE
2 -- (*)  ABORT
```

```
3 --      MORE INFO
Please choose (1--3) >>>>
```

Cycle Conflict

The graph cannot contain cycles, and the server must check each dependency declaration to ensure it does not introduce one. Cycle conflicts are rare, but when they occur, there is no choice other than abortion of the run.

```
VOV: ATTENTION!  Cycle detected! VOV: ATTENTION!
VOV: ATTENTION!  Cycle detected! VOV: ATTENTION!
vw Jun 07 15:01:00 Failed VOV call libconnect.cc,152

vw ERROR Jun 07 15:01:00 Cycle conflict for ${HOME}/tmp/aa
vw Jun 07 15:01:00 This job 008269147 must be forgotten
```

Control Conflict Detection

Conflict detection cannot be disabled. What you can do is to avoid the waiting for user input caused by the pop-up dialogs. This can be controlled by the environment variable `VOV_CONFLICT_CONTROL`, which can take either the value `ABORT` or `CONTINUE`.

During automatic retracing, `VOV_CONFLICT_CONTROL` is always set to `ABORT`.

Firing Jobs

vovfire

The utility `vovfire` is used to run one or more jobs specified by their VovId on the local machine.

`vovfire` takes care of switching environment and of changing working directory.

When using `vovfire`, you take on the responsibility of deciding that the local host has the appropriate resources to execute the job.

For example, the following command runs the jobs with VovId 00000308 and 00000336 on the local host, which happens to be a Windows NT machine (notice the backslashes in the directory names):

```
% vovfire 308 336
vovfire: message: #####
vovfire: message: Env is: DEFAULT
vovfire: message: Cwd is: ${TOP}
vovfire: message: Firing: vw cp aa.txt bb.txt
FIRE JOB 308:  #### FIRE JOB
FIRE JOB 308:      ENV=DEFAULT
FIRE JOB 308:      DIR=C:\VOV\TEST
FIRE JOB 308:      CMD=vw cp aa.txt bb.txt
          1 file(s) copied.
FIRE JOB 308:  #### DONE
vovfire: message: Done: vw cp aa.txt bb.txt

vovfire: message: #####
vovfire: message: Env is: DEFAULT
vovfire: message: Cwd is: ${TOP}
vovfire: message: Firing: vw cp bb.txt cc.txt
```

```
FIRE JOB 336:  #### FIRE JOB
FIRE JOB 336:      ENV=DEFAULT
FIRE JOB 336:      DIR=C:\VOV\TEST
FIRE JOB 336:      CMD=vw cp bb.txt cc.txt
                1 file(s) copied.
FIRE JOB 336:  #### DONE
vovfire: message: Done: vw cp bb.txt cc.txt
```

Command Interception

Another way to build the graph is to intercept all tool calls. The essence of this technique is to activate runtime tracing automatically for all intercepted tools without requiring the user to explicitly prepend the VOV wrapper. This technique is available only on UNIX.

The alias `vovintercept` prepends the directory `$VOVDIR/scripts/intercept` to the `PATH` variable. This directory contains an executable file for each tool that you want to intercept. Each file is a symbolic link to the `$VOVDIR/scripts/intercept/generic`. For example, if you want to intercept all calls to the compiler `gcc`, you need to create a link to `generic` called `gcc`:

```
% cd $VOVDIR/scripts/intercept
% ln -s generic gcc
```

Repeat the process for all tools that you want to intercept.

Now you can turn on "intercept mode" with `vovintercept`, and execute your methodology. When you are finished with interception, call the alias `vovinterceptstop`.

```
% vovintercept
% make target
% vovinterceptstop
```

This approach has some limitations. For example, you may find out that your existing flow generates a lot of input and output conflicts. Some scripts and Makefiles use full paths to invoke tools, thus preventing the interception.

Flowbuilding

The trace is built using a combination of techniques.

Execute the Tools Directly

The trace is built as a side effect of each tool execution. FlowTracer follows the activities of the designers and automatically documents what is being done.

This technique is typically used during the exploratory phase of flow development.

```
% vw cp aa bb
```

Code an Outline of the Flow in a Tcl Script, Assuming Integrated Tools

Building the trace by executing the jobs can be slow and tedious, especially when jobs take a long time to complete. FlowTracer provides a facility called [vovbuild](#) to describe the trace using a [Flow Description Language \(FDL\)](#) which is an extension of Tcl. A trace description in Tcl can be processed by vovbuild without actually executing any of the tools.

```
# Tiny example of FDL.  
T vw cp aa bb  
I aa  
O bb
```

This is the most effective technique to build the trace.

A set of ready-to-use methodologies is available in the Flow Library.



Note: The **authoritative source of dependencies** remains the integrated tool at Altair. The dependencies defined in FDL are to be considered only suggestions.

Coding the Flow Completely in Tcl, Without Tool Integration

This technique is an exception in the sense that it does away with any form of tool integration, so it does not really use runtime tracing. However, it is convenient to build the initial flows quickly.

```
# Tiny example of FDL for non-integrated tools  
SERIAL {  
    TASK make clean  
    TASK make install  
}
```

The documentation for this technique can be found in [Task Oriented Flows](#).

Since there is no tool integration with this technique, there is no runtime tracing and therefore there is no automatic updating of the dependencies. This is a technique where "what you write is all you get".

Convert an Existing Makefile

A set of utilities allow the migration from existing Makefiles to the equivalent Tcl script. This conversion is typically done with [vovmake](#). The success of this technique depends on how good the Makefiles are, to begin with.

Building the Flow with vovbuild

In most deployment scenarios, the flow development team prepares a script that performs a lot of the initial steps such as populating a workspace, starting a VOV project, and building a VOV Flow. This is the script that is given to the users, so that most users do not really have to worry about how the flow is built.

For the purpose of the next few sections, we want to show the step used to build the flow, which is basically one or more execution of the utility `vovbuild`.

If you do not already have a flow, you can build one easily taking one of the flows from the Flow Library which is located in `$VOVDIR/flowlib/training`. For example, use the simplest flow, which is `$VOVDIR/flowlib/training/Copy.tcl`.

```
% vovproject create test_project_$USER
% vovproject enable test_project_$USER
% vovconsole &% mkdir some_test_directory
% cd some_test_directory
% vovbuild -f $VOVDIR/flowlib/training/Copy.tcl
% vovconsole -set System:nodes &
```

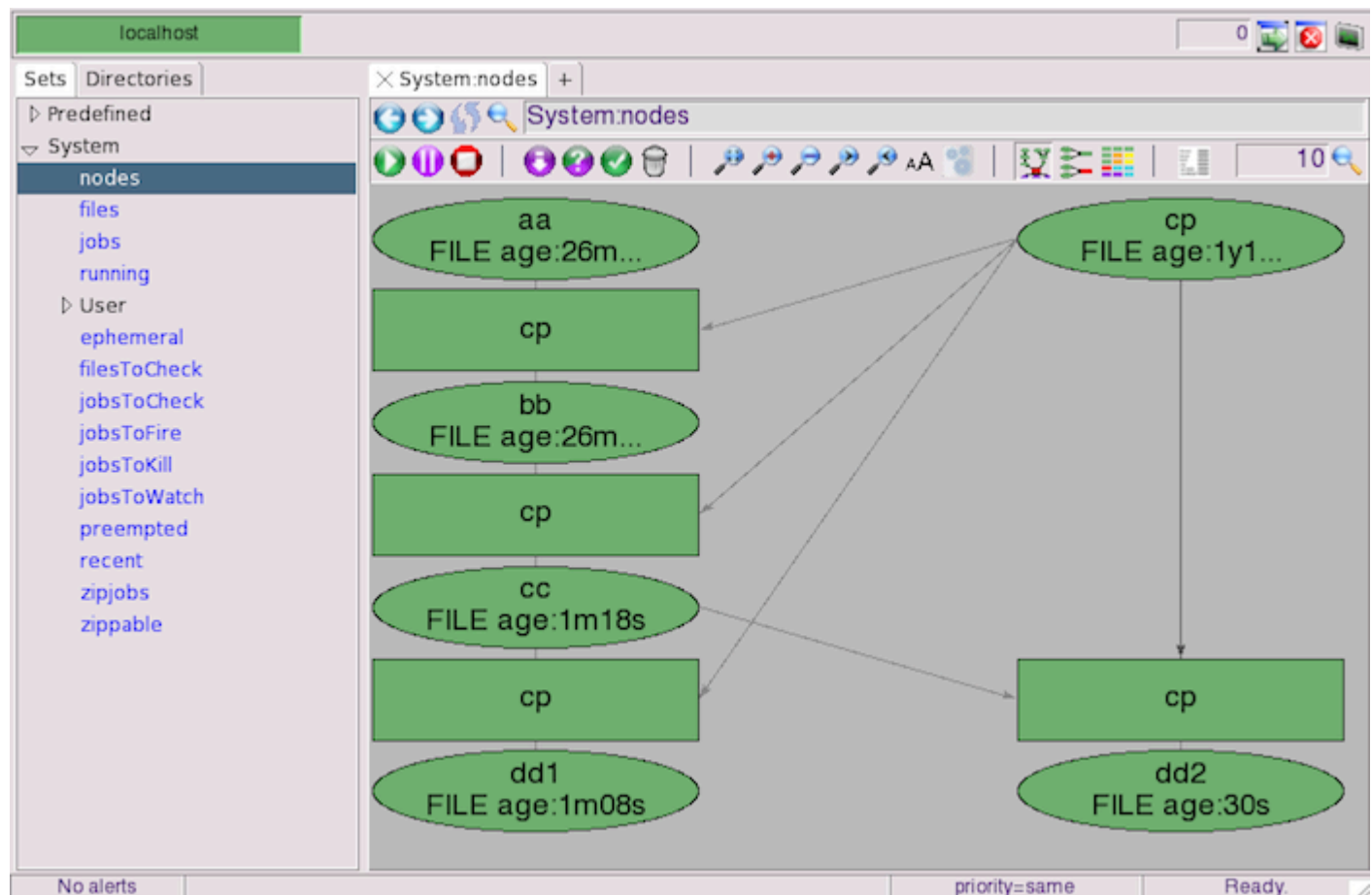


Figure 1: A Small Flow You can Use as a Test

Flow Description Language (FDL)

While it is always possible to build flows by executing one tool at a time, the normal way to use FlowTracer is to:

- Write a description of the flow using the Flow Description Language
- Pass the description to `vovbuild` to import the flow into FlowTracer
- Execute the flow using parallel retracing

This chapter introduces FDL, the Flow Description Language, which is an extension to Tcl. Common uses of the language are demonstrated.

For information about using Makefile to FDL utility, refer to [Makefile to FDL Utility](#).

With FDL, you can concentrate on the flow description. Compared to traditional methods using scripts or makefiles, you do not have to worry about setting up the environment, representing dependencies, deciding the job scheduling, checking the proper completion of each job, capturing stdout and stderr, because FlowTracer takes care of all of that. Instead, you can concentrate on the "recipe" for the flow.

FDL consists of single letter procedures such as J, T, E, R, I, O, N and a few others.

Examples of Flows

This is an example of a very simple flow description:

```
J vw cp aa bb
```

The `J` means that we want a job with command line `vw cp aa bb`. The environment for the job will be `BASE` by default, and the working directory is the current directory.

FDL gives you control on the environment in which the jobs are executed by means of the procedure `E`, which must be invoked before the `J`.

```
E "BASE+D(MYVAR=aaa) "  
J vw cp aa bb
```

Similarly, you can control the resources required by a job by means of the procedure `R`, which must also be called before the `J`:

```
R "linux"  
J vw cp aa bb
```

In alternative to `J`, you can use the procedure `T` to add a job to the flow. In this case, however, it is also important to add explicit dependencies to the flow, using `I` and `O`. Both `I` and `O` must be called **after** `T`. For a detailed description of the difference between `J` and `T`, check the appropriate section.

```
T vw cp aa bb  
I aa  
O bb
```

Since FDL is an extension to Tcl, you can use all Tcl constructs such as `for`, `while`, `foreach` to build arbitrarily complex flows. Here is an example of a `for` loop:

```
E "SYNTHESIS"
for { set i 0 } { $i < $numberOfBlocks } { incr i } {
    J vw runSynthesis $block($i)
}
```

Simple conditionals can be handled with `if` and `switch`:

```
foreach block $listOfBlocks {
    set type $blockType($block)

    switch $type {
        "memory" {
            J vw genMemory $block
        }
        "gates" {
            J vw runSynthesis $block
        }
        default {
            VovFatalError "Unknown block type '$type' for '$block'"
        }
    }
}
```

Also fundamental in Tcl is the ability to create procedures that can be used to make your flow definition more modular:

```
proc compile { cfile } {
    global CASE
    set cc    $CASE(cc)
    set flags $CASE(flags)
    J "vw $cc -c $flags $cfile"
}

set CASE(cc) gcc
set CASE(flags) "-I../include -g"

compile aa.c
compile bb.c
compile cc.c
```

Also in this Section

Flow Language Procedures

FDL Procedures

Name	Description
A	Add annotations to the most recently declared node.

Name	Description
AD	Declare an artificial dependency between two jobs.
D	Set the database for subsequent files.
E	Set the environment for subsequent jobs.
FLAGS	Specify additional flags for the most recent job.
I	Declare an input dependency.
J	Declare a job and use the capsule to compute IO's (see also T and P).
L	Set the legal exit values for subsequent jobs.
N	Set the jobname.
O	Declare an output dependency. All jobs require at least one output to be valid.
P	Add a property to the most recently mentioned object.
PARALLEL	Declare the parallelism between two or more tasks and subflows (See also TASK and SERIAL).
PJ	Declare a job to be executed periodically (see also T and J).
PRIORITY	Set the priority to be used for the subsequent jobs.
R	Set resources for subsequent jobs.
RUNMODE	Set 'runmode' for subsequent jobs.
S	Precise definition of a set of jobs.
S+	Append jobs to a set.
SERIAL	Serially order a number of tasks and subflows (See also TASK and PARALLEL).
START	Start running jobs -- rarely used.
T	Declare a job (see also PJ and J).
TASK	Declare a job with no specific dependencies (see also SERIAL and PARALLEL).
X	Set the expected duration for subsequent jobs.

Name	Description
Z	Set the zippable flag for a list of files.

Flow Library Procedures

Name	Description
Description {doc}	Give a description of the flow.
Keywords {list of words}	Assign keywords to the flow, to aid searches.
Parameter	Describe a parameter of the flow.
GetAllParameters	Get the flow parameters using one of the following methods: <ul style="list-style-type: none">• The command line, using options to vovbuild with the following format - parameterName=parameterValue• An HTML form
RequiredFile file {example ""}	Describe a file that is required by the flow. If the specified file does not exist and example is not null, then the example is copied. If the file does not exist, or if it cannot be copied from example, an error is generated.
RequiredDirectory { directory }	Describe a directory that is required by the flow. If the specified directory does not exist, an error is generated.

Other Useful Procedures

Name	Description
shift	Get the next argument from argv
safeshift	Same as above, but with better error messaging if argv is empty
indir	Execute a code fragment in a directory

Name	Description
RECONCILE_WITH_FILE_SYSTEM	Used to recover the status of a flow based on the current timestamps of files

FDL Procedures Reference

FDL_A

Add annotations to the most recently declared node.

Arguments

`note checkFlag`

Description

Adds annotations to the most recently declared node, be it an input, an output, or a job. If `checkFlag` is true, the procedure checks the list of existing annotations in order to avoid duplicates.

Example

```
J vw sim $block.v $block.sti
A "Run the simulation on block $block"
I $block.sti
A "The stimulus file"

# The binary 'clevercopy' does not need a wrapper
J clevercopy $file $goldDir
A "Publish $file"
```

FDL AD

Declare an artificial dependency between two jobs.

Arguments

`id1 id2 barrier`

Description

Creates an artificial dependency between a job (`id1`) and another job or set (`id2`). Useful if no natural file dependency exists between them. Makes `id2` dependent on `id1`. `barrier` defaults to 0 (off).

Example

```
AD $id1 $id2 ; # id2 is dependent on id1
AD $id1 $id2 1 ; # id2 is dependent on id1 with a barrier on $id2.
```

FDL CAPSULE

Arguments

[options] script

Description

Define a capsule-on-the-fly.

Option

-vars *list_of_vars*

Description

Allows the passing of specified variables for use inside the CAPSULE.

Examples

```
J vw mycopy aa bb
CAPSULE {
  I [shift]
  O [shift]
}
```

```
J vw mycopy aa bb
CAPSULE { I aa; O bb }
```

```
J vw vsyn -cell $cell -corner $corner
CAPSULE -vars [list cell corner] {
  I $cell.$corner.in
  O $cell.$corner.out
}
```

FDL D

Set the database for subsequent files.

Arguments

database

Description

Sets the [Databases](#) for all subsequent files. The default database is "FILE".

Example

```
D FILE
D FILEX
```

```
D GLOB
D JOBSTATUS
D LINK
D PHANTOM
D VOVSETS
D ZIP
```

FDL E

Arguments

environment OPTIONAL_SCRIPT

Description

The default environment is "BASE". The procedure **E** sets the environment for all subsequent jobs. If the optional script is specified, the value of the current environment is reset after evaluation of the script.

The environment for `vovbuild` itself is not affected (use `ves` if you want to change the environment of `vovbuild` itself).

The command `vel` will list the known environments in your installation, Normally the following locations are searched for environment information :

- \$SWD/environments
- \$VOVDIR/local/environments
- \$VOVDIR/etc/environments

Example

```
E BASE
E BASE+SUNCC
E EPIC (5.3)
E BASE+GNU {
    J vw cp aa bb
}
```

FDL FLAGS

Arguments

job-flag ...

Description

The procedure **FLAGS** sets flags for the most recent job. It is to be used after a **T** or a **J**. The flags that can be set are: `autoflow`, `autoforget`, `preemptable`, `doprofile`, `nojournal`, `schedfirst`, and `skip`.



Note: `autoflow` and `skip` are the same flag.

Example

```
J vw cp aa bb
FLAGS skip

J vw date
FLAGS preemptable doprofile autoforget
```

FDL I

Declare an input dependency.

Arguments

[options] FILE FILE ...

Description

This is an input declaration. Its argument is a list of file names. Each file is declared as an input of the job created with the most recent `J` or `T` command. It is an error to call `I` without having called `J` or `T` first. Defining cyclic dependencies generates an error.

Option	Description
-db <i>name</i>	Sets the database name. The default is "FILE".
-sticky	Specify that the file is an input, even if it's not actually a true input at runtime.
-consumed	Specify that the input disappears if the tool completes successfully (e.g. the tool gzip consumes its input).
-normal	This option undoes the other options.
-quote	Use the input name as given, instead of mapping it to the corresponding logical canonical name.
-trigger,run	If the input changes, automatically run the downcone
-trigger,stop	If the input changes, automatically stop all jobs in the downcone
-noop	Ignored
-normal	Ignored
-links	Ignored in this context

Example

```
I $inFile
I -consumed xx.tar
I -db PHANTOM net.h
```

FDL indir

Execute a code fragment in a directory.

Arguments

[options] directory script

Description

Executes Tcl script in the given directory.

Option	Description
-create	Create directory if it does not exist.
-onerror script	Execute "script" if the directory does not exist.

Examples

```
indir -create Subdir {  
    E BASE {  
        J vw cp aa bb  
        J vw cp bb cc  
    }  
}
```

```
indir -onerror {  
    VovMessage "Skip directory Subdir because it does not exist"  
} Subdir {  
    E BASE {  
        J vw cp aa bb  
        J vw cp bb cc  
    }  
}
```

FDL INSTRUMENTED

Arguments

command

Description

Add a job that is implemented by a properly instrumented script, that is a script that outputs FDL if called with the VOV instrumentation library environment variable defined.

Example

```
INSTRUMENTED ./mycopy aa bb
```

FDL J

Declare a job and use the capsule to compute IO's (see also T and P).

Arguments

args

Description

Mnemonic for 'Job'

Declares a job where the associated capsule (if it exists) is used at build time. This procedure is identical to T, except that T does NOT search for corresponding capsule. The arguments form the command line for a new job to be added to the trace. Environment and working directory for the job are taken from the current settings. If an identical job already exists in the trace, nothing happens. The procedure returns the VovId of the job. The status of a newly created job is always INVALID.

If the procedure is passed multiple arguments, each argument is quoted if necessary. If the procedure is passed a single list, all elements in the list are "join'ed" together with spaces.

Example

```
J vw cp aa bb  
J vw vdc_shell -f $script
```

FDL J_FINAL

Similar to T_FINAL, J_FINAL indicates that the inputs and outputs declared through I and O commands associated with the last J declaration are the only inputs and outputs for that J. Any inputs and outputs that were associated with this J from a previous vovbuild but not declared again in this vovbuild will be removed.

FDL L

Set the legal exit values for subsequent jobs.

Arguments

legal_exit_spec

Description

Set the list of legal exit status values for all subsequent jobs.

Example

```
L "0";           # That is the default.  
L "0 1 2";       # Accept as legal the 3 values 0 1 and 2.  
L "12-20";       # Accept as legal any value in the range from 12 to 20.
```

FDL N

Set the jobname.

Arguments

jobname

Description

Set the jobName for **all subsequent jobs**. The jobName may contain any alphanumeric character, in addition to the space and characters from '-_./='. It is an error to use illegal characters. The name is truncated to 256 characters.

Example

```
N "build"  
N "simulation-2"  
N "build_tree"
```

For all products, strict job name checking has been enabled and invalid job name characters will cause an error. For Accelerator and Accelerator Plus, this can be overridden by putting the following in \$VOVDIR/local/vncrun.config.tcl: `set ::jobname_lexicon legacy` or `set ::jobname_lexicon replace`.

Legacy will use the more lax job naming rules from earlier releases.

Replace will identify invalid characters in the job name, replace them with "_", and issue a warning to the console.

FDL N2

Arguments

jobname

Description

Set the jobName for the most recently described job. This procedure performs no checks on the name, but this may change in the future.

Example

```
J vw cp aa bb  
N2 "my_copy"
```

FDL O

Declare an output dependency. All jobs require at least one output to be valid.

Arguments

[options] FILE FILE ...

Description

This is an output declaration. Its argument is a list of file names. Each file is declared as an output of the job created with the most recent `J` or `T` command. It is an error to call `O` without having called `J` or `T` first. Defining cyclic dependencies generates an error. All jobs must have at least one output to be consider valid.

Option	Description
-db <i>name</i>	Sets the database name of the output.
-sticky	Specify that the file is an output, even if it's not actually a true output at runtime. Also useful if it is truly an output, but is not instrumented or declared as an output at runtime.
-normal	This option undoes the other options.
-barrier	Output may have a barrier. The barrier, if it exists, is controlled by the job that generates this output.
-force	Force output declaration even with conflict.
-noforce	Fail on output conflict.
-shared	The output is shared.
-md5	Output has an automatic barrier consisting of its MD5 sum.
-rcpc	Same as -md5
-ignore_timestamp	Specifies that the timestamp of the file should not be used to determine whether the job has succeeded or failed.
-quote	Use the output name as given, instead of mapping it to the corresponding logical canonical name.

FDL P

Add a property to the most recently mentioned object.

Arguments

`propertyName` `propertyValue`

Description

Attach a property with the specified name and value to the most recently mentioned node. The type of the property is determined automatically from the value, that is, the type is integer if the value is numeric. If `P` is called after `T` or `J`, the property is attached to the last job. If `P` is called right after an `I` or `O` declaration, the property is attached to the input or output file respectively.

Example

```
J vw cp aa bb
P TEST "This property will be attached to the job"
I aa
P TEST "This property is attached to file aa"
O bb
P TEST "This property is attached to file bb"
```

FDL PARALLEL

Declare the parallelism between two or more tasks and subflows (See also TASK and SERIAL).

Arguments

script

Description

All of the tasks, jobs, and sets defined in the script are set up to be executed in parallel. Artificial dependencies are created between each task, job, or all the jobs in the set to build the graph in parallel. Can be nested in any order with sets, jobs, tasks, other SERIAL scripts, and other PARALLEL scripts. For examples see the TASK procedure.

FDL PJ

Declare a job to be executed periodically (see also T and J).

Arguments

[OPTIONS] args

Description

Creates a periodic job using J (uses capsule if present). Periodic jobs run with a default period of $\$period$, but will adjust future runs according to how long the job takes to complete. The range of the period is defined by $\$minperiod$ and $\$maxperiod$.

Option	Description
-period <i>TIMESPEC</i>	Specify target period of job. This is the period with which the job is scheduled. If there are enough resources, it will also be the period with which the job is executed, else the job may have to be queued.
-min <i>TIMESPEC</i>	Allow period to be reduced down to this minimum period. This happens if the job is quick enough.
-max <i>TIMESPEC</i>	Allow period to be increased up to this maximum period. This happens if the job takes a long time to execute. The max period is also used to kill the job if it takes longer than max period.

Option	Description
-P <i>TIMESPEC</i>	Short cut specification of period, min period, max period, and autokill, where, with \$P set to the TIMESPEC, min period is set to \$P, max period is set to 10*\$P and autokill is set to 3*\$P.
-autokill <i>TIMESPEC</i>	Kill job if it runs for longer than the specified amount. If this is greater than the maximum period, then the job will be killed if it runs for more than the maximum period.
-resources <i>RESOURCE_SPEC</i>	Override resource specification for the job. By default, this is the empty list.
-env <i>ENV_SPEC</i>	Override environment specification for the job.
-cal <i>CALENDAR_SPEC</i>	Specify when the job is allowed to be executed. A CALENDAR_SPEC consists of two comma-separated lists joined by a colon. The part to the left of the colon expresses the days, and the part to the right specifies the hours, 0..23. Days are numbered 0..7 with 0 for Sunday. A CALENDAR_SPEC format example is: Sun,Mon,Tue:1,2,3,4 which states that the job can be executed on Sunday, Monday, and Tuesday, at the hours of 1AM, 2AM, 2AM 4AM. This option works in conjunction with the period settings describe above. Other CALENDAR_SPEC examples are:0,1,2:* - Sunday, Monday, and Tuesday, all hours.*:15,21 - All days, at 3pm and 9pm.
-systemjob	The job is considered "systemjob" (not keeping server up). If not declared as a systemjob, this job might prevent auto shutdown of an idle project, if the period of the PJ is shorter than the auto shutdown interval. Please note that the setting of make(systemjob) is not used to determine if the PJ is a system job or not. You must use this flag to declare this periodic job to be a system job.

Example

```
PJ -period 1h updateSomeFile
PJ -period 1h -min 40m -max 2h -autokill 2m \
  -resources linux -env BASE updateSomeFile

E BASE
R linux
PJ -P 1m date

set make(jobname) "$tag"
set make(nojournal) 1
PJ -P 30s -systemjob ftlm_parse_flexlm ...
```

FDL PRIORITY

Set the priority to be used for the subsequent jobs.

Arguments

spri xpri

Description

Set the priority to be used for subsequent jobs. The priority can be specified numerically (from 1 to 15) or symbolically (low normal high top). The `xpri` argument is optional. If missing, it is set to `spri`. The value of priority for a job can easily be overwritten by a retrace request.

Example

```
PRIORITY 4 4
PRIORITY normal
J vw cp aa bb
PRIORITY top normal
J vw cp aa bbx
```

FDL R

Set resources for subsequent jobs.

Arguments

resourceList OPTIONAL_SCRIPT

Description

Sets the resources required by the following jobs, i.e. the `R` statement should precede `T` or `J`. Takes one or two parameters. The first parameter is a list of resource names. The second parameter, if present, is a Tcl script which is executed in the context of the caller of the `R` procedure. Side effect: sets the value of `make(resources)`, except when the second (script) parameter is given.

Example

```
R "unix"
R "RAM#512"
R "sun5 dc_shell_license"
R "res1 res2 res3"
R "res1 res2" {
    J vw cp aa bb
}
R {res1 res2} {puts "Resources = '$make(resources)'"}
```

FDL R2

Arguments

resourceList

Description

Similar to `R`, but this only applies to the most recently created job. That is, `R` goes before the `T` or the `J`, while `R2` goes after the `T` or `J`. This procedure is used by the Vil-Tool VovResources when called with `VOV_FDL_ONLY`.

Takes one parameter, a resource expression. No side effects.

Example

```
J vw cp aa bb
R2 "res1 res2"
```

FDL S

Precise definition of a set of jobs.


Arguments

`setname script args`

Description

Collect all jobs created in the body of the procedure in the specified set. The `s` procedure can be nested. If the `setname` at one level begins with a '+' sign or if it has no colons (':'), the full name of the set is the concatenation of the name of the set at the previous level, separated with a colon ':'. If the name contains colons, it is considered to be a complete set name.

As of 2014.*, it is allowed for sets to contain other sets. Because of this, nested `s` procedures now attach the subset to the superset.

 **Note:** Job I/O nodes are not placed in the set.

Synopsis

```
S set_namebody
```

Arguments

```
"-label" -
"-gui_label" {
  set label [shift args]
  if { $label == "" } {
    vtk_prop_del $make(setId) GUI_LABEL
  } else {
    vtk_prop_set -text $make(setId) GUI_LABEL $label
  }
}
"-files" {
  S_update_files $setname
}
```

Examples

Single Set

```
S foo1 { J vw sleep 0 } ## ID001
```

One nested Set (Set nested within a Set):

```
S foo2 {                                ## ID002
  S bar2 { J vw sleep 0 }              ## ID003
}
```

Using the Colon Syntax:

Create a visual representation of a hierarchical set structure, with only one job in one Set:

```
S foo3:bar3 { J vw sleep 0 } ## ID003
```

Create Sets Each Having Jobs using the Colon Syntax

In this example, two Sets are created, and both will contain jobs.

```
S foo4 { J vw sleep 0 } ## ID004
S foo4:bar4 { J vw sleep 0 } ## ID004
```

FDL S+

Append jobs to a set.

Arguments

setname script

Description

Append all jobs created in the body of the procedure to the specified set.



Note:

- If the specified set does not exist, the set is created.
- Job I/O nodes are not placed in the set.
- Must use fully qualified set name when appending to a nested set.

Arguments

```
switch -- $arg {
  "-last" {
    vtk_set_operation $setid $make(lastNodeId) ATTACH
  }
  "-set" {
    set set2name [shift args]
    set set2id [vtk_set_find $set2name]
    vtk_set_operation $setid $set2id UNION
  }
  "-idlist" -
  "-id" {
    set idList [shift args]
```

```
        vtk_set_operation $setid $idList ATTACH  
    }
```

Synopsis

```
S+ set_namebody
```

Examples

Create the first Set

```
S "foo1" {  
    J vw cp aa bb  
}
```

Append a second Set to the first using S+

```
S+ "foo1" {  
    J vw cp bb cc  
}
```

foo1 now has two jobs attached to it. "J vw cp aa bb" and "J vw cp bb cc". Because **S** + was used, rather than just the **S** procedure, the jobs were added to the **foo1** Set, rather than overwriting the original.

Append Jobs to a Non-existing Set

If you try to append jobs to a Set that does not exist yet, the Set is created by default, with the added jobs attached to it.

For example

```
S+ "MyNewSet" {  
    J vw cp cc dd  
}
```

A new Set named "MyNewSet" is created.

FDL SERIAL

Serially order a number of tasks and subflows (See also TASK and PARALLEL).

Arguments

script

Description

All of the tasks, jobs, and sets defined in the script are set up to be executed serially in the order they appear. Artificial dependencies are created between each task, job, or all the jobs in the set to build the graph serially. Can be nested in any order with sets, jobs, tasks, other SERIAL scripts, and other PARALLEL scripts. For examples see the TASK procedure.

FDL shift

Get the next argument from argv

Arguments

<none>

Description

Get the next argument from argv.

Example

```
while { $argv != {} } {  
    set arg [shift]  
    switch -glob -- $arg {  
        "-h"      { help }  
        "--title" {  
            set title [shift]  
        }  
        default {  
            VovError "Unknown argument '$arg'"  
        }  
    }  
}
```

FDL START

Start running jobs -- rarely used.

Arguments

list-of-ids

Description

Starts a retrace in SAFE mode with NORMAL priority for each object-ID passed in. IDs may be of PLACE, TRANSITION, NODE or NODESET. Ignores invalid object types with a message. It is a fatal error to pass in an invalid vovID.

Synopsis

```
START id1 [id2 .. idN]
```

Example

```
set jid [J vw cp aa bb]  
START $jid
```

FDL STOP

Arguments

list-of-ids

Description

Calls `vtk_job_control $id STOP` for each object-ID passed in. Ids may be of PLACE, TRANSITION, NODE or NODESET. Ignores invalid object types with a message. It is a fatal error to pass in an invalid vovID.



Note: Does not stop jobs in a NODESET, but only produces a message.

Synopsis

```
START id1 [id2 .. idN]
```

Example

```
set rjobs vtk_set_get_elements [vtk_set_find System:transitionsRunningRetracing]
@ID@
STOP $rjobs
```

FDL T

Declare a job (see also PJ and J).

Arguments

args

Description

Mnemonic for 'Tool'

Declares a job where the associated capsule (if it exists) is NOT used at build time. This procedure is identical to `J`, except that `J` does search for corresponding capsule. The arguments form the command line for a new job to be added to the trace. Environment and working directory for the job are taken from the current settings. If an identical job already exists in the trace, nothing happens. The procedure returns the VovId of the job. The status of a newly created job is always INVALID.

If the procedure is passed multiple arguments, each argument is quoted if necessary. If the procedure is passed a single list, all elements in the list are "join'ed" together with spaces.

Example

```
set list { aa bb }
T vw cp aa bb;                # Many arguments
T "vw cp $list";              # One argument
T vw vdc_shell -f adder.dcsb
T vw cat xx.v yy.v zz.v > bigfile.v
T vw echo "ciao bello"
```

FDL T_FINAL

Arguments

<none>

Description

T_FINAL indicates that the inputs and outputs declared through **I** and **O** commands associated with the last **T** declaration are the only inputs and outputs for that **T**. Any inputs and outputs that were associated with this **T** from a previous vovbuild but not declared again in this vovbuild will be removed.

FDL TASK

Declare a job with no specific dependencies (see also SERIAL and PARALLEL).

Arguments

args

Description

TASK declares a task. This is a wrapper for the procedure **T**. Its main purpose is to be used with SERIAL and PARALLEL to build simple, intuitive flows.

Example

```
SERIAL {
  TASK make clean
  TASK configure
  TASK make
  TASK make install
}

SERIAL {
  S Run_Three_A_In_Parallel {
    PARALLEL {
      T vw scriptA1.csh
      T vw scriptA2.csh
      T vw scriptA3.csh
    }
  }
  S Run_Three_B_In_Parallel {
    PARALLEL {
      T vw scriptB1.csh
      T vw scriptB2.csh
      T vw scriptB3.csh
    }
  }
}

PARALLEL {
  SERIAL {
    TASK date 1920
    TASK date 1921
    TASK date 1922
  }
}
```

```
PARALLEL {  
  SERIAL {  
    TASK date 1923  
    TASK date 1924  
    TASK date 1925  
  }  
  SERIAL {  
    TASK date 1926  
    TASK date 1927  
    TASK date 1928  
  }  
}
```

FDL X

Set the expected duration for subsequent jobs.

Arguments

xdur

Description

Set the expected duration for subsequent jobs. It takes one argument, which is a time specification.

Example

```
X 120; # Expected duration is 120 seconds.  
X 2m ;  
X 3h10m;
```

FDL Z

Set the zippable flag for a list of files.

Arguments

FILE FILE ...

Description

Set the 'zippable' flag in all the mentioned files. The file must be a place in the trace belonging to the database 'FILE'. A file that has the zippable flag will be automatically compressed and decompressed as required by the flow. If all jobs that use the file have been completed successfully, the file will be compressed. If a job that needs the file is scheduled to run, the file will be decompressed.

J versus T

In FDL there are two procedures that can be used to add a job to a flow: T and J. Which one should you use?

The difference between J and T is that **J evaluates the capsule for the tool while T does not**. If the tool you are adding to the flow has no capsule, there is no difference between J and T. On the other hand, if a capsule is available, J will give you a more compact flow description. Even with a capsule, you may want to consider using T, with some explicit I's and O's in order to build the flow more quickly.

Procedure	Advantages	Disadvantages
J (with capsule evaluation)	More compact flow description. Same as T if there is no capsule.	Requires a capsule to be effective. Capsule evaluation may be slow.
T (no evaluation of capsule)	Faster evaluation.	Must declare explicit I's and O's dependencies to avoid false job executions.

For example, consider the tool `cp` which has a capsule (`$VOVDIR/tcl/vtcl/capsules/vov_cp.tcl`). The two following flow descriptions are equivalent:

```
# This uses the capsule vov_cp.tcl
J vw cp aa bb

# This instead does not use the capsule.
T vw cp aa bb
I aa
O bb
```

Build Flows with vovbuild

To begin, you need to get a **Flow Definition File**. You have the following options:

- Get the flow definition from the Flow Library.
- Create the flow definition yourself.

In the following example, you create the flow yourself. The default name for the flow definition file is `Flow.tcl`. If you use another name, use the option `-f` in `vovbuild` to select the desired flow description.

```
% vovbuild
% vovbuild -f Flow.tcl
% vovbuild -f MyBigFlow.tcl
```

With `vovbuild` you can build large flows with thousands of jobs in a few seconds and then you can execute them efficiently using FlowTracer retracing capabilities.

The Flow Definition File is a Tcl script that describes the structure of the flow by using a [Flow Description Language](#) (FDL) consisting of some extension to Tcl.

A simple `Flow.tcl` file may look like the following:

```
E BASE
N "Copy"
T vw cp aa bb
I aa
O bb
```

The first line calls the procedure `E` which sets the [environment](#) to "BASE". This means that any job to be declared from now on will be executed in the "BASE" environment.

The second line call procedure `N` to assign a name to the subsequent job. In this case the name is "Copy". If the name is not specified, or if the name is the empty string "", then the system will use the tool name (in this case "cp") as the job name.

The next line calls procedure `T` to add a tool invocation with command line "`vw cp aa bb`". This job is to be run in the current working directory, using the environment "BASE".

The next two lines call the procedures `I` and `O` to declare the expected inputs and outputs for the job.

The `Flow.tcl` script is then processed by `vovbuild`.

```
% vovbuild
.
```

`vovbuild` does not execute the commands. Instead, it builds or updates the design trace by adding all jobs defined in the Tcl script.

After the trace is built, you can request a parallel execution with the command `vsr`.

All extra arguments to `vovbuild` are passed to the Tcl script and are available as the global array `argv`.

Tracing the Execution of vovbuild

The tool `vovbuild` itself is an instrumented tool. This means that you will be able to trace the invocation of `vovbuild` itself and to have FlowTracer monitor the changes in the `Flow.tcl` file as well. The input list of a `vovbuild` job includes `Flow.tcl` and possibly many other Tcl scripts. These scripts are now an important part of your design, and it is important to trace the changes to these files. By default, tracing is disabled in `vovbuild`, but you can enable it with the option `-T`:

```
% vovbuild -T
```

In this way, if `Flow.tcl` changes, FlowTracer knows that you must rerun `vovbuild`.

If you have tens or hundreds of `Flow.tcl` files in as many directories, you will end up with many invocations of `vovbuild -T`, all indistinguishable from one another in many of the FlowTracer messages. The option `-l` allows you to add arbitrary descriptive text to the command line; its use purely a matter of style and clarity and is recommended to avoid confusion. Example:

```
% vovbuild -T -l "Label to distinguish this job from other jobs"
```

Conflicts in vovbuild

During the building of a flow, it is possible to generate conflicting flow fragments.

Type	Cause	Handling
Output conflicts	A <code>J</code> or an <code>O</code> statement generate an output dependency for a file that is already the output of another job.	Output conflicts arising from running <code>vovbuild</code> are resolved with "force," that is, in case of output conflict, build rules that are processed later override the preceding rules. To aid the debugging of flows, the old job remains in the trace, with its status set to <code>SLEEPING</code> and with an explanation of what happened. You can access to this explanation by visiting the sleeping job and by asking "why is it sleeping?"
Input conflicts	The flow declares as input a file that is not <code>VALID</code> .	Input conflicts during flow builds are not really conflicts and are always ignored.
Cycle conflicts	A declaration that causes a cycle in the dependency graph	Cycle conflicts are always fatal errors that force <code>vovbuild</code> to abort.

Use `O_CONFLICT` to Choose Behavior in Case of Output Conflict

This procedure sets the handler for output conflicts in FDL. The default behavior is to `ABORT`. It is also possible to choose `RETRY`, which kills the currently running old job and then retries the output declaration.

It is also possible to pass the name of a **custom handler**. This handler takes 3 arguments: `jobId` `outputId` `oldJobId` and must return `"RETRY"` if successful.

Usage:

```
O_CONFLICT <ABORT OR RETRY OR name_of_handler>
```

Examples:

```
O_CONFLICT ABORT

J vw cp aa1 bb
J vw cp aa2 bb ; ## This will abort
```

Example of output-conflict-handler

```
proc VovOutputConflictHandlerAbort { jobId placeId oldJobId } {
    VovFatalError "Cannot declare $placeId as output of $jobId because it is already
    the output of $oldJobId"
}

## Use this handler on output conflict.
O_CONFLICT VovOutputConflictHandlerAbort
```

Usage: O_CONFLICT <ABORT|RETRY|name_of_handler>
This procedure sets the handler for output conflicts in FDL.
The default behavior is to ABORT. It is also possible to
choose RETRY, which kills the currently running old job and then
retries the output declaration.
It is also possible to pass the name of a custom handler.
This handler takes 3 arguments: jobId outputId oldJobId
and must return "RETRY" if successful.

vovbuild Argument Parsing

Like in all Tcl scripts, command line arguments are passed using the variables *argv0* and *argv*. If you need to pass parameters to the Flow Description, you can consider using the command line.

The best way to separate the *vovbuild* options from the options to be passed to the flow is to use the double dash "--". The double slash itself is passed to the flow description.

```
# The following code parses commands of the form:
# % vovbuild -f Flow.tcl -- -type gates -drc

# Common way to parse command line options:
set blockType "undefined"
set doDrc      0
while { $argv != {} } {
    set arg [shift]
    switch -glob -- $arg {
        "--" { }
        "-type" { set blockType [shift] }
        "-drc" { set doDrc 1 }
        "-*" { VovUserError "Unknown option '$arg'" }
        default { VovUserError "Unknown argument '$arg'" }
    }
}
```

Task Oriented Flows

Summary of procedures reviewed in this section:

Procedure	Description
PARALLEL	The tasks described within the script are to be considered parallel, meaning that they can be executed in any order and also concurrently.
SERIAL	The tasks described within the script are to be executed in the specified order.
TASK	Describe a task in a SERIAL or PARALLEL context

Keep it Simple

The idea behind the flows using parallel and serial tasks is to allow quick implementation of simple flows. These flows are based on a serial/parallel description of the relationships between tasks. It is the user's responsibility to make sure that such relationships are correct. Although these flows do not use any form of [Tool Integration](#), they still offer visualization, resource management, and parallel execution of tasks. These flows are not the most efficient that can be written, but may be an easy way to begin using FlowTracer.

Here are some examples of task-oriented flows:

```
SERIAL {
  TASK ./configure
  TASK make clean
  TASK make install
}

PARALLEL {
  foreach dir { dir1 dir2 dir3 } {
    indir $dir {
      TASK make install
    }
  }
}
```

SERIAL and PARALLEL can be deeply nested and are fully compatible with all other FDL directives including S, J, T, etc, as in this example:

```
PARALLEL {
  foreach dir { dir1 dir2 dir3 } {
    indir $dir {
      SERIAL {
        TASK make clean
        TASK make install
        S "PostInstall" {
          PARALLEL {
            J vw run_script1.csh
            T vw run_script2.csh
          }
        }
      }
    }
  }
}
```

```
    }  
  }  
}
```

or in this other example of File: TaskComplex.tcl:

```
PARALLEL {  
  SERIAL {  
    N "PS1"  
    TASK cal 2001  
    TASK cal 2002  
    TASK cal 2003  
    TASK cal 2004  
  }  
  SERIAL {  
    N "PS2"  
    TASK cal 2011  
    TASK cal 2012  
    TASK cal 2013  
    TASK cal 2014  
  }  
}
```

Adding Resources

Regular FDL procedures can be used to enhance the TASK oriented flows. For example, the procedure `R` can be used to specify the resource requirements for a task (and all subsequent tasks):

```
SERIAL {  
  R "License:Synthesis RAM/2000"  
  TASK doSynthesis ALU  
  foreach test $listOfTests {  
    R "License:Simulator RAM/1500"  
    TASK doSimulation ALU $test  
  }  
}
```

Using S (setname) and N (name)

In the following flow, you can see the use of `s` to collect the tasks in this flow into a set and the use of `N` to give names to each task.

```
S "MainTasks" {  
  SERIAL {  
    R "pseudotasks !unix"  
    N "Builds"  
    TASK vsr -set "All:vovbuild"  
  
    N "Headers"  
    TASK vsr -cb -f -recompute -set All:include -timeout 30m  
  
    N "Libraries"  
    TASK vsr -cb -f -recompute -set All:libraries -timeout 30m  
  
    N "AllTheRest"  
    TASK vsr -all -nowait  
  }  
}
```

Dependencies in FDL

In most of the flow examples, we have not shown any explicit dependency, mostly because we have used `J` rather than `T`.

In practice, it is common to use the FDL procedures `I` and `O` to describe basic dependencies in the jobs. Both `I` and `O` must follow the job to which they refer. For example:

```
T vw cp $in $out
I $in
O $out
```



Important:

- The dependencies you describe in the Flow Description only form the starting point of the flow graph.
- In FlowTracer the dependencies for each job are dynamically computed by the tool integration at run time.
- In FlowTracer all jobs must have at least one output dependency in order to be considered valid.
- In Accelerator, there is no runtime tracing, and the flow graph is not updated at run time.

Incorrect Dependencies

Incorrect dependencies may create problems. Given that runtime tracing will fill in the dependency details, you should only specify a minimum set of dependencies in your flow description and let runtime tracing fill in the rest automatically.

- Incorrect Input Dependency

```
# Incorrect input dependency
T vw cp aa bb
I cc; ##### incorrect input
I aa
O bb
```

This example will generate an spurious input in the flow. If the file `cc` changes, it will invalidate the job and will cause it to be re-run needlessly. However, running the job will fix the input dependencies for `bb` by dropping the dependency on `cc`.

- Incorrect Output Dependency

```
# Incorrect output dependency
T vw cp aa bb
I aa
O bb
O cc; ##### Spurious output
```

This example will generate a spurious output in the flow. However, running the job will fix the output dependencies by dropping the output dependency on `cc`.

Arbitrary Dependencies with **DEPENDENCY** or with **AD**

The procedures **DEPENDENCY** (and its synonym **AD**) take two list of IDs as parameters. In the common form, they take two simple IDs:

```
DEPENDENCY $J1 $J2
```

or

```
AD $J1 $J2
```

These procedures create a **PHANTOM** place and declare it as a sticky output to the job with VovId J1 and as a sticky input to the job with VovId J2.

Example:

```
set j1 [J vw cp aa bb]
set j2 [J vw cp aa cc]

# Force the second job to be executed after the first job, even
# if there is no particular reason for it. You can also use the
# short name 'AD' (for Artificial Dependency).
DEPENDENCY $j1 $j2
```

The flow above can be written more concisely as:

```
SERIAL {
  TASK cp aa bb
  TASK cp aa cc
}
```

which also creates an artificial dependency between the two tasks by means of a "sticky phantom". This is one way to force a job to have an output without explicitly declaring it at runtime.

Forcing Dependencies

To force a dependency that is not a "runtime dependency" you can include "Sticky" dependencies to your flow description. Since all jobs require at least one output to be valid, using sticky is a good way to keep a dependency around without having to instrument it or declare it as an output during execution of the job.

Example:

```
J vw cp aa bb
I -sticky /usr/man/man1/cp.1
```

To force all dependencies to be sticky by default, use the option `-s` in `vovbuild` when building the flow:

```
% vovbuild -s -f Flow.tcl
```

vtk_transition_add

The fundamental procedure executed by `vovbuild` is `vtk_transition_add`, which is typically called indirectly by the FDL procedures `T` and `J`. The procedure takes six arguments:

- The working directory
- The environment
- The resources
- The command line
- The list of inputs
- The list of outputs

This procedure is controlled by `make(ctrl,action)`, which can be one of `trace`, `script`, or `echo`.

- If `make(ctrl,action)` is `trace`, this procedure is an interface to `vtk_transition_add_to_trace`, which takes the same arguments. The effect of this procedure is to add the specified transition to the trace. If the transition already exists, its resources will be augmented, if necessary. At least all the files specified in the input list will have arcs into the transition, and similarly for the outputs. Existing transitions with output arcs to files in the list of outputs are deleted.
- If the action is `script`, the procedure prints on stdout an executable script valid for the architecture `make(arch)`.
- If the action is `echo`, the transition information including inputs and outputs is echoed on stdout.

Example:

```
vtk_transition_add . "BASE" "unix" "clevercopy $filein $installdir" "$filein"  
"$installdir/[file tail $filein]"
```

Limits

The VOV system enforces limits on the lengths of some of these parameters. Calls to `vtk_transition_add` with parameters exceeding the limit will fail to add the transition and generate a message.

Current limits (in characters) are:

- **Command** 40960
- **Environment spec** 512
- **Directory** 1024
- **Resource spec** 1024

Difference Between E and ves

In FDL you have two procedures that affect the environment: `E` and `ves`. The procedures operate very differently.

The procedure `ves` changes the current environment.

The procedure `E` determines the environment that will be used by the jobs created with procedures `J` and `T`.

Linear Coding in FDL

A large part of your flow description will look a lot like this fragment of linear coding, in which we define environment, resources, and command line for a set of jobs:

```
E WORK+SYNTH
R "synthesis_license sun5"
J vw runSynthesis $block
I $block.v
O $block.vg

E WORK+VERIFY
R "drc_license"
J vw runDRC $block

E WORK+VERIFY
R "lvs_license"
J vw runLVS $block

# The Altair Accelerator binary 'clevercopy' is a special executable that is
# instrumented.
R ""
J clevercopy -m 644 $block.gds $GDSDIR/$block.gds
```

Set Creation in FDL

Grouping jobs into named Sets enables you to use those Sets to specify retracing targets, create high-level flows, and generate reports. Use the CLI to create the Sets and order them in a hierarchical manner that makes sense to you. Each Set is created by using the `S` or `S+` command.

Important: You can name Sets anything, but you cannot use the same name more than once. If you use the same name for the Set, the previous Set will be overwritten. To add to an existing Set, use the `S+` procedure to append to a Set. See the instructions below.

One Set

To create a small, simple Set, do the following:

```
S foo1 { J vw sleep 0 } ## ID001
```

This create a Set name **foo1**, with one sleep job. For clarification purposes, an ID has been assigned. In the GUI, this is represented as such:

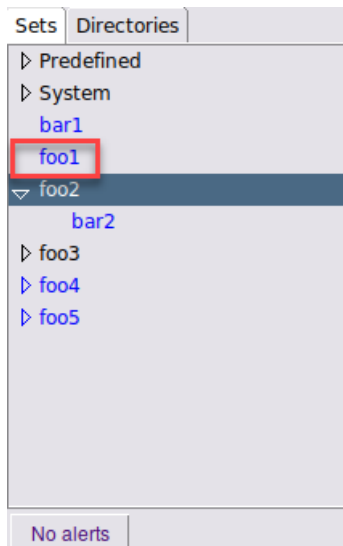


Figure 2: One Set

One Nested Set

To create a Set nested within a Set, do the following:

```
S foo2 {                                ## ID002  
  S bar2 { J vw sleep 0 }                ## ID003  
}
```

Here, two Sets are created, named **foo2** and **bar2**. However, only **bar2** has a job assigned to it. Here is the GUI representation:

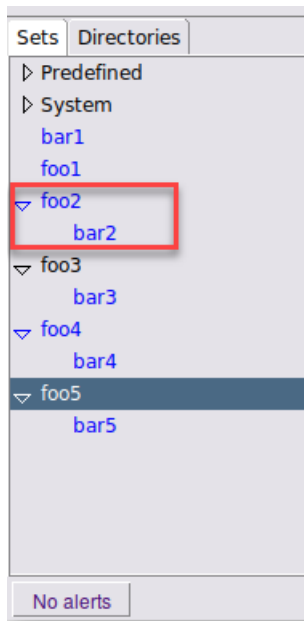


Figure 3: Set Nested with a Set

Two Sets are created, so each receives it's own ID number.

Using the Colon Syntax

You can use a colon ":" to create a visual representation of nested Sets in the Sets tab. This can be useful for organizing Sets without the need to create an actual Set for every level of the desired hierarchy.

Create a visual representation of a hierarchical set structure, with only one job in one Set:

```
S foo3:bar3 { J vw sleep 0 } ## ID003
```

In this example, **foo3** is just part of a single Set named **foo3:bar3**, which has one job. This is represented in the GUI as such:

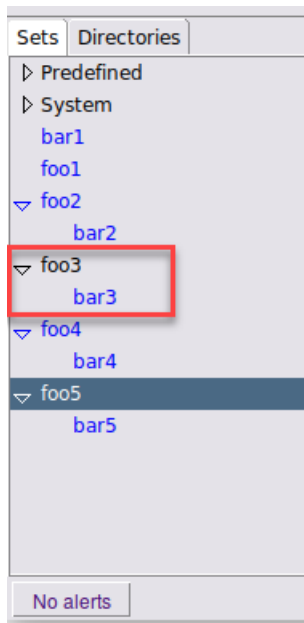


Figure 4: Set Created with Colon Syntax

You can see that **foo3** is shown in black, which indicates it is for visual representation only. **bar3** is shown in blue because it is the last colon-separated name component, so it represents the entire **foo3:bar3** Set, which contains one job.

Because **bar3** is only visually nested within **foo3**, one ID is associated with the Set.

Create Sets Each Having Jobs using the Colon Syntax

In the example above, only one Set had a job. However, you can create multiple Sets where each will have jobs associated with them, and use the colon to appear as nested. In this example, two Sets are created, and both will contain jobs.

```
S foo4 { J vw sleep 0 } ## ID004
S foo4:bar4 { J vw sleep 0 } ## ID004
```

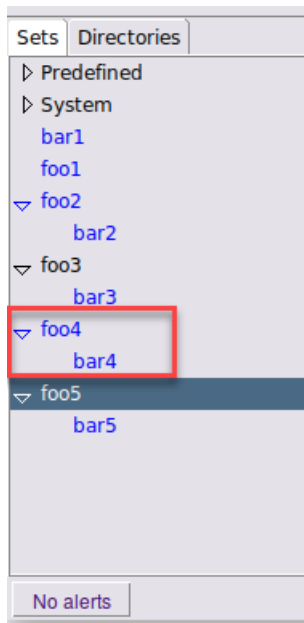


Figure 5: Two Sets

Here, both **foo4** and **bar4** are blue, because both Sets contain functioning objects. This can also be created with this command structure:

```
S foo5 {  
  J vw sleep 0  
  S foo5:bar5 { J vw sleep 0 }  
}
```

Set **foo5** is created, with a job associated with it. Then, **bar5** is created within **foo5**, also with it's own job.

Append Sets

You can append to an existing set by using `s+`. This enables you to add to an existing Set without overwriting the original Set.

An example of using `S+` and colons to create hierarchical sets is shown below:

Create the first Set

```
S "foo1" {  
  J vw cp aa bb  
}
```

Append a second Set to the first using `S+`

```
S+ "foo1" {  
  J vw cp bb cc  
}
```

foo1 now has two jobs attached to it. "J vw cp aa bb" and "J vw cp bb cc". Because `S+` was used, rather than just the `s` procedure, the jobs were added to the **foo1** Set, rather than overwriting the original.

Append Jobs to a Non-existing Set

If you try to append jobs to a Set that does not exist yet, the Set is created by default, with the added jobs attached to it.

For example

```
S+ "MyNewSet" {  
    J vw cp cc dd  
}
```

A new Set named "MyNewSet" is created.

Common Mistakes

Do not use the same Set name more than once. The procedure `s` is used to define exactly the content of the set. Extra elements are eliminated from the set. In the following example, the execution of the FDL fragment terminates with one set called "MySet" which contains the job `vw cp xx yy` and this job is always going to be "INVALID".

```
S "MySet" {  
    J vw cp aa bb  
}  
# At this point the set MySet contains "vw cp aa bb"  
  
S "MySet" {  
    J vw cp xx yy  
}  
  
# At this point the set MySet contains "vw cp xx yy".  
# The job vw cp aa bb is forgotten
```

Conditional Flows

There are many different types of conditions that affect the shape of the flow, from conditions that are known at "build-time", meaning at the time `vovbuild` is first executed, to conditions that are known only after one or more jobs have been executed. In this section, we examine a couple of common types.

Build-time Conditions

Conditions that can be evaluated at build time are the easiest to handle because we can use standard Tcl procedures like `if` and `switch`.

```
if { $blockType == "large" } {  
    J vw runFastRouter $block  
} else {  
    J vw runNormalRouter $block  
}
```

```
}
```

Complex Conditions

Sometimes a flow depends on a computation that involves one or more complex tools. For example, we may have a methodology that says:

If the block meets timing, then publish the block, else improve timing

where the condition "meets timing" is computed by running expensive (in term of software licenses) and time-consuming tools. The way to handle such flows is to build them one piece at a time and to include the invocation of the `vovbuild` itself as part of the flow.

```
# This is BlockFlow.tcl
#
# A procedure to get the slack (timing) of a block.
# A negative slack means that the block does not meet timing.
#
proc getBlockSlack { file step } {
    set fp [open $file]
    set slack [read $fp]
    close $fp
    return $slack
}

#
# In the first part of the flow, we run a few jobs
# and then we run another vovbuild.
#
proc partOne { block } {
    S "Block:$block:Step1" {
        E WORK
        J vw runSynth $block
        J vw measureSlack $block -step 1
        O reports/step1/$block.timing

        J vovbuild -T -f BlockFlow.tcl -- -step 2
        I reports/step1/$block.timing
    }
}

#
# In the second part of the flow, we get the
# timing of the block and then decide what to do
# based in it.
#
proc partTwo { block step } {
    set n1 [expr $step -1 ]
    set slack [getBlockTiming "step$n1/$block.timing"]
    S "Block:$block:Step$step" {
        if { $slack >= 0 } {
            # The block meets timing.
            J clevercopy -m 644 $block.v ../../netlists
        } else {
            # The block does not meet timing.
            E WORK
            R "timing_opt_license"
            J vw refineTiming $block -step $step
            O reports/step$step/$block.timing

            J vovbuild -T -f BlockFlow.tcl -- -step [expr $step + 1]
        }
    }
}
```

```
        I reports/step$step/$block.timing
    }
}

#
# Main controller code for the flow.
#

set block [file tail [pwd]]
set step 1
while { $argv != {} } {
    set arg [shift]
    switch -glob -- $arg {
        "-step" { set step [shift] }
    }
}

switch $step {
    1 { partOne $block }
    2 - 3 - 4 - 5 - 6 { partTwo $block $step }
    default {
        VovFatalError "Too many iterations: timing does not converge"
    }
}
```

Decisions in Flows with IFJOB

Another way to add conditionals into the flows consists of adding the decision code directly into the FDL. This can be done with the procedure `IFJOB`, as illustrated by the following example.

The example is of course contrived. We assume that we do some design jobs (e.g. copying files) and then we check some properties of the results, e.g. the size of the output file, to determine whether we have met some attribute of the design or not. In this example, we pretend that the attribute is "timing", which is a property of a digital circuit. If the timing is met, we do something, otherwise we do something completely different.

```
# Do some design activity...
N "standinDesignActivity"
J vw cp aa bb

# Now add the decision node.
IFJOB -label checkTiming {
    puts "Hello: we are inside checkTiming"
    set timingOk [expr [file size bb] & 1]
    puts "TimingOk = $timingOk Size = [file size bb]"

    set sId [S "Result:decision:checkTiming" {
        if { $timingOk } {
            N "TIMING OK"
            J vw cp bb cc
            J vw cp cc dd
            J vw cp dd ee
            J vw cp ee final
        } else {
```

```
        N "FIX_TIMING"
        J vw cp bb cc1
        J vw cp bb cc2
        J vw cp bb cc3
        J vw cp bb cc4
        J vw cat cc1 cc2 cc3 cc4 > final
    }
}]]
AD $env(VOV_JOBID) $sId
# after 5000; START $sId
}
```

This flow initially creates two jobs, the `cp aa bb` and another job called `vovdecision`, which depends on the first job based on the sequence of the code (or we could have used the option `-jobid ID` in `IFJOB` to specify a dependency with another job). The code fragment provided as last argument to `IFJOB` is attached to the decision node as an annotation.

When the decision node is executed, it executes the code fragment. In this case, the code consists of a measurement (pretend that the size of the file `bb` is an expression of the timing of a circuit), and then decides: if the timing is ok, it creates a flow, otherwise it creates a different flow.

The utility `vovdecision` is essentially an invocation of `vovbuild` with a small flow file that can be embedded right into the top-level flow.

The limit for the length of the script is about 30kB, but we expect the code fragments used in decision nodes to be much simpler. For more complex decisions, you may want to write a separate flow file and execute it normally with `vovbuild`. You can add the `vovbuild` job to the trace by using the option `-T`, as in the following example:

```
% vovbuild -T -f anotherDecisionFlow.tcl
```

Parse Configuration Files

It is common for a flow to depend on an existing configuration file.

We do not recommend the use of configuration files for this purpose, because they tend to introduce a dependency between the file and the flow that is likely to invalidate an entire flow even for simple changes to the config file.

Nevertheless, since the use of such files is widespread, we want to show you how you can easily extract information from those files to control your flows.

Every configuration file has a slightly different syntax. In the following table, it is assumed that you have a configuration file that contains simple assignments, comments and empty lines. We trust you will know how to adapt the code fragments below to your particular application.

```
# Fragment of fictitious configuration file.
VERILOG  = finsim
SYNTH    = autosyn
PLACER   = se
ROUTER   = apollo

# More parameters.
```

```
DO_DRC    = 0
DO_LVS    = 1
```

To parse the configuration file, we can use standard Tcl:

```
proc parseConfigurationFile { file } {
    # An array to collect the parameters.
    global PARAM

    # Open the file and read one line at a time.
    set fp [open $file "r"]
    while { [gets $fp line] >= 0 } {
        if [regexp {^#} $line] continue; # Discard comments.
        if [regexp {(.)=(.)} $line all var value] {
            set var [string trim $var]
            set value [string trim $value]
            set PARAM($var) $value
        }
    }
    close $fp
}
```

RECONCILE_WITH_FILE_SYSTEM

```
RECONCILE_WITH_FILE_SYSTEM { setName "System:jobs" } { verboseFlag 0 }
```

If some jobs are executed outside of FlowTracer, the system considers the unjustified changes of timestamps as errors and invalidates the flow below the changes. This is the correct and safe response. However, occasionally, it may be desirable to avoid rerunning expensive jobs and tell FlowTracer to accept the status of the files as they are.

The procedure `RECONCILE_WITH_FILE_SYSTEM` scans the inputs and outputs of each `INVALID` job. If all the inputs are older than the youngest of all outputs, then the job is turned to `VALID`. The indication that the job was reconciled and not executed is the name of the execution host, which becomes the string `"__reconciled__"`.

Create Directories in FDL

We recommend that you use the flow description to create directories necessary to run the tools and to collect the results.

You can use one of the methods shown below:

```
# Use the Tcl standard 'file mkdir' procedure.
file mkdir $installDir

# Use the VOV extension 'indir -create {}'.
indir -create $subDir {
    # ....
}
```

```
}  
  
# Avoid this method; it is slower and does not work on Windows.  
exec mkdir $subDir
```

File Generation in FDL

We recommend that all files in a flow be generated by some job rather than as part of `vovbuild`. Occasionally, however, it may be more convenient to generate some files as part of the flow construction, that is by `vovbuild` itself.

If that is the case, we recommend that such files be created with a barrier to change propagation. To this effect, you may find convenient to use `VovBarrierOpen` and `VovBarrierClose`, as in the following example.

```
# Recommended method to create a file as part of vovbuild.  
set fp [VovBarrierOpen "config.txt" ]  
puts $fp "THE CONTENT OF THE FILE"  
VovBarrierClose $fp
```

Use the option `-T` to trace the execution of `vovbuild`, as in:

```
% vovbuild -T ...
```

Old Names for VovBarrierOpen

The procedure `barrierOpen` is the old name for `VovBarrierOpen` and has the same functionality. The same applies for `barrierClose` and `VovBarrierClose`.

Generate FDL Using the Instrumentation Library

The instrumentation library (`VovInput`, `VovOutput`, ...) can generate FDL if the variable `VOV_FDL_ONLY` is set. This can be used to ease the integration of makefiles and instrumented scripts.

Example Using C-shell

File: `example.csh`

```
#!/bin/csh -f  
##### Part 1: The FlowTracer Section.  
VovFdl T vov example.csh  
VovResources "RAM/10"  
VovInput aa /bin/cp      || exit 1  
VovOutput bb             || exit 1  
  
if ( $?VOV_FDL_ONLY ) exit  
  
##### Part 2: Do The Real Work Here.
```

```
cp aa bb
exit 0
```

Example using Makefile

File: Makefile_original

```
# Example of a Makefile that uses the VIL-Tools to generate FDL.
target: someInput
    -@mv someOutput someOutput.bak
    ./someScript someInput someOutput
```

File: Makefile_with_fdl

```
# Example of a Makefile that uses the VIL-Tools to generate FDL.
target: someInput fdl_target
    -@mv someOutput someOutput.bak
    ./someScript someInput someOutput

# Additional rules for FlowTracer
fdl_target:
    @VovFdl T vov make target
    @VovInput    someInput
    @VovOutput -ignore_timestamp someOutput
```

```
# Fragment of Flow.tcl code to use the information
# in the 'fdl' target of the Makefile above.
set fdl [exec env VOV_FDL_ONLY=1 make fdl_$target]
catch {eval $fdl}

# Fragment of Flow.tcl code to use the information
# in the 'fdl' target of the Makefile above.
set fdlFile /tmp/vovfdl[pid]
setenv VOV_FDL_ONLY $fdlFile
catch {file delete $fdlFile}
catch {exec make fdl}
if [catch {source $fdlFile} errmsg] {
    VovFatalError "Cannot source $fdlFile: $errmsg"
}
catch {file delete $fdlFile}
```

Run Jobs in an xterm

This section applies only to UNIX. In Windows, it is not possible to run jobs in an xterm.

Using FlowTracer in UNIX, jobs can be run in four modes:

- normal mode, i.e. in the background, with no stdin. We expect most jobs to be using this execution mode.
- xterm_open mode, i.e. within an open xterm
- xterm_icon mode, i.e. within an iconified xterm
- xterm_none mode, similar to "normal" except this is using the same sequence of commands as the xterm_open and xterm_icon modes

The xterm is opened on the display specified by the X11_DISPLAY property, which can be set either on each job or on the object with ID 1.

When executing in xterm, you may specify the name of an additional log file using the property VXT_LOG. This property is ignored in "normal" run-mode.

The run-mode is honored by the wrapper (vw, vov, ...). If you have a job with no wrapper, then you cannot change its run-mode.

The run-mode can be changed from the GUI, or can be specified in FDL. Changing the run mode of a job, does not invalidate the job.

```
# Example of FDL
RUNMODE xterm_open
J vw run_some_job

# Alternative: use option -xt -xi -xn -xb in T,J
J -xt run_som_job
```

The RUNMODE procedure sets the run-mode for all subsequent jobs, while the option -xt, -xi, -xn, -xb only affects one job.

Subtle Differences between Normal and xterm_none

In "normal" run-mode, the command is a direct child of the wrapper, while in "xterm_none" the command is processed by the script "vov_job_xterm". In general "normal" is a bit faster than "xterm_none" and the VXT_LOG property is ignored. Ideally, "xterm_none" should behave the same as "normal", but because of the additional processing, the command may behave differently in the presence of syntax characters like redirections, pipes and quotes.

To have full control on complex commands, we recommend you choose the shell that you want to use as part of the command. Example:

```
X11_DISPLAY my_workstation:0.0
RUNMODE xterm_none
N "complicated command with pipes"
T /bin/sh -c "perl myscript | awk '{print $2}' | sort -u"
P VXT_LOG some_log_used_when_the_runmode_is_xterm_something.log
```

Controlling the DISPLAY

You can set the X11_DISPLAY variable in your flow with the X11_DISPLAY procedure.

```
# Fragment of Tcl code.
X11_DISPLAY -help

Description:
Set the X11_DISPLAY property on an object.
This property is used by the jobs that have
a runmode of xterm_icon or xterm_open.

Usage:
X11_DISPLAY value <objectSpec>

If value is empty, the property is deleted.
If objectSpec is -help, this message is shown.
```

```
If objectSpec is empty or "main", the VovId 1 is used.  
If objectSpec is "lastjob", the most recent job is used.
```

```
Examples:  
X11_DISPLAY localhost:0.0  
X11_DISPLAY localhost:0.0  
X11_DISPLAY localhost:0.0 main  
X11_DISPLAY mac01:0.0 lastjob
```

FDL and make Comparison

The utility `make` manages flow and dependency information in software projects as well as in many other domains. It is portable across multiple platforms, it is free, it is a standard, and everybody familiar with it knows its idiosyncrasies.

In the development of FlowTracer, we wanted to make sure that all the makefiles could still be used to build the design trace. For example, we wanted it to be possible to transfer the flow information from the makefile into the design trace using either `vovmake` or using the "interception" technique.

After many years of FlowTracer development, it has become increasingly hard to justify the effort of maintaining a makefile, with its arcane syntax consisting of `%.c`, `$<`, `$@` and invisible tabs, only to get a representation of dependencies that is, in many cases, incomplete, incorrect, or both.

The many benefits of FDL are:

- All dependencies are maintained up to date automatically.
- Dependency constraints are never violated.
- In a heterogeneous network, one can build targets for any architecture easily. For example, to build a MacOS-X binary from any window with access to the FlowTracer server, we can enter:

```
% vsr ../macosx/bin/vovserver
```

- It is easy to define sets of targets for the user's convenience. For example, in the development of FlowTracer there is a set called `All:include` consisting of all of the generated headers used during compilation. To retrace all the files in this set, use:

```
% vsr -set All:include
```

- The build-rules are expressed with a Tcl script. The Tcl syntax is simpler than that of a traditional makefile and allows proper looping and conditional constructs. With Tcl, rules can be expressed more compactly than with `make`.
- Designers have complete control on the working directory and on the execution environment.
- FlowTracer offers coordination of team effort. Multiple overlapping build requests, coming possibly from different designers, will never cause any tool to be executed more than once.
- Changes to the build rules themselves can also be traced.
- FlowTracer supports parallel development of multiple variants. For example, for each piece of software one typically needs a debuggable version (`-g`), an optimized version (`-O`), and a profilable version (`-pg`).

- FlowTracer supports parallel development on multiple platforms.
- FlowTracer supports [Runtime Change Propagation Control](#). It understands that some changes do not need to be propagated. With proper barriers to change propagation, FlowTracer frees the designers from the tyranny of timestamps.
- In lines of code, the flow written with FDL are much smaller than the corresponding Makefiles, resulting in lower development and maintenance costs.

Run Interactive Jobs with vxt

You may want to run some of your jobs in an interactive fashion where a user can press [Ctrl]-[C] and take the hand on the tool in a terminal in the middle of a run.

FlowTracer's FDL allows you to run some jobs from inside an X-window terminal by giving some option to the `J` or `T` keyword:

- `-xt`: runs the job in an open xterm (steals mouse focus). Logging happens as usual despite the xterm.
- `-xi`: runs the job in an iconified xterm: allows the benefit of running in an xterm without stealing focus from the user when the job starts.
- `-xb`: runs in the background (no need for a `DISPLAY` even). This is similar to not using any option at all and is implemented for completeness of the user interface.

The developer has full control on the interactive terminal used to run the tool and can alter the properties: `VXT_XTERM` and `VXT_LOG` after the `J -xt` line with a `vtk_prop_set`.

Changing the `VXT_XTERM` property would allow for different arguments to be passed to xterm, or a different terminal like `konsole` or `rxvt` to be used as examples.

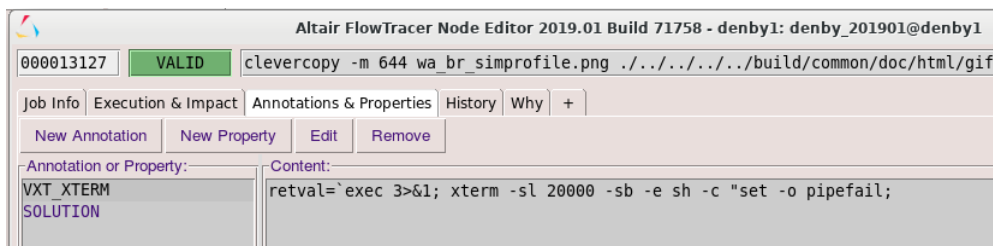


Figure 6:

During `vovbuild`, the developer can modify a job after its definition by changing the property with something similar to:

```
vtk_prop_set $node_id VXT_LOG {retval=\`exec 3>&1; xterm -sl 20000 -sb -e sh -c \"set -o pipefail;\"}
```

Once the developer is happy with the changes, it can be made the default for all further jobs in the build by setting "make(VXT_XTERM)" as in:

```
set make(VXT_XTERM) {retval=`exec 3>&1; xterm -sl 20000 -sb -e sh -c "set -o pipefail;`}
```

Changing the VXT_LOG property would allow for example for capturing the log in a different name file:

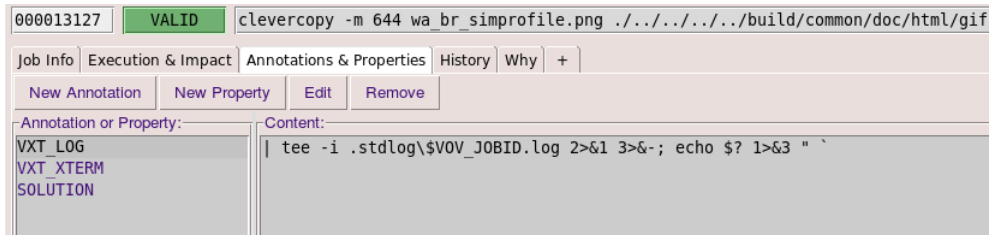



Figure 7:

During vovbuild, the developer can modify a job after its definition by changing the property with something similar to:

```
vtk_prop_set $node_id VXT_LOG {| tee -i .stdlog\${VOV_JOBID}.log 2>&1 3>&-; echo $? 1>&3 " `}
```

Once the developer is happy with the changes, it can be made the default for all further jobs in the build by setting "make(VXT_XTERM)" as in:

```
set make(VXT_LOG) {| tee -i .stdlog\${VOV_JOBID}.log 2>&1 3>&-; echo $? 1>&3 " `}
```

 **Note:** This feature **supports only simple commands** (single tool). If you need some complex commands, write them in a small shell script, and use the small shell script as the command line for vxt.

Tcl Interface to Various Set Operations

List Sets

Use `vtk_trace_list_sets` to list all the sets. Usage details can be found in the *VOV Reference Guide*.

Example:

```
# -- List sets alphabetically
foreach set [lsort [vtk_trace_list_sets]] {
    puts "Set: $set"
}
```

Find Sets by Name

To map a set name to a VovId, use `vtk_set_find` as in:

```
set setId [vtk_set_find $setName]
```

This procedure returns zero (actually 00000000) if there is no set by the given name. All usage details of API procedures for sets are located in the *VOV Reference Guide*.

List Set Elements

To list the elements of a given set, use `vtk_set_get_elements`. This procedure requires at least two arguments, the VovId of a set and a formatting string.

Example:

```
set setId [vtk_set_find "System:files"]
foreach info [vtk_set_get_elements $setId "@ID@ @NAME@" ] {
    puts "$info"
}
```

Smart Sets

Smart sets are sets that automatically update after every node change and include all nodes based on a selection rule given at set creation time. This feature can be very powerful, but very expensive.

Smart sets created with the `-smarc` argument only update on node creation, reducing the load created by smart sets.

Create Sets

From a Tcl script, use the procedure `vtk_set_create`, which requires two arguments, the [name](#) of the new set and the selection rule.

The procedure returns the VovId of the new set. If a set by the same name exists already, it is forgotten and a new set with a new VovId is created.

Example:

```
# Create an empty set by using "" (the null string) as the selection rule.
set setId [vtk_set_create "tmp:myNewSet" ""]

# Create a set of all jobs that are FAILED at the moment of the
# creation of the set
set setId [vtk_set_create "my_failed_jobs" "isjob status==FAILED"]

# Create a set of all jobs that are not VALID at the moment of the
# creation of the set. The set will be destroyed when the calling
# process exists (-client). The set creation will not create events (-transient).
set setId [vtk_set_create -client -transient "tmp:failed" "isjob status!=VALID"]

# Create a smart set of all jobs that are SUSPENDED.
# The smart set is updated automatically by vovserver.
# The set expires after 5 minutes from the creation.
set setId [vtk_set_create -smart -expire 5m "My:Suspended" "isjob status==SUSPENDED"]

# Create a hierarchical set. In the following example, MySet and Subset are created,
# Subset becomes the subset of MySet.
```

```
set setId [vtk_set_create -hier MySet:Subset ""]
```

Forget Sets and Elements of Sets

Use `vtk_set_forget` to forget a set. This procedure returns "ok" if the set has been forgotten and returns an error otherwise, e.g. if the VovId is not valid for a set.

```
vtk_set_forget $setId
```

To forget all the elements of a set, use the option `-elements` as in:

```
vtk_set_forget -elements $setId
```

To forget a set and all the subsets, use the option `-hier` as in:

```
vtk_set_forget -hier $setId
```

To forget all elements in the set and the subsets:

```
vtk_set_forget -hier -elements $setId
```

Rename Sets

To rename a set, use `vtk_set_rename` as in:

```
vtk_set_rename $setId "new set name"
```

Operations on Sets

Many operations may be performed on sets. All operations can be performed from the Tcl interface using a procedure called `vtk_set_operation`. Many operations are also available from the CLI and from the GUI. This section deals only with the Tcl interface.

Attaching and detaching nodes

To attach and detach nodes from a set, you need the VovId of the set and the VovId of the node you want to attach or detach. Then you use one of these two forms:

```
vtk_set_operation $setId $nodeId ATTACH  
vtk_set_operation $setId $nodeId DETACH
```

To detach all elements of a set, use:

```
vtk_set_operation $setId ignored CLEAR
```

The second argument is ignored. Detached nodes remain in the trace.

Testing for membership

To test for membership of a node in a set, you need the VovId of the set and the VovId of the node. The following expression returns 1 if the node is a member of the set, and 0 otherwise:

Note: in 2015.03 and later releases, `ISMEMBER` may be used as an alternate to `CONTAINS`.

```
vtk_set_operation $setId $nodeId CONTAINS
```

Intersection and union of sets

To intersect or unite two sets you need the VovId's of both sets. One of the sets will be the "accumulator" set, because it collects the result of the operation. In the case of intersection, all elements in the accumulator that do not belong in the other set are detached from the accumulator set. In the case of union, all elements in the other set are attached to the accumulator set.

```
vtk_set_operation $accId $setId INTERSECTION  
vtk_set_operation $accId $setId UNION
```

In the `vtk_set_operation` calls, the first argument is always the VovId of the accumulator set.

Complement of a set

This operation also requires two sets. The result, stored into the accumulator set, is effectively the intersection of the accumulator with the complement of the other set.

```
vtk_set_operation $accId $setId COMPLEMENT
```

Input set, Output set and cones

These operations are similar to union and intersection. Upon entry, the accumulator set, which collects the result, should be an empty set.

```
vtk_set_operation $accId $setId INPUTS  
vtk_set_operation $accId $setId OUTPUTS  
vtk_set_operation $accId $setId UPCONE  
vtk_set_operation $accId $setId DOWNCONE
```

Subselection

Given a set, you can subselect the elements from the set that satisfy a selection rule using `vtk_set_subselect`. The result is a new set. For the original set, you can use either the name or the VovId.

```
set newSetId [vtk_set_subselect $nameOrIdOfSet $rule]
```

By default, the name of the new set is computed by `vtk_set_subselect` using the name of the original set and the selection rule in the following form:

```
SUBSELECT(<setName>,<Rule>)
```

You can specify the name of the new set with the option `-setname`:

```
set newId [vtk_set_subselect -name "MyNewSet" $setId $rule]
```

If the original set has subsets and you want to collect elements from subsets as well, use `-hier` option.

```
set newId [vtk_set_subselect -hier -name "MyNewSet" $setId $rule]
```

Subselection generates a large number of detach/attach events, and this may overflow the event queues for the GUI's connected to the server. In those cases in which the subselected set is

immediately forgotten, the attach/detach events are of no interest. It is possible to inhibit the generation of these events by declaring the set as transient with the option `-transient`. Example:

```
set newId [vtk_set_subselect -transient $setId $rule]  
doSomething $newId  
vtk_set_forget $newId
```

Tool Integration

Tool integration is a key ingredient of runtime tracing. It allows the tool to communicate at runtime with the FlowTracer server in order to describe the list of its inputs and its outputs.

There are several integration techniques, differing in terms of the effort and accuracy.

Regardless of the technique, the end result is always the same: enable the integrated tool to communicate its I/O behavior to FlowTracer. A good tool integration is the key for an optimal flow development.

The following table compares the available techniques:

Technique	Features	Effort
Interception	Non-intrusive, system level operation (library based). Available on a limited number of platforms. Often detects more files than the users care for.	Requires no effort on the part of the end user or administrator.
Instrumentation	The best option for those who have access to a tool's source code. Maximum accuracy and low overhead. Requires some knowledge of the behavior of the tool. Superior to encapsulation.	Requires access to tool's source. Most commonly used with scripts rather than with binaries.
Encapsulation	Powerful because it does not require any modification of the tool. Also called black-box instrumentation.	Requires the writing of a separate encapsulation script, one for each tool. The effort depends on tool. As little as a few minutes, typically a few hours per tool.
Encapsulation On-the-Fly	Conceptually the same as encapsulation, except that the capsule code is embedded into the Flow description file, instead of being in a separate file.	Useful for commands that defy encapsulation, and for simple input/output relationships. Attractive because it keeps all flow information in a single place.

Practical Considerations for Tool Integration

- It is possible to build useful flows **without any integration**. Refer to [Task Oriented Flows](#) section for more information.

- Interception with `vrt` is very powerful as a discovery tool, but it is **somewhat risky** in production flows due to the deep level of interaction with the tool. Interception reveals a lot of dependencies, many of which, although true, may be uninteresting in most cases, such as the dependencies with all files in the Perl or Python libraries. This then requires a cleanup using the `exclude.tcl` file mechanism.
- A simple **encapsulation**, meaning one that lists a few of the obvious inputs and outputs, can go a long way. You can quickly arrive at a useful flow, and later you can go back and improve the accuracy of the dependencies by either enabling interception or by enhancing the capsule.


Also in this Section

Wrappers

The choice of the wrapper depends on the integration method used for each of the tools. These methods are three: encapsulation, instrumentation, interception. The following table summarizes the ways to invoke the wrappers:

Technique	Wrapper	Description
Interception	<code>vrt</code>	Use this to also activate interception of the operating system calls. The wrapper also honors encapsulation and instrumentation. The interception is available only on Linux.
Instrumentation	<code>vov</code>	Use this wrapper for tools that have been instrumented . The wrapper also honors encapsulation, if available.
Encapsulation	<code>vw</code>	Use this wrapper for tools that have been encapsulated . If no capsule is available, then the only input is the "executable" and the only output is the stdout and stderr, or any redirection thereof. This wrapper is also used in Accelerator for batch jobs.

The binaries `vovbuild`, `clevercopy`, and `cleverrename` don't need to be run with a wrapper usually as they are fully instrumented. In the rare case of slow NFS servers, a `vov` wrapper may be needed.

 **Note:** There is only one actual binary which all wrappers invoke to activate runtime tracing, and it is called `vw2`. Through symbolic links (on UNIX) or copies, the binary can be called with several different names. The behavior of the wrapper is controlled either by the name used to invoke the binary or by the command line options `-C`, `-V`, `-R` and `-T`.

Track the Origins of Dependencies

The dependency engine tracks the origin of dependencies. This can be useful when debugging complex flows. The origin of a dependency can be one of the following:

1. FDL: this is a dependency described using the Flow Description Language FDL
2. Capsule: this is a dependency described in a capsule
3. Interception: this is a dependency captured by intercepting the operating system calls
4. Instrumentation: this is a dependency captured by calling the VIL or the VILtools
5. Wrapper: this is a dependency related to the use of a wrapper
6. Extra: another type of dependency

The origin of a dependency is shown as a set of 7 flags, with the following meaning:

Flag	Description
f-----	FDL
-c-----	Capsule
--r-----	Interception (most often seen as --rv-----)
---v-----	Instrumentation (VIL or VilTools)
----w-----	Wrapper (vw, vov, vrt, ...)
-----x---	Extra dependency
-----s--	Sticky flag (normally coming from FDL)
-----o-	Old dependency (orange)
-----n	New dependency (yellow)
-----F	File ready flag
-----W	Wait for input flag

Strictly speaking, the flag 's' for 'sticky' is not really an origin, but is nevertheless an important attribute of a dependency.

The origin of dependencies is now part of the default output of `vsx`, where the default format (see option -O), includes the keyword "@ORIGIN@"

Any dependency can have multiple origins, for example from FDL and from a capsule. For example, let's start with one line of FDL:

```
% vovsh -x 'J vw cp aa bb'
% vsx !
```

```
000001847 VALID      f----- aa
>>>> Node 000076523 INVALID vw cp aa bb
000001848 INVALID    f----- bb
```

In this example, both dependencies for `aa` and `bb` are `f-----` because they are coming from the one line of FDL. After execution of the tool, things change:

```
% vw cp aa bb
% vsx !
000001847 VALID      fc-----on-- aa
000001851 VALID      ----w---n-- /bin/cp
>>>> Node 000076524 VALID  vw cp aa bb
000001848 VALID      fc-----on-- bb
```

Now, the origin for the `aa` dependency is `fc-----` meaning that it is coming from both the FDL and from the capsule for the tool "cp". The new dependency for the file `/bin/cp` is coming from the wrapper `vw` which always declares the executable as an input.

Integration by Interception

This section discusses the integration technique called interception, which captures the most complete IO behavior of your programs. This gives the most correct dependency graph for the flow, but in some cases this may be more detail than is wanted.

By interception, we mean that FlowTracer interposes a thin layer between your tool or program and the operating system itself, so that whenever the tool uses a file, that IO behavior is reported to FlowTracer. There are two kinds of interception available, library-based interception and OS-based interception.

Library-based Interception

Most modern operating systems support the concept of dynamic-link libraries. Programs using dynamic-link libraries do not contain all the code necessary to execute, but have external references which are satisfied by the dynamic linker at the time the program is launched. There are many benefits to dynamically-linked programs, including smaller executable sizes, reduced memory consumption due to shared code, and easier maintenance.

When using library-based interception, FlowTracer installs its IO library before the system standard IO library. The FlowTracer library picks off the names and opening modes of the files your tool uses, then passes the call on to the regular system library. This way, your program does not need to be relinked or otherwise modified, and it behaves the same as when not using FlowTracer. More [platform-specific information](#) about library-based interception is available. Library-based interception is activated by using the FlowTracer `vrt` wrapper program in front of tool command lines.

Library-based Interception

This method is a technique where the system I/O library is replaced by a modified version that has been instrumented with FlowTracer. Without changing the source code of the tools, this technique is in fact a form of low-level instrumentation. The routines in the modified I/O library behave exactly as those in the original system library, except that a few additional calls to the VIL functions are made to declare the I/O behavior of the tool.

Because the runtime interception library is sensitive to all file I/O, it is most important to have a well-defined [exclusion file](#) to eliminate uninteresting files from the resulting trace. No other preparations are necessary.

To activate runtime interception, use the wrapper `vrt` or `vw2`. The command line options to `vrt` are similar to those of `vw`, and the method of invocation is the same:

```
% vrt cc -o prog src1.o src2.o src3.o
```

This technique is available for Linux.

Requirements for Linux

`vrt` only applies to executables linked with the dynamic C library. To find out if an executable qualifies, you can use the `utilities` file and `ldd` as in the following example:

```
% file /bin/ls
/bin/ls: ELF 32-bit MSB executable SPARC Version 1, dynamically linked, stripped
```

Integration by Instrumentation

In instrumentation, the tool source code is modified to include some calls to the [VOV Instrumentation Library \(VIL\)](#). This library is also accessible via the [VILtools](#) library, which is a collection of utilities that can be embedded into scripts and batch files.

The instrumentation library provides runtime traceability with minimum change to the source code and with no change to the tool behavior. The instrumentation library is small (about 13KB), fast, and, if FlowTracer is not running, completely passive.

The VIL library is provided also in source code.

The code modifications take only a few minutes of file editing.

```
/* Example of an instrumented "Hello World" program. */
1.  #include <stdio.h>
2.  #include "vil.h"
3.
4.  int main( argc, argv )
5.      int  argc;
6.      char* argv[];
7.  {
8.      int  i, n = 20;
```

```
9.      FILE* fp;
10.
11.      if ( argc > 1 ) {
12.          n = atoi( argv[1] );
13.      }
14.      VovBegin( argc, argv );
15.      VovExecutable( argv[0] );
16.      fp = VovFopen( "Hello", "w" );
17.      for ( i = 1 ; i <= n; i++ ) {
18.          fprintf( fp, "%2d - HELLO WORLD\n", i );
19.      }
20.      fclose( fp );
21.      VovEnd( 0 );
22.  }
```

VOV Instrumentation Library (VIL)

VIL: The Basic Procedures

The essential functions in the VOV C library, together with the few common utilities, allow for the easy instrumentation of tools. VIL can be used with either C or C++ programs.

Type	Procedure	Args	Description
void	VovBegin	(int argc, char** argv)	Starts a channel and opens a connection to it. We recommend the use of VovConnect () instead of this procedure.
void	VovConnect	(void)	Connects to an existing channel, if available, that is if the current process is a child of a wrapper that activated runtime tracing, such as vrt or vov.
int	VovBarrier	(char* db, char* file, char* action)	Declares an output barrier. The third argument action can be either "STOP" or "PROPAGATE". Used by "smart tools" to inform the server whether there are changes propagating to the output. The argument ACTION is either

Type	Procedure	Args	Description
			PROPAGATE or STOP. For more information, see Runtime Change Propagation Control .
VovStatus	VovExecutable	(char* exec)	Declares an input taking advantage of the PATH search mechanism. Equivalent to <code>VovInput("FILE", true_exec)</code> , where <code>true_exec</code> is the full path of the executable <code>exec</code> as produced by the PATH search mechanism. If <code>true_exec</code> cannot be found, an error condition is raised, causing termination of the run.
FILE*	VovFopen	(char* fname, char* mode)	Replacement for <code>fopen()</code> .
VovStatus	VovInput	(char* db, char* name)	Generic input declaration.
VovStatus	VovInputFile	(char* name)	Declares an input file.
VovStatus	VovOutput	(char* db, char* name)	Generic output declaration.
VovStatus	VovOutputFile	(char* name)	Declares an output file.
void	VovFinish	(int status)	Closes connection with channel.
void	VovEnd	(int status)	Closes connection and calls <code>exit()</code> . Use this only if in conjunction with <code>VovBegin()</code> .
VovStatus	VovInputWithFlags	(const char* db, const char* name, int flags)	Offers the most complete interface to declare an input in an instrumented tool. The flags argument is the

Type	Procedure	Args	Description
			<p>OR'ing of the following flags defined in vil.h:</p> <pre>VID_CONSUMED_FLAG VID_NORMAL_FLAGS</pre>
VovStatus	VovOutputWithFlags	(const char* db, const char* name, int flags)	<p>The procedure VovOutputWithFlags (db, name, flags) offers the most complete interface to declare an output in an instrumented tool. The flags argument is the OR'ing of the following flags defined in vil.h:</p> <pre>VOD_NORMAL_FLAGS VOD_FORCE_FLAG VOD_OPTIONAL_FLAG VOD_SHARE_FLAG VOD_IGNORE_TIMESTAMP_FLAG VOD_BARRIER_STOP_FLAG VOD_BARRIER_PROPAGATE_FLAG</pre>

Instrumentation Procedure

1. Find `main()` and add either `VovBegin(argc, argv);` or the call `VovConnect();`
2. Call `VovBegin(...)` before any other VOV procedures and before doing any I/Os.
If FlowTracer is not running, `VovBegin(...)` has no effect. If FlowTracer is running, `VovBegin(...)` initiates a connection with the FlowTracer server, or inherits the connection from the parent process.
`VovConnect()` is similar to `VovBegin()` but it does not initiate any connection with the vovserver. It can only inherit a connection from the parent process.
3. Find where your program terminates. This is normally, but not necessarily, at the end of `main()`. The correct way to terminate a program is either to call `exit(status)` or to return from `main()` with return status.
4. For most UNIX tools, a status equal to 0 normally means that the tool has terminated correctly. Now replace `exit(status)` with `VovEnd(status)`.
In principle, you can ignore any `exit()` called by error trapping routines, because if a tool terminates without calling `VovEnd()`, it is assumed to have failed. In practice, it is better to ensure that every `exit()` is replaced by a `VovEnd()`. If FlowTracer is not running, `VovEnd()` simply calls `exit()`.

If `VovEnd()` is called by the tool that opened the connection with the server, the connection is closed.

5. Find all file I/Os. Use `VovFopen()` to replace all calls to `fopen()`.

If you use other methods to open or create a file, such as `freopen()` or `rename()`, you must call explicitly either:

```
VovInput(char* db, char* name)
```

or

```
VovOutput(char* db, char* name)
```

If the database in these calls is "FILE", you can also call `VovInputFile(char* name)` or `VovOutputFile(char* name)`.

For example, the call to `VovFopen()` in line 16 can be replaced with a call to `VovOutputFile("Hello")`, followed by `fopen()`. If FlowTracer is not running, `VovFopen()` is equivalent to `fopen()`, while `VovInput()` and `VovOutput()` do nothing.

Ignoring the return status from `VovInput()` and `VovOutput()` does not change the behavior of your tool. (See the include file `$VOVDIR/include/vil.h` for more information on the return status of these procedures.)

6. `VovInput()` and `VovOutput()` require two arguments: the name of a database and the name of an object in that database. In most cases, the database is "FILE", and the object name is the file name. This can be a full path starting with `/`, or a relative path. Tilde (`~`) and environment variables are expanded as by the shell.

`VovFopen()` can replace `fopen()` in most cases, except where `fopen()` is used to test the existence of a file, or where temporary files are being created. The use of `fopen()` to test for file existence is a misuse of the function. `stat()` is more appropriate and more efficient.

7. Include the file `vil.h` in all sources that you have modified. Then recompile and relink the tool. To complete the linking, you need the `libvil.a`, a copy of which can be found in `$VOVDIR/lib/libvil.a`.



Note: Altair does not furnish a library for Python tools. However, Python's `os.*` and `subprocess.*` commands can execute other programs, including `VovInput` and `VovOutput`. These are links to the `viltool` library.

VILtools

Another technique to instrument tools is based on a set of command line utilities called **VILtools** which stands for Vov Integration Library Tools. The VILtools, shown in table below, implement the same functions as VIL but are intended for use in shell scripts.

If the environment variable `VOV_FDL_ONLY` is set, then the VILtools generate [FDL code](#).

The VILtools are passive if VOV is not running or if runtime tracing has not been activated by means of one of the wrappers `vov`, `vrt`, or `vw2`.

To instrument a script:

- Add the appropriate calls to `VovInput` and `VovOutput` to declare the I/O behavior of the script. (To be safe, call these tools before you read or write a file and always check the exit status.) Temporary and intermediate files that are removed upon successful execution need not be declared.
- Execute the integrated script within the wrapper `vov` as follows:

```
% vov myscript arg1 arg2 arg3
```

VovInput

Usage

`VovInput [options] <FILE FILE ...>`

Description

Declare the list of files as inputs. It returns 0 if the files are VALID, or 4 if there are input conflicts.

Options

-sticky	Specify that the input is an input, even if it's not actually a true input at runtime.
-consumed	Specify that the input disappears if the tool completes successfully (e.g. the tool gzip consumes its input).
-db <name>	Specifies the database of the input, which defaults to "FILE"
-links	Declare as inputs all symbolic links used to canonicalize the name of file.
-show_barrier	Prints on stdout the name of the files on which there is an input barrier.
-quote	Use the input name as given, instead of mapping it to the corresponding logical canonical name.

VovOutput

Usage

`VovOutput [options] <FILE FILE ...>`

Description

Declare the list of files as outputs. It returns 5 if there are output conflicts and 0 otherwise.

Options

-sticky	Specify that the output is an output, even if it's not actually a true output at runtime.
----------------	---

-ignore_timestamp	Specifies that the timestamp of the file should not be used to determine whether the job has succeeded or failed.
-db <name>	Specifies the database of the output, which defaults to "FILE"
-shared	State the output is the shared by many tools. FlowTracer ignores the timestamp of the shared outputs and serializes the execution of the jobs that contribute to the shared output. In this case, all jobs must use declare the file as shared.
-stop_barrier	Specify there is a barrier on the output, where the barrier is effective in stopping change propagation on and output. See Runtime Change Propagation Control for more details.
-propagate_barrier	Specify there is a barrier on the output, where the barrier is deemed to be ineffective in stopping change propagation on this output. See Runtime Change Propagation Control for more details.
-optional	Specify that the output file may or may not exist.
-links	Declare as input (notice: 'input' and not 'output'!) all symbolic links used to canonicalize the name of file.
-quote	Use the output name as given, instead of mapping it to the corresponding logical canonical name.

VovExecutable

Usage

```
VovExecutable <FILE FILE ...>
```

Description

Declares an executable as an input. This is equivalent to VovInput except that the argument is assumed to be an executable file and is searched for using the PATH variable.

Options

none

VovDelete

Usage	VovDelete <FILE FILE ...>
Description	Declares an executable as an input. This is equivalent to VovInput except that the argument is assumed to be an executable file and is searched for using the PATH variable.
Options	none

VovToolId

Usage	VovToolId -get
Description	Returns the VovId of the current transition, otherwise the string '00000000' is returned if tracing is not active.
Options	none

Integration by Encapsulation

To encapsulate a tool:

- Write an encapsulation script based on the [Encapsulation Procedures](#)
- Invoke the tool with the FlowTracer [Wrappers](#) `vw` or with `vw2`.

For example, if your design flow includes the invocation of `yacc` as in:

```
% yacc -d -t cc.y
```

You would activate tracing by prefixing the above command with `vw` as in:

```
% vw yacc -d -t cc.y
```

The encapsulation script is a Tcl script whose name is derived from the name of the encapsulated tool by prepending `vov_` and appending `.tcl`. For example, the encapsulation script for the tool `yacc` is called `vov_yacc.tcl` and looks like this:

```
set FILEPREFIX "y"
while { [arglength] > 1 } {
    set arg [shift]
    switch -- $arg {
        "-d" { VovOutput "$FILEPREFIX.tab.h" }
        "-t" {}
        "-b" { set FILEPREFIX [shift] }
        default {
            VovFatalError "Unknown option $arg"
        }
    }
}
VovInput [shift]
```

```
VovOutput "$FILEPREFIX.tab.c"
```

For compatibility with Windows NT, in calculating the "name of a tool" we drop the suffixes `.bat.exe` and `.cmd`. Therefore, if the tool is called `mytool.exe`, its capsule is called `vov_mytool.tcl`.

The FlowTracer wrapper `vw` looks for the encapsulation script in the following directories:

- In the property "CAPSULE" attached to the job (see [Capsule On-the-Fly](#))
- In the current working directory
- In the directories indicated by the environment variable `VOV_CAPSULE_DIR`, if it exists; the directory names are separated by a colon on UNIX and by a semicolon on Windows NT
- In the directory `$VOVDIR/local/capsules`
- In the directory `$VOVDIR/tcl/vtcl/capsules`

This search order for the capsules facilitates the development of new capsules and the management of a set of site specific capsules.

Simple Capsules

Encapsulation is a simple and effective method to integrate any tool. While the encapsulation of some tools may be extremely complex, the majority of the capsules consists of very few lines of Tcl code. For example, say we are interested in encapsulating a tool called `simulator` that has the following command line:

```
simulator INPUT_FILE OUTPUT_FILE
```

The capsule for such tool may look like this:

```
# This is vov_simulator.tcl
VovInput [shift]
VovOutput [shift]
```

This is a good starting point. You can build interesting flows using the tool `simulator` and the simple capsule shown above. As you become more familiar with FlowTracer, you will learn how to increase the accuracy of the encapsulation, by looking at real capsules for real tools.

Capsule Template

When you need a new capsule, start from the standard template, which can be found in `$VOVDIR/tcl/vtcl/capsules/vov_capsule_template.tcl`

The template illustrates all the procedures that may be useful to write capsules.

Encapsulation Procedures

In an encapsulation script, you can use the following Tcl procedures:

VovInput

Description	Declares an input of a job
-------------	----------------------------

Usage	VovInput [options] <FILE>	
Aliases	I	
Options	-sticky	Specify that the input is an input, even if it's not actually a true input at runtime.
	-consumed	Specify that the input disappears if the tool completes successfully (e.g. the tool gzip consumes its input).
	-db <name>	Specifies the database of the input, which defaults to "FILE"
	-links	Declare as inputs all symbolic links used to canonicalize the name of file.
	-quote	Use the input name as given, instead of mapping it to the corresponding logical canonical name.
	-normal	Use default flag setting.

VovOutput

Description	Declares an output of a job	
Usage	VovOutput [options] <FILE>	
Aliases	O	
Options	-sticky	Specify that the output is an output, even if it's not actually a true output at runtime.
	-ignore_timestamp	Specifies that the timestamp of the file should not be used to determine whether the job has succeeded or failed.
	-db <name>	Specifies the database of the input, which defaults to "FILE"
	-shared	State the output is the shared by many tools. FlowTracer ignores the timestamp of the shared outputs and serializes the execution of the jobs that contribute to the shared output.

-stop_barrier	Specify there is a barrier on the output, where the barrier is effective in stopping change propagation on and output.
-propagate_barrier	Specify there is a barrier on the output, where the barrier is deemed to be ineffective in stopping change propagation on this output.
-optional	Specify that the output file may or may not exist upon completion of the job. If the file does not exist, it is removed from the list of outputs. If the file does exist, it must obey the rules of all outputs, i.e. have a valid timestamp relative to the lifespan of the job. If an older file already exists and it is not an output of the job, the job will fail for having an output with a bad timestamp, i.e. older than the job start time. If this can happen in your flow, you may want to add the option <code>-ignore_timestamp</code> together with <code>-optional</code> .
-links	Declare as input (notice: input and not output!) all symbolic links used to canonicalize the name of file.
-normal	Use default flag setting.

VovExecutable

Description	Declares an executable as an input. This is equivalent to VovInput except that the argument is assumed to be an executable file and is searched for using the PATH variable.
Usage	<code>VovExecutable <FILE></code>
Aliases	None
Options	None

VovStdoutCtrl

Description	Controls the disposition of both stdout and stderr. By default, everything printed by the tool to stdout or to stderr is saved into a file. The name of the file normally depends on the
--------------------	--

command line and on the value of the environment variable VOV_STDOUT_SPEC.

Usage

VovStdoutCtrl [options]

Aliases

I

Options

- normal** Save stdout and stderr if they are non empty.
- ignore** Stdout and stderr are ignored (they are not saved).
- forget_if_success** If the tool competes successfully, stdout is deleted and forgotten even if it is not empty; stderr, if any, is always saved.
- spec "string"** Change the string used to generate the name of the files used to store stderr and stdout. This string may contain the substrings @OUT@ and @UNIQUE@ and @ID@, which are appropriately replaced by FlowTracer.

Examples

```
# This is constant across retraces.
VovStdoutCtrl -spec savestd/@OUT@-@UNIQUE@.txt

# This changes with every invocation of the job.
VovStdoutCtrl -spec savestd/@OUT@-@ID@.txt
```

VovCorrectExitStatus

Description

Specifies which exit status values signify correct behavior. By default, only the value 0 (zero) is acceptable as the indicator of successful tool termination. For certain tools, other non-zero values may also indicate successful termination. The string is a space-separated list of positive integers and ranges of integers. A range of integers is expressed with a dash separating the lower limit from the upper limit.

Usage

VovCorrectExitStatus "<space-delimited-string-of-exit-value>"

Aliases

None

Options

None

Examples

```
VovCorrectExitStatus "0-5"
```

```
VovCorrectExitStatus "0 10-15 77"
```

Capsule Post-Processing

Sometimes, not all inputs and outputs of a job can be computed by the capsule at start time. Therefore, you may need to do some processing after the tool has finished. Since the capsule is evaluated before the invocation of the tool, FlowTracer provides a hook for you to perform post-processing.

If you define the Tcl procedure `VovCapsulePostProcessing`, it is called immediately after the encapsulated tool has completed.

This procedure is called with a single argument `jobId`, which is the `VovId` of the current job. When this procedure is invoked, the job is still in the `RUNNING` status. All Tcl and vtk procedures are available in the context of this procedure.

Example:

```
##
## This procedure must be defined in the capsule code,## i.e. in the file
##   vov_TOOLNAME.tcl
##

proc VovCapsulePostProcess { jobId } {
  # -- Declare extra outputs for the current transition.
  set outputs [glob -nocomplain *.V??]
  foreach out $outputs {
    set placeId [vtk_place_get_or_create FILE $out]
    vtk_output_declare $jobId $placeId
  }
}
```

Capsule On-the-Fly

If writing a separate capsule for each script seems to be too much work, we offer another way to embed the encapsulation script directly into the flow description, i.e. inside of the FDL. We call this method "capsules on-the-fly" and is based on the FDL procedure "CAPSULE" to be invoked after the `T` or the `J`.

The CAPSULE procedure takes only one argument, the encapsulation script. This script is:

- evaluated at build time
- attached to the job as a property
- evaluated at runtime by the `vw` wrapper

Example:

```
T vw ./run.job $block $step
CAPSULE {
  I -links $block.v
```

```
    O $block.$step.out.gz  
}
```

If a capsule "on-the-fly" is found, all other capsules that could apply to the job are NOT evaluated.

Recommendations on using Capsules on-the-fly

- Each capsule on-the-fly is evaluated by `vovbuild` by the same Tcl interpreter, meaning that there can be cross interference between the capsules. This means that it is important to keep the capsules **simple** and to try to avoid setting variables without cleaning them up.
- Avoid changing the values of global arrays like `env` or `make`.
- The command line is available in the variable `$argv`, without the leading wrapper `vw`. The variable `$argv0` is set to "capsule_on_the_fly".

Capsules for VHDL and Verilog Tools

Capsules for VHDL Tools

A useful tool for VHDL capsules is `vhdlpp`. This is based on a proprietary VHDL parser. For more details on how to use this tool, see the capsule `$VOVDIR/eda/Synopsys/vov_vhdlan.tcl`

Capsules for Verilog Tools

A package of procedures to deal with Verilog-based tools is found in `$VOVDIR/eda/Verilog/vovverilog.tcl`.

In particular:

- `vv_find_libraries` helps parse the `vsystem.ini` files
- `vv_find_modules` finds all module definitions
- `vv_do_instances` processes all instances

A generic capsule for a verilog simulator is `$VOVDIR/eda/Verilog/vov_generic_verilog.tcl`.

For example, if you need the capsule for the Verilog simulator `silos` by `Simucad`, you can get it by creating a link with the generic verilog capsule, as in:

```
% cd $VOVDIR/eda/Verilog  
% ln -s vov_generic_verilog.tcl vov_silos.tcl
```

Integrate Difficult Tools

Tools that Run in Place

It may happen that a file is declared **both as an input and as an output**.

The **order** in which the declarations occur is of great importance, because it determines the structure of the resulting dependency graph.

If a file is declared **first as an output**, all subsequent input declarations for the same file are ignored, the logic being that a job can do anything it wants with its own outputs, including reading and writing them multiple times.

If a file is declared **first as an input** and later as an output, FlowTracer assumes that the tool modifies its input. This adds some complication to the management of the flow, because now we have two places with the same name, one to represent the input and one to represent the output of the tool. If a tool modifies one or more of its inputs, we say that the tool **runs in place**.

The two places that identify the file that is being modified are said to be in a **chain of places**. A minimum chain contains 2 places, but if a file is modified in place by many tools the chain for that file can get arbitrarily long. Chains of places are managed automatically by FlowTracer and most of the times you need not worry about them.

Start a Subflow Before a Job is Completely Done with **vovfileready**

There are jobs that run for a long time and produce several outputs along the way. If it is known that one of the outputs is "ready to be used", it would be nice to be able to process that output before the current job is complete. This violates the normal rule that an output of a job becomes VALID only when the job that creates it completes successfully.

The utility **vovfileready** can be used during the job runtime to make the output available for immediate processing. This utility changes the dependency tree so that the output becomes the VALID output of a new special job, and therefore is immediately usable.

vovfileready

If you want to use this utility **vovfileready** you need to make sure that you first call **vovfileready -clean** before any other call to **vovfileready <output_name>**. See example in the usage message below.

```
vovfileready: Usage Message

USAGE:
  % vovfileready <NAME_OF_FILE_THAT_IS_READY> ...

OLD USAGE: the following options are now ignored:
  % vovfileready -start
```

```
% vovfileready -clean
```

This script can only be executed as part of a running job.
From within the running job, when you know that one or more
output files are ready to be used, you can call

```
vovfileready NAME_OF_FILE_THAT_IS_READY ...
```

and this utility will mark all the files specified
on the command line as VALID.

EXAMPLE: PSEUDO-CODE

```
-- #!/bin/sh -f
-- # This could be a script called ./my_tool
-- DoSomeInitialization
-- DoSomeWorkFor5Minutes
-- vovfileready my_first_output ; ### The flow can start processing
--                               ### my_first_output even
--                               ### if ./my_tool is not quite done.
--
-- DoSomeOtherWorkFor1Hour
-- vovfileready another_output and_another_one
-- MoreWorkToFinish
-- exit 0
```

Then invoke the script with a wrapper, e.g.
% vov ./my_tool arg1 arg2 ...

Aggressive Retrace is Required

Due to the use of barriers in the current implementation of `vovfileready`, aggressive retrace is required for it to work. This disrupts the normal use of barriers, so flows that already use barriers will not work as intended if aggressive retrace is always enabled.

Aggressive retrace can be enabled in FlowTracer using several different methods. For a FlowTracer console that is already open, choose the **aggressive** option in the **Retrace Priority & Flags** dialog. When retracing from the command line, use `vsr -aggressive`. To make the setting apply to every FlowTracer console for a particular project, add the following to `$VOV_SERVER_DIR/gui.tcl`:

```
set ::VovGUI::retrace(mode) "aggressive"
```

Example

```
#!/bin/csh -f

# Clean vovfileready from previous run(s)
vovfileready -clean

#
# Do some stuff
#

foreach extraction_corner $extraction_corners {
    #
    # Complete extraction for this corner; could take a while
    #
}
```

```
# When files are complete, call vovfileready on them
vovfileready data/$extraction_corner.spef.gz
}

#
# Do more stuff
#

# Full job completes, potentially much later
exit 0
```

Runtime Change Propagation Control

The patented technique of Runtime Change Propagation Control (RCPC) provides the ability to **stop the propagation of insignificant changes** through the flow.

Motivation

A dependency management system based exclusively on timestamps, such as `make`, assumes that any change to the timestamp of a file is a change that needs to be propagated, even if the information contained in the file has not changed significantly or has not changed at all, as happens when you use the program `touch` on a file.

There are changes that do not need to be propagated to all dependent files. How can these changes be recognized? How can the change propagation safely be stopped? These are the issues addressed by the Runtime Change Propagation Control (RCPC) facilities supported by FlowTracer.

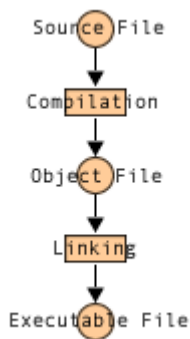


Figure 8:

In software development the comments in the source code do not affect the behavior of the executable code. If we have a consistent (completed) build process and we change a comment in a source file, we do not really need to recompile the code or to repeat the testing. However, if we are using a dependency system that is based exclusively on timestamps, the build will be considered inconsistent, with out-of-date and untested executables, until both compilation and testing are run again. This approach wastes compute resources when source code changes are only in the comments.

This example demonstrates that a dependency graph does not prevent unnecessary work from being performed. However, with the addition of run time control of change propagation, the extra work can be avoided.

Runtime Control of Change Propagation

A better system than depending on timestamps is to introduce a step that analyzes the change and tells the build system if the change should be blocked in the dependency graph. When change propagation is blocked, the system can suppress running the dependent steps.

The model of FlowTracer's Runtime Change Propagation Control is that an executable program used in the job sequence is assigned the responsibility to decide the change propagation case for a given file and to notify FlowTracer about it during the run. This program is customized to be the single agent that has that responsibility because it has specific knowledge about the meaning of the data.

Notification of FlowTracer can be done directly by including calls in the checking program to the appropriate procedures in the VIL library. Another way is to use a shell script and command line calls to helper programs that notify FlowTracer about the change propagation status.

In the software build scenario above, consider the introduction of a program into the job sequence and dependency graph that can check on changes to the source file, and then block the propagation of the change when the changes to the source file are comments only, or to allow the change to propagate when the changes are significant.

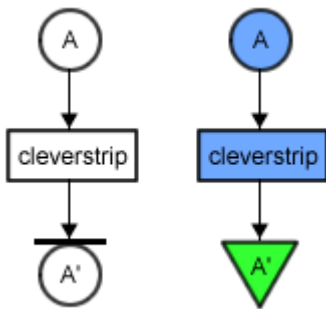


Figure 9:

Imagine this program is called `cleverstrip`. It can reduce the new version of the source file to a minimized form which is stripped of all the comments, and can compare it to the minimized form of the previous version to see if changes were only to comments. If the minimized versions are the same, then the change was only to comments and change does not need to propagate. If the minimized versions differ, then the change was to more than comments and the change should propagate.

This approach is to create a derived file from the file that is to be analyzed. In the diagram of this situation, the source file to consider is file A. The derived source file is A'.

The analysis of the file creates a decision point in the dependency graph which is called a **"barrier"**. This part of the graph is the dark segment shown at the tip of the output arrow of `cleverstrip`. It marks that output file as a *"barrier file"*. The checking program is responsible for notifying FlowTracer on the state of that *"barrier file"*. Is it open, allowing the change to propagate, or is it closed to stop the change from propagating?

The original diagram changes with the introduction of this decision point. The logical steps change from the simple two step segment of "Source File" to "Compilation" into a four step segment of "Source File" to "`cleverstrip`" to "Source File Prime" to "Compilation".

If `cleverstrip` is fast and efficient relative to the compilation and testing steps, computation time is saved while still maintaining integrity in the compiled and tested release products.



Note: A graphical view of the dependency graph in the FlowTracer GUI console displays a *"barrier file"* as a hexagon instead of a circle.

Introducing runtime change propagation logic into a job sequence is a useful way to save compute resources. Using a derived file as the *"barrier file"* is a straight-forward way to implement the logic. Another way is to have the basic file become the barrier file. This is done by changing the job that creates that basic file to contain the runtime change propagation control logic.

Also in this Section

Triggers on Files

There are two possible triggers that can be set that depend on a file's timestamp:

- **Run Trigger:** when this trigger is set, and the timestamp of the file changes, the downcone of the file is automatically run.
- **Stop Trigger:** when this trigger is set, and the timestamp of the file changes, all running jobs in the downcone of the file are automatically stopped.

If both triggers are set, first the jobs are stopped, then the downcone is scheduled. The run is always performed at NORMAL priority.

Set the Triggers

The simplest way to set triggers in FDL is to use the options `-trigger,run` and `-trigger,stop` in the `I` call that registers the file as input.

```
# Example of setting a trigger flag in FDL
J vw cp aa bb
I -trigger,run aa
```

Other Ways to Change the Triggers

Triggers on a file can be set using the vtk API.

```
# Example of manipulation of both trigger flags using the VTK API.
set placeId [vtk_place_find "FILE" "aa"]
vtk_place_get $placeId placeInfo

set placeInfo(trigger,run) 1
set placeInfo(trigger,stop) 1

vtk_place_set $placeId placeInfo
```

Barriers to Change Propagation

There are two types of barriers supported by FlowTracer: output barriers and input barriers.

Barrier Type	Computed by:	Used by	Comments
Output barrier	The clever tool, at runtime	vovserver to control change propagation to outputs of job	Has to be explicitly declared in flow (<code>-md5</code> option) or by the clever

Barrier Type	Computed by:	Used by	Comments
			tool. Commonly used barrier.
Input barrier	vovserver, upon each input declaration	the tool	Always computed by vovserver. Rarely used in the tools.

Output Barriers

Output barriers are the most common type. They are implemented by the tools and used by the `vovserver`. Output barriers are associated with the outputs of a clever tool, which is represented by a hexagon rather than by a circle. In some representations, the output barrier is also shown as a dark horizontal line in a tool's output arcs.

Some tools can determine if one or more of their own outputs are significantly affected in a design run. If a clever tool computes that its output exists before the execution of the tool, and the current execution does not significantly modify that output, the tool informs the FlowTracer server. Thus, the tool is responsible for stopping the propagation of changes to one or more of its own outputs. FlowTracer recognizes this by adding a barrier to those outputs.

When an input of a clever tool is either modified or invalid, the outputs with a barrier do not become invalid. Only when the clever tool is executed again can the tool stop or propagate changes to its outputs.

The simplest way to add an output barrier to a flow is to use the `-md5` flag in the 'O' (output) procedure in [FDL](#). This is simpler than inserting a 'clevercopy' job in the middle of the flow.

```
# Example of output barrier.
T vw cp aa bb
O -md5 bb
```

The effect of the option `-md5` is to add a property called MD5BARRIER to the output file 'bb'. If the property is present, then the wrapper (in this case `vw`) computes the MD5 sum upon completion of the job. If the MD5 sum does not change then the barrier is active and change propagation is stopped by the barrier. If the MD5 sum changes, the change propagates through the barrier and the new value of the MD5 sum is stored in the property.

Input Barriers

Input barriers are implemented by `vovserver` and used by the tools. Input barriers exist on all input arcs of each tool. In other words, the input barrier flag is one of the results of calling `VovInput` to declare an input dependency.

Whenever a tool declares an input, the FlowTracer server informs the tool whether the file has changed since the last successful invocation; if no change has occurred, then there is an input barrier on that input. Whether such type of information can save computation time, depends completely the tool.

Here you can see an example of how a script can use input barriers. The first code fragment shows a simple script without barriers, where a list of input files is being processed:

```
## Example without input barriers.
foreach input ( $listOfInputs )
  VovInput $input || exit 1
  doSomeProcessing $input
end
```

If we assume that `doSomeProcessing` implies a lot of work, it may be useful to try to short-cut that work whenever the script can determine that an input has not changed since the last successful run of same script:

```
## Example with input barrier.
foreach input ( $listOfInputs )
  set barrierInfo = `VovInput -show_barrier $input || exit 1`
  if ( "$barrierInfo" == "" ) then
    doSomeProcessing $input
  else
    echo "Skip processing of input $input because of input barrier"
  endif
end
```

Another example of script with input barriers can be found in `$VOVDIR/training/basic/script_with_input_barriers.csh`

vovbarrier

vovbarrier in shell scripts

While writing shell scripts, you may also find that the script `vovbarrier` may be useful. For example, the following code is also equivalent to a simple implementation of `clevercopy`:

```
#!/bin/csh -f
VovInput $1 || exit $status
vovbarrier $1 $2
exit $status
```

VovBarrier in C and C++

RCPC is implemented with the procedure `VovBarrier()` that has been added to [VIL](#), and with the tool [vovbarrier](#). In both cases, the key argument is the action to be performed on the barrier, which is either "STOP" or "PROPAGATE".

There are three options for a tool to control change propagation on one of its outputs:

- Call `VovBarrier(..., "STOP");` This does not invalidate the fanouts of the output file immediately, allowing time to erect a barrier so the output file is not invalidated by invalidating this tool.
- Call `VovBarrier(..., "PROPAGATE");` This invalidates the fanouts of the output file immediately, but also erects a barrier so the output file won't be invalidated by invalidating this tool.

- Do not call `VovBarrier(...)`; This invalidates the fanouts of the output file immediately, allowing the file to be invalidated in the future if this tool is invalidated.

clevercopy / cleverrename

clevercopy (copy files intelligently)

The simplest tool that offers change propagation control is a clever version of copy. Rather than blindly copying a file onto another one, `clevercopy` begins by comparing the input with the output and starts the actual copying only if it detects any difference between them. If there is no difference, `clevercopy` informs the `vovserver` that the output has not changed.

Any tool can be made to exhibit this effect by inserting a `clevercopy` on one or more of its outputs.

```
usage: clevercopy [-fil] [-l label] [-m mode] [-n] [-t filter_tcl_script]
               [-T filter_tcl_script] [-uv]
  -f:          Force writing of output even if the output file is not
               writable. This controls only the permissions on the output
               file, not its directory. This option may still fail if the
               output directory is not writable.
  -i:          Disable input barriers. See on-line documentation for more
               information on input barriers.
  -l:          The label is ignored. It is used to annotate the command line.
  -m:          UNIX: set access mode for output files. The argument 'mode'
               should be in octal and has the same form as for chmod().
               WindowsNT: the access mode for output files is set to 'Read
               Only'. The argument is ignored. By default, the mode is set to
               allow writing.
  -n:          Do not activate VIL
  -t:          Use script as filter. Output is the filtered input. See man
               page for information on filter scripts.
  -T:          Same as -t, but copies unfiltered input
  -u:          Unlink old file if there is a change. Useful when the output
               is an executable which may be in use.
  -v:          Verbose flag.
```

`Clevercopy` can automatically copy many files from one directory to another. In response to the barriers on the inputs, `clevercopy` can activate one or more of its [output barriers](#). This stops change propagation without having to compare the input with the corresponding output.

```
# Clevercopy file 'bb' to file 'cc'
% clevercopy bb cc

# Declare a job clevercopying 'aa' to 'bb'.
J clevercopy aa bb
```

Please make note that `clevercopy` is a fully instrumented Altair binary that usually requires no wrapper. (You may need to wrap with a `vov` if you experience NFS cache issues).

Use filters with `clevercopy` to:

- transform files to a canonical form
- control barrier based on a comparison of the canonical form

Example of a filter:

```
#
# -- A filter to reduce a file to lowercase.
#
set fp1 [open [shift] "r"]
set fp2 [open [shift] "w"]

puts -nonewline $fp2 [string tolower [read $fp1]]

close $fp1
close $fp2
```

Use the option `-t` in `clevercopy` to apply a filter while copying. Example:

```
% clevercopy -t tolower.tcl bb clever_cc clevercopy
```

cleverrename (copy files intelligently, then remove the inputs)

A variation of `clevercopy` is `cleverrename`, which reduces storage requirements by deleting the input after the copy is made.

```
usage: cleverrename [-fil] [-l label] [-m mode] [-n] [-t filter_tcl_script]
               [-T filter_tcl_script] [-uv]
  -f:          Force writing of output even if the output file is not
               writable. This controls only the permissions on the output
               file, not its directory. This option may still fail if the
               output directory is not writable.
  -i:          Disable input barriers. See on-line documentation for more
               information on input barriers.
  -l:          The label is ignored. It is used to annotate the command line.
  -m:          UNIX: set access mode for output files. The argument 'mode'
               should be in octal and has the same form as for chmod().
               WindowsNT: the access mode for output files is set to 'Read
               Only'. The argument is ignored. By default, the mode is set to
               allow writing.
  -n:          Do not activate VIL
  -t:          Use script as filter. Output is the filtered input. See man
               page for information on filter scripts.
  -T:          Same as -t, but copies unfiltered input
  -u:          Unlink old file if there is a change. Useful when the output
               is an executable which may be in use.
  -v:          Verbose flag.
```

Please make note that `cleverrename`, just like `clevercopy`, is a fully instrumented Altair binary that usually requires no wrapper. (You may need to wrap with a `vov` if you experience NFS cache issues).

Search Path for Filters

Filters are looked for in the following directories:

- The current directory
- `$VOVDIR/local/cleverfilters`
- `$VOVDIR/etc/cleverfilters`

Flow Library

FlowTracer captures your design in design flows described by Flow.tcl files. The Flow Library is meant to help users locate and deploy instances of packaged flows bound to their specific design data.

Flow Library Components

The Flow Library consists of three main components:

Component	Description
addflow.cgi	Browser-based wizard to guide user
vovflowcompiler	Script to package information about a flow
vovflowlib	Tcl library procedures for packaged flows

How to Package a Flow for the Flow Library

The information about a flow is extracted from the prepared Tcl description of the flow and placed in a file with a **.pp** extension. This file is placed in the Flow Library directory structure so that information about the flow may be presented in the browser.


You package a flow for the library by placing some procedures from the Tcl library `vovflowlib.tcl` in your regular `Flow.tcl` file that defines the steps of the flow, then run `vovflowcompiler`, which produces and optionally installs the `.pp` file.

How to Select and Deploy a Flow from the Flow Library

Point your browser to the home page of your FlowTracer project and click the **Add Flow** button at the bottom, or go directly to the AddFlow browser-based UI to add a flow. Follow the steps outlined on the left-hand side of the browser pages.

Add a Flow

The `addflow.cgi` script is used to add a flow to the current project.

 **Important:** You must have already created and started the VOV project. The files and jobs defined by the flow are added to the database of the server which is serving the CGI pages.

The purpose of this script is for users to be able to select from a library of preconfigured flows for doing various tasks, for example, to place and route a random logic block, or to synthesize a datapath block. This CGI script guides the user through the steps needed.

The first phase is to select the `Flow.tcl` file.

1. Click **SelectFlow** on the left-hand side.

This will be highlighted in orange to indicate that it is selected, and is the default when you first open the CGI script.

2. Click the **flow library** link in the right-hand pane.
This will help you navigate through the predefined flows. If there is a .pp file for the flow, information about the flow will be displayed. The .pp files are created by the `vovflowcompiler` program.
3. Once you have located the desired flow, click on **Select this flow** link.
This sets a browser cookie identifying the flow you chose, and takes you back to the menu page.

Select the Directory

4. Click the **SelectDirectory** menu item on the left-hand side.
This becomes highlighted, and displays 'Select Directory'.
5. Click the **visiting a directory** link in the right-hand pane.
This takes you to a page showing all the 'workspaces', or filesystems which have logical names, known to your VOV project.



Note: If the directory in which you want to run the jobs of the chosen flow does not have a logical name, you need to create one by modifying the `equiv.tcl` configuration file for your project. Please see [Equivalence File](#).

Assign a label

6. Click on the **AssignLabel** menu item on the left-hand side.
This will then be highlighted, and then show 'Assign Label' in the right-hand pane.
7. Enter a label for this instance of the flow, if desired.
If you will have many similar instances of the flow, a label can help you tell them apart.

Set the parameters

8. Click on **SetParameters**.
If the flow you chose has no parameters, you will receive a message saying that you do not need to set any parameters.
9. If there are any parameters for the flow, fill them in appropriately.

Flow Library Packager vovflowcompiler

The tool `vovflowcompiler` creates a packaged flow, or .pp file from a Tcl format flow description file. You use some additional procedures defined in `$VOVDIR/tcl/vtcl/vovflowlib.tcl` to specify the description and parameters of the flow.

For example, here is the information added to the hourglass training example flow.

```
Description "Build a large flow as a training exercise."
Keywords    "Training toolx hourglass"
Parameter MAX_ROW -default 2 -doc "Number of levels in the flow (between 2 and 10)"
Parameter RESOURCES -default "unix" -doc "The resources required by the flow"

if [GetAllParameters] {
  # ...
}
```

```
# regular Flow.tcl contents for the flow
# ...
}
```

The above flow has two parameters. The first is the depth of the flow, such as the number of levels from the top to the middle job of the hourglass-shaped graph. This is limited to the range of 2 to 10, so that the graph does not become degenerate or too large. The second parameter is the requested resource for running the jobs, which defaults to 'unix' so the jobs may be placed on any UNIX-like system.

Here is the brief usage information for the `vovflowcompiler` command.

```
vovflowcompiler:
vovflowcompiler: DESCRIPTION:
vovflowcompiler:     Generate a .pp file from a flow description in Tcl.
vovflowcompiler:
vovflowcompiler: USAGE:
vovflowcompiler:     % vovflowcompiler [options] <flow_definition_file>
vovflowcompiler: OPTIONS:
vovflowcompiler:     -install    <dir>          -- Install the .pp file into
vovflowcompiler:                               the specified directory
vovflowcompiler:
vovflowcompiler:
```

Flow Library Procedures

This section documents the procedures for use with `vovflowcompiler`, the tool for packaging `Flow.tcl` scripts for inclusion in the Flow Library. You use these procedures in the `Flow.tcl` file you submit to `vovflowcompiler` to assign values to the metadata about the flow. These metadata values are displayed to the user when they browse the Flow Library using the [Add Flow CGI script](#).

These are the procedures:

Tcl Procedure	Description
Description	Assign the description of the purpose of a flow
Keywords	Assign the keywords associated with a flow
Parameter	Assign characteristics of a parameter of flow. The facets you can assign are: <ul style="list-style-type: none"> • Default value • Documentation string • Valid range • Data type • List of choices for a select type parameter • Field width

Tcl Procedure	Description
GetAllParametersFromTk	Get flow parameters from a Tcl/Tk application
GetAllParametersFromHTML	Get flow parameters from a browser-based application
GetAllParametersFromCLI	Get flow parameters from a command-line application
RequiredFile	Test for presence of file needed by flow, copy from sample if any
RequiredDirectory	Test whether a directory needed by the flow exists

Files, Equivalences, Exclusions

The files used in a project reside in one or more directories spread all around your filesystems. Each logical name defined in the `equiv.tcl` file implicitly declares the root of a workspace. You can browse the workspaces using the browser interface.

Files and File Names

The files used in a project reside in one or more directories spread all around your /filesystems.

Each **logical name** defined in the `equiv.tcl` file implicitly declares the root of a workspace.

You can browse the workspaces using the browser interface by visiting the Workspaces page and the `/dir?dir=dirname` URL. (You need to enable cookies in the browser, because the directory most recently visited is stored in a cookie.)

Canonical and Logical File Names

VOV clients and server exchange dependency information by using file names; each file needs a single name that is valid on both the client and the server.

It may seem that each file, could use its full path as its unique name. However, a file may and will have more than one name for the following reasons:

- Links, both hard and symbolic, allows multiple full paths for the same file.
- For any file, it is possible to generate an infinite number of full paths by using the "dot" and "dot-dot" notation (for example, `/usr/bin/ls` can also be written as `/usr/../usr/bin/./ls`).
- The same file may have different full paths on different hosts due to how the file systems are network mounted.

Canonical Names

VOV defines the *canonical name* of a file to be the full path obtained by removing all symbolic links and all "dots".

In this example, a file system contains the following link:

```
/users/john/projects --> /sandbox/projects
```

With the relative path `~/projects/vhdl/vtech/../../syn/vtech.v`, with respect to the user `john`, the following transformations would apply:

The non-canonical path	<code>~/projects/vhdl/vtech/../../syn/vtech.v</code>
after tilde expansion becomes	<code>/users/john/projects/vhdl/vtech/../../syn/vtech.v</code>

after removing the symbolic link becomes	/sandbox/projects/vhdl/vtech/../../syn/vtech.v
by removing the double dot becomes canonical	/sandbox/projects/vhdl/syn/vtech.v

Logical Names

A canonical name is then turned into a *logical name*. A logical name is one in which the file name begins with the value of a variable.

For example, the name `${HOME}/foo.c` is logical, while `/users/home/john/foo.c` is not.

The use of logical names is critical because the value of the variable used in the name is allowed to be different on different hosts. This is to account for the different ways the file systems are mounted across the network.

For example, the variable `${HOME}` may point to `/users/home/john` on a UNIX machine and to `h:/john` on a Windows NT machine.

All filenames in VOV are logical and canonical names. The logical names are formed according to the rules defined in the [equiv.tcl](#) file.

There are two further advantages in using logical canonical names:

- The average length of names is reduced, which reduces the storage requirements for the trace.
- The trace can be easily moved from one file system to another.

Databases

Design files may reside in different databases. The most common database is the filesystem. VOV offers a generic database interface in the sense that it traces dependencies between named entities residing in persistent databases.

VOV requires a method to obtain the timestamp of each named entity in the database. For example, the most common database is called "FILE", its entities are files, and the method to obtain the timestamp of a file is the OS call `stat()`.

VOV supports multiple databases. Some are supported internally. Others must be supported externally by special clients called database proxies, which provide the server with the needed timestamp for the entities in a database.

Check the Databases page to see the list of databases supported by the current project.

Internally Supported Databases

The following databases are currently supported internally:

FILE

The entities are files. The names are logical canonical path names. The timestamp is the modification time of the file. If a file does not exist, the timestamp is 0.

FILEX

The entities are files, but in this case only the existence of the file, is important; the timestamp is not important. The names are logical canonical path names. If the file exists, the timestamp is fixed in the distant past. If a file does not exist, the timestamp is 0.

To specify a file where you just care about the existence of the file, regardless of its timestamp, the syntax is different whether the file is an input or an output:

	FDL	Instrumentation
Input	I -db FILEX myfile D FILE	VovInput -db FILEX myfile
Output	O -db FILEX - ignore_timestamp myfile D FILE	VovOutput -db FILEX - ignore_timestamp myfile

The change of database is persistent. Next Inputs & Outputs will use the FILEX database, unless you add a -db FILEX or a D FILE line in your flow.

GLOB

The entities in this database are "glob expressions." The syntax is the one used by the glob procedure in Tcl and by "globbing" in C-shell, that is "*" stands for any sequence of zero or more characters and "?" stands for any single character. If the expression can be expanded into a set of files, the timestamp of the entity is the timestamp of the most recent file. If the set of files corresponding to the expression is empty, the entity is considered non-existing and its timestamp is 0.

This database is useful to represent the situation where a job depends on a directory and on all the files in the directory.

JOBSTATUS

This database represents the status of a job. The name of the place has the form *TYPE/JOBID* where TYPE is one of DONE, FAIL, or SUCCESS and JOBID is the id of a job. The timestamp of the place is a fixed timestamp in the past and never changes. What changes is the status of the place depending on the status of the job it is attached to. For more information on this database, please refer to [The JOBSTATUS Database](#).

LINK

This database behaves like the FILE database, except that the entities are symbolic links. An easy way to add links to the flow is to use the option `-links` in the calls to VovInput and VovOutput (either encapsulation or instrumentation)

```
...  
VovInput -db LINK nameOfLink  
VovInput -links someInputFile  
VovOutput -links someOutputFile  
...
```

PHANTOM

This database behaves like the FILE database, except that the timestamp of a missing file is, by convention, a time in the distant past (by Computer Science standards), i.e. sometime in the year

1970. This database is useful to represent situations where a job is sensitive to the fact that a file may exist or not, for example when a tool uses a search path to locate files.

VOVSETS

The entities in this database are sets of nodes. The timestamp of the entity is the timestamp of the set creation. Each set in the trace has a name and a timestamp.

ZIP

The entities in this database are members in an ZIP archive. The name of the entity has the form ARCHIVENAME (MEMBERNAME) .

The timestamp of the entity is the timestamp of the member inside the archive. In encapsulation scripts, be careful to escape the parentheses appropriately as in the following example:

```
...
VovOutput -db ZIP -ignore_timestamp $archiveName\($memberName\)
...
```

The LINK Database

To establish dependencies with symbolic links, the LINK database must be used.

This section assumes that are familiar with the "symbolic link" in a UNIX file system. For information about symbolic links, you can use the UNIX command `man ln`.

For example, the tool "ln" can be used to create a link. In FDL, write:

```
# Create a link called 'bar' for to a file called 'foo'
#   bar → foo
R "unix"
T vw ln -s foo bar
O -db LINK bar
```

Notice that the file `foo` in the example above does not even need to exist, and therefore it is not considered an input to "ln". The link `bar → foo` is created whether `foo` exists or not. The job `ln -s ...` can be run at any time, even before `foo` exists, but not before the jobs that use the symbolic link `bar`.

The timestamp of the link is not the same as the timestamp of the file. You can have an old link that points to a young file. Recreating the link will normally change the timestamp of the link.

It is possible to create automatic dependencies for all links used in path expansion. If you are using the VIL tools, you can use the option `-links` at runtime:

```
### ---- in the middle of a script ....

### All symbolic links used to expand foobar and abc will be declared
### as inputs to the current job.
VovOutput -links foobar || exit 1

VovInput -links abc || exit 1
```

All symbolic links are declared as input dependencies. (The only tool that has a LINK as output is "ln" or some other application that call the `symlink()` system call.)

Alternatively, you can set the environment variable `VOV_VW_TRACK_LINKS` before you invoke the tool to track all symbolic links.

```
setenv VOV_VW_TRACK_LINKS 1
vw cp aa bb
```

Examples

In the following examples, the following structure are assumed (links shown in **bold**):

```
/project/ivy/releases/current      -> milestoneA
/project/ivy/releases/milestoneA   -> serials/00014
/project/ivy/releases/serials/0014/file.txt
```

Focus on runtime declarations using the VIL-Tools `VovInput` and `VovOutput`. Assume you are in directory `/project/ivy/releases`:

- ```
VovInput current/file.txt
VovInput -db FILE current/file.txt
```

These commands are equivalent, because `FILE` is the default database. They both create an input dependency with `FILE serials/0014/file.txt`, because the links "current" and "milestoneA" are both traversed.

- ```
VovInput -db LINK -quote current
```

This creates an input dependency for the symbolic link `current` → `milestoneA`. The path "current" is not expanded because it is quoted by the option `-quote`.

- ```
VovInput -links current/file.txt
```

This command creates three dependencies: one for `FILE serials/0014/file.txt` and two for the links that have been traversed. This is equivalent to these three lines:

```
VovInput -db FILE /project/ivy/releases/serials/0014/file.txt
VovInput -db LINK -quote /project/ivy/releases/serials/current
VovInput -db LINK -quote /project/ivy/releases/serials/milestoneA
```

- ```
VovInput -db LINK current
```

Without the `-quote` option, this creates an input dependency for the directory `serials/0014`, but in the "LINK" database, which is actually a silly thing to do. Behaviorally, this is not much different from having a dependency on a "FILE" `serials/0014`.

- ```
VovInput -links -db LINK current
```

This is the same as above, but in addition we also have dependencies on the two links that have been traversed.

- ```
VovInput -links -db FILE current
```

This is better than above, because the expanded path "current" is actually a directory, and not a link.

Option -links in FDL

The option -links is available also in the procedures I and O in FDL, but they have no effect. The symbolic links are only added as inputs when the jobs are actually run.

```
## This is legitimate FDL.  
T vw cp aa bb  
O -links bb  
I -links aa
```

```
# More useful is to use -links in a capsule-on-the-fly  
T vw ./myjob aa bb  
CAPSULE {  
  I -links aa  
  O -links bb  
}
```

Debugging symlinks

If you want to get more verbose information about handling of symlinks, you need to add the following statement in the `equiv.tcl` file:

```
# This goes in the equiv.tcl file.  
# It activates debugging for both the equivalence subsystem  
# and the symlink subsystem  
vtk_equivalences_debug 1
```

This change will have an effect when you run another job. Remember to revert the change after you are done debugging.

Links into the DesignSync Caches


Some links in particular need not be traversed. One example of this are the links into the DesignSync caches. To define which caches to ignore, use the variable `VOV_SYNC_CACHE_DIR`, which is a colon-separated list of paths.

```
##  
## Avoid expanding links into the DesignSync caches.  
## This is normally set in the setup.tcl file.  
W#  
setenv VOV_SYNC_CACHE_DIR /some/location/sync1/cache:/other/location/sync33/cache
```

The JOBSTATUS Database

Places in the JOBSTATUS database represent the status of jobs.

The name of the place is of the form `TYPE/JOBID`: TYPE represents DONE FAIL or SUCCESS; JOBID is typically the ID of the job containing the place.

 **Note:** JOBID could be any unique string.

This database does not require any operation on any filesystems. It is used in Accelerator to represent jobs that have no output log. It is also used to represent jobs whose execution depends on the success or failure of a job.

To attach a JOBSTATUS place to a job, use the following FDL code:

```
# Fragment of FDL
set jobid1 [J vw cp aa bb]
O -db JOBSTATUS -sticky -quote "DONE/$jobid1"

# Run this job after jobid1, whether it passes or fails.
set jobid2 [J vw cp bb cc]
I -db JOBSTATUS -sticky -quote "DONE/$jobid1"
```

Type

Behavior

SUCCESS

This is the simplest case. The place becomes VALID when the containing job also becomes VALID. If there was a barrier on the place, the barrier is removed.

FAIL

The place becomes VALID only when the containing job becomes FAILED. At the same time, a barrier is added to the place in order to preserve the trace consistency, since it is not legal for an output of a FAILED job to be VALID unless there is a barrier on the output place.

DONE

This is a combination of the two cases above. The place becomes VALID only when the containing job completes, when it becomes either VALID or FAILED. At the same time, a barrier is added or removed depending on the status of the job.

Define Equivalences for File Names

There are multiple methods to define the equivalences used to compute the canonical names of files and directories.

For example:

- Instruct the server to parse the `equiv.tcl` file and provide entries to clients. This is the default behavior. Note that for this case, equivalences that reference an environment variable should not resolve the variable in this file, in environments that will have both UNIX and Windows clients. Instead, they will need to be resolved by the client upon receipt. This is done by enclosing the equivalence value inside curly braces and referring the environment variable as `$VARNAME` as opposed to the Tcl format of `$env(VARNAME)`.
- Instruct clients to read the file directly. This is a legacy method that requires that all clients have access to the server working directory so they can parse the `equiv.tcl` file for entries, and read/write access to the `equiv.caches` directory so the entries can be written to a host-based cache file for future use. In this mode, environment variables may be resolved in this file, but the behavior will be the same as not allowing them to be resolved. To resolve them in this file,

the equivalence value should not be wrapped with curly braces and the environment variable should be referred to in the Tcl format of `$env(VARNAME)`. This method is enabled by setting the `VOVEQUIV_CACHE_FILE` environment variable to "legacy".

- Instruct clients to read a specific cache file only. This is a special method used in corner cases where directories may not be the same but should be forced to be considered the same. This is utilized mainly by Monitor agent single-file distributables. In this mode, environment variables may be resolved in this file, but the behavior will be the same as not allowing them to be resolved. To resolve them in this file, the equivalence value should not be wrapped with curly braces and the environment variable should be referred to in the Tcl format of `$env(VARNAME)`. This method is enabled by setting the `VOVEQUIV_CACHE_FILE` environment variable to a valid equivalence cache file path.

Equivalence File

The equivalence file (`equiv.tcl`) defines the rules to generate logical names. This file is used by all clients as well as by the server. This file is a Tcl script. The fundamental procedure used in this script is `vtk_equivalence`.

The procedure `vtk_equivalence` has the following purposes:

- The main purpose is to define an equivalence between a logical name and a physical path, as in:

```
vtk_equivalence TOP /export/projects/cpu
vtk_equivalence TOP p:/cpu
```



Note: The physical path need not be canonical. The definition is silently ignored if the physical path does not exist.

- The secondary purpose is to control the case sensitivity for file names, using the options `-nocase` or `-case`. With `-nocase`, all names are canonicalized to lowercase, which is useful when the vovserver is running on a Windows NT machine.
- The third purpose is to control whether the AFS paths should be supported. If the `-afs` option is used, then all paths of the type `/.automount/hostname1/root/aaa` become `/net/hostname1/aaaa`

The procedure `vtk_equivalence` also has side effects:

- The environment variable corresponding to the logical name is set, if it does not exist already (that is, the variable `$env(TOP)`).
- The Tcl global variable corresponding to the logical name is set to the value of the environment variable (that is, the variable `$TOP`).

Example Uses of `vtk_equivalence`

See the following example:

```
# -- HOME should not be used in multi-user projects, because it has a
# -- different value for each user. Use it only in single-user projects.

vtk_equivalence HOME $env(HOME)

# -- VOVDIR is always defined and these equivalences are always useful.

vtk_equivalence VOVDIR $VOVDIR
```

```
vtk_equivalence VOVDIR $VOVDIR/../../common

# -- Data directories.

vtk_equivalence TOP /export/projects/cpu; # This is for Unix
vtk_equivalence TOP p:/cpu                ; # This is for Windows

# Uncomment this if you need AFS paths.

# vtk_equivalence -afs
```

For another example of equivalence file, see the default file for the "generic" project type \$VOVDIR/local/ProjectTypes/generic/equiv.tcl.

Define Host-specific Overrides for the Server-side Equivalence Cache

The server-side equivalence cache can be accessed via the VTK Tcl API using `vtk_equivalence_get_cache OPTION`. When passing a host name in for `OPTION`, the equivalences for that host will be returned. When passing an empty string in for `OPTION`, the list of host names that have entries is returned. By default, a special host name of `"_default_"` is used for the server-side cache that applies to all clients.

The server-side equivalence cache can be set with via the VTK Tcl API using `vtk_equivalence_set_cache HOSTNAME VALUES`, where `HOSTNAME` is the name of a host or the `"_default_"`, and `VALUES` is a Tcl list with an even number of elements in the form `LOGICAL_NAME PHYSICAL_PATH`.

```
vtk_equivalence_set_cache lin0201 "HOMES /homes VOVDIR /tmp_mnt/tools/rtda/current/"
```

The equivalences can also be viewed and managed via the web UI on the Equivalences page.

Equivalence Cache

To avoid executing the `equiv.tcl` script over and over for each tool invocation, the results of the evaluation are stored in the subdirectory `equiv.caches` in the server configuration directory, one for each host. The caches are recomputed automatically when the equivalence file changes.

To force a refresh of the caches, use one of the following methods:

- Restart all taskers using

```
% vovtaskermgr restart
```

- Use the option `-r` in `vovequiv`, to refresh the cache on the current host:

```
% vovequiv -r
```

- Use the option `-rs` in `vovequiv`, to refresh the cache and show the result:

```
% vovequiv -rs
Directory of equiv files: .
Executable equiv file:    /Users/john/projects/mac81/vovadmin/mac81.swd/equiv.tcl
```

```
Cached      equiv file:      /Users/john/projects/mac81/vovadmin/mac81.swd/  
equiv.caches/mac05  
* VOVDIR          -> /Users/john/rtda/2015.09/mac05  
VOVDIR          -> /Users/john/rtda/2015.09/common  
* BUILD_TOP       -> /Users/john/projects/mac81  
* HOME            -> /Users/john
```

Use vovequiv to Check Equivalences

To test the equivalence file, use the command `vovequiv`. The option `-s` shows all the equivalences valid on the current host plus those computed automatically:

```
% vovequiv -s  
Directory of equiv files: ${JOHN}/vov  
Executable equiv file:   ${JOHN}/vov/italia.equiv.tcl  
Cached      equiv file:  ${JOHN}/vov/italia.equiv.caches/saturn  
(Hits      0) VOVDIR      -> /remote/proj2/evolve_cdrom/evcdrom/common  
* (Hits      0) VOVDIR      -> /remote/proj2/evolve_cdrom/evcdrom/sun5  
(Hits      0) SUNW        -> /export/home/opt/SUNWspro  
* (Hits      0) TOP        -> /home/john/Italia  
* (Hits      0) VENDORS    -> /remote/vendors  
(Hits      0) VENDORS    -> /rtda/vendors  
* (Hits      0) SUNW       -> /opt/SUNWspro  
* (Hits      0) JOHN       -> /home/john
```

To test the effect of equivalences on a specific file, use the option `-p`. For example, in your home directory you can try:

```
% cd ~  
% vovequiv -p file  
${HOME}/file
```

Notice how FlowTracer correctly handles dots and symbolic links:

```
% vovequiv -p ~/.file  
${HOME}/file  
% ln -s file foobar% vovequiv -p foobar  
${HOME}/file
```

Taskers: Mixed Windows NT and UNIX Environment

It is common to run VOV in a mixed network of UNIX and Windows NT workstations. Following is a set of guidelines to facilitate the deployment in such a network.

To avoid case-based ambiguities with file names, use only lowercase names in your design.

Define all appropriate [equivalences](#) in the `equiv.tcl` file. For example, if a project directory is mounted as `/projects/local/top` in UNIX and as `p:/local/top` on Windows NT, you can define the following equivalences:

```
vtk_equivalence TOP /projects/local/top  
vtk_equivalence TOP p:/local/top
```

In the `taskers.tcl` file, be explicit about specifying the directory of the VOV installation as seen from the Windows NT machines as well as the working directory of the server as seen from the Windows NT machines. Example:

```
vtk_tasker_set_default -vovdir v:/vov/winnt -serverdir p:/local/  
top/vovadmin  
vtk_tasker_define nthost1 vtk_tasker_define nthost2
```

The tool `vovtaskermgr` normally uses a remote shell (either `rsh`, `remsh` or `ssh`) to start taskers on remote UNIX machines. Windows NT does not offer an effective equivalent to the UNIX remote shell, so you can either start taskers manually on each Windows NT workstation, or you can use [vovtsd](#).

Exclude Files From the Graph

The power of FlowTracer is its ability to capture all inputs and outputs for each job. When determined as more efficient, such as ignoring details that are considered as unnecessary or of little importance, selected files can be excluded from the dependency graph.

Files are sometimes excluded from a dependency graph when a file exists on one machine but not another. For example, suppose you are running `vovserver` on a 32-bit machine (`lnx32`) but you are compiling a C-file on a 64-bit machine (`lnx64`). The compilation probably requires the file `/lib64/libgcc_s.so.1` which exists on `lnx64` but not on `lnx32`. If the compiler declares `/lib64/libgcc_s.so.1` as input, the `vovserver` (running on `lnx32`) will not find such file and therefore declare the compilation as failed. Having such a file in the dependencies is not really useful and the exclusion mechanism described in this section is an easy way to avoid these dependencies.

The exclusion rules are defined in the `exclude.tcl` file. There are two types of rules relating to a file's name that will lead to a file being excluded from the graph:


- Based on a regular expression.
- Based on a prefix: these rules are a special case of regular expressions but they execute faster,

Both rules are defined with the Tcl procedure `vtk_exclude_rule`. For example, to exclude all files in `/usr/tmp`, a prefix rule can be defined. Example:

```
vtk_exclude_rule -prefix /usr/tmp
```

To exclude all files that end with the `.tmp` suffix, a rule can be based on a regular expression. Example:


```
vtk_exclude_rule -regexp {\.tmp$}
```

 **Note:** The braces are necessary to protect the regular expression from the undesired evaluation of special characters such as \$ and [].

While parsing the `exclude.tcl` file, the global variables `$argv0` and `$argv` are set to the command line of the tool, including the wrapper. This makes it possible to create different exclusion rules depending, for example, on the tool name. Additionally, the variables `EXCLUDE_TOOL`, `EXCLUDE_JOBNAME`, and `EXCLUDE_JOBCLASS` are available.

```
if { $EXCLUDE_TOOL == "dc_shell" } {  
    vtk_exclude_rule -regexp {/command.log}  
}
```

The exclude file is used by the tools and it is ignored by the server. Any change to the exclude file is effective immediately for all the tools that are executed after the change. The change, however, has no retroactive effect. Files to be excluded that are already in the graph must be forgotten explicitly with other commands.

Procedure	Arguments	Description
<code>vtk_exclude_rule</code>	<code>[-prefix </code> <code>-regexp </code> <code>-clear </code> <code>-tool TOOLNAME_RX </code> <code>-jobname JOBNAME_RX </code> <code>-jobclass JOBCLASS_RX]</code> <code>string</code>	<p>Define an exclusion rule. This procedure is used in the <code>exclude.tcl</code> file and can also be used in the encapsulation files. Examples:</p> <ul style="list-style-type: none"> Exclude all files that end begin with <code>/usr/tmp</code> <pre>vtk_exclude_rule - prefix /usr/tmp vtk_exclude_rule - tool dc_shell -regexp command.log vtk_exclude_rule - jobclass synth -regexp {.vvv\$}</pre> Exclude all files that end with a tilde. <pre>vtk_exclude_rule {~\$}</pre> Reset the exclude rules. <div>  Note: This procedure is rarely used. </div> <pre>vtk_exclude_rule - clear</pre>

Procedure	Arguments	Description
		<ul style="list-style-type: none"> Exclude the <code>command.log</code> file, but only if the tool is <code>dc_shell</code>. <pre>vtk_exclude_rule - tool dc_shell -regex command.log</pre> <ul style="list-style-type: none"> Exclude files ending in <code>.vvv</code> if the job is running in the job class "synth". <pre>vtk_exclude_rule - jobclass synth -regex {.vvv\$}</pre>
<code>vtk_path_exclude</code>	<code>path</code>	Test whether a path is excluded by the rules that have been defined. This procedure is normally not used in the <code>exclude.tcl</code> file.

Examples of `vtk_exclude_rule`

For an example of exclude file, see the default file for the "generic" project type at `$VOVDIR/local/ProjectTypes/generic/exclude.tcl`.

Another example:

```
# The files that start with these prefixes will not be added to the graph.
vtk_exclude_rule -prefix ${VOVDIR}}
vtk_exclude_rule -prefix /dev/
vtk_exclude_rule -prefix /devices/
vtk_exclude_rule -prefix /etc/
vtk_exclude_rule -prefix /proc/
vtk_exclude_rule -prefix /lib/
vtk_exclude_rule -prefix /opt/CC
vtk_exclude_rule -prefix /tmp/
vtk_exclude_rule -prefix /tmp_mnt/usr/
vtk_exclude_rule -prefix /usr/
vtk_exclude_rule -prefix /var/
vtk_exclude_rule -prefix c:/
vtk_exclude_rule -regex {Dependency.state$}
vtk_exclude_rule -regex {/gcc-lib/}
vtk_exclude_rule -prefix /lib64;    ### Added for multi-arch compilations.

# -- Exclude the NT executables from the current installation.
vtk_exclude_rule -regex {VOVDIR.*/[a-z]+\..exe$}

# -- Tool dependent exclusion rule.
# -- (the first arg is the wrapper, the second the tool)
# -- Could also use the variable EXCLUDE_TOOL
switch -- [lindex $argv 1] {
```

```
"toolx" {  
    vtk_exclude_rule -regexp /TOOLXCACHE/  
}  
}
```

Debugging the Exclusion Mechanism

Use the environment variable VOV_STRICT_TRACING to disable the exclusion mechanism in your shell.

Use the environment variable VOV_DEBUG_FLAGS to force the exclusion routines to print out debugging messages:

1.

```
% setenv VOV_DEBUG_FLAGS 128
```

2. Run the tools.

Additional Files with Exclusion Rules

The variable VOV_EXCLUDE_FILES can be used to list a number of additional files that contain exclusion rules. Examples:

```
% setenv VOV_EXCLUDE_FILES $VOVDIR/local/exclude/exclude.cdn.tcl  
% setenv VOV_EXCLUDE_FILES $VOVDIR/local/exclude/exclude.cdn.tcl:$VOVDIR/local/  
exclude/exclude.snps.tcl
```

Change the exclude.tcl File

The exclude file is used by the tools and not by the server; when an exclude file is changed, no immediate action will occur. The files that are excluded will remain in the trace if the files are already present. However, as tools are executed, the dependencies with the excluded files will be dropped.

In the following example, all dependencies with files in the directory /home/tools/bin are being excluded from the trace.

1. Add to the exclude.tcl file the line:

```
vtk_exclude_rule -prefix /home/tools/bin
```

2. Create a set of all the files to be excluded that are already in the trace:

```
% vovset create Tmp:exclude "isfile name~/home/tools/bin"
```

3. Forget all those files:

```
% vovforget -elem Tmp:exclude
```

4. Forget the temporary set:

```
% vovset forget Tmp:exclude
```

Automatic Zipping and Unzipping Files

FlowTracer can automatically compress (zip) and uncompress (unzip) files in the flow using the utilities `gzip` and `gunzip`. This service is provided for all places that satisfy the following conditions:

- In the database "FILE"
- Have the "zippable" flag set

Rule 1: Compression

A zippable file is automatically compressed when all the jobs that require compression have completed successfully. The check to see if compression is required is performed about every 30 seconds.

Rule 2: Decompression

A zippable file is automatically uncompressed if any of the jobs that requires it is queued and ready to be executed. Jobs will not be executed with compressed inputs.

If a file is compressable, it can be in either the normal or the compressed state. For a file called 'X', if the file does not exist but 'X.gz' does, then the file is considered compressed.

Directing vovzip Jobs to Certain Hosts

The work to compress/uncompress a file is performed by a system job that is executed by the owner of the project. These jobs are scheduled to be executed by the appropriate vovtaskers.

To direct the `vovzip` jobs to particular hosts, create a tool map in the project's `resources.tcl` file, where the right-hand side of the map is the necessary resource.

For example, the following statement could be used:

```
vtk_resourcemap_set Tool:vovzip UNLIMITED vovzip_host
```

The vovtasker resource `vovzip_host` can be placed on the desired hosts in the project's `taskers.tcl` file. The number of concurrent `vovzip` jobs can be limited: instead of `UNLIMITED`, insert an integer such as 10.

Defining Zippable Files

In a flow description, the procedure `z` can be used to define the files that are zippable. Example:

```
J vw cp aa bb  
J vw cp bb cc  
Z bb cc
```

Another method is using the command `vovset` with the subcommand `zippable`.

Examples:

```
% vovset zippable SETNAME 1  
% vovset zippable SETNAME 0
```

Using Tcl, `vtk_place_get` and `vtk_place_set` can be used to test and set the zippable flag.

```
set id [ vtk_place_find "FILE" "aa"]
```

```
vtk_place_get $id info
puts "The current value of the zippable flag is $info(zippable) "
puts "The current value of the zipped flag is $info(zipped) "

set info(zippable) 1
vtk_place_set $id info
```

Recommendations for Zippable Flags

Using the zippable flag is recommended for relatively large files, such as greater than 100kB. A tradeoff needs to be evaluated: the space consumed and flow complexity and additional CPU/IO time, versus the degree of the compressibility of the files. For example, JPEG and PDF files are already highly compressed; zipping these files may produce little or not benefit. Test vector files, however often compresses to 3-8% of their native size.

vovzip Jobs and Log Files

The information in this section may be helpful for troubleshooting zippable files. The jobs that compress and expand the zippable files:

- Are created automatically by the vovserver.
- Use the tool `vovzip`, supplied by FlowTracer.
- Are retraced in "FAST" mode.
- Their log file goes into the directory `projectName.swd/vovzipdir`.
- The name of the log file is of the form `FILEID.log`, where `FILEID` is the `VovId` of the zippable file.

Makefile Conversion

If you have Makefiles, you can use `vovmake` to extract flow information from the Makefile. This approach is intrinsically less capable and less reliable than using FlowTracer's [Flow Description Language \(FDL\)](#) to describe your flows, because of the limitations of the Makefiles. For example, the Makefiles do not have information about the resources (e.g. licenses, RAM) required for each job in the flow, in which case you may want to augment the description contained in the Makefile with a [vovmake.config.tcl](#) file.

There are some limitations to using Makefiles to maintain large flows:

- `make` is commonly used recursively, so there is no one instance of `make` that contains the complete dependency graph
- `make` does not have any graphical user interface, so you need to read log and dump files carefully when troubleshooting.
- the language of makefiles is complicated and arcane, and uses many tersely-named special variables. This is OK for experts, but is a barrier to new or casual users.

With some exceptions, running your Makefiles with `vovmake` can provide a valuable visualization of the structure of the Makefiles.

You can use your Makefiles as a starting point, then run your computation with FlowTracer, or continue to maintain the makefiles, and just use FlowTracer to visualize what they are doing.

Often the flow can be brought up-to-date more quickly by FlowTracer than by `make`, by running multiple jobs in parallel over the network. Since FlowTracer is client-server, all users see the same state of the computation in their GUI or browser.

Prerequisites for Running vovmake

You must first start a FlowTracer *project*, managed by the `vovserver` program. This can run on the same or a different host from where you run `vovmake`.

The command for starting a project is:

```
% vovproject create <project_name>
% vovproject enable <project_name>
```

In the simplest form, use `vovmake` just like you would use your regular `make`, that is, instead of typing:

```
% make all
```

you will type:

```
% vovmake all
```

Other common invocations are:

```
% vovmake -help
% vovmake -clean
```

vovmake -help

DESCRIPTION:

- This is a script used to convert makefiles into FlowTracer flows.
1. First the makefiles are interpreted by a modified gmake, called `gmake_with_vov_extension`, which contains some Altair Engineering extension and supports the option `-F` to dump the dependencies into Tcl file.
 2. The resulting Tcl file is then mapped into a flow by means of the script `$(VOVDIR)/tcl/vtcl/vovmaketoflow.tcl`.

USAGE:

```
% vovmake [VOVMAKEOPTIONS] [gmake_options]
```

VOVMAKEOPTIONS:

- | | |
|---|--|
| <code>-help</code> | -- Print this help. |
| <code>-clean</code> | -- Just cleanup the generated files. |
| <code>-nocleanup</code> | -- Do not cleanup temporary files (for debugging). |
| <code>-gmake <GMAKEBIN></code> | -- Use specified gmake binary
This must be one with the Altair Engineering extensions.
Default <code>gmake_with_vov_extension</code> |
| <code>-build "options"</code> | -- Options passed to <code>vovbuild</code> , with flow
<code>\$(VOVDIR)/tcl/vtcl/vovmaketoflow.tcl</code> . |
| <code>-version</code> | -- Show gmake version and exit |
| <code>-wrapper <dfltwrapper></code> | -- Specify default wrapper for jobs in flow. It is passed to <code>vovbuild</code> . |
| <code>-run [ft no]</code> | -- Run Flowtracer to get more information about input makefile. Default is <code>ft</code> . |

EXAMPLES:

```
% vovmake -help

% vovmake install
% vovmake <SOME_MAKE_TARGET>
% vovmake -clean

% vovmake -nocleanup all
% vovmake -gmake ~/bin/my_gmake_with_vov_extension
% vovmake -wrapper vrt
% vovmake -build "-env BASE" install
```

To find the options of `vovmaketoflow.tcl`, you can use

```
% vovbuild -f $VOVDIR/tcl/vtcl/vovmaketoflow.tcl -- -help
```

How vovmake Works

The utility `vovmake` operates as follows:

1. It calls `gmake_with_vov_extension` to parse the Makefiles and generates an intermediate flow file called `/tmp/vovmakeUSERNAME/TIMESTAMP/vovmakePID.tcl`.
2. This intermediate flow file is then processed by the Tcl script `vovmaketoflow.tcl` which transfers the dependencies derived from the Makefiles into the FlowTracer server.

The processing of the Makefile rules occurs as follows:

- Simple rules (one-line commands) are preserved and appear as a job in the flow

- Complex rules, i.e. rules that have either multiple lines, or multiple commands, or redirections, are encapsulated into a local shell script in the directory `vmake_scripts`
- The dependencies in your flow graph will be those in your makefile

3. `vovmake` issues the command `vsr` to request that the flow graph should be brought up-to-date

Limitations on makefiles

There are many different versions of `make` available, that support different commands. GNU `make` is the most common, and it has some functionality that is not supported by `vovmake`.

Makefiles that simply define variables, targets, and recipes to make the targets should work OK.

Suggestions

Use `vovmake` only to "build" stuff, not to run targets that clean up. Many makefiles have a target named 'clean' or the like which will remove intermediate and output files, leaving only the sources. It is common to compensate for incorrect or incomplete dependencies by using the 'make clean; make all' sequence.

When dealing with recursive `make`, we recommend you start from the leaves, i.e. the deepest directories. Try to do one directory at a time.

Some Results Using Compilation Examples

The following table describes the results obtained at Altair by using `vovmake` on some common open-source packages.

Package Name	Comments	Size of resulting Flow
GNU make 3.80	One shot build with the usual: <pre>% ./configure % vovmake</pre>	less than 400 nodes
mysql 5.0.19	Complete build succeeds. You can observe multiple passes: <pre>% ./configure % vovmake</pre>	1000 jobs and 4400 files
GNU gdb 6.3	On Linux, you get a flow graph with about 2300 nodes. Exclude the "maybe-*" targets by adding <code>vtk_exclude_rule -regexp {/maybe-}</code> to <code>exclude.tcl</code> file. <pre>% ./configure % vovmake % vovmake</pre>	About 2300 nodes

Package Name	Comments	Size of resulting Flow
	<pre>% vsr -f -all -retry 4</pre>	
valgrind 2.4.0	<p>The package almost builds valgrind in one shot, but shows a very "interesting" makefile system with a few conflicts between addrcheck and memcheck and some failures because of missing '*.Tpo' dependencies.</p> <pre>% ./configure % vovmake % vovmake % vsr -f -all -retry 4</pre>	About 120

Use Makefiles

If your flow is currently managed by Makefiles, you may perform a conversion of the Makefile into a trace using the utility `vovmake`.

- To convert Makefiles to TCL flows use the following command line:

```
% vovmake -f Makefile
```

- `vovmake` executes the following steps:
 1. Parse the Makefile with `gmake_with_vov_extension` to generate a Tcl representation of the dependencies in the Makefile
 2. Use `vovbuild` to parse the file and to build a flow
 3. Request an update (a retrace, in FlowTracer lingo) of all the jobs thus created

There are limitations to this approach, because not all Makefiles represent a proper flow. The conversion may not be 100% complete. This is not necessarily bad, as an improper flow will not produce predictable results anyway.

Options for vovmake

In addition to a list of Makefile targets, `vovmake` accepts a set of options to change the conversion process.

Option	Description
-help	Show the usage.

Option	Description
-clean	Cleanup the auxiliary directories and scripts created by vovmake.
-wrapper <WRAPPER>	Choose the wrapper to prefix each tool invocation. By default, no wrapper is used, so that runtime tracing is not active on the resulting flow. This means that the default behavior is to use exactly the dependencies specified in the Makefile.
-res <RESOURCES>	Resource expression to be used with the jobs in the flow. By default, the resource expression is empty.

vovmake -help

DESCRIPTION:

This is a script used to convert makefiles into FlowTracer flows.

1. First the makefiles are interpreted by a modified gmake, called gmake_with_vov_extension, which contains some Altair Engineering extension and supports the option -F to dump the dependencies into Tcl file.
2. The resulting Tcl file is then mapped into a flow by means of the script \$(VOVDIR)/tcl/vtcl/vovmaketoflow.tcl.

USAGE:

```
% vovmake [VOVMAKEOPTIONS] [gmake_options]
```

VOVMAKEOPTIONS:

```
-help                -- Print this help.
-clean              -- Just cleanup the generated files.
-nocleanup          -- Do not cleanup temporary files
                    (for debugging).
-gmake <GMAKEBIN>   -- Use specified gmake binary
                    This must be one with the Altair Engineering
                    extensions.
                    Default gmake_with_vov_extension
-build "options"     -- Options passed to vovbuild, with flow
                    $(VOVDIR)/tcl/vtcl/vovmaketoflow.tcl.
-version            -- Show gmake version and exit
-wrapper <dfltwrapper> -- Specify default wrapper for jobs in
                    flow. It is passed to vovbuild.
-run [ft|no]        -- Run Flowtracer to get more information about
                    input makefile. Default is ft.
```

EXAMPLES:

```
% vovmake -help

% vovmake install
% vovmake <SOME_MAKE_TARGET>
% vovmake -clean

% vovmake -nocleanup all
% vovmake -gmake ~/bin/my_gmake_with_vov_extension
% vovmake -wrapper vrt
% vovmake -build "-env BASE" install
```

```
To find the options of vovmaketoflow.tcl, you can use
% vovbuild -f $VOVDIR/tcl/vtcl/vovmaketoflow.tcl -- -help
```

Conversion Examples

Some simple example makefiles are available in \$VOVDIR/training/vovmake. The simplest example is called Makefile_1, as shown here:

```
##### Simple Makefile for training with vovmake

# Experiment 1:    % vovmake all
# Experiment 2:    % vovmake -wrapper vw all

all: dd ee

bb: aa
   cp aa bb

cc: bb
   cp bb cc

dd: cc
   cp cc dd

ee: cc
   cp cc ee
```

```
% cp $VOVDIR/training/vovmake/Makefile_1 Makefile
% vovconsole -view graph -set System:nodes &
% touch aa
% vovmake
% vovforget -allnodes
% vovmake -wrapper vw
```

vovmake Uses GNU gmake with Some Extensions

The **vovmake** script uses `gmake_with_vov_extension` to parse the Makefile and dump it into a Tcl script. The program `gmake_with_vov_extension` is a variant of GNU-make 3.81. In compliance with the GNU license, the source code for `gmake_with_vov_extension` is included in the distribution in \$VOVDIR/src/vmake381_src.tgz.

Configure vovmake

The behavior of `vovmake` can be configured with a file called `vovmake.config.tcl`, which is also useful to augment the information already contained in the Makefile, for example by adding new dependencies or resource constraints.

The name of the configuration file is `vovmake.config.tcl` but this value may be overridden with the environment variable `VOVMAKE_CONFIG`, which can be used to specify a full path.

The search path for the file contains the following directories:

- `./` (working directory of `vovmake`)

- ../ (parent directory of vovmake)
- PROJECT.swd/ (project directory)
- \$VOVDIR/local

The procedures that belong in a `vovmake.config.tcl` file are the following:

- `vovmakeTarget`
- `vovmakeAddInputForTarget`
- `vovmakeAddOutputForTarget`
- `vovmakeSetWrapperForTool`
- `vovmakeConfig`

In the following, fragments of Makefiles are shown in this font while fragments of `vovmake.config.tcl` will be shown in this different font.

Procedure `vovmakeTarget`

The procedure `vovmakeTarget` defines additional information for a specific target. For example, it can define which licenses are required to update the target. The target name has to be the same as the one specified in the Makefile, or a glob expression.

```
chip.plan: chip.netlist
runPlanner chip

# The floorplanner requires a special license and 5GB of RAM.
vovmakeTarget chip.plan -res "License:floorplanner RAM/5000"

# Also specify a job name and an expected duration.
vovmakeTarget *.plan -name "Planner" -xdur 3h
```

Some targets can be skipped, because they do not map into actionable scripts:

```
all: place route

# There is no need to add the 'all' target to the flow.
vovmakeTarget all -skip

%.o:%.c

vovmakeTarget *.o -name Compile -res "RAM/20"
```

Procedures `vovmakeAddInputForTarget` and `vovmakeAddOutputForTarget`

The procedures `vovmakeAddInputForTarget` and `vovmakeAddOutputForTarget` can be used to declare more I/O dependencies for a target.

```
chip.plan: chip.netlist
runPlanner chip

# The floorplanner generates also a report file
vovmakeAddOutputForTarget chip.plan chip.rpt chip.log

# Also floorplanner uses a 'planner.constraints' file as input.
vovmakeAddInputForTarget chip.plan planner.constraints
```

Procedure vovmakeSetWrapperForTool

This procedure takes two arguments, a tool and a wrapper. By default, vovmake uses the default wrapper for all jobs that have simple (one-line) makefile rules and the wrapper "vov" for all jobs that have complex rules.

```
# Simple example of invocation.  
vovmakeSetWrapperForTool cp vw
```

Procedure vovmakeConfig

The procedure vovmakeConfig changes the behavior of vovmake. It takes two arguments, a name and a value.

```
# Example of each meaningful invocation of vovmakeConfig  
vovmakeConfig strictTracing 1  
vovmakeConfig doRun 0  
vovmakeConfig setname "abc"  
vovmakeConfig wrapper,default "vw"  
vovmakeConfig res,default "RAM/200"  
vovmakeConfig env,default "SNAPSHOT"  
vovmakeConfig wrapper,tool,cp "vrt"; # This can also be called  
vovmakeSetWrapperForTool
```

Makefile to FDL Utility

Makefile to FDL is a utility to quickly create a new Flow Description Language for projects that already have a working Makefile. This utility reads a Makefile and then generates an output FDL file, which is the FlowTracer equivalent of the Makefile. This utility provides a quick way to get started with a "near enough" FDL equivalent.

Makefiles can be very complex, which can affect the results. We strongly recommend checking the FDL output for accuracy and completeness. For example, it is known that self-generating Makefiles are not supported; FDL files generated from self-generated Makefiles are not guaranteed to be a complete equivalent of the original Makefile.

Access and Set Up Makefile to FDL

Access Makefile to FDL utility through the vovconsole.


1. Go to the **Tools** menu and then select the Makefile to FDL menu item.
A dialog opens, which enables you to specify a working directory, an input Makefile and an output FDL file.
2. Enter the working directory or navigate to it using the ellipsis (...) button on the right.



Note: The working directory is often the directory containing the Makefile. It allows relative pathnames in the Makefile to be resolved.

3. Use the Makefile text field to enter the path to the Makefile to be translated. Ensure that the Makefile is readable.
The Makefile loads into the left pane.
4. Enter the pathname to be used to store the output FDL file (default: FDL.out).
The system is now ready to generate FDL.

Generate FDL


1. To generate FDL, click **Convert** at the bottom of the dialog.
The FDL is shown in the right pane. The FDL will then be generated. Details can be viewed as follows:
 - Clicking on a line in the FDL pane on the right will cross-highlight the corresponding rule in the Makefile from which the FDL line was derived.
 - Clicking a line in the Makefile, highlights the corresponding line in the FDL.
- 
- Note:** Cross-highlighting may not work for all lines, but it will work for the most significant lines in the FDL and the Makefile.
2. If satisfied with the FDL, click **Save FDL** to capture it into the named output file for use with FlowTracer.
 3. Optional: Click **Clean** to delete the intermediate files that were created during FDL generation. We recommend reviewing the FDL and ensure that all dependencies and generation rules have been captured correctly.

Advanced Tasker Topics

Hardware Resources

All taskers offer a predefined set of hardware resources that can be requested by jobs.

All taskers offer a predefined set of hardware resources that can be requested by jobs. These resources are listed in the following table.

Hardware Resource	Type	Description
ARCH	STRING	The VOV architecture of the machine, for example "linux64", "win64", "armv8"
CORES	INTEGER	Consumable resource: the number of logical CPUs/processors used by a job.
CORESUSED	INTEGER	The total number of cores used by the running jobs. It is assumed that each job uses at least one core.
CLOCK	INTEGER	The CPU-clock of at least one of the CPUs on the machine in MHz. If the machine allows frequency stepping, this number can be smaller than expected.
GPUS	INTEGER	<div> Note: The environment variable "VOV_GPU_SUPPORT_ENABLE" must be set in the tasker environment to enable GPUS support. The value of the environment variable is ignored.</div> <p>This will consume the specified number of GPUS resources and, in addition, the associated GPUs' GPU:<UUID>, GPU:<Model_Name> and GPU:RAM resources.</p> <p>Example:</p> <pre>nc run -v 1 -r GPUS/1 - sleep 10</pre>
GPU: <uuid>	INTEGER	Request GPUs by device name (UUID). This will consume one GPUS resource, and in addition, the associated GPUs' GPU:<UUID>, GPU: <Model_Name> and GPU:RAM resources.

Hardware Resource	Type	Description
		<p>Example:</p> <pre>nc run -v 1 -r GPU:GPU-ab3207e3-6dff-fcbe-d93b-0f91cb2d45c3/1 --sleep 10</pre>
GPU: <model_name>	INTEGER	<p>Request GPUS(s) by GPU model name. This will consume the specified number of GPU:<Model_Name> resources and, in addition, the specified number of GPUS resources, the corresponding GPU:<UUID> resources, and the corresponding GPU:RAM resource quantity.</p> <p>Example:</p> <pre>nc run -v 1 -r GPU:Quadro_K3100M/1 -- sleep 10</pre>
GPU:RAM	INTEGER	<p>This will request a GPU and the requested GPU:RAM from it, 1 GPUS resource and, in addition, the associated GPUs' GPU:<UUID> and GPU:<Model Name> resources.</p> <p>Example:</p> <pre>nc run -v 1 -r GPU:RAM/1024 - sleep 10</pre> <p>Note that even though the entire GPU:RAM might not be requested, the entire GPUS is consumed.</p>
GROUP	STRING	The tasker group for this tasker. Each tasker can belong to only one tasker group.
HOST	STRING	The name of the host on which the tasker is running. Typically this is the value you get with <code>uname -n</code> , except only the first component is taken and converted to lowercase, so that if <code>uname -n</code> returns <code>Lnx0123.my.company.com</code> the value of this field will be <code>lnx0123</code> .
LOADEFF	REAL	The effective load on the machine, including the self-induced load caused by jobs that just started or finished.
L1	REAL	On UNIX, the load average in the last one minute.
L5	REAL	On UNIX, the load average in the last five minutes.
L15	REAL	On UNIX, the load average in the last fifteen minutes.
MACHINE	STRING	Typically the output of <code>uname -m</code> .
MAXNUMACORES	INTEGER	Highest total number of NUMA cores in a single NUMA node.

Hardware Resource	Type	Description
MAXNUMACORESFRE	INTEGER	Highest number of free cores in a single NUMA node. Note that free NUMA cores are correctly accounted for only if the user specified -jpp pack or -jpp spread for all jobs on the tasker.
NAME	STRING	The name of the tasker.
OS	STRING	The name of the operating system: "Linux" or "Windows".
OSCLASS	STRING	This can be <code>unix</code> or <code>windows</code> .
OSVERSION	STRING	The version of the OS. On Linux, this can usually be found in <code>/etc/system-release</code> .
OSRELEASE	STRING	Typically the output of <code>uname -r</code> .
PERCENT	INTEGER	Consumable resource: The percentage of the machine that is still available.
POWER	INTEGER	The effective power of the tasker, after accounting for both raw power and the effective load.
RAM	INTEGER	A consumable resource expressing the remaining RAM available to run job: <code>RAMTOTAL-RAMUSED</code> , in MB.
RAMFREE	INTEGER	The amount of RAM available to run other jobs. This metric comes from the OS, and on linux it includes both free memory and buffers. In MB.
RAMTOTAL	INTEGER	The total amount of RAM available on the machine, in MB.
RAMUSED	INTEGER	The aggregate quantity of RAM used by all jobs currently running on the tasker, in MB. For each job, the amount of RAM is calculated as the maximum of the requested RAM resource (<code>REQRAM</code>) and the actual RAM usage of the job (<code>CURRAM</code>).
RELEASE	STRING	On Linux machines, this is the output of <code>lsb_release -isr</code> , with spaces replaced by dashes. For example, <code>CentOS-6.2</code>
SLOT	INTEGER	A consumable resource indicating how many more jobs can be run on the tasker.
SLOTS	INTEGER	Same as SLOT
SLOTSUSED	INTEGER	Corresponding to the number of jobs running on the tasker.

Hardware Resource	Type	Description
STATUS	ENUMERATED TYPE	Possible values are BLACKHOLE, BUSY, DEAD, DONE, FULL, OVRLD, NOLIC, NOSLOT OK, PAUSED, READY, REQUESTED, SICK, SUSP, WARN, WRKNG
SWAP	INTEGER	A consumable resource. The swap space in MB.
SWAPFREE	INTEGER	The amount of free swap.
SWAPTOTAL	INTEGER	Total about of swap configured on the machine.
TASKERNAME	STRING	Same as <i>NAME</i>
TASKERHOST	STRING	Same as <i>HOST</i>
TIMELEFT	INTEGER	The number of seconds before the tasker is expected to exit or to suspend. This value is always checked against the expected duration of a job.
TMP	INTEGER	On UNIX, free disk space in /tmp, in MB.
USER	STRING	The user who started the vovtasker server, which is usually the same user account associated with the vovserver process.
VOVVERSION	STRING	The version of the vovtasker binary (such as '2015.03').

Request Hardware Resources

Each job can request hardware resources.

 **Note:** The consumable resources are CORES, CPUS, PERCENT, RAM, SLOT, SLOTS, and SWAP.

- To request a machine with the name *bison*, request *NAME=bison*. To request any linux64 machine, request *ARCH=linux64*.
- Consumable resources are added together. For example *-r CORES/2 CORES/4 CORES/6* is a request for a total of 12 cores.
- If redundant resources are specified, the largest value will be taken. For example, if *-r RAMTOTAL#2000 RAMTOTAL#4000* is specified then *RAM#TOTAL4000* will be the resource that is used.

Request examples are listed in the following table:

Request Objective	Syntax for the Request
A specific tasker	<i>NAME=bison</i>
Not on bison	<i>NAME!=bison</i>

Request Objective	Syntax for the Request
One of two taskers	NAME=bison, cheetah
A preference: bison, if it is available; otherwise, cheetah	(NAME=bison OR NAME=cheetah)
A specific architecture, such as Linux	ARCH=linux
A specific tasker group, such as prodLnx	GROUP=prodLnx
2 GB of RAM	RAM/2000
Two cores	CORES/2
Two slots	SLOTS/2
Exclusive access to a machine	PERCENT/100
1 minute load less than 3.0	L1<3.0

HOST_OF_jobId and HOST_OF_ANCECEDENT

Sometime, a job needs to run on the same host as another job. This condition can be expressed with the special resource "HOST=HOST_OF_<jobid>" and "HOST=HOST_OF_ANCECEDENT".

If a job requires "HOST=HOST_OF_<jobid>", it will be run on the same host as the job with the specified id. If such job does not exist, the job will not run ever.

If a job requires "HOST=HOST_OF_ANCECEDENT", the scheduler looks for any of the jobs that generate at least one input of the job, and schedules the job on the same host. If the job has no antecedent, then the job will never run.

Examples of HOST= HOST_OF_ANCECEDENT

In the following example, both jobs are going to be executed on the same host.

```
J vw cp aa bb
R "HOST=HOST_OF_ANCECEDENT"
J vw cp bb cc
```

In the following example, the job j2 runs on the same host as job j1

```
R "RAM/10"
set j1 [J vw prepare_host_for_job]

R "RAM/1000 License:expensive HOST=HOST_OF_$j1"
set j2 [J vw run_big_job]
```

Resource Management

The Resource Map Set represents the collection of resource maps that have been assigned to a project and the constraints associated with such resources.

Resource Map Set

Each resource map is described in the file `resources.tcl`, by means of the Tcl procedure `vtk_resourcemap_set`. This procedure requires the following arguments:

1. The name of a resource (in the form "type:name" with optional "type:").
2. The number of units available for the resource, which is either a positive integer or the keyword "unlimited" (case insensitive).
3. The third argument is optional. If it exists, it is used to indicate the name of a resource expression into which the first resource is to be mapped.
4. The fourth argument is also optional; it represents the number of units for that resource that are currently in use. This argument is intended to be used only by [vovresourced](#).
5. The fifth argument is also optional; it represents the expiration date that show when this resource expires (and is automatically removed from the set).

Example

```
# -- We have 4 licenses of DesignCompiler (Synopsys)
vtk_resourcemap_set License:dc_shell -max 4

# -- We have 1 license for PathMill (Synopsys), it can only
# -- run on Linux machines
vtk_resourcemap_set License:pathmill -max 1 -map linux

# -- Specify a resource as the sum of 2 other resources
vtk_resourcemap_set License:msimhdl -sum -map 'License:msimhdlsim OR
License:msimverilog'
```

Also in this Section

Resource Mapping

As the vovservers determines which tasker is most suited to execute a particular job, it performs a *mapping* of the job resources, followed by a *matching* of the mapped resources.

When dispatching a job, the vovservers does the following:

- Gets the list of resources required by a job.
- Appends the resource associated with the priority level, such as `Priority:normal`.
- If it exists, it appends the resource associated with the name of the tool used in the job (reminder: the tool of a command is the tail of the first command argument after the wrappers). The tool resource has type `Tool` and looks like this: `Tool:toolname`.

- Appends the resource associated with the owner of the job, such as `User:john`.
- Appends the resource associated with the group of the job, such as `Group:time_regression`.
- Expands any special resource, i.e. any resource that starts with a "\$".
- For each resource in the list, the vovservers looks for it in the resource maps. If the resource map is found and there is enough of it, that is, the resource is available, the vovservers maps the resource. This step is repeated until one of the following conditions is met:
 - The resource is not available. In this case, the job cannot be dispatched and is left in the job queue.
 - A cycle in the mapping is detected; in this case the job cannot be dispatched at all and is removed from the job queue.
 - The resource is not in the resource map.
- VOV appends the resource associated with the expected job duration to the final resource list. For example, if the job is expected to take 32 seconds, the resource `TIMELEFT#32` will be appended.
- Finally, the vovservers compares the resulting resource list with the resource list of each tasker. If there is a match - all resources in the list are offered by the tasker - the tasker is labeled as eligible. If there is no eligible tasker, the job cannot be dispatched at this time and remains in the queue; otherwise, the server selects the eligible tasker with the greatest effective power.

Local Resource Maps

Resource maps can be designated as *local*, using the `local` flag.



Important: This flag is only available and supported for a FlowTracer installation, utilizing `vovwxd` and an LSF interface.

Resource maps designated as local will be managed on the "local" (FT) side of the `vovwxd` connection instead of the normal case where resource specifications are expected to be managed on the base queue side.

For example, to limit jobs to running 5 at a time from a specific FlowTracer project, do the following:

1. Enable local resources with run: `vovservermgr configure vovwxd.localresources 1`



Note: Alternatively, you can add the following to the `policy.tcl` file:

```
set config(vovwxd.localresources) 1
```

2. Create the local resource.

- a. Run `vovresourcemgr set mylocallimit -max 5 -local`



Note: Alternatively, you can add the following to the `resource.tcl` file:

```
vtk_resourcemap_set mylocallimit -total 5 -local
```

This results in limiting running jobs with the local resource `mylocallimit` to a maximum of 5 jobs at a time.

For example, FDL to use a local resource named "mylocallimit:

```
R mylocallimit  
J vov /bin/sleep 0
```

Limitations on the number of Resource Expressions

While the number of OR expressions allowed in a job resource request is limited and controlled by a policy setting, the number of AND expressions including plain expressions without an explicit AND, also has limits.

The scheduler evaluates the resource expressions counting them as it goes. For example a list of 4 separate resource requests (without maps) will result in a max count of 7; 4 from the explicit resource requests and 3 from the automatically added resources (Group, Priority and User). Traversal into a resource map increments the count and a return from a resource map restores the count to value prior to the traversal into the map.

During this traversal, OR operators are recorded and used to influence scheduler operation. The count represents the cumulative depth of the resources. Whenever the count exceeds 30, any traversal that increases the depth is curtailed. This is done silently.

Any OR operators that occur at a depth deeper than 30 are ignored and those scheduling solution are effectively ignored by the scheduler resulting in unexpected behavior.

Large numbers of resource expressions do impact scheduler performance. The general guidance is to keep the explicit resource expressions to fewer than 8 including any mapped ones.

For applications that need a much larger number and where the depth may exceed the 30 limit, it is recommended to place the OR operators early in the resource requests (for example, the left hand side) and to place large numbers of ANDed resource into a resource map.

Resource Reservation

Resource reservation ensure that specific groups always has access to licenses, regardless of the number of jobs.

For example, if you have 10 licenses of spice but you want to make sure that the 'Production' group has always access to 2 of those licenses, you can use resource reservation.

A resource map can be reserved in whole or in part for a group, a user, or a specific job. When a resource is reserved for a group, jobs in other groups cannot use that resource. Similarly, the same happens for user reservation.

With resource reservation, you are willing to take a hit in the overall utilization of the resources in order to provide instant access to a specific group or user. The reserved resources remain idle even if there is demand for those resources by other groups or users. If you care about overall utilization, consider the other mechanisms of FairShare and preemption.

This functionality is available only to ADMIN users by means of the procedure `vtk_resourcemap_reserve` which accepts the following arguments:

ResourceName	The name of the resource you want to reserve.
TypeOfReservation	One of the following strings "USER" "GROUP" "JOBID" "JOBCLASS" "JOBPROJ"
NameOfUserOrGroupOrJob	Depending on the value of the previous argument, this is the name of the user or the name of a group or the ID of the job for which the resource map is to be reserved. The reservation for a specific job is used by the preemption daemon.
HowMany	The quantity of resources to be reserved. If this is greater or equal than the total available quantity, then the whole resource map is reserved.
Duration	The duration of the reservation. After the expiration of the reservation, the resources become available to all.
Explanation	A message to document why this reservation has been placed.

Examples of vtk_resourcemap_reserve

```
# Reserve 1 License:abc to user John for the next 3 days.
vtk_resourcemap_reserve License:abc  USER john      1 3d  "decided by the boss"

# Reserve 3 licenses for a group
vtk_resourcemap_reserve License:abc  GROUP  /app/alpha  3  8h  "agreed by the board"

# Reserve 3 licenses for a jobclass
vtk_resourcemap_reserve License:abc  JOBCLASS  hsim  3  8h  "as discussed in meeting"
```

Cancel a Resource Reservation

To cancel all resource reservations, use `vtk_resourcemap_clear_reservations`

```
# Cancel all reservations on a resource.
vtk_resourcemap_clear_reservations License:abc
```

A possible way to cancel a specific reservation is to make a new reservation that expires very quickly.

```
# Cancel a reservation for user john.
vtk_resourcemap_reserve License:abc  USER john      1 2s  "cancel"

# Cancel a reservation for a group
vtk_resourcemap_reserve License:abc  GROUP  /app/alpha  3  2s  "cancel"

# Cancel a reservation for a jobclass
vtk_resourcemap_reserve License:abc  JOBCLASS  hsim  3  3s  "cancel"
```

Delete Resources

To delete a resource map, use the Tcl procedure `vtk_resourcemap_delete` in `vovsh`. Resource maps may be deleted by ID or by name.

For example:

```
# Delete a resource by name
vtk_resourcemap_delete License:dc_shell

# Delete a resource by id
vtk_resourcemap_forget 00023456
```

Resource maps are automatically deleted after they expire, as soon as no job is using the resource.

The `License:` resources that are maintained by `vovresourced` and by `Allocator` are created with relatively short (typ. 5m) expiration times, but these are intended to be several times the duration at which they are refreshed by the daemon from license in-use information.

Rarely, you may observe jobs where `nc info` or the browser UI will show waiting on INFO: being deleted. This is shown when a job is waiting on a resource map that is in the process of being deleted or expiring.

Resource Daemon Configuration

vovresourced

Table 1:

Working directory	<code>vnc.swd/vovresourced</code>
Config file	<code>vnc.swd/vovresourced/resources.tcl</code>
Auxiliary config file	<code>\$VOVDIR/local/resources.tcl</code>
Info file	<code>vnc.swd/vovresourced/resourced.pid</code>

The daemon `vovresourced` is the main agent that defines the resources of the `vovservers`. The configuration file is `resources.tcl`, which is located in the server configuration directory. This file defines which resources are to be used by the server by calling the procedure `vtk_resourcemap_set`. Examples are available in the `vnc.swd/resources.tcl` file.

The procedures `vtk_flexlm_monitor` and `vtk_flexlm_monitor_all` are used to define resources that are derived from licenses. If the `resources.tcl` file calls for monitoring FlexNet Publisher features with the command `vtk_flexlm_monitor`, or when Monitor receives notification of an event from LICMON, `vovresourced` then retrieves the information about the event. Refer to the `vnc.swd/resources.tcl` file for examples.

Refresh Rate

The frequency of the checks can be configured in the `resources.tcl` file as shown below:

```
set RESD(refresh) 10000;    #Set refresh time to 10,000 milliseconds.
```

Starting vovresourced

The program `vovresourced` is normally started automatically by the server. `vovresourced` can also be invoked manually from any directory. After reading the `vovresourced` configuration file, `vovresourced` then reads the auxiliary configuration file `$VOVDIR/local/resources.tcl` (if the auxiliary file has been created). When the `resources.tcl` file is changed, the `vovresourced` daemon is restarted with the following commands:

```
% vovproject reread          ; # Generic method for any vovserver.  
% nc cmd vovproject reread   ; # Specific for Accelerator.  
% ncmgr reset                ; # Specific for Accelerator.
```

vovresourcemgr

`vovresourcemgr` is a utility for managing VOV resource maps. It may be used to create, modify, forget, and reserve resource maps.

The `vovresourcemgr` utility command connects to and acts on the VOV project enabled in the shell where it is launched. To act on Accelerator, use `vovproject enable vnc`, or precede it with `nc cmd` as shown in the examples below.

```
vovresourcemgr: Usage Message  
  
USAGE:  
  % vovresourcemgr COMMAND [options]  
  
COMMAND is one of:  
  show          Show summary info about all resource maps  
  show [R1..RN] Show info about specified resource map(s)  
  matches RESMAP Show license matching info  
  ooq RESMAP    Show out of queue license handles  
  create RESMAP map-options  
                  Create a new resource map  
  set RESMAP map-options  
                  Create a new or modify an existing resource map  
  reserve RESMAP TYPE WHO HOWMANY HOWLONG WHY [-exclusive]  
                  Place a reservation on a resource map  
  forget [-force] R1 [R2..RN]  
                  Remove resource map(s) from the system  
  
MAP-OPTIONS:  
  -expire specify expiration (timespec) relative to now  
  -max     specify quantity  
  -map     specify map-to value  
  -rank    specify rank when setting  
  -nooq    do not track out-of-queue  
  -local   specify that this is a local resource (when using vovwxd)
```

For reserve, TYPE is one of: USER, GROUP, JOBCLASS, JOBPROJ, JOBID.

EXAMPLES:

```
% vovresourcemgr show
% vovresourcemgr show Limit:abc
% vovresourcemgr matches Limit:abc
% vovresourcemgr create License:spice -max 8
% vovresourcemgr set License:spice -max 10
% vovresourcemgr set License:spice -map "Policy:spice"
% vovresourcemgr ooq License:spice
% vovresourcemgr reserve License:spice USER john,jane 3 3d ""
% vovresourcemgr reserve License:spice USER bill 1 1w "" -exclusive
% vovresourcemgr forget License:spice
% vovresourcemgr forget -force License:spice
```

```
% vovresourcemgr show ; # show defined resource maps
% vovresourcemgr show R ; # show details of resource map R
% vovresourcemgr set ; # set/create a new resource map
% vovresourcemgr reserve ; # reserve resource map for user or group
% vovresourcemgr ooq ; # show Out-Of-Queue use (for Accelerator)
% vovresourcemgr forget ; # remove a resource map from vovserver
```

Dynamic Resource Map Configuration

Persistent resource maps are defined in the `resources.tcl` configuration file for a project. The `vovresourcemgr` command is useful to make changes to the resource maps on the fly.



Note: Unlike resource maps defined in `resources.tcl`, changes made with `vovresourcemgr` do not persist across restarts of `vovserver`.

The `create` command checks for existence of the named resource map and exits with a message if it already exists. The `set` command will create or replace an existing resource map with the given values with no confirmation.

The following example creates a new resource map named `Limit:spice`, which is created with a quantity of 10 and an empty map-to value.



Note: `nc cmd` shows this example is to act on Altair Accelerator.

```
% nc cmd vovresourcemgr set Limit:spice 10 ""
```

Resource Map Reservation

Following is an example of using `vovresourcemgr` to place a reservation on a resource map. In this case, two of the resource maps called `License:spice` are reserved for user `john` for an interval of 4 hours. The resource map reservation will automatically expire after 4 hours.

```
% nc cmd vovresourcemgr reserve License:spice USER john 2 4h "library char"
```

Workaround for Misspelled Resource


Sometimes users submit jobs to Altair Accelerator that request nonexistent resources, which causes the jobs to be queued indefinitely. Such jobs can be made to run by creating the missing resource, or by modifying the jobs to request the correct resources.

The following example creates four temporary `License:sspice` resources that are mapped to the correct `License:spice` resource. `License:sspice` is an incorrect request - that resource does not exist. A temporary resource is created with that name that will be mapped to the correct resource, `License:spice`

```
% nc cmd vovresourcemgr create License:sspice -max 4 "License:spice"
```

Forget Unneeded Resource Maps

Continuing the above example - the temporary resource map may be removed after the malformed jobs have run. Or, you can just let it expire.

 **Note:** There is no confirmation; the command acts immediately.

```
% nc cmd vovresourcemgr forget License:sspice
```

Policies and Resources

You can define site-specific policies on how resources are allocated to various projects by creating a file called `$VOVDIR/local/resources.tcl`. This file contains the definition of a procedure called `vtk_resources_policy_hook` and can be used to limit the amount of resources assigned to a given project.

The default definition for this procedure is:

```
# Default definition
proc vtk_resource_policy_hook { projectName resName max } {
    return $max
}
```

```
}
```

For example, if you have 70 license of SPICE and do not want any project to have more than 20, except special projects, you can write the following definition for `vtk_resources_policy_hook`:

```
# Example of definition of vtk_resource_policy_hook
proc vtk_resource_policy_hook { projectName resName max } {
    switch -glob $resName {
        "spice*" {
            switch -glob $projectName {
                "special*" { return $max }
                default {
                    return [expr $max > 20 ? 20 : $max ]
                }
            }
        }
        default {
            return $max
        }
    }
}
```

Add Resources

Generic resources are added to Altair Accelerator via the `vtk_resourcemap_set` procedure call:

```
vtk_resourcemap_set <name> <quantity> [map]
```

Below are two examples:

```
vtk_resourcemap_set myres 2
vtk_resourcemap_set myunlimitedres UNLIMITED
```

Example: Node Locked License

In the scenario of this example, a license does not utilize FlexNet Publisher or another dynamic license management solution, but does require a tool to run only on one specific host. In this example, the tool is `spice`, the host is `pluto`, and the license is for two concurrent instances of `spice`. Following are the steps to correctly handle this constraint:

1. Choose a name (in the form `name` or `type:name`) to represent the node locked resource (such as `License:spice_pluto`) and a name to be announced to the users (such as `License:spice`). In this way, it is hidden to the users that the `spice` license is locked to a given node. Also, if there is an upgrade to a floating license or multiple node-locked licenses, that can be carried out without having to announce it to the users.
2. Add the following lines to the `resources.tcl` file:

```
vtk_resourcemap_set License:spice UNLIMITED License:spice_pluto
vtk_resourcemap_set License:spice_pluto 2 pluto
```

```
% nc cmd vovproject reread
```

3. Let the job declare that it requires the resource `License:spice`; use option `-r` in `nc run` as shown below:

```
% nc run -r License:spice -- spice -i chip.spi
```

CGI Interface

Webpages in the Altair Accelerator programs are typically produced using `.cgi` files ("Common Gateway Interface").

The `.cgi` files reside in 4 specific locations:

- `$VOV_PROJECT_NAME.swd/cgi` (with respect to the server working directory)
- `./cgi` (with respect to the server working directory)
- `$VOVDIR/local/cgi`
- `$VOVDIR/etc/cgi`

The scripts are usually Tcl and you can find a list at `/cgi/listcgi.cgi`. The scripts are executed by a `vovserver` subprocess, use the current working directory, and use the environment of the server.

If no script is found, a simple error page is reported. To get a list of all available CGI scripts, click `${VOVDIR}/etc/cgi/list.html`.

In a CGI URL you can specify the option `timeout=N` where `N` is the time in seconds the server allows for a CGI script to complete. The default for `N` is 120 (2 minutes). Example:

```
http://host:port/cgi/listcgi.cgi?timeout=600
```

Another commonly used option is `redirect=newurl` which tells the server to produce a page and then redirect automatically to the new URL. For example:

```
http://host:port/welcome?redirect=/project
```

Write CGI Scripts

These are guidelines for writing cgiscripts to work with `vov`.

The script must generate a legal HTTP reply, taking into account that `VOV` already provides the first line which is always

```
HTTP/1.0 200 OK
```

Typically, the script must begin by printing to `stdout` the following data:

```
Content-Type: text/html
Content-Length: the length of the HTML page
<<-- Will not work without this empty line!!
The content of the HTML page
```

This task is greatly simplified by:

- writing the CGI script as a Tcl script;
- taking advantage of the procedures defined in `VOVDIR/tcl/vtcl/vovhtmlgen.tcl`.

Simplest CGI script

```
#!/bin/csh -f
# The rest is -*- Tcl -*- \
exec vovsh -f $0 $*

# Try this by means of the URL /cgi/ciaobello.cgi

VOVHTML_START
HTML {
    HEAD { TITLE "Ciao bello example" }
    BODY {
        OUT "CIAO BELLO"
    }
}
VOVHTML_FINISH
```

Retrieving VOV information

You can use the vtk library to get information from the trace. Here is a simple example, that also shows a way to generate tables.

```
#!/bin/csh -f
# The rest is -*- Tcl -*- \
exec vovsh -f $0 $*

# This script responds to a URL of type: /cgi/getnode.cgi?nodeid
set id [shift]

VOVHTML_START
HTML {
    HEAD { TITLE "Get Node $id" }
    BODY {
        H1 { OUT "Node $id" }
        set nodeInfo [vtk_node_get $id]
        TABLE {
            foreach field { status barrier timestamp type } {
                TR {
                    TH { OUT $field }
                    TD { OUT [shift nodeInfo] }
                }
            }
        }
        OUT "CIAO BELLO"
    }
}
VOVHTML_FINISH
```

VOV CGI Procedures

This document describes the Tcl procedures provided by VOV for creating HTML pages using CGI programs. CGI (Common Gateway Interface) is a way of using external programs to dynamically create pages which are returned by your project's vovserver to a user's web browser.

This is greatly simplified by:

- Writing the CGI script as a Tcl script;
- Using the VOV procedures described in this document

The CGI programs may be written in any language which can produce an executable program that makes text output.


General-purpose web servers such as Apache and Microsoft IIS support different languages such as Perl, Tcl, Python, Php, Visual Basic and C. The VOV server can also use a CGI programs written in any language, but it is best if your CGI scripts are written in Tcl, because then your programs may use the `vtk_*` Tcl API functions to interact with the vovserver.

Your Tcl script must generate a correct HTTP reply, considering that VOV already provides the first line which is always:

```
HTTP/1.0 200 OK
```

Typically, the script must print to stdout the following data:

```
Content-Type:  text/html
Content-Length: the length of the HTML pagetext of the HTML page
```


 **Note:** The blank line following the `Content-length:` is required for the HTML to be correctly displayed.

When using the VOV CGI procedures, you will not need to be concerned with this type of detail. The VOV CGI procedures help you create the text, and automatically compute the proper length, and close all tags correctly.

HTML is a hierarchical language, where the various types of tags may only appear in certain contexts. For example, it is an error to use the `` (list item) tag outside the context of a `` (unordered list) or `` (ordered list). Each tag `<TAG>` defines such a context, and must be properly closed by the corresponding `</TAG>` for the HTML to be correct.

In practice, modern browsers attempt to present even incorrect HTML, so that many simple tags like `<P>` and `` do not need to be closed properly. For tables, however, closing the tags properly is essential for correct presentation. The VOV CGI procedures take care of this chore for you. The scope of each tag begins with the tag's procedure, and finishes with the final closing brace `}`.

VOV provides procedures to handle a subset of the tags available in the HTML specification, but this set is sufficient to quickly build informative, dynamically- created web pages about your project.

 **Important:** The names of these procedures are mainly taken from the type of HTML tag which they implement, to make them easy to remember for persons familiar with HTML. However, they are not HTML tags; each of the following is a Tcl procedure which takes zero or more arguments, so the script that you write using them must be in valid Tcl syntax.

VOV CGI Procedure Overview

Procedure	Descriptions
VOVHTML_START	Begin a new HTML page
VOVHTML_FINISH	Finish the current page
OUT	Insert text in page without newline
OUTLN	Insert text in page with newline
HTML	Build the HTML part of the page
TITLE	Set the page title
BODY	Build the main body of the page
COLOR	Set color of text
FONT	Specify font of text
SPAN	Build a span element
FORM	Begin an HTML form for user interaction
SCRIPT	Insert script code (e.g. Javascript) into page
OPTION	Build an option element
SELECT	Build a select element
VOV_SET_COOKIE	Set a cookie
VOV_GET_COOKIE	Retrieve a cookie's value
TABLE	Start a new table in the page
TR	Start a new row in the current table
TH	Insert a table header cell in the current table row
TD	Insert a table data cell in the current table row
HREF	Insert a hyperlink into the page
BR	Begin a new line
HR	Insert a Horizontal Rule into the page
H1	Format text as Header1 style (most emphasis)


Procedure	Descriptions
H2	Format text as Header2 style
H3	Format text as Header3 style
H4	Format text as Header4 style
H5	Format text as Header5 style
H6	Format text as Header6 style (least emphasis)
Other Formatting Items	Format text bold, italic, etc.

Example CGI Scripts

This section presents two example scripts written in Tcl. The first is an elaboration of the traditional 'Hello, World!' program, and the second shows how to use VOV's Tcl API to get information about a VOV project from its server.

Example -- Basic "Hello, World" CGI script

This script exploits a difference between the handling of comments in the Tcl language, and C-shell to produce an executable script written in Tcl. The C-shell starts, using -f to specify not reading your ~/.cshrc file, and then immediately executes the VOV 'vovsh' program, which is a Tcl-language shell. That program then receives the remainder of the file as a Tcl script. The Tcl interpreter sees the 'exec' line as a comment, but C-shell does not, and executes it.

 **Note:** The `-*-` on either side of the word 'Tcl' allows Emacs to recognize the file as a Tcl script.

```
#!/bin/csh -f
# For the -*- Tcl -*- interpreter the next line is a comment\
exec vovsh -f $0 $*

# This file is provided by the VOV install as $VOVDIR/etc/cgi/ciaobello.cgi
#
# Once you have created a project using 'vovproject create' and enabled it
# in your shell using 'vovproject enable your-project-name', you may use
#   vovbrowser
# to find the port on which your project's server listens for HTTP requests.
#
# You may view this CGI program's output by pointing your browser to the URL
#   http://your-server-host:your-project-port/cgi/ciaobello.cgi

VOVHTML_START                                ;# start a new page
HTML {                                        ;# begin the page's HTML
    HEAD { TITLE "Ciao bello example" }      ;# set page title
    BODY {                                    ;# begin page body
        H1 { OUT "CIAO BELLO" }              ;# "Ciao bello" is hello world
        in Italian
        H3 { OUT "It is now [clock format [clock seconds]]" }
        TABLE {
            TR {                               ;# first table row is the
table hdr
```

```

    TH { OUT "Example of a simple CGI script with VOV." }
  }
  TR BGCOLOR="white" {                                     ;# second table row, white
background
    TD {                                                    ;# only 1 table data cell is
in this row
        SMALL {                                             ;# make the following text
smaller
            PRE {                                           ;# and preformatted in
teletype font
                set fp [open $argv0] ;# open the file of this
script
                OUT [read $fp]      ;# insert its text into the
page
                close $fp          ;# close file nicely
            }
        }
    }
  }
}
VOVHTML FINISH ;# finish the page

```

Example -- Retrieving VOV Project Information

You can use VOV's Tcl API `vtk_*` procedures to get information from your project's flow. A `nodeid` is an integer which is VOV's internal database ID for an object.

Here is a simple example that also shows a way to generate tables.


```
#!/bin/csh -f
# For the -*- Tcl -*- interpreter the next line is a comment\
exec vovsh -f $0 $*

# This script responds to a URL of type: /cgi/getnode.cgi?nodeid
# see comments in preceding example for how to connect to your project's server
set id [shift]

VOV_HTML_START
HTML {
    HEAD { TITLE "Get Node $id" }
    BODY {
        H1 { OUT "Node $id" }
        set nodeInfo [vtk_node_get $id]
        TABLE {
            foreach field { status barrier timestamp type } {
                TR {
                    TH { OUT $field }
                    TD { OUT [shift nodeInfo] }
                }
            }
        }
        OUT "CIAO BELLO"
    }
}
VOV_HTML_FINISH
```

Procedure Descriptions

This section describes the VOV Tcl CGI procedures in more detail, describing the arguments and actions of each.

 **Note:** The names of these procedures are mainly taken from the type of HTML tag which they implement, to make them easy to remember for persons familiar with HTML. Again, they are not HTML tags; each of the following is a Tcl procedure which takes zero or more arguments, so the script that you write using them must be in valid Tcl syntax.

Pay special attention to this for any parameter which is named `args` or `script`. See the description of the `TABLE` procedure and the example scripts for more information.

VOVHTML_START {}

Description

The VOVHTML_START procedure begins a new HTML page. It opens a temporary file to receive the text of the page. This procedure **must** be called before any of the following ones.

Arguments

None

VOVHTML_FINISH {[contentType HTMLtype]}

Description

The VOVHTML_FINISH procedure closes a HTML page. It writes the HTML header information, then the text which was stored in that page's temporary file, and removes the file.

This is the procedure which actually returns the page text.

Arguments

1. `content-type` (optional); default is `text/html`

OUT {text}

Description

The OUT procedure adds the text of its argument to the page, **without a newline**.

Arguments

1. `text` text to insert into the page

OUTLN {text}

Description

The OUT procedure adds the text of its argument to the page, **with a newline**.

Arguments

1. `text` text to insert into the page

HTML {script}

Description

The HTML procedure builds the HTML section of the page, starting with the `<html>` tag, the text of the page, and then closes it properly with the `</html>` tag. It executes the Tcl script passed as its argument, and inserts the resulting text into the page.

Arguments

1. `script` Tcl script that builds the HTML of a page

TITLE {title}

Description

The TITLE procedure inserts its text as the page title in the HEAD section of the page.

Arguments

1. `title` text of page title

BODY {script}

Description

The BODY procedure executes the Tcl script, and inserts the text which the script returns as the page body in the BODY section.

Arguments

1. `script` Tcl script that builds text to form the page body

COLOR {color text}

Description

The COLOR procedure surrounds its text argument with HTML code so that it will display using the named color.

Arguments

1. `color` HTML color name or #code
2. `text` text to be displayed in `color`

FONT {args}

Description

The FONT procedure surrounds its text argument with HTML code so that it will display using the named color. All except the last argument are taken as attributes of the FONT tag.

Arguments

1. `args` text to be displayed in using the named `font`.

SPAN {args}

Description

The SPAN procedure defines a region of args to which a style rule may be applied.

Arguments

1. `args` variable-length argument list.

FORM {args}

Description

The FORM procedure defines an HTML form for user interaction.

Arguments

1. `args` attributes and Tcl script defining the form

SCRIPT {text}

Description

The SCRIPT procedure inserts its text as the code of a script into the current page. Typically, the code will be JavaScript.

Arguments

1. `text` text to be inserted as script code

OPTIONS {args}

Description

Inserts an OPTION element into the current SELECT element. Must be used in the scope of a SELECT.

Arguments

1. `args` variable-length argument list

SELECT {args}

Description

Inserts an SELECT element into the current FORM. Must be used in the scope of a FORM. The SELECT element forms a one-of-many choice using a drop-down list. Each item in the list is defined by an OPTION item.

Arguments

1. `args` variable-length argument list

VOV_SET_COOKIE {name value [expires]}

Description

Store a cookie in the browser memory for future reference.

Arguments

1. `name` the name of cookie to be stored
2. `value` the value of cookie to be stored
3. `expires` (optional) the expiration timestamp of the cookie. Without this argument, the cookie will survive the session, i.e., the cookie expires after all browser windows are closed.

VOV_GET_COOKIE {name}

Description

Retrieve a cookie value.

Arguments

1. `name` the name of cookie to retrieve

Returns

1. value the value of the cookie retrieved or blank("") if no cookie with name name is found

TABLE {attributes} script

Description

Defines a table in the current HTML page. The table may contain nested TR, TH, and TD procedures.

The HTML <table> tag may have attributes such as cellpadding=0 border="1" , etc. between the <table part and the closing >.

Example:

```
<TABLE ALIGN="CENTER" BORDER="1" { .... }
```

All items except the last one are taken as attributes. The final argument is evaluated as a Tcl script, and the resulting text is inserted into the page between the <table> and </table> tags.

Arguments

1. `attributes` for the table, such as ALIGN="CENTER" BORDER="1" etc.
2. `script` Tcl script that builds the rows of the table.

Returns

1. value the value of the cookie retrieved or blank("") if no cookie with name name is found

TR {attributes} script

Description

The TR procedure defines a row in a table. It must appear in the scope of a TABLE procedure.

Example:

```
TR { ...script... }
```

```
TR BGCOLOR="white" { ...script... }
```

Arguments

1. `attributes` optional attributes for the row (e.g. bgcolor)
2. `script` Tcl script that builds the content of the rows.

TH {attributes} script

Description

The TH procedure defines a **header** cell in a table. It must appear in the scope of a TR procedure. The text of a table header cell is formatted **bold** and is centered in the cell.

Example:

```
TH { OUT "Ciao" }
```

```
TH VALIGN="top" BGCOLOR="white" { ...script... }
```

Arguments

1. `attributes` optional attributes for the cell
2. `script` Tcl script that builds the content of the header cell.

TD {attributes} script

Description

The TD procedure defines a table data cell in a table. It must appear in the scope of a TR procedure. The text of a table data cell is formatted plain and is usually aligned to the top left of the cell.

Example:

```
TD { OUT "Some data" }
```

```
TD VALIGN="right" BGCOLOR="red" { ...script... }
```

Arguments

1. `attributes` optional attributes for the cell
2. `script` Tcl script that builds the content of the cell.

HREF {url label}

Description

The HREF procedure defines a hyperlink in the current page.

Arguments

1. `url` destination URL of the link
2. `label` visible label of the link, which appears in the current page

BR {}

Description

The BR procedure begins a new line in the current page.

By default, HTML text in a page is flowed, rather than fixed-format. Without tags to define the structure of the document, its text will be flowed together to fit into the current window size whenever you change your browser window.

Arguments

None

HR {}

Description	The HR procedure inserts a horizontal rule (line) into the current page. The width of the line will vary as the window displaying the page is resized.
Arguments	None

Section Header Procedures

HTML defines six section header styles, ranging from H1 through H6, with H1 being the most emphatic (largest point size, and bold-face), and H6 being the least emphatic. VOV provides the corresponding H1 through H6 procedures.

H1 {script}

The H1 procedure evaluates its script as a Tcl script, and inserts the resulting text, surrounded by H1 tags, into the current page.

Arguments: `script`

H2 {script}

The H2 procedure formats its text using the Header2 style.

H3 {script}

The H3 procedure formats its text using the Header3 style.

H4 {script}

The H4 procedure formats its text using the Header4 style.

H5 {script}

The H5 procedure formats its text using the Header5 style.

H6 {script}

The H6 procedure formats its text using the Header6 style.

Other Formatting Procedures

VOV CGI also provides procedures to support other common formatting requests, including:

- **B** display text **boldface**
- **EM** display text *emphasized*
- **TT** display text in TTY (fixed width) font
- **BIG** display text in larger size
- **SMALL** display text in smaller size
- **P** start a new paragraph
- **OL** start a ordered (numbered) list
- **UL** start an unordered (bulleted) list
- **LI** start a List Item in an ordered or unordered list
- **PRE** start section of pre-formatted text (displayed in TTY font, not flowed)

The procedures described are defined in `VOVDIR/tcl/vtcl/vovhtmlgen.tcl`.

Events

Every change in the trace is recorded as an *event*, which is then sent to all clients that subscribe to them.

To attach to the event stream use

```
vtk_event_control START
```

The client implicitly subscribes to all events. Each event can be retrieved with a call to

```
vtk_event_get $timeout eventDesc
# Example to show which fields are available
parray eventDesc
eventDesc(auxid)      = 1094842
eventDesc(auxstatus)  = 0
eventDesc(description) =
eventDesc(id)         = 473865
eventDesc(priority)   = 0
eventDesc(subject)    = NODE
eventDesc(verb)       = CONNECT
eventDesc(x)          = 1094842
eventDesc(y)          = 0
```

Each event is characterized primarily by a 'subject' and a 'verb'.

In order to reduce network traffic, a client application may subscribe to a subset of the events by adding filters. This is done with

```
vtk_eventfilter_append $subjectList $verbList
```

For example, to receive only the events that have to do with jobs that start, you would call

```
vtk_eventfilter_append "JOB" "START"
```

To keep the GUI up to date, all events are streamed asynchronously from the VOV server to the GUI. It is possible for a GUI to be slow with respect to the stream of events, causing some internal buffers to overflow. In such case, the VOV server generates an *overflow event* and stops sending events to the slow client until the client has caught up.

For more information on handling the events, please refer to the documentation for the `vtk_event*` procedures.

Monitor Events with voveventmon

An easy way to look at the event stream is with the utility `voveventmon`.

```
voveventmon: Usage Message

DESCRIPTION:
  voveventmon  --  A utility to show the event stream
                   Feel free to look at the source to
                   learn how you can yourself tap into
                   the event stream.
```

This utility is also used in vovresourced to allow asynchronous update of the resourcemap information as soon as LicenseMonitor has new information.

USAGE:

```
% voveventmon [OPTIONS]
```

OPTIONS:

```
-h                This help

-f <subjects> <verbs>
-filter <subjects> <verbs>
                    Define a filter for events.
                    For definitions of <subjects> and <verbs>
                    please refer to on-line documentation.

-m <N>
-maxevents <N>     Max number of events to show
                    Default is 1000.
                    For unlimited events, set <N> to 0.

-n <N>             Same as -m

-s <bool>          If set, show event statistics
-t <S>
-timeout <S>       Default is 10.
-host <HOST>       Host name override.
-port <PORT>       Port name override.
-project <PROJECT> Project name override.
```

FOR TESTING ONLY:

```
-d <DELAY_IN_MS>  Add a delay after processing each event.
```

EXAMPLES:

```
% voveventmon                -- Show 1000
% voveventmon -m 0            -- Unlimited events
% voveventmon -f FILE INFORM
% voveventmon -f LICDAEMON ALL
```

Here is some example output:

```
% voveventmon
1 NODE CONNECT 02709649 1132452736 {746499}
2 NODE CHANGE 00746499 1132452736 {"RUNNING\"}
3 NODE CONNECT 00745066 1132452736 {2709649}
4 NODE CONNECT 02709649 1132452736 {746500}
5 NODE CHANGE 00746500 1132452736 {"RUNNING\"}
6 NODE CONNECT 02709649 1132452736 {746501}
7 NODE CHANGE 00746501 1132452736 {"RUNNING\"}
8 NODESET DETACH 02699306 1132452737 {2709643 0 9}
9 NODESET DETACH 02699306 1132452737 {2709646 0 9}
10 NODE CONNECT 02709649 1132452737 {746502}
...
% voveventmon -f LICDAEMON CHANGE
...
```

Miscellaneous

Controlling Jobs on Tasker Host

From Tcl, you can control both the taskers and the jobs running on the remote taskers with the procedure `vtk_job_control`.

This is the main procedure to send controlling signals and modifications to jobs running on remote taskers. The argument `taskersId` can either be a legal VovId, "ALL", or the number 0 which is equivalent to "ALL". The argument `action` can be one of STOP KILL DEQUEUE SUSPEND RESUME SIGUSR1 SIGUSR2 SIGTSTP CHECK EXT MODIFY

- If `jobId` is 0, then all jobs on the specified tasker are affected.
- If `jobId` is the string "TASKER", then the tasker is stopped gracefully.
- If `jobId` is the string "TASKER/FORCE", then the tasker is stopped with force. Taskers can only be stopped.

Table 2: Options

Option	Action EXT	Action MODIFY
opt1	signalName	fieldName
opt2	procNameIncludeRx	newValue
opt3	procNameExcludeRx	unused



Note:

- STOP and KILL are equivalent
- SUSPEND may use colon delimiters to specify which signals to use and include/exclude lists, formatted as:
SUSPEND:[comma-delimited signal list]:[comma-delimited include list]:[comma-delimited exclude list] . See Examples.
- STOP may use colon delimiters to specify which signals to use, include/exclude lists and delay in seconds between sending signals, formatted as:
STOP:[comma-delimited signal list]:[comma-delimited include list]:[comma-delimited exclude list]:[delay between signals in seconds] . See Examples.
- The include and exclude lists used by STOP and SUSPEND are mutually exclusive; if both are specified the exclude list will apply.
- DEQUEUE does not reach the tasker and is processed by vovserver
- CHECK forces the taskers to scan the process table and gather information about the processes and send it to the vovserver
- EXT uses the script `vovjobctrl` to execute the job control. Check the documentation about `vovjobctrl` for more information about this type of job control.

Examples of vtk_job_control

```
# Stop all taskers
vtk_job_control ALL STOP TASKER

# Stop tasker 000123456 with force
vtk_job_control 000123456 STOP TASKER/FORCE

# Stop all jobs on tasker 000123456
vtk_job_control 000123456 STOP 0

# Stop job 223344 on tasker 000123456
vtk_job_control 000123456 STOP 223344

# Stop job 223344 running on any tasker
vtk_job_control ALL STOP 223344

Suspend all sleep jobs using SIGINT followed by SIGHUP on tasker 12345:
vtk_job_control 12345 SUSPEND:INT,HUP:sleep

Kill all jobs except sleep jobs using the default signal cascade with 2 seconds
between signals on tasker 23456:
vtk_job_control 23456 STOP::sleep:2
```

Customize the Browser Interface

The templates used to build the browser interface are in looked for in the following directories:

Directory	Description
html/	Computed with respect to the server configuration directory, for project specific templates.
./html/	Computed with respect to the server working directory, for project specific templates.
\$VOVDIR/local/html	For site specific templates.
\$VOVDIR/etc/html	Default installation templates.

The server caches the templates used to generate the HTML pages. To clear the caches, visit the /sanity page.

You may also use the command listed below in your own window to perform a sanity check.

```
% vovproject sanity
```

TclXML Parser

TclXML is an open source XML parser written in Tcl. Altair Accelerator ships TclXML with it, so no 3rd party installation is needed to use TclXML. For detailed information, please go to <http://tclxml.sourceforge.net/tclxml.html>.

The goal of the TclXML package is to provide an API for Tcl scripts that allows "Plug-and-Play" parser implementations; e.g. an application will be able to use different parser implementations without change to the application code.

The TclXML package provides a streaming, or "event-based", interface to an XML document. An application using TclXML creates a parser "object", sets a number of callback scripts and then instructs the parser to parse an XML document. The parser scans the XML document's text and as it finds certain constructs, such as the start/end of elements, character data, and so on, it invokes the appropriate callback script.

Parser Object (::xml::parser)

The ::xml::parser command creates an XML parser object. The return value of the command is the name of the newly created parser.

The parser scans an XML document's syntactical structure, evaluating callback scripts for each feature found. At the very least the parser will normalise the document and check the document for well-formedness. If the document is not well-formed then the -errorcommand option will be evaluated. Some parser classes may perform additional functions, such as validation. Additional features provided by the various parser classes are described in the section Parser Classes.

Parsing is performed synchronously. The command blocks until the entire document has been parsed. Parsing may be terminated by an application callback, see the section Callback Return Codes. Incremental parsing is also supported by using the -final configuration option.

The following table lists all options to the parser object. To find a complete explanation on all script arguments, please find online documentation at <http://tclxml.sourceforge.net/tclxml.html>.

Option	Script Args	Description
-attlistdeclcommand <script>	name attrname type default value	Specifies the prefix of a Tcl command to be evaluated whenever an attribute list declaration is encountered in the DTD subset of an XML document.
-baseurl URI	N/A	Specifies the base URI for resolving relative URIs that may be used in the XML document to refer to external entities.
-characterdatacommand <script>	data	Specifies the prefix of a Tcl command to be evaluated whenever character data is encountered in the XML document being parsed.
-commentcommand <script>	data	Specifies the prefix of a Tcl command to be evaluated whenever a comment is encountered in the XML document being parsed.
-defaultcommand <script>	data	Specifies the prefix of a Tcl command to be evaluated when no other callback has been defined for a document feature which has been encountered.
-defaultexpandinternalentities <boolean>	N/A	Specifies whether entities declared in the internal DTD subset are expanded with their replacement text. If entities are not expanded then the entity references will be reported with no expansion.
-doctypecommand <script>	name public system dtd	Specifies the prefix of a Tcl command to be evaluated when

Option	Script Args	Description
		the document type declaration is encountered.
-elementdeclcommand <script>	name model	Specifies the prefix of a Tcl command to be evaluated when an element markup declaration is encountered.
-elementendcommand <script>	name args	Specifies the prefix of a Tcl command to be evaluated when an element end tag is encountered.
-elementstartcommand <script>	name attlist args	Specifies the prefix of a Tcl command to be evaluated when an element start tag is encountered.
-endcdatasectioncommand <script>	-NONE-	Specifies the prefix of a Tcl command to be evaluated when end of a CDATA section is encountered. The command is evaluated with no further arguments.
-enddoctypecommand <script>	-NONE-	Specifies the prefix of a Tcl command to be evaluated when end of the document type declaration is encountered. The command is evaluated with no further arguments.
-entitydeclcommand <script>	name args	Specifies the prefix of a Tcl command to be evaluated when an entity declaration is encountered.
-entityreferencecommand <script>	name	Specifies the prefix of a Tcl command to be evaluated when an entity reference is encountered.
-errorcommand <script>	errorcode errmsg	Specifies the prefix of a Tcl command to be evaluated when a fatal error is detected. The error may be due to the XML document not being well-formed.

Option	Script Args	Description
		In the case of a validating parser class, the error may also be due to the XML document not obeying validity constraints. By default, a callback script is provided which causes an error return code, but an application may supply a script which attempts to continue parsing.
<code>-externalentitycommand <script></code>	<code>name baseuri uri id</code>	Specifies the prefix of a Tcl command to be evaluated to resolve an external entity reference. If the parser has been configured to validate the XML document, a default script is supplied that resolves the URI given as the system identifier of the external entity and recursively parses the entity's data. If the parser has been configured as a non-validating parser, then by default external entities are not resolved. This option can be used to override the default behaviour.
<code>-final <boolean></code>	N/A	Specifies whether the XML document being parsed is complete. If the document is to be incrementally parsed then this option will be set to false, and when the last fragment of document is parsed it is set to true.
<code>-ignorewhitespace <boolean></code>	N/A	If this option is set to true then spans of character data in the XML document which are composed only of white-space (CR, LF, space, tab) will not be reported to the application. In other words, the data passed to every invocation of the <code>-characterdatacommand script</code>

Option	Script Args	Description
		will contain at least one non-white-space character.
-notationdeclcommand <script>	name uri	Specifies the prefix of a Tcl command to be evaluated when a notation declaration is encountered.
-notstandalonecommand <script>	-NONE-	Specifies the prefix of a Tcl command to be evaluated when the parser determines that the XML document being parsed is not a standalone document.
-paramentityparsing <script>	-NONE-	Controls whether external parameter entities are parsed.
-parameterentitydeclcommand <script>	name args	Specifies the prefix of a Tcl command to be evaluated when a parameter entity declaration is encountered.
-parser <script>	-NONE-	The name of the parser class to instantiate for this parser object. This option may only be specified when the parser instance is created.
- processinginstructioncommand <script>	target data	Specifies the prefix of a Tcl command to be evaluated when a processing instruction is encountered.
-reportempty <boolean>	N/A	If this option is enabled then when an element is encountered that uses the special empty element syntax, additional arguments are appended to the -elementstartcommand and -elementendcommand callbacks. The arguments [-empty 1] are appended.
-startcdatasectioncommand <script>	-NONE-	Specifies the prefix of a Tcl command to be evaluated when the start of a CDATA section is encountered. No

Option	Script Args	Description
		arguments are appended to the script.
<code>-startdoctype declcommand <script></code>	-NONE-	Specifies the prefix of a Tcl command to be evaluated at the start of a document type declaration. No arguments are appended to the script.
<code>-unknownencodingcommand <script></code>	-NONE-	Specifies the prefix of a Tcl command to be evaluated when a character is encountered with an unknown encoding. This option has not been implemented.
<code>-unparsedentity declcommand <script></code>	system public notation	Specifies the prefix of a Tcl command to be evaluated when a declaration is encountered for an unparsed entity.
<code>-validate <boolean></code>	N/A	Enables validation of the XML document to be parsed. Any changes to this option are ignored after an XML document has started to be parsed, but the option may be changed after a reset.
<code>-warningcommand <script></code>	warningcode warningmsg	Specifies the prefix of a Tcl command to be evaluated when a warning condition is detected. A warning condition is where the XML document has not been authored correctly, but is still well-formed and may be valid. For example, the special empty element syntax may be used for an element which has not been declared to have empty content. By default, a callback script is provided which silently ignores the warning.
<code>-xmldeclcommand <script></code>	version encoding standalone	Specifies the prefix of a Tcl command to be evaluated

Option	Script Args	Description
		when the XML declaration is encountered.

Application Example

The following is a crude application example on how to pretty print an XML tree. Note, singleton elements are not printed correctly.

```
# Import 'xml' Tcl package
package require xml

set indent 0 ; # Global indentation variable, for pretty printing

# Declare various callback procedures which will be used by parser
# to crudely pretty print an XML tree.

# Define callback for when an XML element start tag is encountered
proc startCmd {name attlist args} {
    global indent
    puts [format "%[expr $indent * 4]s<$name $attlist $args>" ""]
    incr indent
}

# Define callback for when an XML element end tag is encountered
proc endCmd {name args} {
    global indent
    incr indent -1
    puts [format "%[expr $indent * 4]s</$name $args>" ""]
}

# Define callback for when an XML element's CDATA is encountered
proc cdataCmd {data args} {
    global indent
    puts [format "%[expr $indent * 4]s$data $args" ""]
}

# Create XML parser object with certain callbacks
set parser [::xml::parser -ignorewhitespace 1 -elementstartcommand startCmd -
elementendcommand endCmd -characterdatacommand cdataCmd]

# Open XML file for parsing or read from stdin
if {$argc} {
    set file [shift]
    if {![file exists $file]} {
        VovFatalError "The specified file '$file' does not exist"
    }
    set fp [open $file "r"]
    $parser parse [read $fp]
    close $fp
} else {
    VovMessage "Waiting for STDIN"
    $parser parse [read stdin]
}
```

Create New Tcl Commands

The Tcl interface allows you to manipulate the trace using Tcl scripts. This chapter covers the basics about writing Tcl scripts and executing them in VOV. In fact, most VOV commands such as `vsr` and `vovforget` are implemented by means of Tcl scripts.

Tcl Interpreters with VOV Extensions

Many VOV tools contain a Tcl interpreter.

- For interactive use, choose **Tools > Tcl Interpreter** in the Console.
- For batch processing you can choose between `vovbuild` and `vovsh`. The two tools differ in their behavior after the script has been executed.

Use `vovbuild` with the `-f` option. After the script has completed, `vovbuild` exits.

```
% vovbuild -f <tclScript>
```

Use `vovsh` with the `-f` option. After the script has completed, `vovsh` enters a loop to manage the user interface and exits when the top level window of the user interface is dismissed. If no window is created by the script, `vovsh` exits immediately.

```
% vovsh -f <tclScript>
```

The packages Tk and Tix are available in these interpreters, but for performance reasons they are disabled by default and they need to be activated explicitly calling `vovClientInit`. For example:

```
vovClientInit tk ; # Initialize Tk.  
vovClientInit tix; # Initialize Tk and Tix.  
vovClientInit all; # Initialize all packages, i.e. Tk and Tix.
```

These interpreters also have access to all the VOV extensions, that is all the "vtk" procedures. For more information on the vtk procedures, refer to the VOV/Tcl Interface section in the *VOV Reference Guide*.

Tcl Interpreters without VOV Extensions

The VOV distribution contains two other Tcl interpreters which are already familiar to those using Tcl. One is `vtclsh`, which is the VOV version of `tclsh`. The differences between `tclsh` and `vtclsh` are :

- we use a different name to avoid conflicts with other versions of `tclsh`;
- `vtclsh` is **instrumented** with **VIL**, the VOV Instrumentation Library;
- by default, `vtclsh` uses the libraries included in the VOV distribution.

The other interpreter is `wish`, which differs from the original only because it is an instrumented tool.

Many scripts in the VOV distribution use `vtclsh`.

You can invoke these interpreters with the proper options as illustrated below:

```
% vtclsh <script>  
% wish -f <script>
```

Set Names

Every set has a name and the name must be unique.

The name of a set can be any string, including spaces. However, obeying the following rules will make it easier for you to use and browse sets:

- The maximum length of a set name is 512 ASCII characters
- Consider the set name as a "path" in which components are separated by the colon ':'.

Examples:

```
User:joe:tmp:myset
All:clevercopy
```

- A colon at the beginning of the name is not necessary.
- Avoid using spaces in the name.
- Names beginning with three semicolons ';;;' used to be reserved for system sets, which are **protected**. These sets cannot be forgotten or modified. The user is now allowed to create sets with such names, but we discourage it.
- Names beginning with three '@@@' or with 'tmp:' indicate special transient sets. A **transient set** does not generate attach/detach events, thus avoiding unnecessary traffic between the server and the GUI clients.

vovexport

Generates a file in FDL (i.e. Tcl), C-Shell, or Makefile syntax from a subset of the trace.

vovexport: Usage Message

DESCRIPTION:

The utility vovexport generates a file in FDL (i.e. Tcl), C-Shell, or Makefile syntax from a subset of the trace.

USAGE:

% vovexport [OPTIONS]

OPTIONS:

-h	-- Help usage file.
-v	
-set <SETNAME>	
-dir <DIRNAME>	-- Dump jobs in this directory and subdirectories.
-type {make fdl csh}	
-make	-- Generate Makefile.vov
-sh	-- Generate FlowExport.csh
-fdl	-- Generate FlowExport.tcl
-out <OUTFILE>	
-comment	-- Output additional comments.

EXAMPLES:

```
% vovexport -type make -out Makefile.vov
% vovexport -make
% vovexport -make -dir .
```

```
% vovexport -set All:compile -type fdl -out Flow.export  
% vovexport -set All:compile -type csh -out RunAll.csh
```

GUI_Label Property

The GUI_LABEL property can be added to jobs, files, and sets to control what text is shown in the VovConsole. If the property is missing, the system decides what text to show based on available space.

The value of GUI_LABEL can be multiple lines. In the case of files and jobs, the GUI_LABEL can contain fields in the form @FIELDNAME@. Example:

```
# Note: it is easier to pass new line chars using the Tcl API, rather than the shell  
command.  
% vovsh -x 'vtk_prop_set 123456 GUI_LABEL "id=@ID@ host=@HOST@\nNAME=@JOBNAME@  
\n@DURATIONPP@"'
```

- The GUI_LABEL for subsets does not support fields
- Keep labels short; long labels are truncated to fit into the geometry of the node being shown.
- Not all fields are meaningful in the GUI_LABELS, because the fields are expanded using cached information on the client side and some fields are only available on the vovserver side (example: @TASKERNAME@ for a job).

When updating the GUI_LABEL from a Tcl script, it may be useful to notify all vovconsoles that the node needs to be updated. This can be done with the following call:

```
vtk_prop_set -text $nodeId "GUI_LABEL" "Hi there!\nMy id is @ID@"  
vtk_event_hurl $nodeId NODE GUIUPDATE
```

Adding a GUI_LABEL to a set in FDL

If you are using the procedure [S](#), you can use the option -gui_label to attach the property GUI_LABEL to the set.

```
S "AAA" {  
  ...  
} -gui_label "My gui label"
```

Limits for Objects and Strings

Numerical Limits

Description	Applies to	Default	Min	Max	Policy Variable	Notes
File tail name	Files	255	-	-	-	This is everything after the directory name
Full path name	Files	1023	128	16000	maxPathLength	Windows limit is 259, Linux is 1023 or 4095
NFS delay	Files	0	0	300	nfsdelay	The number of seconds to wait before trusting the NFS cache for file properties
Time tolerance	Files	0	0	600	timeTolerance	The max time skew between the server and any host used as a tasker
Child processes	Jobs	40	-	-	-	This is the number that are tracked for job statistics
Command length	Jobs	40960	128	100000	maxCommand	
Job array size	Jobs	10000	10	100000	maxJobArray	

Description	Applies to	Default	Min	Max	Policy Variable	Notes
Job class length	Jobs	No limit	-	-	-	Limited by available memory only
Job directory	Jobs	1023	-	-	-	Limited by OS
Job environment	Jobs	512	128	16000	maxEnvLength	Used named environments if more characters are needed
Job legal exit codes	Jobs	100	-	-	-	Space-separated list
Job name length	Jobs	No limit	-	-	-	Limited by available memory only
Job project length	Jobs	No limit	-	-	-	Limited by available memory only
Job resources	Jobs	1024	128	16000	maxResources	
Resource map length	Jobs	1024	512	64000	resmap.max.n	
Dependency levels	Nodes	4095	-	-	-	This is the max number of dependency levels in a flow graph including files and jobs
Number of smart sets	Sets	50	10	5000	maxSmartSets	Too many smart sets can result in reduced server performance

Description	Applies to	Default	Min	Max	Policy Variable	Notes
Selection rule	Sets	511	-	-	-	
Set name	Sets	511	-	-	-	
Property name length	Properties	255	16	1024	prop.maxName	
Property value length	Properties	131071	128	1048576	prop.maxStrin	
Tcl variables	Scripts	2147483647	-	-	-	Tcl variable values are limited to 2GB which may restrict some strings

Character Type Restrictions

Description	Applies to	Notes
Job name	Jobs	Special characters and spaces are okay to use in job names.
Job command	Jobs	Most special characters are okay to use in commands. Redirects (e.g. > and <), pipes (e.g.), logical operators (e.g. && or) and semicolons need to be escaped unless they are meant to be used as shell operators.
Job substitutions	Jobs	@JOBID@, @NCJOBID@, and @IDINT@ in job names are substituted with the job's values for the job name, environment, and command.
Job array substitutions	Jobs	@INDEX@, @ARRAYJOBID@, @ARRAYID@, and @ARRAYIDINT@ in jobs that are in job arrays are substituted with the job's values

Description	Applies to	Notes
Set name	Sets	Special characters and spaces are okay to use in set names. Colons are used to denote hierarchy. Sets starting with "@@@" or "tmp:" are temporary sets and should not be used.

Upgrading to 2019.9 and Beyond

Changes to Job State Transition Model

Altair FlowTracer 2015.09 introduced a major change in the state transition model for jobs in a flow. This change in the model will affect existing scripts that have code which depends on details of the earlier model.

In earlier versions, a work task might be represented by more than one job node in the flow. This would happen when a job node reached the RETRACING state. This would cause a new related job node to be created in a RUNNING state with a different job ID. This new job node in the RUNNING state represented the same work task. The two nodes with different node IDs and different state would exist to represent the work task. If the work task successfully completed, then the original job node in RETRACING state would vanish and the new job node that was in RUNNING state would continue to exist to represent the work task. If the work task failed, then the new job node in the RUNNING state would vanish and the original job node would remain to represent the work task.

With Altair FlowTracer 2015.09, the state transition model changed so that there is only one job node to represent the work task. It maintains the same job ID over the entire life of the work task. When a given job reaches the RETRACING state, it then changes to a RUNNING state, and from there to the other states in its future. No related job nodes are created to represent the work task.

Script Changes

Existing scripts may depend on the earlier state model while checking the state of the system. Below are some common cases with suggestions on how to change the scripts to fit the new model.

RETRACINGID

Anything that asks for the RETRACINGID of a job is no longer valid. A request for RETRACINGID simply returns a zero.

Previous:

```
set runningJobId30 [vtk_node_format $jobId30 "@RETRACINGID@"]
```

New:

Delete all the code that uses \$runningJobId30 if it no longer has meaning, or use @ID@ instead of @RETRACINGID@.

Counting Running Jobs

In the old model, finding the count of running jobs could be done simply by looking for job nodes in the RETRACING state. In the new model, finding the count requires looking for job nodes in both RETRACING and RUNNING state.

Previous:

```
set count_running = vovset count JOBRUNNINGDURINGCRASH RETRACING
```

New:

```
set count_running = vovset count JOBRUNNINGDURINGCRASH RETRACING+RUNNING
```

ISRUNNING to select Running Jobs

In the old model, counting running jobs only needed to consider the RETRACING status of a job. In the new model, you can use the ISRUNNING keyword instead.

Previous:

```
set SELRULE "isjob status==RETRACING"
```

New:

```
set SELRULE "isjob ISRUNNING"
```

Previous:

```
set subsetId [vtk_set_subselect -transient $setId "ISJOB STATUS==RETRACING  
ISNONEEXEC==0"]
```

New:

```
set subsetId [vtk_set_subselect -transient $setId "ISJOB ISRUNNING  
ISNONEEXEC==0"]
```

Is Job Started

In the old model, checking if a job started could consider just the RETRACING status. In the new model, you must check for both the RETRACING and RUNNING status.

Previous:

```
if { $nodeInfo(status) != "RETRACING" } {  
}
```

New:

```
if { $nodeInfo(status) != "RETRACING" && $nodeInfo(status) != "RUNNING" } {  
}
```

Check that the Job is Still Running

In the old model, your code to check if a job were still running could consider only the RETRACING status. In the new model, your code should check on both the RETRACING and RUNNING status.

Previous:

```
vtk_node_get $jobId a
```

```
if { $a(status) == "RETRACING" } {  
    # job is still running  
}
```

New:

```
vtk_node_get $jobId a  
  
if { $a(status) == "RETRACING" || $a(status) == "RUNNING" } {  
    # job is still running  
}
```

Avoid Hitting the Same Job Twice

In the old model, you would need code to prevent processing the same work task twice while looking through the job nodes, because two job nodes might exist for the same work task. (A job node in RETRACING and a job node in RUNNING state can exist at the same time for the one task). In the new model, you do not need to have such code. Each work task is represented by only one job node.

Previous:

```
foreach { id st sn jn } $JOBS {  
  
    # Prevent processing more than one job node for one work task,  
    # skip nodes in RETRACING state, because the related node in RUNNING state will  
    # be the one that is processed  
    {$st == "RETRACING"} continue WACost::InitCosts? $id  
}
```

New:

```
foreach { id st sn jn } $JOBS {  
  
    # every job node represents a work task  
    WACost::InitCosts? $id  
}
```

Using Set Statistics to Count Running Jobs

In the old model, you could get set statistics to count running jobs by limiting the search to just job nodes with RETRACING status. In the new model, you should get set statistics to count running jobs by searching for job nodes with either RETRACING or RUNNING status.

Previous:

```
vtk_set_statistics $setIdVovlmd setStats set stillRunning  
    $setStats(jobs,RETRACING)
```

New:

```
vtk_set_statistics $setIdVovlmd setStats set stillRunning [expr  
    {$setStats(jobs,RETRACING) + $setStats(jobs,RUNNING)}]
```

Legal Notices

Intellectual Property Rights Notice

Copyrights, trademarks, trade secrets, patents and third party software licenses.

Copyright ©1986-2025 Altair Engineering Inc. All Rights Reserved.

This Intellectual Property Rights Notice is exemplary, and therefore not exhaustive, of the intellectual property rights held by Altair Engineering Inc. or its affiliates. Software, other products, and materials of Altair Engineering Inc. or its affiliates are protected under laws of the United States and laws of other jurisdictions.

In addition to intellectual property rights indicated herein, such software, other products, and materials of Altair Engineering Inc. or its affiliates may be further protected by patents, additional copyrights, additional trademarks, trade secrets, and additional other intellectual property rights. For avoidance of doubt, copyright notice does not imply publication. Copyrights in the below are held by Altair Engineering Inc. or its affiliates. Additionally, all non-Altair marks are the property of their respective owners. If you have any questions regarding trademarks or registrations, please contact marketing and legal.

This Intellectual Property Rights Notice does not give you any right to any product, such as software, or underlying intellectual property rights of Altair Engineering Inc. or its affiliates. Usage, for example, of software of Altair Engineering Inc. or its affiliates is governed by and dependent on a valid license agreement.

Altair HyperWorks®, a Design & Simulation Platform

Altair® AcuSolve® ©1997-2025

Altair® Activate® ©1989-2025

Altair® Automated Reporting Director™ ©2008-2022

Altair® Battery Damage Identifier™ ©2019-2025

Altair® CFD™ ©1990-2025

Altair Compose® ©2007-2025

Altair® ConnectMe™ ©2014-2025

Altair® DesignAI™ ©2022-2025

Altair® DSim® ©2024-2025

Altair® DSim® Cloud ©2024-2025

Altair® DSim® Cloud CLI ©2024-2025

Altair® DSim® Studio ©2024-2025

Altair® EDEM™ ©2005-2025

Altair® EEvision™ ©2018-2025

Altair® ElectroFlo™ ©1992-2025
Altair Embed® ©1989-2025
Altair Embed® SE ©1989-2025
Altair Embed®/Digital Power Designer ©2012-2025
Altair Embed®/eDrives ©2012-2025
Altair Embed® Viewer ©1996-2025
Altair® e-Motor Director™ ©2019-2025
Altair® ESAComp® ©1992-2025
Altair® expertAI™ ©2020-2025
Altair® Feko® ©1999-2025
Altair® FlightStream® ©2017-2025
Altair® Flow Simulator™ ©2016-2025
Altair® Flux® ©1983-2025
Altair® FluxMotor® ©2017-2025
Altair® GateVision PRO™ ©2002-2025
Altair® Geomechanics Director™ ©2011-2022
Altair® HyperCrash® ©2001-2023
Altair® HyperGraph® ©1995-2025
Altair® HyperLife® ©1990-2025
Altair® HyperMesh® ©1990-2025
Altair® HyperMesh® CFD ©1990-2025
Altair® HyperMesh® NVH ©1990-2025
Altair® HyperSpice™ ©2017-2025
Altair® HyperStudy® ©1999-2025
Altair® HyperView® ©1999-2025
Altair® HyperView Player® ©2022-2025
Altair® HyperWorks® ©1990-2025
Altair® HyperWorks® Design Explorer ©1990-2025
Altair® HyperXtrude® ©1999-2025
Altair® Impact Simulation Director™ ©2010-2022
Altair® Inspire™ ©2009-2025
Altair® Inspire™ Cast ©2011-2025
Altair® Inspire™ Extrude Metal ©1996-2025

Altair® Inspire™ Extrude Polymer ©1996-2025
Altair® Inspire™ Form ©1998-2025
Altair® Inspire™ Mold ©2009-2025
Altair® Inspire™ PolyFoam ©2009-2025
Altair® Inspire™ Print3D ©2021-2025
Altair® Inspire™ Render ©1993-2025
Altair® Inspire™ Studio ©1993-20245
Altair® Material Data Center™ ©2019-2025
Altair® Material Modeler™ ©2019-2025
Altair® Model Mesher Director™ ©2010-2025
Altair® MotionSolve® ©2002-2025
Altair® MotionView® ©1993-2025
Altair® Multi-Disciplinary Optimization Director™ ©2012-2025
Altair® Multiscale Designer® ©2011-2025
Altair® newFASANT™©2010-2020
Altair® nanoFluidX® ©2013-2025
Altair® NLView™ ©2018-2025
Altair® NVH Director™ ©2010-2025
Altair® NVH Full Vehicle™ ©2022-2025
Altair® NVH Standard™ ©2022-2025
Altair® OmniV™©2015-2025
Altair® OptiStruct® ©1996-2025
Altair® PhysicsAI™©2021-2025
Altair® PollEx™ ©2003-2025
Altair® PollEx™ for ECAD ©2003-2025
Altair® PSIM™ ©1994-2025
Altair® Pulse™ ©2020-2025
Altair® Radioss® ©1986-2025
Altair® romAI™ ©2022-2025
Altair® RTLvision PRO™ ©2002-2025
Altair® S-CALC™ ©1995-2025
Altair® S-CONCRETE™ ©1995-2025
Altair® S-FRAME® ©1995-2025

Altair® S-FOUNDATION™ ©1995-2025
Altair® S-LINE™ ©1995-2025
Altair® S-PAD™ © 1995-2025
Altair® S-STEEL™ ©1995-2025
Altair® S-TIMBER™ ©1995-2025
Altair® S-VIEW™ ©1995-2025
Altair® SEAM® ©1985-2025
Altair® shapeAI™©2021-2025
Altair® signalAI™©2020-2025
Altair® Silicon Debug Tools™©2018-2025
Altair® SimLab® ©2004-2025
Altair® SimLab® ST ©2019-2025
Altair® SimSolid® ©2015-2025
Altair® SpiceVision PRO™ ©2002-2025
Altair® Squeak and Rattle Director™ ©2012-2025
Altair® StarVision PRO™ ©2002-2025
Altair® Structural Office™ ©2022-2025
Altair® Sulis™©2018-2025
Altair®Twin Activate®©1989-2025
Altair® UDE™ ©2015-2025
Altair® ultraFluidX® ©2010-2025
Altair® Virtual Gauge Director™ ©2012-2025
Altair® Virtual Wind Tunnel™ ©2012-2025
Altair® Weight Analytics™ ©2013-2022
Altair® Weld Certification Director™ ©2014-2025
Altair® WinProp™ ©2000-2025
Altair® WRAP™ ©1998-2025

Altair HPCWorks®, a HPC & Cloud Platform
Altair® Allocator™ ©1995-2025
Altair® Access™ ©2008-2025
Altair® Accelerator™ ©1995-2025
Altair® Accelerator™ Plus ©1995-2025
Altair® Breeze™ ©2022-2025

Altair® Cassini™ ©2015-2025
Altair® Control™ ©2008-2025
Altair® Desktop Software Usage Analytics™ (DSUA) ©2022-2025
Altair® FlowTracer™ ©1995-2025
Altair® Grid Engine® ©2001, 2011-2025
Altair® InsightPro™ ©2023-2025
Altair® InsightPro™ for License Analytics ©2023-2025
Altair® Hero™ ©1995-2025
Altair® Liquid Scheduling™ ©2023-2025
Altair® Mistral™ ©2022-2025
Altair® Monitor™ ©1995-2025
Altair® NavOps® ©2022-2025
Altair® PBS Professional® ©1994-2025
Altair® PBS Works™ ©2022-2025
Altair® Simulation Cloud Suite (SCS) ©2024-2025
Altair® Software Asset Optimization (SAO) ©2007-2025
Altair® Unlimited™ ©2022-2025
Altair® Unlimited Data Analytics Appliance™ ©2022-2025
Altair® Unlimited Virtual Appliance™ ©2022-2025

Altair RapidMiner®, a Data Analytics & AI Platform
Altair® AI Hub ©2023-2025
Altair® AI Edge™ ©2023-2025
Altair® AI Cloud ©2022-2025
Altair® AI Studio ©2023-2025
Altair® Analytics Workbench™ ©2002-2025
Altair® Graph Lakehouse™ ©2013-2025
Altair® Graph Studio™ ©2007-2025
Altair® Knowledge Hub™ ©2017-2025
Altair® Knowledge Studio® ©1994-2025
Altair® Knowledge Studio® for Apache Spark ©1994-2025
Altair® Knowledge Seeker™ ©1994-2025
Altair® IoT Studio™ ©2002-2025
Altair® Monarch® ©1996-2025

Altair® Monarch® Classic ©1996-2025

Altair® Monarch® Complete™ ©1996-2025

Altair® Monarch® Data Prep Studio ©2015-2025Altair® Monarch Server™ ©1996-2025

Altair® Panopticon™ ©2004-2025

Altair® Panopticon™ BI ©2011-2025

Altair® SLC™ ©2002-2025

Altair® SLC Hub™ ©2002-2025

Altair® SmartWorks™ ©2002-2025

Altair® RapidMiner® ©2001-2025

Altair One® ©1994-2025

Altair® CoPilot™©2023-2025

Altair® Drive™©2023-2025

Altair® License Utility™ ©2010-2025

Altair® TheaRender® ©2010-2025

OpenMatrix™ ©2007-2025

OpenPBS® ©1994-2025

OpenRadioss™ ©1986-2025

Third Party Software Licenses

For a complete list of Altair Accelerator Third Party Software Licenses, please click [here](#).

Technical Support

Altair provides comprehensive software support via web FAQs, tutorials, training classes, telephone and e-mail.

Altair One Customer Portal

Altair One (<https://altairone.com/>) is Altair's customer portal giving you access to product downloads, Knowledge Base and customer support. We strongly recommend that all users create an Altair One account and use it as their primary means of requesting technical support.

Once your customer portal account is set up, you can directly get to your support page via this link: www.altair.com/customer-support/.

Altair Training Classes

Altair training courses provide a hands-on introduction to our products, focusing on overall functionality. Courses are conducted at our main and regional offices or at your facility. If you are interested in training at your facility, please contact your account manager for more details. If you do not know who your account manager is, e-mail your local support office and your account manager will contact you

Index

A

A [39](#), [42](#)
A, FDL [42](#)
access and set up makefile to FDL [144](#)
AD [39](#), [42](#)
AD, FDL [42](#)
add a flow [117](#)
add resources [159](#)
advanced tasker topics [146](#)
advanced taskers topics [146](#)
alias, Tcl procedure [12](#)
automatic zipping and unzipping files [135](#)

B

barriers to change propagation [112](#)
build flows with vovbuild [60](#)
building the flow with vovbuild [37](#)

C

cache, equivalence [129](#)
canonical file names [121](#)
capsule on the fly [105](#)
capsule post-processing [105](#)
CAPSULE, FDL [43](#)
capsules for VHDL and Verilog tools [106](#)
CGI interface [161](#)
change the exclude.tcl file [134](#)
check environment [20](#)
clevercopy/cleverrename [115](#)
command interception [35](#)
command line representation [28](#)
composite environments [11](#)
conditional flows [74](#)
configure vovmake [142](#)
conflict detection [33](#)
conflicts in vovbuild [62](#)
create directories in FDL [78](#)
create new Tcl commands [184](#)
create sets with selection rules [185](#)
customize the browser interface [176](#)

D

D [39](#), [42](#)

- D, FDL [43](#)
- database, JOBSTATUS [126](#)
- database, LINK [124](#)
- databases [122](#)
- decisions in flows with IFJOB [76](#)
- define equivalences for file names [127](#)
- delete resources [155](#)
- dependencies in FDL [66](#)
- develop environments [12](#)
- difference between E and ves [69](#)
- dynamic resource map configuration [157](#)

E

- E [39](#), [42](#)
- E, FDL [44](#)
- encapsulation procedures [101](#)
- environment debugging [20](#)
- environment examples [11](#)
- environment management [8](#)
- environment management limits [22](#)
- environment management: commands [16](#)
- environments in taskers, cache [12](#)
- equivalence cache [129](#)
- events [173](#)
- exclude files from the graph [131](#)
- execute jobs from the command line [26](#)
- exit status [28](#)

F

- FDL [47](#)
- FDL A [42](#)
- FDL AD [42](#)
- FDL and make comparison [82](#)
- FDL CAPSULE [43](#)
- FDL D [43](#)
- FDL E [44](#)
- FDL FLAGS [44](#)
- FDL I [45](#)
- FDL indir [46](#)
- FDL INSTRUMENTED [46](#)
- FDL J [47](#)
- FDL L [47](#)
- FDL N [48](#)
- FDL N2 [48](#)
- FDL O [48](#)
- FDL P [49](#)

- FDL PARALLEL [50](#)
- FDL PJ [50](#)
- FDL PRIORITY [52](#)
- fdl procedures reference [42](#)
- FDL R [52](#)
- FDL R2 [52](#)
- FDL S [53](#)
- FDL S+ [54](#)
- FDL SERIAL [55](#)
- FDL shift [56](#)
- FDL START [56](#)
- FDL STOP [57](#)
- FDL T [57](#)
- FDL T_FINAL [58](#)
- FDL TASK [58](#)
- FDL X [59](#)
- FDL Z [59](#)
- file generation in FDL [79](#)
- file names [121](#)
- FILE, databases [122](#)
- files and file names [121](#)
- files, equivalences, exclusions [121](#)
- FILEX, databases [122](#)
- firing jobs [34](#)
- FLAGS [39](#), [42](#)
- FLAGS, FDL [44](#)
- Flow Description Language (FDL) [38](#)
- flow language procedures [39](#)
- flow library [117](#)
- flow library packager vovflowcompiler [118](#)
- flow library procedures [119](#)
- flowbuilding [36](#)
- FlowTracer Developer Guide [5](#)
- FlowTracer roles [6](#)
- forcing dependencies [67](#)
- forget unneeded resource maps [158](#)

G

- generate FDL [145](#)
- generate FDL using the instrumentation library [79](#)
- GLOB, databases [122](#)
- GUI_Label property [186](#)

H

- handling pipes and redirection with VOV wrappers [32](#)
- hardware resources [146](#)

HOST_OF_jobId and HOST_OF_ANCECEDENT [150](#)
host, change [134](#)

I

I [39](#), [42](#)
I, FDL [45](#)
indir, FDL [46](#)
instrumentation procedure [96](#)
INSTRUMENTED, FDL [46](#)
integrate difficult tools [107](#)
integration by encapsulation [100](#)
integration by instrumentation [93](#)
integration by interception [92](#)

J

J [39](#), [42](#)
J vs T [60](#)
J_FINAL, FDL [47](#)
J, FDL [47](#)
JOBSTATUS database [126](#)
JOBSTATUS, databases [122](#)

L

L [39](#), [42](#)
L, FDL [47](#)
library based interception [93](#)
limits for objects and strings [187](#)
linear coding in FDL [69](#)
LINK database [124](#)
LINK, databases [122](#)
local resource maps [151](#)
logical file names [121](#)

M

makefile conversion [137](#)
makefile to FDL utility [144](#)
manage umask [21](#)
miscellaneous [175](#)

N

N [39](#), [42](#)
N, FDL [48](#)
N2, FDL [48](#)

O

O [39](#), [42](#)
O, FDL [48](#)
owner, change [134](#)

P

P [39](#), [42](#)
P, FDL [49](#)
PARALLEL [39](#), [42](#)
PARALLEL, FDL [50](#)
parameterized environments [9](#)
parse configuration files [77](#)
PHANTOM, databases [122](#)
PJ [39](#), [42](#)
PJ, FDL [50](#)
policies and resources [158](#)
PRIORITY [39](#), [42](#)
PRIORITY, FDL [52](#)
project host, change [134](#)

R

R [39](#), [42](#)
R, FDL [52](#)
R2, FDL [52](#)
RECONCILE_WITH_FILE_SYSTEM [78](#)
refresh environments [12](#)
refresh rate [155](#)
rehost project [134](#)
request hardware resources [146](#)
resource daemon configuration [155](#)
resource map expiration [155](#)
resource map reservation [158](#)
resource map set [151](#)
resource mapping [151](#)
resource reservation [153](#)
rules to write environments in c-shell [12](#)
rules to write environments in tcl [12](#)
run interactive jobs with vxt [83](#)
run jobs in an xterm [80](#)
RUNMODE [39](#), [42](#)
runtime change propagation control [110](#)

S

S [39](#), [42](#)
S, FDL [53](#)

- S+ [39, 42](#)
- S+, FDL [54](#)
- SERIAL [39, 42](#)
- SERIAL, FDL [55](#)
- set creation in FDL [69](#)
- shift, FDL [56](#)
- standard environments [18](#)
- START [39, 42](#)
- start a subflow before a job is completely done with vovfileready [107](#)
- START, FDL [56](#)
- starting vovresourced [155](#)
- stdout and stderr [31](#)
- STOP, FDL [57](#)

T

- T [39, 42](#)
- T_FINAL, FDL [58](#)
- T, FDL [57](#)
- TASK [39, 42](#)
- task oriented flows [64](#)
- TASK, FDL [58](#)
- taskers: mixed Windows NT and UNIX environment [130](#)
- Tcl interface to various set operations [84](#)
- Tcl-language API [22](#)
- TclXML parser [177](#)
- tool integration [89](#)
- track the origins of dependencies [91](#)
- triggers on files [112](#)

U

- upgrade to 2019.9 and beyond [190](#)
- use makefiles [140](#)
- use o_conflict to choose behavior in case of output conflict [62](#)
- use vovequiv to check equivalences [130](#)

V

- VILtools [97](#)
- VOV CGI procedures [162](#)
- VOV CGI procedures descriptions [167](#)
- VOV instrumentation library (VIL [94](#)
- VOV_UMASK examples [21](#)
- vovbarrier [114](#)
- vovbuild argument parsing [63](#)
- vovequiv [129](#)
- VOVEQUIV_CACHE_FILE [127](#)

- vovexport [185](#)
- vovforget [134](#)
- vovresourced [155](#), [155](#)
- vovresourcemgr [156](#)
- vovserver host [134](#)
- vovset [134](#)
- VOVSETS, databases [122](#)
- vovzip [135](#)
- vovzip_host [135](#)
- vovzipdir [135](#)
- vtk_equivalence [127](#)
- vtk_equivalence_get_cache [127](#)
- vtk_equivalence_set_cache [127](#)
- vtk_exclude_rule [134](#)
- vtk_job_control [175](#)
- vtk_place_find [135](#)
- vtk_place_get [135](#)
- vtk_place_set [135](#)
- vtk_resourcemap_delete [155](#)
- vtk_resourcemap_forget [155](#)
- vtk_resourcemap_set [135](#)
- vtk_transition_add [68](#)

W

- workaround for misspelled resource [158](#)
- wrappers [90](#)
- write CGI scripts [161](#)
- write environments [12](#)

X

- X [39](#), [42](#)
- X, FDL [59](#)

Z

- Z [39](#), [42](#)
- Z, FDL [59](#)
- zip, automatic [135](#)
- ZIP, databases [122](#)