

A Configuration-based Verification Environment

Thomas J. Sheffler
sheffler@rambus.com

Kathryn M. Mossawir
kmossawir@rambus.com

Kevin D. Jones
kdj@rambus.com

ABSTRACT

The design of a Verification Environment is a software engineering project in and of itself. This paper presents a Verification Environment design pattern in which all design, testbench and tool elements are abstracted through a named configuration database. This organization allows the design of a virtual environment defined in terms of a virtual design and abstract flows. A concrete instance is defined by a named configuration. This approach promotes a unified Verification Environment that supports multiple projects and their variants at one time and promotes code reuse across designs and testbenches, as well as the environment infrastructure itself.

1. Introduction

A Verification Environment (VE) is a Software Development Environment tailored for the design and debugging of a verification testbench and a hardware design or module. A typical VE consists of multiple test scenarios, a set of design files describing a design, and a number of commands that run tests against the design to expose potential errors.

While this definition of a VE is completely general, the realization of a VE does not express general concepts. Most often, a VE is tightly coupled to a particular instance of a design, so that tailoring a VE for a new design involves destructive modifications to many different files distributed through the environment. This organization leads to a separate copy of a VE for each design or variant.

This paper presents an alternative VE organization, in which all design, testbench and tool elements are abstracted through a level of indirection - a named configuration database. This organization allows the design of a virtual VE defined in terms of a virtual design and testbench. A particular VE instance is defined by a named configuration. This organization promotes a unified VE that supports multiple projects and their variants simultaneously. The arrangement encourages code reuse across designs and testbenches as well as the VE infrastructure itself. It offers additional benefits when used with a Revision Control System.

This paper is organized as follows. Section 2 defines the components of a simple Verification Environment and defines a Configuration-based Verification Environment. Section 3 proposes a simple representation of a configuration as a text file and defines configuration inheritance. Section 4 describes writing generic flows as parameterized Makefiles. Template processors are introduced in Section 5. The relationship of the Configuration-based Verification Environment and the Revision Control System is described in Section 6. Section 7 discusses the implementation of an example Configuration-based Verification Environment. Section 8 presents experiences we have had with our users and Section 9 makes some concluding remarks.

1.1 Related Work

VHDL [1] and Verilog 2001 [4] include directives for HDL configuration management in the language themselves. Tools that move configuration to a separate language-neutral pre-processor have also been described [3]. These facilities do not address the configuration of other elements of the Verification Environment, as we do here.

In his book on testbenches, Bergeron gives a number of recommendations and techniques for configuring a testbench [2]. Most of those recommendations are endorsed by our approach and expanded upon. We do not include the "test" in the configuration, and we include arbitrary settings related to tools. We also advocate a multi-project Verification Environment, which is an approach not addressed in his book.

The vManager product from Cadence [5] also addresses workspace configuration and automation. While it allows parameterization of tools, it is not clear that it supports named configurations. Its focus is on the executable Verification Plan, while our ideas are independent of such software.

2. Verification Environment

A Verification Environment is a workplace for performing analyses of models and designs. Most typically, the analysis involves simulation and the design is a Verilog module. The environment is an organized file hierarchy with fixed locations for different types of components. The main components of a typical environment are listed below.

- dut
- testbench
 - tests
 - test harness
- commands and scripts
 - run simulations or certain types of analyses
 - data collection routines (errors, coverage, performance)
- logs, summaries and reports

The following figure shows the organization of a simple Verification Environment.

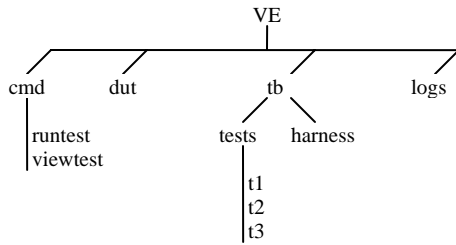


Figure 1. Verification Environment Structure.

The "dut" (design under test) is the representation of the design we are simulating. The DUT has a top-level shell represented in Verilog, but its modules may be represented in Verilog, C PLI code or HSPICE.

The "testbench" (tb) includes two main components. The "test harness" is a Verilog shell that includes clock generators, registers for nets that must be driven, and wires for nets that are observed by the testbench. The "test harness" also includes miscellaneous tasks supporting vendor-specific waveform dumping commands, and site-specific error reporting and logging facilities.

Each "test" in the testbench applies a particular stimulus scenario to the design through a stimulus generator. For Specman-based tests, this describes a particular aspect of the design. For vector-based tests, this is a specific trace testing for a specific behavior. Ordinarily, all of our tests are self-checking, but we support the degenerate case of no checking for those opportunities when such scenarios are useful.

Coordinating operations over the data in the environment are a collection of commands and scripts. The variety of operations performed over the environment can be large. At the minimum, an environment supports running a single Verilog simulation with a particular test, and running a regression consisting of many tests. Commands accept simple arguments that name environment components in standard locations.

```
% cmd/runtest t1
```

Summaries and reports are produced as the result of simulation and analysis. A logfile is produced by each Verilog simulation. Many types of post-processing and data collection are performed to help the Verification engineer take stock of a vast amount of simulation and coverage data. Regression and coverage reports aggregate data over large numbers of simulation runs.

2.1 Verification Environment Parameters

The parameters of a VE are those items that receive new values when a VE is migrated from one project to another. VE parameters may include:

- the list of HDL modules describing the design
- the values of Verilog `defines

- the chosen simulator and version
- test harness files
- stimulus generator and library files
- command line arguments to tools
- arguments to job scheduling software (LSF)

2.2 Parameterized Verification Environment

A parameterized VE moves all parameters into one database, providing a single registry for all project-dependent parameters. With this organization, migration is simplified because all customization occurs in the unified parameter database. Commands and scripts operate largely as before, but are implemented to use the parameter database to look up project-dependent information.

The following figure illustrates the organization of a Parameterized Verification Environment using a registry file as a database.

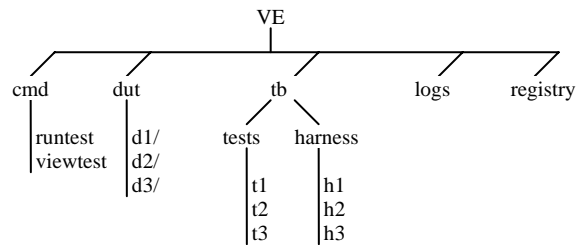


Figure 2. Parameterized Verification Environment Structure

2.3 Configuration-based Verification Environment

A Configuration-based Verification Environment is a variant on the structure above where multiple designs and testbenches are present in the same VE. The set of commands are expressed so as to accept an additional argument: the configuration name. From this argument each command extracts the list of design files, tool settings and other parameters that are related to the design. All commands in the environment are expressed in terms of placeholders that are filled in with values from the configuration file.

The following figure shows the top-level structure of a Configuration-based Verification Environment.

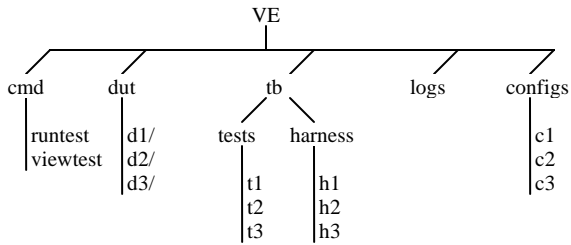


Figure 3. Configuration-based Verification Environment Structure

The configuration name becomes an explicit argument to every command. The following listing shows the use of the "runtest" command modified to accept an explicit configuration name.

```

% cmd/runtest c1 t1
% cmd/runtest c2 t1

```

This organization has the advantage that it factors the same commands over many projects. If the number of designs is large, then this organization avoids replicating commands and testbench elements that are common to many designs. When sharing is not encouraged, and replication is allowed through copying, design environments quickly get out of sync and confusion results.

This definition of a configuration is very broad. A configuration corresponds to a particular design, testbench, set of tools, tool versions and job scheduling requirements. It is possible to create configurations that differ only in tool version, or only in a testbench file.

3. Configuration Representation

A configuration file is a text file describing sectioned, attribute:value pairs. The listing below shows an example configuration file fragment.

```

[verilog]
DUT: dut1.v
LIBS: mod1.v mod2.v
LIBDIRS: modules/dir1 modules/dir2

[simulator]
CMD: /apps/bin/ncverilog5.3
OPTS: +librescan +librescan +notimingchecks

[specman]
SPECRUN: /apps/bin/specrun4.3
SPECRUNARGS:
    -pre "conf gen -seed=random"
GC_THRESHOLD: 256M
MAX_SIZE: 768M

```

Each section corresponds to a related group of attributes. Sections help users to understand the context in which each attribute is used, which is an important hint. Attribute values give semantic intent, rather than a particular use. For example, each of the Verilog library files in the LIBS attribute is typically

prepended with the "-v" flag for use by a particular simulator. The "-v" flag is not part of the LIBS attribute. Commands formulate these strings as appropriate for specific tools from the fundamental information given in the configuration file. This makes it possible to write scripts to process fundamental information for many different applications.

3.1 Inheritance

In a Configuration-based VE, the number of elements in each configuration file is large, and many configurations share similar settings. An inheritance mechanism is a useful means to populate each configuration with default values for all attribute slots. Customizing a particular configuration requires overriding only those values that are unique to that configuration.

Figure 4 below illustrates a simple inheritance hierarchy from a common base. This base configuration defines default values for each of the tools used by the VE, and leaves attributes that describe designs and testbenches empty. The base configuration always has *usable* values for all attribute slots and if used directly produces legal, but mostly empty, output logs.

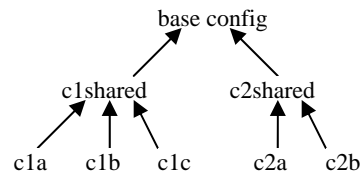


Figure 4. Configuration Inheritance

The use of inheritance allows the VE infrastructure to evolve and improve independent of derived configurations. For example, if a new command is added that makes use of a new attribute value, a default value may be added to the base configuration so that any derived configuration inherits the default value, unless it overrides it. New versions of tools are released to the general community in this manner too. When a new tool release is installed, the base configuration is updated to reflect the new default version. Configurations that override the default can select a specific version, but those that do not are always up to date with the latest release.

The base configuration also serves as a document for users. Comments in the base configuration describe for each attribute why or how a user might override the definition in a derived configuration. In this way, the structure of configuration files remains somewhat self-documenting.

4. Generic Flow

Much of the complexity of a Verification Environment arises from the intricate flows involved. Here, a flow means a chain of programs that must be executed in order to perform a task. Flows involve combining tools from different vendors, who make slightly different assumptions about the way their tools will be

used. Because of this, flow creation and maintenance can be a difficult task.

A flow can be described as a parameterized Makefile. Each parameter is a place-holder whose value is derived from one or more attribute values from a configuration file. Such a parameterized Makefile describes a skeleton of a flow, with the instances of all tools, design and testbench files provided by the configuration file.

The listing below shows a fragment of a parameterized Makefile "skeleton" described using parameters gathered from a configuration file. In the listing, the parameter "<C>" is substituted with the configuration name, and the parameter "<sect.attr>" is substituted with the attribute value from the named section. The fragment shown illustrates part of the Make target for running a Specman test in the a "tests" directory against a design and set of tools described by a configuration.

```
sim/%-<C>.log: tests/%.e <verilog.DUT>
  <specman.SPECRUN> <specman.SPECRUNARGS> \
  -pre "load tests/$*.e" \
  <verilog.CMD> <verilog.OPTS> <verilog.DUT> ...
```

The design of a flow is a delicate job because the dependencies of the various tools are complicated by references to libraries and other included files. It is important to design Make rules so that all of these dependencies are tracked. Some flows may have conditional execution of steps, which is difficult to express in the Make language. Illustrating techniques to handle these issues is beyond the scope of this paper.

5. Templates

The Makefile skeleton pre-processing step is a simple example of a template file with variable substitution from configuration file attributes. This concept is generalized to support template file preprocessing for all file types in a Configuration-based VE. Examples of pre-processors in use in many EDA groups today include cpp, m4, vpp (Verilog pre-processor), empy (Python-based pre-processor) and ppp (Perl-based pre-processor).

A template preprocessor may generate arbitrary program text using values derived from a configuration file. For example, an empy Verilog template fragment that generates register definitions is shown below.

```
@for i in range(@N):
  reg temp_@i;
@end
```

This template uses the value of "N," derived from a configuration file, to generate the definition of an arbitrary number of registers, whose names are of the form "temp_0", "temp_1", etc. The configuration must provide the value of "N" in a section appropriate for the template generator.

```
[empy]
N: 4
```

Template generators are powerful tools for algorithmically generating HDL structures such as the test harness for a parameterized architecture. By describing testbenches and designs as a combination of actual files and files created by a template preprocessor, a Configuration-based VE can automatically produce boiler-plate code.

For example, a parameterized design may be defined in two widths with corresponding configurations named "by4" and "by8." A single test-harness template named "top.v.empty" is created that generalizes the test-harness for all device configurations. When running simulations over the "by4" configuration, the test-harness named "top-by4.v" is generated automatically when it is needed. Similarly, the test-harness "top-by8.v" is generated when it is needed. The use of a single template for defining the test-harness for all device variations simplifies programming and reduces errors.

The named configuration serves two purposes here. It is the repository for the information defining the hardware variant, and its name provides a unique name component of the corresponding HDL object created in the filesystem.

6. Revision Control System

The use of a Revision Control System (RCS) geared for concurrent development is an integral part of a Configuration-based Verification Environment. With the adoption of the Configuration-based organization, the set of files in the VE includes all design and testbench files in the enterprise. Most users do not wish to check out the entire set of files and instead prefer to work only with those files relevant to a few configurations.

With the appropriate operations provided by the Revision Control System, it becomes possible for a user to check out only those files of interest to a particular project. The figure below illustrates a projection of the Configuration-based Verification Environment over only those files relevant to a particular user. Elements grayed out are not physically present in the projection, but the overall structure remains unchanged.

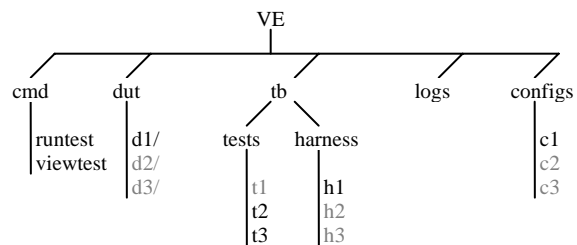


Figure 5. File Structure Projection.

With the use of a RCS, the Configuration-based Verification Environment use-model is the following.

Make new directory for environment, populate from RCS repository using tag.

Build local per-environment binaries (PLI binaries, specman binary) that are independent of configuration.

Development Loop:

- run cmd
- edit files
- ci/co
- repeat

In this model, all design, testbench and environment files are revision controlled. (This includes the commands of the VE itself.) Our goal is to allow a checkpoint of a development environment to completely identify the state of not only the design files but the state of the tools and settings used with all tools. To ensure that a checkpoint tag also identifies the revision of the tools used, we adopt a policy of explicitly naming each tool and its version in the configuration files within the environment; i.e., our environment ensures that the version of any particular tool used in a flow is determined by the configuration file and not by the user's PATH environment variable.

6.1 The Importance of Configuration Names

A configuration file is an enterprise-wide revision-controlled, object, whose name has special significance in the Configuration-based Verification Environment. With a single RCS checkpoint and configuration name, it should be possible to completely recreate the result of a simulation. This leads to the following requirement on the use of configuration files.

Elements used by the Verification Environment but not included in the Verification environment are referenced in a manner that explicitly identifies their version.

We have adopted a policy in which all tools remain mounted on the fileserver in perpetuity, with pathnames that identify their version. Thus, a RCS checkpoint and a configuration name serve to identify the version of all elements (internal and external) relevant to the verification environment of a particular design at a particular point in time.

When an enterprise has multiple sites, if it is guaranteed that the path names and versions of elements outside the VE are identical, it is possible for engineers at geographically distributed sites to re-create test scenarios with identical results. In our system, a scenario can be completely defined by the following parameters: RCS tag, configuration name, test name and stimulus generator random number seed. With just this information, two engineers - at the same or different sites - are guaranteed to be able to reproduce test results on a particular design with equivalent results. This elimination of ambiguity increases productivity.

The uniqueness of configuration names in the enterprise provides a simple mechanism for constructing the commands of the environment so that intermediate files generated in the same workspace from concurrent execution of tools have unique names.

Any file object generated as the result of an operation on a configuration file-specific operation includes the configuration file as part of its name.

This serves two purposes. Most obviously it ensures that derived files are easily related to their precursors. It also ensures that concurrent compute runs over multiple configurations can proceed with the guarantee that generated files do not interfere with one another. The ability to run concurrent jobs over a set of configurations in the same Configuration-based VE is a powerful capability.

For example, when evaluating NC-Verilog 5.1, we compared it to simulation results produced by NC-Verilog 3.4 by creating two configurations for the same design, with names "projNC34" and "projNC51." With our Configuration-based Verification Environment, a complete regression was run over the existing design with both configurations in the same workspace. Result logs produced for each test had names of the form "test-projNC34.log" or "test-projNC51.log." The regressions were run concurrently. It was easy to compare the results of the old and new version of the tool with certainty of which version produced which result.

7. Implementation

We implemented a Configuration-based Verification Environment with a control program called Yve (YYY Verification Engine). The Yve program itself is fairly simple. Figure 6 illustrates its architecture. Most of the complexity of the Yve system resides in the Makefile template, which describes our flows. The system also required the development of a large amount of "plumbing" to route configuration parameters to all the contexts they were used: as tool arguments, as Verilog values, as the result of PLI calls, in C code, in Specman code, and a number of other contexts. We also developed a Remote Job Execution system on top of LSF that addresses the problem of how a configuration names tool binary versions and plugin (PLI) binary versions in an architecture-neutral, but completely specific manner in our heterogeneous compute farm.

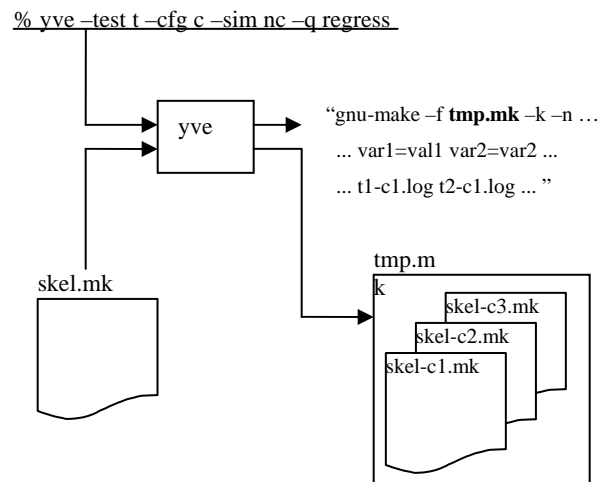


Figure 6. Yve Architecture.

The Yve program accepts user-friendly command line arguments and options and processes them in turn. The job of Yve is to accept simple options and produce the set of flags for subordinate

processes necessary to accomplish the requested task. It produces two things as output:

- a Makefile
- a gnu-make command with argument list

The Makefile skeleton is a template for an actual Makefile, with placeholders for substitution variables from a configuration file. For each configuration that is referenced by an Yve call, Yve makes a copy of the skeleton with configuration parameters substituted. For the resulting gnu-make command, Yve builds a list of options, variable definitions and targets. Each target thus generated causes a flow to be run.

- options (-x -y)
- variables (var1=val1 var2=val2)
- targets (t1-c1.log t1-c2.log)

One of the benefits Yve offers its users is an enormous amount of checking of the consistency of the command line arguments given to it. It checks for the existence of configuration files and tests requested. Where possible, it also checks for the parents of configuration files and for the presence of testbench files. It complains if non-sensical arguments are given to it, and performs simple sanity checks wherever possible. The helpful consistency checking Yve performs ensures that only well-formed commands are launched for execution by gnu-make.

8. Experiences

Our resulting Configuration-based Verification Environment has succeeded in providing a virtual verification environment with pluggable components for the test generator, simulator, DUT, models and target execution host. We support Specman and a home-grown test generator with simulators NC, VCS, Modelsim and Nanosim-cosim. PLI model inclusion is configurable. Optional flows support test vector capture and assertion synthesis. The release level of each tool in each flow is explicit and under configuration control. Remote execution on HPUX, Solaris and Linux hosts is transparent. Dozens of designs have been supported concurrently in this shared environment.

The amount of reuse provided by the Yve approach to environment creation has been obvious in productivity increases. Where it used to take weeks to build an environment for a new project, we are now able to build a simple testbench for a new design in just a few days. A derivative configuration for a new project very similar to an existing one is created in a matter of hours.

Yve has turned out to be a useful meta-tool and is now in use by dozens of people across many groups. Its success has been proven by the fact that the "system" allows non-verification engineers to run many different types of very complicated flows that they would normally require assistance with. The structure of the Yve program and the way the environment is managed through our RCS makes this possible.

The very structure of this environment allows a few "experts" to support a large number of users in a very manageable way. One of the principles it embraces is that complexity is managed and compartmentalized. The hard stuff is done by a few and used by many.

Each "flow" records an incredibly complicated sequence of steps involving many different tools, in a way that a flow is applicable to most any project. Only a handful of people edit a flow, because changes in a flow affect all Yve users. Casual users make use of flows, but are not presented with the complexity of the flow representation.

The use of common flows has increased everyone's productivity. A good example is our use of the 0in check assertion tool. Once integrated into a flow for one project, all projects immediately had access to the new flow. In the past, the introduction of a new tool could be complicated enough that only those projects with "slacktime" could try using them.

Another example is the introduction of a mixed-signal simulation flow. The use of a mixed-signal simulator did not require a new environment. We simply had one expert develop a general mixed-signal simulation flow that added HSPICE files and some setup information. Once adopted, all projects benefited from being able to do mixed-signal simulations fairly easily, by mimicing a previously successful mixed-signal configuration.

Configuration file inheritance structures complexity and hides much of it. The base configuration contains default values for all attributes used by Yve and the process of creating a derived configuration becomes one of customization. Once created, derived configurations are used by non-verification engineers. Their text representation is simple enough that non-experts can read them and change minor things, like the list of Verilog files needed for a design.

Configuration of our testbenches makes it easy to simulate structural variants of an architecture at the same time in the same environment, or to simulate the same architecture over different versions of tools. Of course this is a benefit, but it also has a downside. Yve made it too easy to run lots of simulations. We need more licenses and compute servers to keep up with the demand.

Moving LSF arguments as well as tool selection arguments into the configuration file has proven extremely useful for special cases. For our parameterizable designs, we want to do most simulation with small hardware configurations. When a large configuration requires an execution CPU with a large memory, it is possible to record the requirement in the configuration.

For example, configurations make it easy to create a normal configuration "projNormal" and a large configuration "projLarge" that includes not just the Verilog parameterization of the large design, but large memory allocation parameters to Specman and LSF host selection for a 2GB memory execution host. We can then run the same test ("test1") over both configurations to produce outputs "test1-projNormal.log" and "test1-projLarge.log" knowing that all requirements needed for the large configuration have been encoded and enforced in "projLarge."

The monolithic structure of Yve also has its downsides. Yve automates complicated flows but makes it difficult to deviate

from an encoded flow. We only encounter this problem when debugging a particular tool or when attempting to integrate a new tool into a flow. These situations inevitably required a Vendor representative who would have preferred to run each step of the Yve flow manually. A simple solution to this problem would be for Yve to record the entire Unix command produced for each step in a flow for later playback in such situations. Such a playback script could be edited while tool debugging is being performed.

9. Summary

Yve is a program and an environment structure. The rigidity of the environment structure enforces many methods but allows the flexibility needed by design and verification engineers to innovate in areas where it matters. Yve clearly divides an environment into

infrastructure, and
project and test files.

This partitioning allows Verification experts to support infrastructure, and a much larger community of design and

verification engineers to use the environment and gain the benefits of using the flows developed by experts.

10. References

- [1] Ashenden, P.J., *The Designer's Guide To VHDL. Second Edition*. Morgan Kaufmann Publishers, San Francisco, CA, 2002.
- [2] Bergeron, J., *Writing Testbenches: Functional Verification of HDL Models Second Edition*. Kluwer Academic Publishers, Norwell, Mass, 2003.
- [3] Lange, H., and Radetzki, M. IP Configuration Management with Abstract Parameterizations. Design And Reuse. <http://www.us.design-reuse.com/article5327.html>
- [4] Sutherland, S., *Verilog-2001: A Guide to the New Features in the Verilog Hardware Description Language*. Springer, 2001.
- [5] Verification Automation Management. Whitepaper. <http://www.verisity.com/resources/whitepaper/vmanager.html>