

Verification Planner User Guide

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2012 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

"This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclipse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (sm)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.

ARM and AMBA are registered trademarks of ARM Limited.

Saber is a registered trademark of SabreMark Limited Partnership and is used under license.

All other product or company names may be trademarks of their respective owners.

Contents

1. Verification Planner

Introduction to Verification Planner	1-2
Understanding the Verification Planner Structure	1-3
Verification Planner-Enabled Applications.	1-4
Reporting HVP Annotated Results in XML Format	1-6
Use Model	1-7
XML Files	1-7
XML Format	1-7
session	1-8
hvp	1-8
datadef	1-8
scope	1-10
plan.	1-12
feature	1-13
measure	1-15
source/region	1-15
Verification Planner Data Sources.	1-16
Synopsys Data.	1-16

External User Data.	1-17
Multiple Value Columns in a Row.	1-19
Comments in HVP userdata Format	1-19
Using the HVP Language	1-20
Plan Declaration	1-21
Attribute Declaration	1-22
Annotation Declaration.	1-24
Metric Declaration	1-25
Built-In Metrics	1-27
Metric Types and Aggregators	1-28
Feature Declaration	1-34
Attribute Value Specification	1-37
Annotation Value Specification.	1-38
Goal Specification	1-38
Measure Specification	1-40
Data Regions in the Synopsys Coverage Database	
Data Source	1-41
Data Regions in External User Data Sources	1-44
Using Attributes in Source Specifications.	1-45
Subplan Declaration	1-51
A Complete HVP Subplan Example.	1-53
Plan Modifiers	1-55
The override Modifier	1-56
filter.	1-58
until Statement – Time-Based Modifier	1-59
Compiler Directives	1-60
Comments	1-61

How to Use HVP Files	1-62
2. Verification Planner Spreadsheet Annotator	
Introduction to Spreadsheet Annotator	2-2
Getting Started with Spreadsheet Annotator	2-4
Using HVP Spreadsheet Meta-Tags	2-7
HVP Plan Sheet.	2-10
hvp plan plan-name	2-11
plan.	2-11
feature	2-12
subplan.	2-13
\$attribute-name, \$annotation-name	2-15
skip.	2-16
goal.metric-name	2-16
measure measure-name.source	2-17
value measure-name.metric-name.	2-18
include	2-19
HVP Metric Definition Sheet	2-21
hvp metric plan-name.	2-22
name	2-23
type.	2-23
aggregator	2-24
goal.	2-24
skip.	2-24
HVP Attribute Definition Sheet.	2-24
hvp attribute plan-name	2-25
name	2-25
type.	2-26

propagate	2-26
default.	2-26
Applying Modifiers	2-27
Sharing a Metric/Attribute Definition Sheet	2-27
Using Spreadsheet Annotator Commands	2-28
Using the HVP_ARCH_OVERRIDE Variable	2-34
Troubleshooting.	2-35

1

Verification Planner

This chapter contains the following sections:

- [“Introduction to Verification Planner”](#)
- [“Understanding the Verification Planner Structure”](#)
- [“Verification Planner-Enabled Applications”](#)
- [“Verification Planner Data Sources”](#)
- [“Using the HVP Language”](#)
- [“Compiler Directives”](#)
- [“Comments”](#)
- [“How to Use HVP Files”](#)

Introduction to Verification Planner

Verification Planner is a verification planning tool which is incorporated into several Synopsys products. Verification Planner is a technology that allows you to think about the verification process at a high-level overview while working with the real objective of low-level verification data. Verification Planner allows you to convert the low-level data into useful information to plan and track the progress of verification projects.

An HVP (Hierarchical Verification Plan) is a comprehensive model that allows you to hierarchically describe a verification plan. The verification plan contains feature declarations, attributes, goals, and metrics. Attributes are named values specified in the plan, whereas metrics are named values annotated by API from HVP data files. Metrics can be coverage information extracted from merged simulation runs. Metrics can also include project specific information, for example, code churn (a measurement of the amount and frequency of source code changes), bug information, die size, clock speed, and so on.

Each hierarchical section of a verification plan is called a feature. A feature may consist of the following:

- Attribute value assignments
- Metric measurement specifications
- Subfeatures

In addition, features can also reference tests. Verification Planner can import pass and fail test results, and report their associated feature values. This allows you to determine at a glance which portions of the design have not been tested completely.

Because features in a verification plan are arranged hierarchically, Verification Planner also traverses the feature sub-trees to aggregate the feature information. When low-level feature information is annotated to a plan, that information can propagate up the feature hierarchy. Therefore, you can determine the status of high-level features at a glance without explicitly defining what metrics contribute to each high-level feature. Furthermore, you can change and tune low-level metrics without manually propagating those changes up the feature hierarchy. The method of aggregation is defined in the metric declaration for each metric being measured. For example, Verification Planner sums up all pass and fail test results, and averages the coverage score.

This user guide explains the basic concepts of Verification Planner technology and the Hierarchical Verification Plan (HVP).

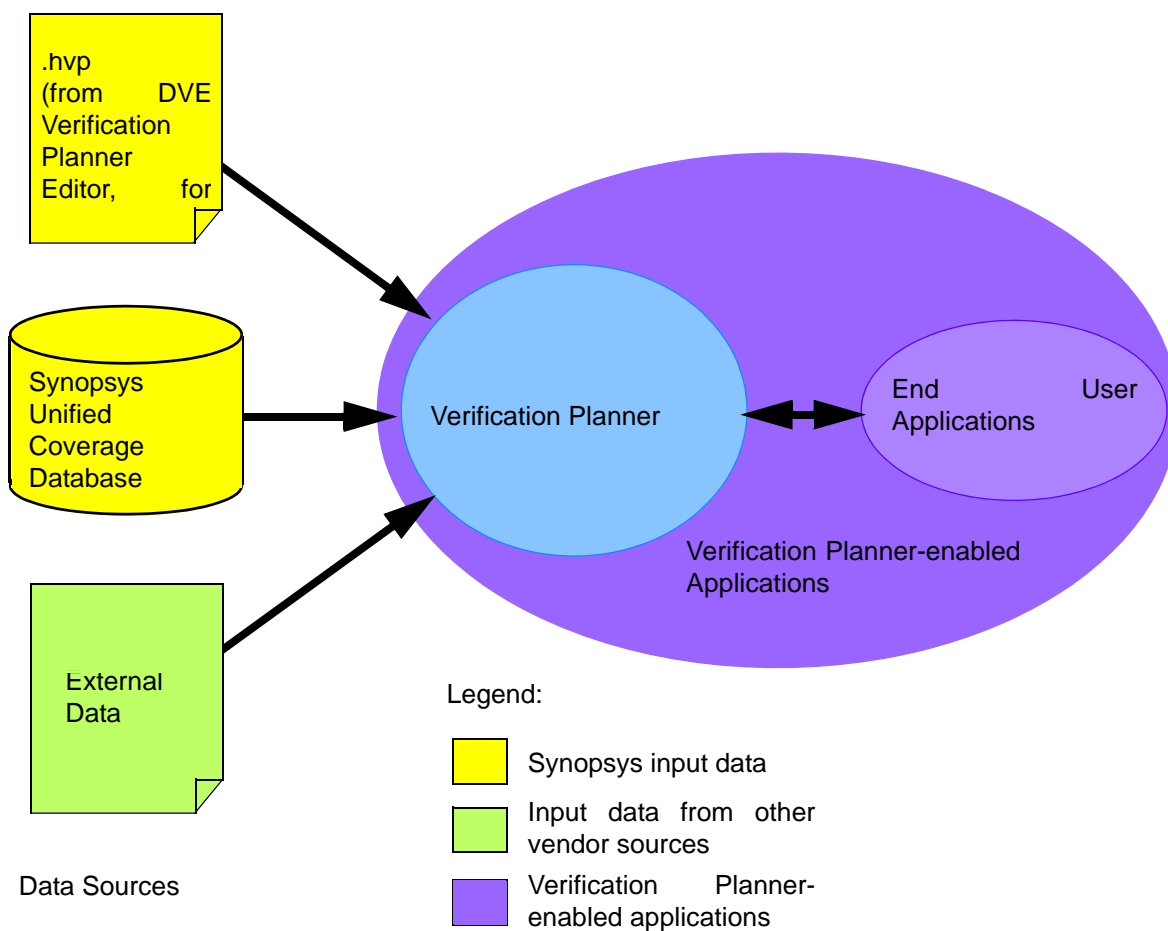
Understanding the Verification Planner Structure

This section provides an overview of Verification Planner-enabled applications. [Figure 1-1](#) illustrates the common usage of Verification Planner. The following indicates the use of color in this figure:

Color	Description
Purple	Verification Planner-enabled applications
Yellow	Synopsys Input data sources to Verification Planner
Green	Input data sources from other vendors

Subsequent sections in this chapter further describe each of these color-coded elements.

Figure 1-1 Common Verification Planner Structure



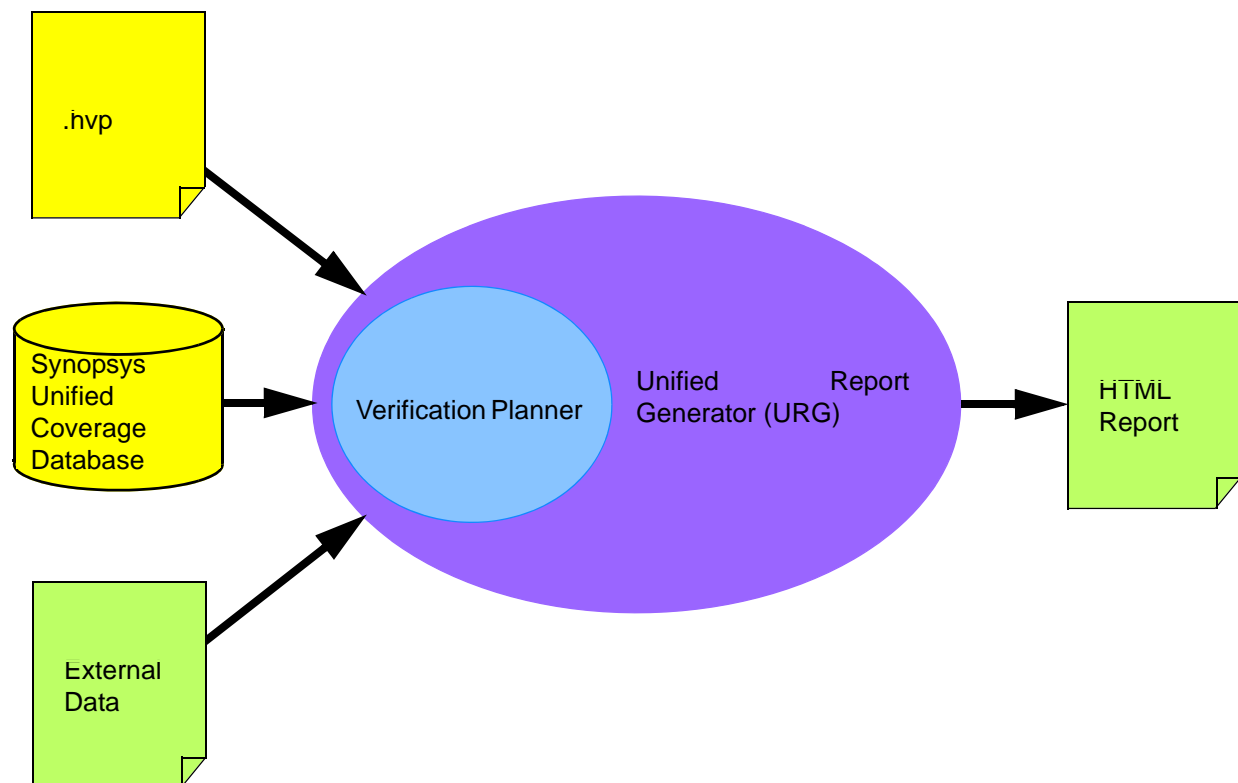
Verification Planner-Enabled Applications

Verification Planner-enabled applications are:

- [Verification Planner Spreadsheet Annotator](#)
- Verification Planner History Annotation (Execution Manager)
- Unified Report Generator (URG)

In addition to the Verification Planner Spreadsheet Annotator and History Annotation, Verification Planner also reports information about a verification plan via URG (see [Figure 1-2](#)). URG is used to generate combined reports for all types of coverage information and includes a separate document. For information regarding URG, see *Viewing the Coverage Report Using Unified Report Generator* in *Coverage Technology User Guide*.

Figure 1-2 Verification Planner-Enabled Application: URG



Reporting HVP Annotated Results in XML Format

The HVP annotated results are reported in the Unified Report Generator (URG) XML format. With this XML report, you can extract all types of coverage information using a common XML reader and create a coverage report in your own format.

Note:

At present, the XML format is only supported for HVP annotated results and not for all coverage row data.

Use Model

To report HVP annotated results in XML format, the `-xmlplan` URG option is added.

When you specify the `-xmlplan` option in URG, it generates an additional `plan.xml` file under the `urgReport` directory. The `urgReport` directory contains an annotated HVP report in XML format.

For example:

```
% urg -dir simv.vdb -xmlplan
```

XML Files

When you specify the `-xmlplan` option in URG, the URG HVP report is supplemented by an additional `plan.xml` file in the URG report directory. The `plan.xml` file contains the same information as a normal URG HVP HTML or text report, but in XML format.

If you want to process the HVP result further, then you may find it easier to use the existing tools to read the XML HVP annotated report and then generate your own formatted coverage report.

XML Format

This section provides the XML Document Type Definitions (DTD) of a URG HVP annotated report, syntax, and an example for each DTD.

session

This is the top-most tag in all XML files.

Syntax

```
<!ELEMENT session (hvp)*>
<!-- ATTLIST session version CDATA #REQUIRED -->
<!-- ATTLIST session release CDATA #REQUIRED -->
<!-- ATTLIST session timestp CDATA #REQUIRED -->
```

hvp

This is the root element in the `hvp.xml` file.

Syntax

```
<!ELEMENT hvp (datadef,scope)>
```

Example

```
<hvp>
  <datadef>
    ...
  </datadef>
  <scope type="plan" name="top_plan" active="1">
    ...
  </scope>
</hvp>
```

datadef

This tag lists all the definitions of metrics, which are used in the current report.

Syntax

```
<!ELEMENT datadef (metdef)+>
```

```
<!ELEMENT metdef (enumdef|aggrdef)*>
<!ATTLIST metdef name ID #REQUIRED>
<!ATTLIST metdef type (ratio|percent|integer|real|enum)
#REQUIRED>
<!ATTLIST metdef aggregator (max|min|sum|average|ERROR)
#REQUIRED>
<!ATTLIST metdef isaggr (0|1) #IMPLIED>
<!ATTLIST metdef goal CDATA #IMPLIED>
<!ATTLIST metdef builtin (1|0) #REQUIRED>
```

When **metdef type** is **enum**, this **metdef** element contains one or more **enumdef** elements.

```
<!ELEMENT enumdef EMPTY>
<!ATTLIST enumdef name CDATA #REQUIRED>
```

When **metdef's** attribute **isaggr** is **1**, this **metdef** element contains one or more **aggrdef** elements.

```
<!ELEMENT aggrdef EMPTY>
<!ATTLIST aggrdef name CDATA #REQUIRED>
```

Example

```
<datadef>
  <metdef name="Line" type="ratio" aggregator="average"
builtin="1" />

  <metdef name="test" type="enum" aggregator="sum"
builtin="1">
    <enumdef name="pass" />
    <enumdef name="fail" />
    <enumdef name="warn" />
    <enumdef name="assert" />
    <enumdef name="unknown" />
```

```

    </metdef>

    <metdef name="SnpsAvg" type="percent" isaggr="1"
aggregator="average" builtin="1">
      <aggrdef name="Line" />
      <aggrdef name="Cond" />
      <aggrdef name="FSM" />
      <aggrdef name="Toggle" />
      <aggrdef name="Branch" />
      <aggrdef name="Assert" />
      <aggrdef name="Group" />
    </metdef>

    <metdef name="bugs" type="integer" aggregator="sum"
builtin="0" />
  </datadef>

```

scope

This is the basic element for coverage scope. This scope could be the **plan type**, **feature type**, **measure type**, **source type**, and **region type**.

Syntax

```

<!ELEMENT scope
(attrdef|attrval|goalover|scorelist|metref|scope)*>
<!ATTLIST scope type (plan|feature|measure|source|region)
#REQUIRED>
<!ATTLIST scope name CDATA #REQUIRED>
<!ATTLIST scope active (1|0) #IMPLIED>

```

The **active** attribute is only used for plan and feature <scope>s.

The **attrdef** element is only in plan scope.

```

<!ELEMENT attrdef EMPTY>
<!ATTLIST attrdef type (string|integer|real|ERROR)
#REQUIRED>

```



```

<!ATTLIST attrdef name CDATA #REQUIRED>
<!ATTLIST attrdef default CDATA #REQUIRED>
<!ATTLIST attrdef propagate (1|0) #REQUIRED>
<!ATTLIST attrdef builtin (1|0) #REQUIRED>

```

Only the plan and feature type <scope>s can contain <attrval> and <goalover> elements.

```

<!ELEMENT attrval EMPTY>
<!ATTLIST attrval name CDATA #REQUIRED>
<!ATTLIST attrval value CDATA #REQUIRED>

```

This goalover element is to override the goal expression.

```

<!ELEMENT goalover EMPTY>
<!ATTLIST goalover name CDATA #REQUIRED>
<!ATTLIST goalover value CDATA #REQUIRED>

```

This scorelist element is a group of metric scores. <scorelist> with total type is general for all scopes, which aggregates all scores in its sub hierarchies. The feature <scope> can have one more <scorelist> with measure type, which aggregates the scores of immediate measures in the feature. The scores in its subhierarchy, except measures, are excluded from the scores.

```

<!ELEMENT scorelist (score)*>
<!ATTLIST scorelist type (total|measure) #REQUIRED>
<!ATTLIST scorelist name CDATA #IMPLIED>

```

This score element is for score of each metric. The name of score must be defined by metdef in datadef element.

```

<!ELEMENT score (enumval)*>
<!ATTLIST score name CDATA #REQUIRED>
<!ATTLIST score value CDATA #REQUIRED>
<!ATTLIST score goalstat (pass|fail|error|none) #IMPLIED>

```

```

<!ELEMENT enumval EMPTY>
<!--ATTLIST enumval name CDATA #REQUIRED-->
<!--ATTLIST enumval value CDATA #REQUIRED-->

[value format]
integer, enum:  "1", "256"
real :  "15.0", "15.25". "15", ".35"
ratio :  "15/25" (note: denominator != 0)
percent:  "15.22%", "14%", ".35%"

<!ELEMENT metref EMPTY>
<!--ATTLIST metref name IDREF #REQUIRED-->

```

The `<metref>` element represents what metrics are monitored in the current measure. `<metref>` can be defined in measure type `<scope>` element. The name of `metref` must be defined in `<datadef>` at the top.

plan

The plan `<scope>` can contain plan and feature type `<scope>`. The feature `<scope>` can contain only total type `<scorelist>`.

Syntax

```

<!ELEMENT scope
(attrdef|attrval|goalover|scorelist|scope)*>

```

Example

```

<scope type="plan" name="top_plan" active="1">
  <scorelist type="total">
    <score name="FSM" value="1/10" />
  </scorelist>
</scope>

```

```

    <score name="Line" value="56/65" />
    <score name="Toggle" value="35/64" />
    <score name="bugs" value="14" />
    <score name="effort" value="29%" />
    <score name="fixed" value="8/20" />
    <score name="phase" value="0.31" />
    <score name="status" value="5">
      <enumval name="developing" value="1" />
      <enumval name="patched" value="2" />
      <enumval name="released" value="2" />
    </score>
    <score name="Score" value="50.28%" />
  </scorelist>

  <attrdef name="owner" type="string" default=""
propagate="1" builtin="1" />
  <attrdef name="description" type="string" default=""
propagate="0" builtin="1" />
  <attrdef name="at_least" type="integer" default="0"
propagate="1" builtin="0" />

  <attrval name="owner" value="Qiaohui" />

  <goalover name="Group" value="Group>=0.8"

  ...

  <scope type="feature" name="feature_1">
    ...
  </scope>
  <scope type="plan" name="subplan_1">
    ...
  </scope>
  ...
  </scope>

```

feature

The feature <scope> can contain zero or more plan, feature, and measure type <scope>.

The feature <scope> can contain total and measure type <scorelist>.

Syntax

```
<!ELEMENT scope (attrval*, goalover*, scorelist*, scope*)>
```

Example

```
<scope type="feature" name="builtin_feat" active="1">
  <scorelist type="total">
    <score name="FSM" value="1/10" />
    <score name="Line" value="56/65" goalstat="fail" />
    <score name="Toggle" value="35/64" />
    <score name="Score" value="50.28%" />
  </scorelist>
  <attrval name="owner" value="" />
  <attrval name="at_least" value="0" />
  <attrval name="weight" value="1" />
  <attrval name="description" value="" />
  <goalover name="Line" value="((Line > 0.90) &&
(Line <= 1.00))" />
  <scorelist type="measure">
    <score name="FSM" value="1/10" />
    <score name="Line" value="56/65" goalstat="fail" />
    <score name="Toggle" value="35/64" />
    <score name="Score" value="50.28%" />
  </scorelist>
  <scope type="measure" name="mymeas1">
    ...
  </scope>
<scope type="feature" name="feature_1">
  ...
</scope>
<scope type="plan" name="subplan_1">
  ...
</scope>
...
</scope>
```

measure

The `measure` type `<scope>` can contain zero or more `source` type `<scope>`.

The `measure` type `<scope>` can contain only `total` type `<scorelist>`. The `measure` type `<scope>` must contain one or more `<metref>`.

Syntax

```
<!ELEMENT scope (metref*, scorelist*, scope*)>
```

Example

```
<scope type="measure" name="m">
  <metref name="Line"/>
  <metref name="FSM"/>
  <scorelist type="total">
    <score name="Line" value="25/84" goalstat="pass" />
    <score name="FSM" value="33/68" />
    <score name="Score" value="55.53%" />
  </scorelist>
</scope>
<scope type="source" name="module: fsm_mod">
  ...
</scope>
<scope type="source" name="instance: test*">
  ...
</scope>
</scope>
```

source/region

The `source` type `<scope>` can contain zero or more `region` type `<scope>`.

Syntax

```
<!ELEMENT scope (scope*)>
```

Example

```
<scope type="source" name="module: fsm_mod*">
  <scope type="region" name="fsm_mod1">
    <scorelist type="total">
      <score name="Score" value="55.53%" />
      <score name="Line" value="25/84" goalstat="error" />
      <score name="FSM" value="33/68" />
    </scorelist>
  </scope>
</scope>
```

Verification Planner Data Sources

Verification Planner can process two types of data sources: Synopsys data and external data (see [Figure 1-2](#)).

This section contains the following subsections:

- [“Synopsys Data”](#)
- [“External User Data”](#)

Synopsys Data

Verification Planner can directly annotate two types of Synopsys generated data such as coverage database and test results dump files. The two types of data are:

- Coverage data obtained from the Synopsys unified coverage database.
- Test pass and fail results created by the Execution Manager. To dump test pass/fail data from the Execution Manager, use the `eman vedump` command.

External User Data

Verification Planner can also annotate external database generated by tools from other vendors. Note that you need to convert the data files into HVP userdata file format before feeding them to Verification Planner.

The userdata file format is a simple table of name and value pairs. The name and value pairs are separated by the "=" character. The format of an HVP userdata file is as follows:

```
HVP metric = metric1, metric2, ...
source_name1 = value1, value2, ...
source_name2 = value1, value2, ...
```

Example of an integer type metric

```
HVP metric = integer_metric
source_name1 = 1
source_name2 = 2
```

Example of a real type metric

```
HVP metric = real_metric
source_name1 = 1.0
source_name2 = 1
```

- Integer and real values are accepted.

Example of a ratio type metric

```
HVP metric = ratio_metric
source_name1 = 1/2
source_name2 = 2/3
```

- The numerator and the denominator must be integer numbers.
- The numerator must be greater than or equal to the denominator.
- The denominator must be greater than 0.
- White spaces within a ratio expression of a value are ignored. For example, the following expressions are equivalent:

```
source_name1 = 5/10
source_name1 = 5 / 10
```

Example of a percent type metric

```
HVP metric = percent_metric
source_name1 = 0.535
source_name2 = 53.5%
```

- Both real (from 0.0 to 1.0) and % format are allowed.
- 1.0 is internally interpreted as 100%.

Example of an enum type metric

```
HVP metric = enum_metric
source_name1 = item1
source_name2 = item2
```

- The value must be one of the enum entries specified for the metric definition.

```
HVP metric = enum_metric.enum_entry
source_name1 = 1
```



```
source_name2 = 2
```

- The value of the `enum_entry` must be an integer.

Multiple Value Columns in a Row

If multiple metrics share the same source string, you can assign as many values as you need in a row.

```
HVP metric = metric1, metric2, metric3
source_name1 = value1, value2, value3
source_name2 = , value2, value3
source_name3 = , , value3
source_name4 = value1, ,
```

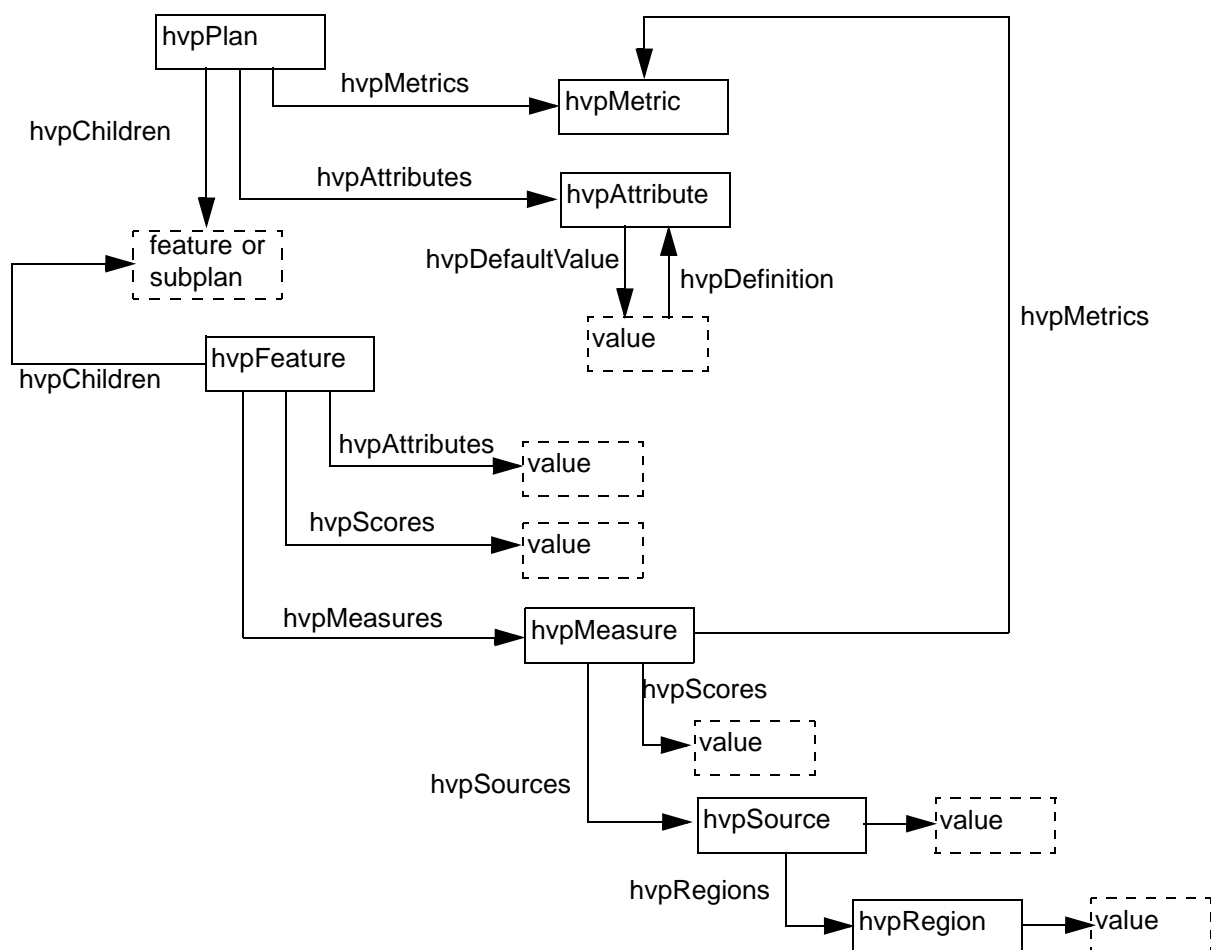
- If no value is available for a specific metric, you can enter a null string in its place.
- The format of each value must match the metric type.

Comments in HVP userdata Format

If “#” is found at the beginning of a line, the line is regarded as a comment and is skipped. Any number of white space is allowed before the “#” in a comment line. However, no non-whitespace characters are allowed before the “#”.

[Figure 1-3](#) illustrates the hierarchical data structure of HVP.

Figure 1-3 Data Structure of HVP



Using the HVP Language

The HVP language was designed to describe a verification plan. The focus of this chapter is to create a verification plan in text format using the HVP language, so Verification Planner can annotate data based on the plan.

Alternatively, you may also choose to use a spreadsheet to describe your verification plan. Use the `hvp annotate` command with the option, `--plan_in xml_in_file`, to tell Verification Planner to automatically translate the spreadsheet into the HVP language (see [“Using Spreadsheet Annotator Commands”](#)).

This section contains the following subsections:

- [“Plan Declaration”](#)
- [“Attribute Declaration”](#)
- [“Annotation Declaration”](#)
- [“Metric Declaration”](#)
- [“Feature Declaration”](#)
- [“Attribute Value Specification”](#)
- [“Annotation Value Specification”](#)
- [“Goal Specification”](#)
- [“Measure Specification”](#)
- [“Subplan Declaration”](#)
- [“Plan Modifiers”](#)
- [“until Statement – Time-Based Modifier”](#)

Plan Declaration

The top-level keyword, `plan`, defines a single hierarchical verification plan which basically contains two sections:

- Definitions of attributes, annotations, and metrics

- Feature trees which are the basic building blocks of the plan

The syntax to declare a plan is as follows:

```
plan identifier;  
    {attribute-declaration}  
    {annotation-declaration}  
    {metric-declaration}  
    {feature-declaration}  
    {subplan-declaration}  
endplan
```

The identifier can consist of alphabetic characters, underscores (`_`), and numbers. Reserved keywords must not be used as identifiers. An identifier must not begin with a number.

Attribute Declaration

Declare attributes outside of features. An attribute can be of type integer, real, string, or enum. The syntax to declare an attribute is as follows:

```
attribute attribute-type identifier =  
    default-value-literal;
```

For example:

```
attribute string team_name = "system";
```

You must specify a default value for an attribute declaration. The default value should be an appropriate literal depending on the given attribute type. For example:

```
attribute integer phase = 1;  
attribute real fraction = 1.0;  
attribute string Specification = "";
```

```
attribute enum{orange, apple, banana} fruitName = orange;
```

Attribute default values are automatically propagated down all the way to leaf node features of hierarchy unless the attributes are explicitly overridden at the plan or feature level. If attributes are explicitly overridden at the plan or feature level, the values are propagated down their child hierarchy.

If you do not want the value to be propagated down the hierarchy, use `annotation` instead of `attribute` (see [“Annotation Declaration”](#) for more information).

Verification Planner provides built-in attributes that are implicitly imported during compilation for every verification plan. Therefore, you can use the built-in attributes without declarations. Note that you cannot redefine the built-in attributes, but you can override the values of the built-in attributes in feature hierarchy (see [“The override Modifier”](#)).

The built-in attributes are `owner` and `at_least`, for example:

```
attribute string owner = "";  
attribute integer at_least = 0;
```

The `owner` attribute does not have any impact on the annotation flow unless you do not use this attribute in the filtering condition. It can be used for your annotation purpose.

However, if you set the `at_least` attribute with non-zero value, it might impact the `Group` or `Assert` metric scores. If the value of the `at_least` attribute is greater than zero, Verification Planner regards the bins with hit count $> \text{at_least}$, and assertions with success count $\geq \text{at_least}$ as covered objects. If `at_least` is not overridden (which means 0), Verification Planner relies on UCAPI to determine covered or not.

Note: The `Assert` or `Group` score in Verification Planner can be different from the traditional URG report if you override the `at_least` value.

Annotation Declaration

You should declare annotations outside of features.

The syntax to declare an annotation is as follows:

```
annotation attribute-type identifier =  
    default-value-string-literal;
```

For example:

```
annotation string group_name = "CPU Group";  
annotation integer weight = 1;  
annotation real myweight = 1.0;
```

Verification Planner also provides a built-in annotation which is implicitly imported during compilation for every verification plan. Therefore, you can use the built-in annotation without declaration.

Note:

You cannot redefine the built-in annotation, but you can override its value in feature hierarchy.

The built-in annotations are `weight` and `description`.

```
annotation real weight = 1;  
annotation string description = "";
```

The `weight` annotation is used when annotated scores are aggregated. You might want to set the weight to 2 or higher for important features or subplans, so that scores in the feature or subplan will be doubled.

You can set "weight = 0" for the features of no interest.

Metric Declaration

Metrics represent values to be annotated to a verification plan from a verification database. Metrics should be declared globally for the entire plan, not on a per feature basis (see [“Measure Specification”](#)).

You declare external metrics at the same level of the verification plan hierarchy as attributes, not inside features. Metric declarations specify the type, goal, aggregator, and owner attribute for each metric. They do not specify how metrics are collected and extracted from the verification database.

The syntax to declare a metric is as follows:

```
metric metric-type identifier;  
    goal = goal-specification;  
    aggregator = aggregator-specification;  
endmetric
```

The following is the Backus-Naur form (BNF):

```
metric-type :  
    ratio | integer | real | enum-type | aggregate  
  
enum-type :  
    enum { enum-identifier-list }  
  
enum-identifier-list ::=  
    enum-identifier { ,enum-identifier }
```

```

metric-identifier ::= identifier

metric-declaration-body ::=
    [ goal-specification ] [ aggregator-specification ]

goal-specification ::=
    goal = goal-expression ;

goal-expression ::= expression

expression ::=
    primary |
    expression binary-op expression |
    unary-op expression |
    inside-expression |
    string-expression

primary ::=
    ( expression ) |
    identifier |
    attribute-value-literal

binary-op ::=
    || | && | > | < | >= | <= | == | !=

unary-op ::=
    !

string-expression ::=
    string-op ( string-primary , string-primary )

string-op ::=
    match | substr

string-primary ::=
    identifier | string-literal

aggregator-specification ::=
    aggregator = max | min | sum | average ;

owner-specification ::=

```



```
owner = string-literal;
```

Built-In Metrics

Verification Planner provides built-in metrics which are implicitly imported during compilation for every verification plan. Therefore, you can use the built-in metrics without declarations.

Note:

You cannot redefine the built-in metrics, but you can override the goals of the built-in metrics in feature hierarchy (see [“The override Modifier”](#)).

The following table lists the types and the associated aggregators of each built-in metric:

Table 1-1 Types and Aggregators for Built-in Metrics

Name	Type	Goal	Aggregator
Line	ratio	No Default Goal	average
Cond	ratio	No Default Goal	average
FSM	ratio	No Default Goal	average
Toggle	ratio	No Default Goal	average
Branch	ratio	No Default Goal	average
Assert	ratio	No Default Goal	average
Group	percent	No Default Goal	average
SnpsAvg	Aggregate {Line, Cond, FSM, Toggle, Branch, Assert, Group}	No Default Goal	
test	enum {pass, fail, warn, unknown, assert}	No Default Goal	sum
AssertResult	enum {successes, failures}	No Default Goal	sum

The `Line`, `Cond`, `FSM`, `Toggle`, `Branch`, `Assert`, and `Group` metrics are the Synopsys built-in code and functional coverage metrics.

The `AssertResult` metric annotates the success and failure counts in Assertion coverage data. This metric works with the `property` keyword in source strings.

The `SnpsAvg` built-in metric is an aggregate metric type which consists of `Line`, `Cond`, `FSM`, `Toggle`, `Branch`, `Assert`, and `Group` built-in metrics as sub-metrics. Use the `average` aggregator to compute aggregation for each individual submetric.

The `test` built-in metric is an enum type consisting of `pass`, `fail`, `warn`, `unknown`, and `assert` named identifiers.

Metric Types and Aggregators

Verification Planner supports six metric types: `ratio`, `percent`, `integer`, `real`, `enum`, and `aggregate`. It also has four operators: `max`, `min`, `sum`, and `average` aggregator.

Note:

An aggregate metric type is not an aggregator.

There are four available aggregators, but you cannot apply every aggregator to every metric type. In other words, not all the combinations of aggregators and metric types produce valid results. The following table summarizes the meaningful combinations:

Table 1-2 Metric Types and Aggregators

	Metric Type
--	-------------

Aggregator	ratio	percent	integer	real	enum	aggregate
sum	X	X	OK	OK	OK	X
average	OK	OK	OK	OK	X	X
min	X	X	OK	OK	X	X
max	X	X	OK	OK	X	X

ratio

Verification Planner stores the `ratio` type metric as a (numerator, denominator) pair. The purpose of this type is to provide meaningful aggregation of two coverage metrics. A coverage metric of type `ratio` can use the numerator to store the number of covered items, and the denominator to store the total coverable items under test. When two coverage metrics are aggregated by an `average` operator, Verification Planner adds all the numerators and the denominators separately. The ratio of the final numerator over the final denominator represents the coverage ratio.

percent

Verification Planner stores the `percent` type metric as a double in the range 0.0 to 1.0. Like the `ratio` type, the `percent` type metric supports only `average` aggregator. When two or more percent values are aggregated, Verification Planner makes an arithmetical mean of the percent values.

In the following example, a feature `fa` has two subfeatures, `fa1` and `fa2`, which both contain two metrics of type `ratio` and `percent`:

```
plan metdef_example;
  metric ratio metric_ratio;
    aggregator = average;
  endmetric
  metric percent metric_percent;
    aggregator = average;
```

```

endmetric

feature fa;
  feature fa1;
    measure metric_ratio, metric_percent m1;
      source = "...";
    endmeasure
  endfeature
  feature fa2;
    measure metric_ratio, metric_percent m1;
      source = "...";
    endmeasure
  endfeature
endfeature
endplan

```

Suppose that the raw data originating in the form of raw ratio coverage data is annotated to `metric_ratio`. The same raw ratio coverage data which converts to a percentage is also annotated to `metric_percent`. The raw data is:

```

fa1 -> 1/2 -> 0.5    (50% covered)
fa2 -> 8/8 -> 1.0    (100% covered)

```

Annotated value for each metric:

```

fa1.metric_ratio    == 1/2;
fa1.metric_percent  == 0.5;
fa2.metric_ratio    == 8/8;
fa2.metric_percent  == 1.0

```

Applying average aggregator to compute aggregation for the parent feature `fa`:

```

fa.metric_ratio    == (1+8)/(2+8) == 9/10;
fa.metric_percent  == (0.5+1.0)/2 == 0.75;

```

Converting each number to a percentage yields 90% and 75% for `fa.metric_ratio` and `fa.metric_percent`, respectively.

The choice of selecting a `ratio` or `percent` metric type depends on the expected use of the results. Typically, coverage information is often represented as:

$$\frac{\text{number-of-items-covered}}{\text{total-number-of-coverable-items}}$$

Therefore, it is more efficient to represent coverage data as `ratio` type, because `ratio` metric type preserves the magnitude of the total coverable items.

real

The `real` type allows the `average` aggregator to compute the arithmetic mean of several metrics.

enum

For `enum` type, use the `sum` aggregator to compute aggregation. The sum of two `enum` metrics is the individual total count of each `enum` named value in the two metrics. Take the built-in `test` metric in the following `system` block as an example:

```
feature system;
...
feature block1;
  measure test my_test1;
    source = "test1", "test3", "test7";
  endmeasure
...
endfeature

feature block2;
```

```

        measure test my_test2;
            source = "test1", "test3", "test7";
        endmeasure
    ...
endfeature
...
endfeature

```

Suppose that the individual values of `test1`, `test3`, and `test7` for `measure my_test1` are `pass`, `fail`, and `pass` respectively:

```

test = 3;
test.pass = 2;
test.fail = 1;

```

Also, suppose that the individual values of `test1`, `test3`, and `test7` for `measure my_test2` are `fail`, `fail`, and `pass`, respectively:

```

test = 3;
test.pass = 1;
test.fail = 2;

```

Since `test` is a built-in metric of `enum` type which only takes `sum` as an aggregator. The aggregation results are:

```

test = 6;
test.pass = 3;
test.fail = 3;

```

aggregate

The `aggregate` type metric has totally different nature from all the other metric types because it is a meta metric which contains a set of predefined metric names as sub-metrics. It computes aggregation of each individual sub-metric score based on the aggregator type of each sub-metric.

To compute the aggregation of two or more aggregate metrics, Verification Planner aggregates individual sub-metrics from each aggregate metric according to the individual sub-metrics' aggregators. Notice that each individual sub-metric must have the same metric type and aggregator to perform sub-metric aggregation. One exception is that you can define a `ratio` type with `percent` type metric as a sub-metric.

goal

Verification Planner evaluates a goal expression to determine if the goal of a metric is met. The identifier in the expression can only be the name of either the metric being evaluated or a qualified enum named identifier. To evaluate a goal expression, Verification Planner substitutes the metric identifier in the expression with the actual value of the metric, and then evaluates the whole expression.

To specify a goal, use the keyword `goal` to describe an expression that evaluates to a Boolean value. For more details on goal specification, see [“Goal Specification”](#).

For example:

```
metric real ecov;
    goal = ecov >= 1.0;
    aggregator = average;
endmetric

metric integer num_bugs;
    goal = num_bugs <= 0;
    aggregator = sum;
endmetric

metric enum {created, reviewed, completed} spec_status;
    aggregator = sum;
    goal = ((spec_status>=10)&&(spec_status.created<=5));
endmetric
```

```
metric aggregate{Line, Cond, FSM, Toggle} SnpsCodeCov;
    goal = (SnpsCodeCov >= 90%);
endmetric
```

Feature Declaration

Features are the basic building blocks in a verification plan, therefore, a verification plan hierarchy is actually its feature hierarchy.

Features can contain attribute and annotation value specifications, goal override specifications, measure specifications and declarations of child features and subplans. Notice that subfeatures collaborate to constitute feature hierarchy. The following is a simple feature example:

```
feature block1;

    //overriding annotation value of a built-in
    //annotation 'weight'
    weight = 3;

    //overriding the goal of a built-in metric 'test'
    test = (test.fail <= 1);

    //overriding the goal of a built-in metric 'Group'
    Group = Group >= 0.8;

    measure test my_test ;
        //specifying test names to be annotated
        source = "test1", "test3", "test7";
    endmeasure

    measure Group my_group ;
        //specifying Group regions to be annotated
        source = "group: groupA";
    endmeasure
```



```
endfeature
```

The above example defines a feature `block1` which corresponds to a module instance `groupA` as indicated by the `source` of measure, `my_group`. The first measure, `my_test`, lists three tests, `test1`, `test3`, and `test7`, to verify the design intent associated with `block1`. The second measure `my_group` tests the coverage ratio of `block1`. This feature block also has a specified weight value of 3.

Notice that feature `block1` specifies a goal value of 0.8 for the `Group` metric. This means that the plan for this feature is to achieve a group coverage of at least 80 percent.

In the following example, the top-level feature is `system` and contains two subfeatures, `block1` and `block2`:

```
feature system;
  phase = 1;
  description = "System level verification feature"

  feature block1;
    phase = 3;
    measure test my_test;
      source = "test1", "test3", "test7";
    endmeasure
    weight = 2;
    Group = Group >= 0.8;
    measure Group my_group;
      source = "group: groupA";
    endmeasure
  endfeature

  feature block2;
    weight = 2
    measure test my_test;
      source = "test1", "test2", "test4";
    endmeasure
```

```

        measure Line, FSM my_groupB;
            source = "groupB";
        endmeasure
        measure Line, Toggle, num_bugs my_cov;
            source = "top.d*";
        endmeasure
    endfeature
endfeature

```

The feature, `system`, itself does not contain any test attribute or measure specification (although it could).

In feature `block2`, the `phase` attribute is not assigned, so it inherits the value 1 from feature `system`, by default. This illustrates the principle that attribute values propagate down the feature hierarchy.

Notice that `test1` is in the test source of both feature `block1` and feature `block2`. It is sometimes useful to assign a test to more than one feature if the test verifies aspects of multiple features. In this example, the complete test list of feature, `system`, consists of `test1`, `test2`, `test3`, `test4`, and `test7`.

Note:

Only alphabetical characters, numbers, and underscores (_) can be used in identifiers such as plan, feature, measure, attribute or metric names. A numerical character cannot be used as the first letter of an identifier. For example, valid identifiers are MyFeatureName1, my_plan, _myPlan, _1_mine, while 1MyFeature, and My-Feature are invalid.

Attribute Value Specification

Specify attribute values using the attribute name followed by the "=" character and a value. You can specify attribute values inside features, metric declarations, or measure specifications. The BNF of attribute value specification is as follows:

```
attribute-value-specification ::=  
    attribute-identifier = attribute-value-literal  
  
attribute-value-literal ::=  
    integer-literal  
    real-literal  
    string-literal
```

For example:

```
weight = 1.0;  
owner = "user1";  
phase = 1;
```

Annotation Value Specification

Specify annotation values using the annotation name followed by the "=" character and a value. You can specify annotation values inside features, metric declarations, or measure specifications. The BNF of annotation value specification is as follows:

```
annotation-value-specification ::=  
    annotation-identifier = string-literal;
```

For example:

```
description = "System level verification features";
```

Goal Specification

You can specify a **goal** expression in a **metric** declaration, **plan/feature** hierarchy, or **override** declaration (see [“The override Modifier”](#)). The final evaluation of a **goal** expression is either true or false. Verification Planner first replaces the **metric** name identifier with the actual value of the metric in the measure, and then evaluates the expression to a Boolean value.

The following table lists the operator precedence for goal expressions from highest to lowest precedence.

Table 1-3 Operator Precedence and Associativity

Operator	Priority	Associativity
* , /	highest	left
+ , -		left
> , < , >= , <= , == , !=		left
! (logical not)		
&&		left

	lowest	left
--	--------	------

The following are restrictions on goal expressions. Invalid usage of expressions in any of these ways might cause errors at goal evaluation time.

- Numerical operators (*, /, +, -) cannot be used with non-numerical values such as ratio or string values.
- No numerical value may be divided by 0.
- Comparison operators cannot be used to compare string values.
- Internally, integer 0, real 0.0 and string "" values are regarded as FALSE if a Boolean value is expected. All the other values are regarded as TRUE.

The syntax to describe a goal expression is slightly different depending on where you define the expression. To specify a goal in a metric declaration, for example, the syntax is:

```
metric ratio MyLine;
    aggregator = average;
    goal = MyLine >= 80%;
endmetric
```

The above expression sets a `MyLine` metric goal of at least 80%.

If you want to override the goal of a specific metric in a feature or a plan, you must specify the metric name. For example:

```
feature MyFeature1;
    MyLine = MyLine >= 50%;
    ...
```

This example sets a Line coverage goal of 50% or higher.

You might want to override the goal in HVP modifier override (see [“The override Modifier”](#)). You need to specify the full path hierarchy with the metric name, as shown in the following example:

```
override MyOv;  
    MyPlan.TopFeature.MyLine >= 50 %;  
endoverride
```

Identifiers in goal expressions must be valid metric names, enum names or *metric-name.enum* names. The metric name is replaced with the actual score of the metric. If the metric is a ratio type, it is automatically converted to a percentage value while evaluating the goal.

The following are examples of valid goal expressions:

```
test = test > 10 && fail == 0;  
test = (pass + warn / test) > 80 % ;  
Line = Line >= 30%
```

Measure Specification

You can use measure specifications to declare which metrics to annotate a feature from the verification database. Measure specifications do not specify how values are extracted from the verification database. In general, a metric represents the declaration of a type, while a measure contains the actual instance of the source to be annotated. You should declare a measure specification within a feature (see [“Feature Declaration”](#)). Also, you must declare all metrics used in a measure specification in the same plan, except for built-in metrics.

Verification Planner typically attaches measure statements to the lowest level of a feature hierarchy. Metric values for high-level features are computed by aggregating metric values from lower-level child features.

The syntax to declare a measure is as follows:

```
measure metric-reference-list measure-name-identifier;  
    source = source-list;  
endmeasure  
  
metric-reference-list = metric-name [, ...]  
source-list = "string" [, ...]
```

You must specify one or more valid metric reference, and also specify **source** strings in the **measure** statement block.

The *source-list* is composed of one or more strings separated by commas. A source string can contain arbitrary data regions mixed with Synopsys coverage database files or HVP userdata files. Data regions are specified differently in the Synopsys coverage database than in the HVP userdata file, as described below.

Data Regions in the Synopsys Coverage Database Data Source

To specify Synopsys coverage database source regions, you must follow a predefined source format according to the metric type.

The syntax to declare a source for Synopsys coverage is:

```
"[keyword]:full hierarchical path separated by dots"
```

For example:

```

source = "module : module_name"
source = "instance : dut.hierarchy"
source = "tree : dut.hierarchy"
source = "group : covergroup_name"

```

[Table 1-4](#) describes the source format of each built-in metric.

Table 1-4 Source Formats for Built-In Metrics

Keyword	Available Metrics	Usage
module:	Assert, Code coverage metrics: Line, Cond, Toggle, FSM, Branch	Module name
instance:	Assert, Code coverage metrics: Line, Cond, Toggle, FSM, Branch	DUT instance hierarchy separated by dot Annotating scores only in the matched instance, not sub-hierarchy.
tree:	Assert, Code coverage metrics: Line, Cond, Toggle, FSM, Branch	DUT instance hierarchy separated by dot Similar to 'instance:', but annotating scores not only in the matched instance, but also sub-hierarchy.
property:	Assert, AssertResult	DUT instance hierarchy separated by dot followed by assertion or property name. The last element in the given hierarchy is regarded as the name of the property or assertion.
property: categoryMask "h###", property: severityMask 'h###' :	Assert	Similar to property: keyword usage. You can specify a three-digit (hex) filtering mask by category and severity, respectively.
group:	Group	Name of covergroup or covergroup.coverpoint can be specified.
group bin:	Group	Name of covergroup.coverpoint.bin can be specified. You can also specify covergroup.coverpoint.bin1-bin2 for crossbin.
group instance:	Group	Name of covergroup.instance or covergroup.instance.coverpoint can be specified.

group instance bin:	Group	Name of covergroup.instance.coverpoint.bin can be specified.
---------------------	-------	--

The easiest way to get the right region hierarchy name after the keyword is to refer to the URG reports:

- URG hierarchy report – shows existing DUT hierarchy
- URG assert report – shows existing property name with full DUT hierarchy
- URG group report– shows existing covergroup, coverpoint and bin names

You can copy and paste the hierarchy after the right keyword.

In order to match multiple regions in a measure, you can enumerate multiple source strings separated by commas, as follows:

```
measure Line, Cond m1;
    source = "tree: top.dut.u1", "tree: top.dut.u2",
            "tree:top.dut.u3";
endmeasure
```

A source name can be a wildcard matching pattern that contains “?” for a single character, “*” for an arbitrary string within a scope (matching a string between two dots), or “***” for an arbitrary string regardless of scope. The following are examples of the usage of each type of wildcard:

Source name pattern	Matching examples	Non-matching examples
tree: top.dut.u?	top.dut.u1 top.dut.u2 top.dut.uN	top.dut.u11 top.dut.u
tree: top.dut.*	top.dut.u1 top.dut.nnn	top.dut.u1.sub1 top.dut

tree: top.dut.*.sub1	top.dut.u1.sub1	top.dut.u1.u2.sub
tree: top.dut.**	top.dut.u1 top.dut.u1.u2	top.dut

Data Regions in External User Data Sources

You can specify source strings without keywords for non-Synopsys coverage metrics for which the data sources are userdata files. For a source without keywords, region strings have no hierarchy but is simply flat. Dots in the strings are not hierarchical separators. Therefore, you can use “?” wildcards to match single characters and “*” wildcards to match multiple characters in the strings.

The following are some examples using each of the wildcards:

Source name pattern	Matching examples
/test/block1/read_write*	/test/block1/read_write_rand_123 /test/block1/read_write/rand_123
top.u1.*	top.u1.sub1 top.u1.sub1.read1

Note that the “*” wildcard matches regardless of the dot scope.

Performance Tip

While Verification Planner internally traverses and annotates given data regions in the Synopsys coverage database and the HVP userdata file, it tries to match the current region path with every single source string in your verification plan. For example, suppose that your coverage database has 100,000 hierarchy regions including instances, assertions, coverpoints, bins, and so on. Also, suppose that your verification plan has 1,000 source strings in leaf node measures overall. Then Verification Planner tries to string match 100,000 * 1,000 times. More source strings result in longer loading times.

You can reduce the number of source strings by using wildcards in measures. For example, instead of using four source strings as in the following:

```
source = "tree: top.dut.u1", "tree: top.dut.u2",  
         "tree: top.dut.u3", "tree: top.dut.u4";
```

you can use a single pattern, for example:

```
tree: top.dut.u?
```

or

```
tree: top.dut.u*
```

Using Attributes in Source Specifications

You can also specify a source using attributes and annotations. For example:

```
plan cache_plan;  
  attribute string root_mod = "";  
  feature cache1;  
    root_mod = "top.";  
    measure Line cov;  
      source = "instance: ${root_mod}level2"  
    endmeasure  
  endfeature  
  feature cache2;  
    measure Line cov;  
      source = "instance: ${root_mod}level2"  
    endmeasure  
  endfeature  
endplan
```

In the above example, the attribute of `root_mod` in feature `cache1` is `"top."`, so Verification Planner interprets the source of measure `cov` in feature `cache1` as `instance:top.level2`. Because the attribute `root_mod` in feature `cache2` is a null string, the source measure `cov` of feature `cache2` is `instance:level2`.

You can also use integer and real type attributes, which will be replaced by their decimal values in the measure source string, but their numeric formats are different.

You can use a reserved variable, `${objpath}`, to retrieve the full path of the current measure hierarchy in the source string. For example:

```
plan cache_plan;
  feature cache1;
    measure bugcount mBug;
      source = "${objpath}";
    endmeasure
  endfeature
endplan
```

In example above, the `${objpath}` variable in measure `mBug` will be replaced with `cache_plan.cache1.mBug` at compile time.

The `Assert` metric supports selection of assertions by category and severity value masking. To set mask values, use the 24-bit hex number in either `property categorymask 'h###` or `property severityMask 'h###` source.

The `SnpsAvg` built-metric is an aggregate type which consists of `Line`, `Cond`, `FSM`, `Toggle`, `Assert`, and `Group` built-in metrics, so you can use any source format listed in [Table 1-4](#). For details on built-in metrics, see [“Built-In Metrics”](#).

Examples

Example 1-1 A Property and the Corresponding Vera Code

```
source = "property: **.tinv", "property: **.t3";

//Vera sample code
unit test_u( logic clk, logic seq_start, logic seq_bit,
  logic t);

  clock posedge(clk) {
    event e_tinv: if (seq_start) then (
      (!seq_bit || (any #1 !seq_bit)) #1
      seq_bit #1 !seq_bit
    );

    event e_t3: if (seq_start) then (
      (any *[0..3]) #1
      ((matched event_result1) #1 (tdown))
    );
  }

  assert tinv: check(e_tinv);
  assert t3: check(e_t3);

endunit

bind module test : test_u test_b( clk, seq_start,
  seq_bit, t);
```

Example 1-2 Assert Metric Using Vera

```
unit test_u( logic clk, logic seq_start, logic seq_bit,
  logic t);

  clock posedge(clk) {
    event e_tinv: if (seq_start) then (
      (!seq_bit || (any #1 !seq_bit)) #1
      seq_bit #1 !seq_bit
    );

    ...
  }
```

```

        event e_t3: if (seq_start) then (
            (any *[0..3])#1
            ((matched event_result1) #1 (tdown))
        );
    }

    assert tinv: check(e_tinv);
    assert t3: check(e_t3);

endunit

bind module test : test_u test_b( clk, seq_start,
    seq_bit, t);

module: test_u
tree: test.test_b
property: test.test_b.tinv

```

Example 1-3 Assert Metric Using PSL

```

module m;
    reg clk = 0;
    reg a = 1, b = 0;

    // psl default clock = (posedge clk);
    // psl AA: assert always {a ; b} ;

    initial begin
        #6 b = 1;
        #4 $finish;
    end
    always #1 clk = !clk;
endmodule

module top
    m m1; //instantiating module m
endmodule

"module: m"
"tree: top.m1" or "tree: **.m1"
"property: top.m1.AA" or "property: **.m1.AA"

```

Example 1-4 Assert Metric Using SystemVerilog

```
module m;
    reg a, clk;
    always @(posedge clk)
    begin
        AST : assert (a)
            $display("Pass");
            else $display("Fail");
    end

    initial
    begin
        ...
    end
endmodule

module top
    m m1; //instantiating module m
endmodule

"module: m"
"tree: top.m1" or "tree: **.m1"
"property: top.m1.AST" or "property: **.m1.AST"
```

Example 1-5 Group Metric Using Vera

```
class div1 {
    ...
    coverage_group trans_type {
        ...
        sample covvalue {
            state ssix(3'b101);
            trans ttwo1 ("szero"->"stwo");
        }

        cross cross_t (mask_hi, mask_lo);
    }
}
```

```

coverage_group Test_cov(sample bit[2:0] covvalue) {
    ...
}

program veracov {
    ...
    div1 marg1 = new();
    ...
}

group: Test_cov
group: div1::trans_type
group: div1::trans_type.covvalue
group: div1::trans_type.cross_t
group bin: div1::trans_type.covvalue.ssix

```

Example 1-6 Group metric using SystemVerilog

```

module my_mod();
    bit success;
    reg clk;
    int a,b;
    integer c,d;
    bit [9:0] ra;
    covergroup gc (int abm) @ (posedge clk);
        type_option.per_instance = 1;
        my_auto_cp: coverpoint ra {
            option.auto_bin_max = abm;
        }
    endgroup

    initial begin
        gc c2 = new (32);
        gc c3 = new (50);
    end

endmodule

group: gc
group: gc.my_auto_cp
group instance: gc.c2

```



```
group instance: gc.c2.my_auto_cp
```

Example 1-7 Measure Specification

```
measure Line line_cov;  
    source = "module: mod*";  
endmeasure  
  
measure Line, Cond, Group m_cov;  
    source = "instance: top.cpu", "instance: top.reg";  
endmeasure  
  
measure Assert ast_cov;  
    source = "property: top.**.readbuf";  
endmeasure  
  
measure Assert cat_mask;  
    source = "property categoryMask `h00f: top.cpu.*"  
endmeasure  
  
measure Assert sev_mask;  
    source = "property severityMask `h010: top.cpu.*"  
endmeasure
```

Subplan Declaration

A plan or feature can contain one or more subplans as if they contain subfeatures. To add a plan under another plan or feature, you need to use the keyword `subplan` with one mandatory plan name identifier as illustrated in the following example.

Note:

You need to first declare a plan identifier where the subplan resides (see [“Plan Declaration”](#)), otherwise Verification Planner-enabled applications (see [“Verification Planner-Enabled Applications”](#)) issue an error message of “unknown plan”. After you add the subplan, use the subplan name identifier to access the subplan.

The BNF for subplan is as follows.

```
subplan-declaration ::=  
    subplan plan-identifier[#(subplan-parameters)];  
  
plan-parameters ::= attribute-value-specification  
    {,attribute-value-specification}  
  
attribute-value-specification ::= attribute-identifier  
    = attribute-value-literal
```

In the following example, `cache_plan` is a subplan of `cpu_plan`:

```
plan cache_plan;  
    attribute string root_mod = "";  
    feature cachel;  
        measure Line cov;  
            source = "instance: ${root_mod}level2"  
        endmeasure  
    endfeature  
endplan  
  
plan cpu_plan;  
    subplan cache_plan #(root_mod="top.");  
    ...  
endplan
```

One set of verification plan files may have multiple plan definitions (each plan definition is encapsulated in a `plan...endplan` block). However, all plans must be used as subplans except for the last plan

definition. Only the last plan can be the top-level plan. In the following example, only `plan C` is the top-level plan, while `plan A` is a subplan of `plan B` and `plan B` is a subplan of `plan C`.

```
plan A
plan B ( subplan A)
plan C ( subplan B )    //plan C is the top-level plan
```

If you specify several plans without declaring them as subplans, this results in more than one top-level plan and Verification Planner-enabled applications will issue an error message. In the following example, none of the plans are declared as a subplan, therefore `plan A`, `B`, and `C` are all top-level plans. This results in an error condition.

```
plan A
plan B
plan C                //Error, plan A, B, C are all top-level plans
```

If you specify several plans without declaring them as subplans, only the last plan is processed as top-level plan and the rest of them are ignored. In the following example code snippet, plans `A` and `B` are ignored, and `plan C` is the top-level plan.

```
...
plan A    // ignored
plan B    // ignored
plan C    // top-level plan
...
```

A Complete HVP Subplan Example

Example 1-8 HVP Subplan Example

```
plan my_sub_plan;
    attribute integer phase = 0;
    feature system;
```

```

        owner = "R&D team";
    endfeature
endplan

plan my_plan;
    metric integer num_bugs;
        goal = num_bugs == 0;
    endmetric

    attribute integer phase = 0;
    annotation string spec_url = "";
    subplan my_sub_plan;

feature system;
    owner = "QA team";
    phase = 1;
    spec_url = "http://www.synopsys.com/verification/
        spec/system.htm";

feature block1;
    phase = 3;
    spec_url = "http://www.synopsys.com/
        verification/spec/block1.htm";
    measure test my_test;
        source = "test1", "test3", "test7";
    endmeasure
    weight = 2;
    measure Group my_groupA;
        source = "groupA";
    endmeasure
endfeature

feature block2;
    weight = 2;
    measure test my_test;
        source = "test1", "test2", "test4";
    endmeasure
    measure SnpsAvg, Line, FSM my_groupB;
        source = "groupB";
    endmeasure
    measure Line, Toggle, num_bugs topd_cov;
        source = "top.d*";

```

```
        endmeasure
    endfeature

endfeature
endplan
```

Plan Modifiers

You can modify a verification plan hierarchy outside of the verification plan definition files using plan modifiers: `override` and `filter` (each plan definition is encapsulated in a `plan...endplan` block). Verification Planner-enabled applications first process the entire verification plan in the HVP language, and then apply modifiers after the plan hierarchy is loaded. Furthermore, Verification Planner-enabled applications apply multiple `override` and `filter` modifiers sequentially in the order they are given.

Modifiers are useful to create derivatives or alter plans based on a common master plan. For example, use the `override` modifier to relax goals during early phases of a project and then gradually tighten those goals later as tests are created and improved.

Use the `filter` modifier to generate reduced views for specific purposes of a large master plan. For example, apply a filter to limit the results to only those features assigned to phase 2 of a project.

This section contains the following subsections:

- [“The override Modifier”](#)
- [“filter”](#)

The override Modifier

The `override` modifier changes values of attributes, annotations, or goal expressions in metrics for one or more features in a verification plan. Apply the `override` modifier to attribute values, annotation values, and metric goal expressions only. The modifiers use an XMR-style full pathname to reference an attribute, an annotation, or a metric, and override it with a different value or goal expression:

```
plan.subplan-or-feature. ... .attribute = value;  
plan.subplan-or-feature. ... .annotation = value;  
plan.subplan-or-feature. ... .metric = expression
```

Encapsulating one or more `override` statements within an `override...endoverride` block enables Verification Planner to execute each statement sequentially. Verification Planner passes the override values (or goal expressions) down the plan hierarchy to override attribute assignments (or goal expressions) in the hierarchy sub-trees.

The BNF for `override` is:

```
override-specification ::=  
    override override-identifier;  
        ( override-assignment )  
    endoverride  
  
override-assignment ::=  
    attribute-full-identifier = attribute-value; |  
    annotation-full-identifier = annotation-value; |  
    metric-full-identifier = metric-value;
```

The following is an example using `override` to change the goal of Line coverage for `myplan.DVD_RW` to be greater than 85%:

```

override milestone_1;
    myplan.DVD_RW.Line = Line > 85%;
endoverride

```

An attribute value that is assigned in an **override** statement is immediately propagated all the way down to the leaf node features. This means that if you specify an attribute override statement of a child node followed by the same attribute override of a parent node, the child node override, specified first, becomes meaningless. However, the same is not true for annotation overrides because annotation values are not propagated downward. In the following example, the resulting owner is Second Owner for all instances in the `topplan.subplan1` hierarchy level and below, including the `topplan.subplan1.mem` instance.

```

override ov_owner;
    /* change the weight attribute in feature top.feas.mem */
    topplan.subplan1.mem.owner = "First Owner";
    topplan.subplan1.owner = "Second Owner";
endoverride

```

You can use wildcard patterns in an `override...endoverride` block to specify scope names such as plans and features, but not attribute names. [Table 1-5](#) lists the available wildcard characters:

Table 1-5 Using Wildcards to Specify Plans and Features

Wildcard	Meaning
?	Matches a single occurrence of any character in the name.
*	Matches zero or more consecutive characters in a single hierarchy (between two "." hierarchy separators).
**	Matches zero or more consecutive characters regardless of hierarchy.

Examples

```

top.*.weight = 2;
top.**.weight = 2;

```

```
top.u?.weight = 2;
```

filter

Specify `filter...endfilter` block outside the plan definition to extract a set of features. This is useful when you want to view part of the HVP reports pertaining to a sub-project only. The BNF for `filter` modifier is as follows:

```
filter-specification ::=
    filter filter-identifier; {filter-statement}
endfilter

filter-statement ::=
    keep filter-expression; / remove filter-expression;
filter-expression ::=
    expression
```

The `filter-expression` consists of the keywords `feature` where followed by an attribute or an annotated name, and a condition. Any identifier other than an attribute or annotated name is not interpreted.

The following example shows a filter `my_view` which defines specific criteria for a given collection of feature attributes.

```
filter my_view;
    remove feature where phase > 2;
    keep feature where phase == 5;
    keep feature where name == "N1" || name == "N2";
endfilter
```

Initially, all the features in `my_view` are selected by default. After applying the filtering operation: `remove` and `keep`, the first filter expression eliminates features whose phase values are greater than

2, the second filter expression extracts features whose values are equivalent to 5, and the last expression extracts features with the names of either N1 or N2.

If a feature is filtered out, Verification Planner excludes the corresponding measure score of the feature from propagating through the entire plan hierarchy.

You can also include the following arbitrary complex expressions in filter patterns:

>	<	>=	<=	==
+	-	*	/	
	&&	()	

Currently, Verification Planner does not support the following expressions:

```
phase inside {1:10 }  
match (owner, "bo*")
```

until Statement – Time-Based Modifier

As a project develops, you can specify a time-based selective block, using an **until** statement, to apply different override or filter rules. The **until ... elseuntil ... enduntil** block can wrap **override** and **filter** statements, and only the block that meets the current time conditions is applied to the verification plan.

The syntax of the **until** statement is:

```
until date-format;  
    override-statement | filter-statement;  
{ elseuntil date-format;  
    override-statement | filter-statement; }
```

```
{ else;
    override-statement | filter-statement; }
enduntil
```

Example

```
until 1-31-2010;
    /* applied before 1/31/2010 */
    override ov_phase1;
        MyPlan.phase = 1;
        MyPlan.owner = "R&D";
    endoverride
elseuntil 04-30-2010;
    /* applied between 2/1/2010 - 4/30/2010 */
    override ov_phase1;
        MyPlan.phase = 2;
        MyPlan.owner = "Verification Team";
    endoverride
else;
    /* applied after 5/1/2010 */
    override ov_phase1;
        MyPlan.phase = 3;
        MyPlan.owner = "Field Engineer";
    endoverride
enduntil
```

Compiler Directives

Unlike Verilog modules which are instantiated separately from their declarations, feature instantiations in a verification plan appear where the declarations exist. However, you can move feature declarations using compiler directives. For example, use the ``include` compiler directive to accept a file containing feature declarations.

The Verification Planner compiler directives allow simple macro definition similar to the Verilog Standard, IEEE Std. 1364-2001.

For example:

```
`define foo \  
    feature my_feature; \  
    ... \  
endfeature
```

Verification Planner supports the following compiler directives which have the same definition as in Verilog:

```
`undef macro_name  
  
`include "file_name"  
  
`ifdef macro-identifier  
    group-of-lines  
{`elsif macro-identifier  
    group-of-lines }  
[`else  
    group-of-lines ]  
  
`ifndef macro-identifier  
    group-of-lines  
{`elsif macro-identifier  
    group-of-lines }  
[`else  
    group-of-lines ]
```

Note that the ``include "file_name"` compiler directive searches the `file_name` based on the current file directory.

Comments

There are two ways to introduce comments using the HVP language: the line comments and the block comments. You can use double forward slashes `("//")` to denote the start of a line comment which stops

at the end of the line. For a block comment, the block starts with "/*" and ends with "*/". A block comment which spans multiple lines may not be nested. A line comment token "//" has no meaning inside a block comment.

How to Use HVP Files

You can use HVP plan and modifier files with URG for annotation and DVE for editing. Also, you can pass modifier files into Verification Planner spreadsheet annotator with plan sheets. Refer to the *Unified Coverage Reporting* and *Discovery Visual Environment* user guides for additional information.

2

Verification Planner Spreadsheet Annotator

This chapter describes how to use the Verification Planner Spreadsheet Annotator application, how to tag spreadsheets, and the annotation command-line options.

This chapter contains the following sections:

- [“Introduction to Spreadsheet Annotator”](#)
- [“Getting Started with Spreadsheet Annotator”](#)
- [“Using HVP Spreadsheet Meta-Tags”](#)
- [“Using Spreadsheet Annotator Commands”](#)
- [“Using the HVP_ARCH_OVERRIDE Variable”](#)
- [“Troubleshooting”](#)

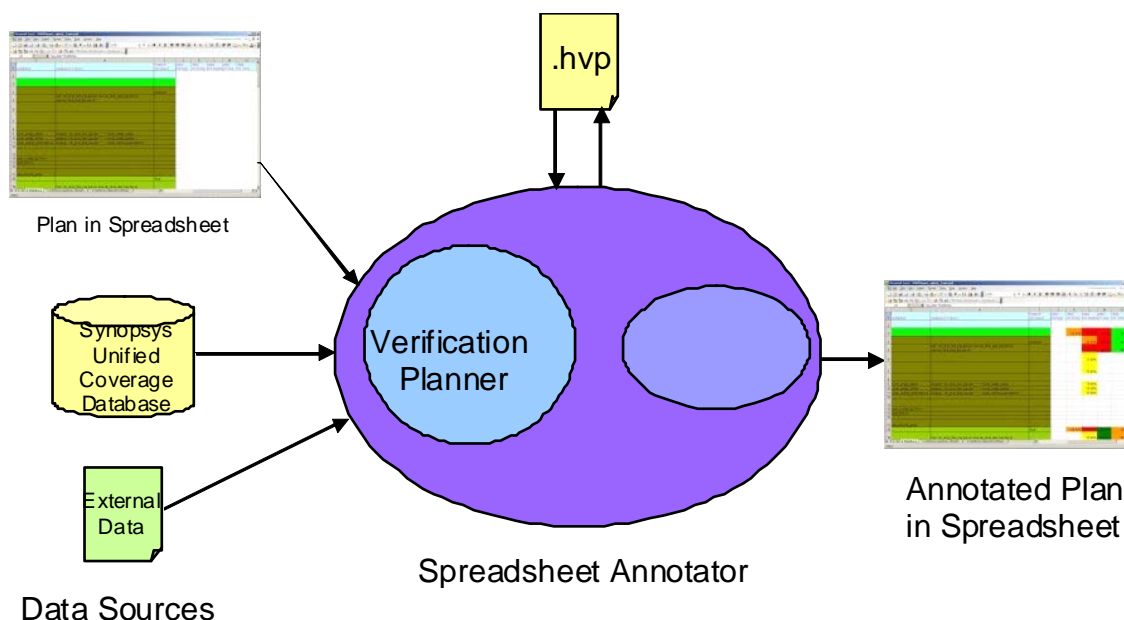
Introduction to Spreadsheet Annotator

There are two ways to describe a verification plan:

- Using the Synopsys proprietary HVP language is the most straightforward method (for additional information on how to create a plan, refer to [“Using the HVP Language” on page 20](#)).
- Describe an implicit verification plan using spreadsheets which are properly formatted with the inclusion of identifying meta-tags.

To take the implicit verification plan approach, you need to use either MS Excel or OpenOffice 2.0 (or later) to edit HVP spreadsheets for back-annotation purpose. The back-annotation process is accomplished via the Spreadsheet Annotator. [Figure 2-1](#) depicts the usage model of the Verification Planner Spreadsheet Annotator:

Figure 2-1 Verification Planner-Enabled Application: Spreadsheet Annotator



The purposes for spreadsheet annotation application are to provide:

- Reasonably simple hooks in MS Excel compliant spreadsheets to interface with Verification Planner-enabled applications.
- Enough degrees of freedom in formatting spreadsheets to promote reusability. Verification Planner uses the same spreadsheets as a verification plan and annotation files. You may use spreadsheet formatting features such as font, attributes, insertion of images, and cell attributes. This allows you the flexibility to design spreadsheets for conventional documentation purposes.
- A mechanism for using the features of Verification Planner by means of tagging spreadsheets with reserved meta-tags.

Getting Started with Spreadsheet Annotator

The following steps describe the basic usage model of the Verification Planner Spreadsheet Annotator:

1. Create spreadsheets to capture a verification plan.

The verification plan contains at most one HVP Plan Sheet, an optional HVP Metric Sheet, and an optional HVP Attribute Sheet. Label columns and rows of each spreadsheet with meta-tags which are interpreted by Verification Planner (see [“Using HVP Spreadsheet Meta-Tags”](#) on how to tag a spreadsheet).

2. Save spreadsheets as XML file format.

If you are using Excel 2007 and you want to be able to load the saved spreadsheets into Excel 2003, choose the XML Spreadsheet 2003 option to Save As, not the XML Data option.

3. Execute the `hvp annotate` command with the appropriate options, for example:

```
hvp annotate -plan_in=xml_in_file
```

For additional information, see [“Using Spreadsheet Annotator Commands”](#).

4. Examine the back-annotated spreadsheets. The annotated spreadsheets are almost the same as the original ones except for the cells that are annotated with metrics and color-coded according to goal values.

After annotation, an annotated cell is highlighted in green if the corresponding goal value of the cell is met, otherwise the cell is highlighted in red. If no goal is specified, Synopsys coverage and test scores are annotated with various colors as shown in [Figure 2-3](#).

[Figure 2-2](#) shows an example of an original spreadsheet versus an annotated spreadsheet.

Figure 2-2 *Original Spreadsheet vs. Annotated Spreadsheet*

Original Spreadsheet

	A	B	C	D	H	I	J
1	hvp plan	feature	subfeature	goal.bugs	measure	value	value
2	myplan	top1			m2.source	m2.Line	m2.Condition
3			sub1	bugs <= 1.5	blank		
4			sub2	bugs <= 1.0	module: kp_fsm		
5			sub3		module: coin_fsm		
6			sub4		instance: test_jukebox.fifo1		
7					instance: test_jukebox.jb1		

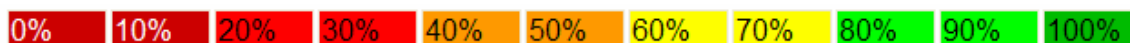
Annotated Spreadsheet

	A	B	C	D	H	I	J
1	hvp plan	feature	subfeature	goal.bugs	measure	value	value
2	myplan	top1			m2.source	m2.Line	m2.Condition
3			sub1	bugs <= 1.5	blank	89.66%	54.05%
4			sub2	bugs <= 1.0	module: kp_fsm	90.12%	42.59%
5			sub3		module: coin_fsm		
6			sub4		instance: test_jukebox.fifo1	Line:96.00%	
7					instance: test_jukebox.jb1	Cond:42.86%	2.86%

After you apply the `hvp annotate` command, the corresponding annotated cells are highlighted in green when goals are met, and in red otherwise, provided that the goals of metrics are specified. If you do not specify goals, the annotated cells will not be highlighted.

However, for a ratio or percent type metric or test metric, even if no goal is given, the cell is highlighted in one of six colors from red (low) to green (high) depending on the metric's percentage value, as shown in [Figure 2-3](#).

Figure 2-3 Color Legend for Synopsys Coverage Metrics



Note:

For the `test` metric, the cell color is based on the (pass count)/(total test count) percentage.

5. In an annotated spreadsheet, a cell for a measure source or feature may have a pop-up window that shows details of its scores and matched region. The information in the pop-up window is useful in determining the exact matching region strings and scores, if the source string contains wildcards or if there are multiple sources in the cell.

For example, [Figure 2-4](#) shows the pop-up window for the source string `/cfg1/CPU1/L1CACHE/*` and seven matched testcases, with an annotated score of `total=7 pass=6 fail=1`. The pop-up window illustrates details about how this score was aggregated.

/cfg1/CPU1/L1CACHE/*	<div> <div>total=7 pass=6 fail=1</div> <div>il=9 warn=1</div> <div>s=6 warn=1</div> </div>
/cfg1/CPU1/L2CACHE/*	
/cfg1/CPU1/ALU/*	
<div> <div>/cfg1/CPU1/L1CACHE/test1: test(total=1 pass=1)</div> <div>/cfg1/CPU1/L1CACHE/test2: test(total=1 pass=1)</div> <div>/cfg1/CPU1/L1CACHE/test3: test(total=1 pass=1)</div> <div>/cfg1/CPU1/L1CACHE/test4: test(total=1 pass=1)</div> <div>/cfg1/CPU1/L1CACHE/test5: test(total=1 fail=1)</div> <div>/cfg1/CPU1/L1CACHE/test6: test(total=1 pass=1)</div> <div>/cfg1/CPU1/L1CACHE/test7: test(total=1 pass=1)</div> </div>	

If you use the **-report** *urg-report-path* switch, you will see hyperlinks at source and feature cells. Clicking one of those hyperlinks opens the URG (Unified Report Generator) report in your Web browser, so you can see details in the URG/HVP report. See [“Using Spreadsheet Annotator Commands”](#) and the *Unified Coverage Reporting User Guide* for details about using the **-report** switch.

Verification Planner allows you to describe a verification plan using one or more spreadsheets depending on the contents of the verification plan. Use the following sheet types to label each spreadsheet:

- ## Verification Planner Spreadsheet Annotator

- “HVP Metric Definition Sheet”
- “HVP Attribute Definition Sheet”
- “Applying Modifiers”
- “Sharing a Metric/Attribute Definition Sheet”

In the following example, the plan `my_plan` is written in the HVP language and it's corresponding to three different types of HVP spreadsheets. You use HVP Metric Definition Sheet to describe the `metric...endmetric` section, HVP Attribute Definition Sheet to describe the `attribute` section, and HVP Plan Sheet to describe the `feature...endfeature` section. In addition to meta-tags for each sheet type, each HVP spreadsheet also has unique meta-tags for annotation purposes.

Table 2-1 Plan Definition and Spreadsheet Types

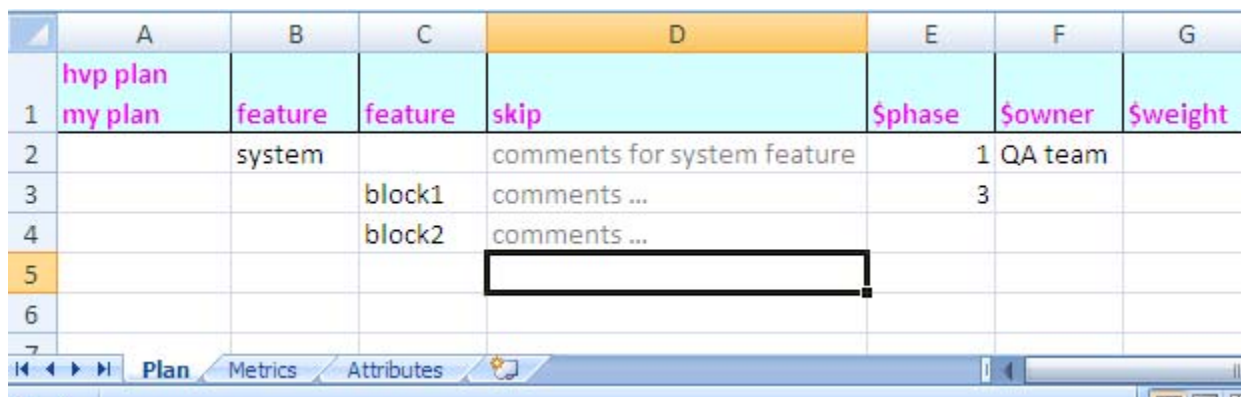
<code>plan my_plan;</code>	
<code>metric integer num_bugs; goal = num_bugs == 0; aggregator = sum; Endmetric</code>	HVP Metric Definition Sheet
<code>attribute integer phase = 0;</code>	HVP Attribute Definition Sheet

<pre> feature system; owner = "QA team"; phase = 1; feature block1; phase = 3; measure test my_test; source = "test1", "test3", "test7"; endmeasure weight = 2; Group = Group >= 80.0 measure Group my_groupA; source = "groupA"; endmeasure endfeature feature block2; weight = 2; measure test my_test; source = "test1", "test2", "test4"; endmeasure measure Group my_groupB; source = "groupB"; endmeasure measure Toggle, num_bugs top_cov; source = "top.d*"; endmeasure endfeature endfeature </pre>	HVP Plan Sheet
<pre> endplan </pre>	

HVP Plan Sheet

HVP Plan Sheet describes the feature hierarchy of a verification plan. A verification plan in spreadsheet format consists of, at most, one HVP Plan Sheet. [Figure 2-5](#) illustrates a simple HVP Plan Sheet.

Figure 2-5 An HVP Plan Sheet



	A	B	C	D	E	F	G
1	hvp plan my plan	feature	feature	skip	\$phase	\$owner	\$weight
2		system		comments for system feature	1	QA team	
3			block1	comments ...	3		
4			block2	comments ...			
5							
6							

The screenshot shows a spreadsheet interface with tabs labeled 'Plan', 'Metrics', and 'Attributes'. The 'Plan' tab is active, displaying the HVP Plan Sheet. The spreadsheet has columns A through G and rows 1 through 7. The data is organized into a hierarchy: 'hvp plan my plan' (A1) points to 'feature' (B1), which points to 'feature' (C1). 'feature' (B1) points to 'system' (B2), which points to 'comments for system feature' (D2). 'feature' (C1) points to 'block1' (C3) and 'block2' (C4), which point to 'comments ...' (D3) and 'comments ...' (D4) respectively. The 'comments for system feature' (D2) points to '1 QA team' (E2). The 'comments ...' (D3) points to '3' (E3). The 'comments ...' (D4) points to an empty cell (D5). The 'comments for system feature' (D2) also points to '3' (E3). The 'comments ...' (D3) points to an empty cell (D4). The 'comments ...' (D4) points to an empty cell (D5). The 'comments for system feature' (D2) points to '1 QA team' (E2). The 'comments ...' (D3) points to '3' (E3). The 'comments ...' (D4) points to an empty cell (D5). The 'comments for system feature' (D2) points to '1 QA team' (E2). The 'comments ...' (D3) points to '3' (E3). The 'comments ...' (D4) points to an empty cell (D5).

This section contains the following subsections:

- “hvp plan plan-name”
- “plan”
- “feature”
- “subplan”
- “\$attribute-name, \$annotation-name”
- “skip”
- “goal.metric-name”
- “measure measure-name.source”

- “value measure-name.metric-name”
- “include”

hvp plan *plan-name*

In [Figure 2-5](#), the cell which contains the `hvp plan plan-name` meta-tag identifies this sheet as an HVP Plan Sheet. The row and column corresponding to this cell are called tag row and tag column, respectively. In the example, the meta-tag cell uniquely specifies the verification plan as `my_plan` and row 2 is the tag row.

plan

The “plan” tag is located in the tag column, while all the other tags are located in the tag row. The row with the “plan” tag is used to set attribute or goal values at the plan level, and to annotate plan-level scores. [Figure 2-6](#) uses row 2 for plan-level hierarchy.

Figure 2-6 Location of the “plan” Tag in the Tag Row

	A	B	C	D	E	F	G	H
1	hvp plan myplan	feature	feature	skip	Sphase	Sowner	Sweight	goal.test
2	plan				1			test.fail == 0
3		system		comments for system feature		QA team		
4			block1	comments...	2		1	
5			block2	comments...			2	
6								
7								
8								
9								

feature

The columns with a meta-tag feature indicate a feature/subfeature hierarchy level. Each HVP Plan Sheet can have one or more feature columns. The order of these columns from left to right represents feature hierarchy level from top to bottom. In [Figure 2-5](#), “system” is the top-level feature, and it consists of two subfeatures, “block1” and “block2”.

The corresponding feature cells of the same row are mutually exclusive. In other words, you can have at most one feature value for any given row. If a row does not have any feature name, the row is automatically a subfeature row to the row above.

In [Figure 2-7](#), rows 6,7 and 8 do not have any feature values. (Internally, Verification Planner Spreadsheet Annotator creates a dummy feature value for each of these rows). By default, these three rows automatically become subfeatures to the row above, that is `instance1.test_jukebox_total.st0_total`.

Figure 2-7 HVP with Meta-Tag: feature

	A	B	C	D	E	
1	hvp plan myplan	feature	feature	feature	measure m1.source	\$p
2		instance1			blank	
3			test_jukebox_total			
4				jb1	instance: test_jukbox.jb1	
5				st0_total		
6					instance: test_jukebox.st0	
7					instance: test_jukebox.st0.coin1	
8					instance: test_jukebox.st0.kp1	
9				st1_total		

Although the HVP language allows only alphabetical characters, numbers, and underscores (_) in identifiers such as feature names, you can use any arbitrary strings as feature names. Internally, special characters or white spaces are replaced with "_" when the spreadsheet is converted to the HVP language. Because of this conversion, different strings might end up with the same feature name internally, which causes a naming conflict. For example, both "CPU READ-WRITE!" and "CPU READ-WRITE#" are converted to "CPU_READ_WRITE_".

If a feature name conflict occurs, `hvp annotate` shows the locations of the features for which the conflict occurs. You must change one of the feature names to eliminate the feature name conflict.

subplan

The `feature` or `subfeature` column can also have a subplan name instead of a feature name as a child subfeature. To put a subplan name in a cell, enter `subplan planname` and add the

subplan file name in the column with the `include` meta-tag in the plan spreadsheet. You can put the include filename in any row in the "include" column (for additional information, see ["include"](#)).

In the following example, `cache_plan` was defined in `cache_plan.xml`, and was used as a subplan in `myplan`.

Figure 2-8 `cache_plan` embedded as a subplan in `myplan`

	A	B	C	D	E	
1	hvp plan myplan	include	feature	subfeature	measure m1.source	valu m1.
2		cache_plan.xml				
3			instance1			
4				test_jukebox		
5			subplan cache_plan			
6						

When you embed a subplan, you can override attribute values in the subplan using `#(attribute-name=value, ...)` syntax. This is useful if you want to change attribute values in the subplan without editing the plan XML file. The overridden attribute must already have been defined in the subplan.

Figure 2-9 Overriding root_mod and weight attribute values in cache_plan

	A	B	C	D	E
1	hvp plan myplan	include	feature	subfeature	measure m1.source
2		cache_plan.xml			
3			instance1		
4				test_jukebox	
5			subplan cache_plan #(root_mod="top.", weight=2)		
6					
7					

\$attribute-name, \$annotation-name

Verification Planner Spreadsheet Annotator interprets any meta-tag that starts with the “\$” character as an attribute or annotation. For example, in [Figure 2-10](#), \$phase is an attribute. You must define all attributes used in the HVP Plan Sheet, except for the built-in owner attribute and the built-in description and weight annotations.

Figure 2-10 An HVP Plan Sheet Showing Meta-tag: skip

	A	B	C	D	E
1	skip				
2	hvp plan myplan	feature	feature	skip	\$phase
3		system			
4			block1		
5			block2		
6					
7					
8					
9					

skip

For annotation purposes, you can skip a row or column by entering the `skip` keyword in the desired tag row or tag column, as shown in [Figure 2-10](#). The Verification Planner Spreadsheet Annotator ignores column D because of the keyword `skip`. This is useful when you want to use spreadsheet functionality to annotate row or column cells rather than applying HVP aggregators. You can also use `skip` with a column meta-tag to temporarily ignore a column during the annotation process. For example, enter `skip feature` to ignore column C, the `feature` meta-tag column.

goal.metric-name

To override the goal of a metric, enter a Boolean expression in the corresponding cell of the column containing the `goal` meta-tag. The new goal expression will be inherited by the entire hierarchy just like attributes. For example, column J in [Figure 2-11](#) contains the goal of a Group metric to be 80 percent or higher. If you do not specify goals in the HVP Plan Sheet, Verification Planner Spreadsheet Annotator uses goals defined in the metric definition by default.

You must define a metric in the HVP Metric Sheet before you can use it in a goal expression except for Synopsys built-in metrics: Line, Cond, FSM, Toggle, Assert, Group, SnpsAvg, and test. The section, [“Goal Specification” on page 38](#), describes the syntax and rules for goal expression.

Figure 2-11 An HVP Plan Sheet Showing Meta-Tag: goal

G	H	I	J	K	L	M
	measure	value		measure	value	measure
\$weight	my_test.source	my_test.test	goal.Group	my_group.source	my_group.Group	top_cov.source
2	test1, test2, test3		Group >= 80%	group: GroupA		blank
2	test1, test2, test3			group: GroupB		instance: top.d*

measure *measure-name*.source

This meta-tag column contains cells which identify sources for measure *measure-name*. You can specify one or more source names or patterns, which may include wildcards: "?", "*", and **. In [Figure 2-11](#), the cell content `top.d*` in column M indicates that `top.d*` is the source of measure `top_cov`.

You can also enter more than one source in a single cell using the “,” character as a delimiter, for example, `test1, test2, test3` in column H. Notice that the *measure-name* is a user-defined name and should be unique within a plan sheet (for details, see [“Measure Specification” on page 40](#)).

If the source of a measure is not given, Verification Planner Spreadsheet Annotator annotates the corresponding `value` cell with the aggregated value of its subfeatures (for details, see [“value *measure-name*.metric-name”](#)).

value *measure-name.metric-name*

This `value` meta-tag column contains values to be back-annotated by the given metric *metric-name* to the corresponding sources for each measure *measure-name*. Notice that cells in `measure measure-name.source` column provide sources for the measure *measure-name*. For example, `jukebox` is a source for measure `m1`. Verification Planner Spreadsheet Annotator associates each source and value using the *measure-name* in a spreadsheet.

In [Figure 2-12](#), cell D2 of `measure m1.source` column does not have a `source` value, therefore, the value 65.41% in cell E2 is the aggregated value of 7 subfeatures, namely cells E3 to E9. Verification Planner Spreadsheet Annotator computes the aggregation result (E2) based on the aggregator in its metric definition. If you do not want Verification Planner to annotate a particular `value` cell, enter the keyword `blank` in the corresponding `source` cell.

Figure 2-12 An HVP Plan Sheet Showing Meta-Tag: value

	A	B	C	D	E
1	hyp plan PlanCodeCov	feature	subfeature	measure m1.source	value m1.Line
2		modules			65.41%
3			kp_fsm	module: kp_fsm	89.66%
4			coin_fsm	module: coin_fsm	41.98%
5			jukebox	module: jukebox	96.00%
6			station	module: station	
7			test_jukebox	module: test_jukebox	62.86%
8			fifo	module: fifo	88.24%
9			cd	module: cd	100.00%
10		instances			45.32%
11			test_jukebox	instance: test_jukebox	62.86%
12			st all	instance: test jukebox.	53.64%

include

To use other spreadsheets as subplans, enter the spreadsheet XML file names in the `include` meta-tag column. The external file name can have a relative path from the current file path. Verification Planner Spreadsheet Annotator will ignore all other cells in the row which has an include file except for the cell in the `include` column.

You can put multiple files in a single cell separated by commas. For example:

```
sub1/JukeST0_2.xml, sub2/JukeST3_4.xml
```

You can have a plan definition to be used as subplan in HVP language format, not in spreadsheet XML format. You can also include the HVP file and use the plans from a spreadsheet.

If you use `ralgen` to generate SystemVerilog files based on your RAL register description, you get HVP files from `ralgen` automatically. The HVP files can be included in your spreadsheet verification plan, and the plan in the HVP file can be used as a subplan.

You can include common metric and attribute definition sheets, as well as, plan XML files. If you have a list of metric and attribute definitions that are used across multiple plans, and you do not want to copy and paste them in multiple XML plan files, you can create a common metric and attribute definition XML, and simply include it. Then the metric and attribute definitions are automatically integrated into the current plan (for more information, see [“HVP Metric Definition Sheet”](#) and [“HVP Attribute Definition Sheet”](#)).

Figure 2-13 HVP Plan Showing Meta-Tag: include

hvp plan				
PlanJukeboxTop	include	feature	subfeature	mea
	sub1/JukeST0_2.xml			m1.s
	sub2/JukeST3_4.xml			
		top		
			test_jukebox	insta
			cd1	insta
			fifo1	insta
			jb1	insta
			subplan PlanJukeboxST0_2	
			subplan PlanJukeboxST3_4	

HVP Metric Definition Sheet

Metrics are values annotated to a verification plan from the verification database. Metric declarations specify the name, type, goal and aggregator for each metric. Descriptions of Verification Planner metrics and how they are aggregated for multiple scores are provided in [“Metric Declaration”](#).

[Figure 2-14](#) illustrates an example of an HVP Metric Definition Sheet:

Figure 2-14 HVP Metric Definition Sheet

	A	B	C	D	E	F	G
1	hvp metric						
2	my_plan	name	type	aggregator	goal		
3		num_bugs	integer	sum	num_bugs == 0		
4							
5							
6							
7							

This section contains the following topics:

- “hvp metric plan-name”
- “name”
- “type”
- “goal”
- “skip”

hvp metric *plan-name*

The cell in a spreadsheet containing `hvp metric plan-name` meta-tag identifies this spreadsheet as an HVP Metric Definition Sheet. The row and column corresponding to this cell are tag row and tag column, respectively. In [Figure 2-14](#), `my_plan` is the name of this verification plan. Use the same plan name as specified in the HVP Plan Sheet and HVP Attribute Definition Sheet to refer to the same verification plan.

If you have multiple plans and they share the same metric definitions, you can create a spreadsheet that only contains `hvp metric` without a plan sheet. You can also add `hvp attribute`. To define common HVP metrics, the `hvp metric` tag should not have plan name. The spreadsheet can be included in XML for other plan spreadsheets, and the metrics will be integrated automatically.

	A	B	C	D	E
1	hvp metric	name	type	aggregator	goal
2		bugs	integer	sum	bugs <1
3					
4					
5					
6					

metrics attributes

name

Use the `name` meta-tag column to define user-defined metrics. You must define all metrics used in HVP Plan Definition Sheet in this column, with the exceptions of Synopsys built-in metrics.

type

Use the `type` meta-tag column to specify value types. The available types for user-defined metrics are integer, real, ratio, percent, enum, and aggregate.

aggregator

Use the `aggregator` meta-tag column to specify aggregation operators. The available aggregation operators are `sum`, `max`, `min`, and `average` (for more information, see [“Metric Declaration” on page 25](#)).

goal

Use the `goal` meta-tag column to define the default goal expression of a given metric. Refer to [“Goal Specification” on page 38](#) for more details on how to describe a goal expression.

skip

Use the `skip` meta-tag to ignore a column or row.

HVP Attribute Definition Sheet

Figure 2-15 An HVP Attribute Definition Sheet

	A	B	C	D	E	
1	hvp attribute myplan	type	propagate	name	default	
2		string	no	desc	def desc	
3		integer	no	phase	1	
4		string	yes	stratt		
5		integer	yes	intatt	0	
6						

Plan Metrics **Attrs**

This section contains the following topics:

- [“hvp attribute plan-name”](#)

- “name”
- “type”
- “propagate”
- “default”

hvp attribute *plan-name*

The cell in a spreadsheet containing `hvp attribute plan-name` meta-tag identifies this spreadsheet as an HVP Attribute Definition Sheet. The row and column corresponding to this cell are called tag row and tag column, respectively. In [Figure 2-15](#), `my_plan` is the name of the verification plan. Use the same plan name as in the HVP Plan Sheet and HVP Metric Definition Sheet to refer to the same verification plan.

Just as we can create a common HVP metric definition sheet, we can also create a common HVP attribute definition sheet. Normally, common HVP metric and attribute sheets can reside in the same XML, which can be included by other XML plan spreadsheets.

name

Use the `name` meta-tag column to specify user-defined attributes. You must define all attributes used in HVP Plan Definition Sheet in this column, with the exceptions of Synopsys built-in attributes and annotations.

type

Use the `type` meta-tag column to specify value types. The available types for user-defined attribute annotation are integer, real, string, and enum.

propagate

Use the `propagate` meta-tag column to specify the permission for attribute values to propagate down to child features. A value of `yes` indicates that a child feature without an explicit attribute value automatically inherits the parent's attribute value. However, if the child feature has an explicit attribute value, the parent value cannot override it. A value of `no` indicates that a child feature takes the default value regardless of the parent's attribute value.

default

Verification Planner Spreadsheet Annotator automatically annotates the value specified in the `default` meta-tag column to a feature if neither its parent nor itself has an attribute value in the verification plan. If a feature has no explicit value and the respective attribute value has no permission to propagate, the feature takes the default value regardless of the parent's attribute value. Note that a parent attribute value has precedence over the `default` meta-tag value if a child attribute value is not specified.

For an enum type attribute, a default value is mandatory.

Applying Modifiers

Verification Planner spreadsheet annotator supports both `override` and `filter` modifiers written in the HVP language format, not in the spreadsheet (for more details, see [“Plan Modifiers” on page 55](#)).

You can pass modifier files with the `-plan` and `-mod` options. Based on your modifiers, part of scores, which are in disabled features, will be excluded in score aggregation.

Note:

If you use a modifier file to override a goal, attribute, or annotation value, the original value will still be shown in the annotated spreadsheet, not the modified value.

Sharing a Metric/Attribute Definition Sheet

The above metric/attribute definition sheets are normally added in the plan workbook as additional worksheets. However, you might want to create an independent metric/attribute definition sheet that can be reused in many different plan sheets.

To define a plan-independent metric/attribute definition sheet, omit the name of the plan under the `hvp attribute` and `hvp metric` tags as shown in the example plan sheet in [Figure 2-16](#).

Figure 2-16 Defining a plan-independent metric/attribute Definition Sheet

	A	B	C	D	E	F
1	hvp attribute	name	type	propagate	default	
2		phase	integer	YES	1	
3		fruit	enum{orange, apple, banana}	YES	orange	
4						
5						
6						
7						

	A	B	C	D	E	F	G
1	hvp metric	name	type	aggregator	goal		
2		bugs	integer	sum	bugs < 2		
3							
4							
5							
6							
7							

You can include a defined, metric/attribute sheet, as shown in [Figure 2-16](#), in a plan sheet or any other subplan sheet. All metric/attribute definitions will be automatically added.

Using Spreadsheet Annotator Commands

The syntax for Verification Planner Spreadsheet Annotator commands is as follows:

```
hvp annotate -plan xml-in-file [-mod mod-file]
[-plan_out xml-out-file]
[-feature "hierarchies" | -featurefile txt-file]
```



```
[-dir covdb-path | -f txt-file]
[-userdata vedata-file | -userdatafile txt-file]
[-group ratio] [-group merge_across_scopes] [-v]
[-show ratio] [-show_incomplete] [-q]
[-userdata_out out-vedata] [-metric_prefix metric-prefix]
```

Enter either of the following commands to list all of the command-line options:

```
%hvp annotate -h
```

Spreadsheet Annotator Commands

-h

Lists descriptions of Spreadsheet Annotator command options.

-plan=*xml-in-file*

Specifies a spreadsheet XML file containing the verification plan. This option is mandatory.

-plan_out=*xml-out-file*

Specifies a spreadsheet XML file containing annotated scores. This option is optional.

-mod=*mod-files*

Applies filter or override files in HVP language format. You can use multiple modifier files, and they are applied in order.

-feature "*hierarchies*"

Specifies the HVP scopes, which you want to annotate with the covdb or HVP userdata database. You can use multiple scopes and wildcards (*, **, ?), but you should enclose the string with the double quotes. Sub-hierarchies of matched scope are automatically annotated. If the `-feature` option is not given, covdb and HVP userdata are annotated to the entire plan. For example:

```
-feature "myplan.rec_feat.* myplan.play_feat*"
```

```
-featurefile=test-file
```

Specifies a text file that contains a list of hierarchy filters.

```
-dir=covdb-path
```

Specifies one or more coverage database directories. To specify multiple coverage directories, use at least one white space as a delimiter between them and enclose them with the double quotes (") characters. For example,

```
-dir "simv.vdb wb_dma.vdb"
```

```
-f=list-of-covdb-paths
```

Specifies one text file that contains a list of coverage .db directories. This command is optional.

```
-userdata=userdata-file
```

Specifies one or more HVP userdata files. To specify multiple HVP userdata files, use at least one white space as a delimiter between them and enclose them with the double quotes (") characters. For example:

```
-userdata "data1.txt data2.txt"
```

-userdatafile=*list-of-userdata-files*

Specifies one text file which contains a list of paths for multiple userdata files. This command is optional.

-elfile=*exclusion-file*

Specifies the exclusion file, used to exclude specific coverage objects from coverage scores. This exclusion is performed at the UCAPI layer.

-map=*mapped-module*

Specifies the mapped module name. See the *Unified Coverage Database API User Guide* and the *Unified Coverage Reporting User Guide* for details about module mapping.

-mapfile=*map-file-path*

Specifies the path to the map file. See the *Unified Coverage Database API User Guide* and the *Unified Coverage Reporting User Guide* for details about module mapping.

-group ratio

Allows you to aggregate Group score as ratio metric type (numerator/denominator) rather than the default, percent type.

-group merge_across_scopes

Aggregates Group metric score as ratio type (covered/total) instead of percent type.

-show_ratio

Displays ratio type score in the format of ratio instead of percentage.

-show_incomplete

Displays incomplete scores with the [inc] notation.

-log=logfile

Specifies the logfile in which all screen output messages are to be saved.

-report=urg-report-path

Generates hyperlinks to the given URG report path at feature and measure source cells. The *urg-report-path* must exist in the path of the annotated spreadsheet.

-runurg

Automatically run URG to generate a URG report at the path specified by the **-report** switch after the Verification Planner process is finished. If the **-report** switch is not given, a URG report is generated in the default `urgReport` subdirectory in the working directory.

When URG runs, some Verification Planner switches such as **-plan**, **-mod**, **-userdata**, **-map**, **mapfile**, **-elfile**, **-trend**, and **-report** are bypassed to URG.

If the `VCS_HOME` environment variable is not set, URG in the VCS deployment for Verification Planner is used. You might want to set a different `VCS_HOME` to choose a different URG path.

-trend sub-arguments

This switch only works when the **-runurg** switch is also given. When URG runs, it triggers the trend chart feature. All **-trend** arguments are bypassed into URG.

-v

Shows progress status messages.

-q

Enables the quiet mode to suppress all warning messages.

-userdata_out *out-userdata-file*

Dump annotated scores of all measures into the userdata file specified by *out-userdata-file*.

-metric_prefix *metric-prefix*

You prepend the given prefix to the metric name in the output HVP userdata file specified with `-userdata_out`.

-ss_show_sources *metric enum1+enum2+...*

Set enum entries that you want to appear in the measure source cell popup. If the `-ss_show_sources entry` is not set, all matching sources appear in the popup. For example:

```
-ss_show_sources test warn+fail
```

-map=*module-name*

Report on merging mapped modules coverage.

-mapfile=*map-file*

Report on merging mapped modules coverage given in *map-file*.

-elffile=*exclusion-file*

Exclude coverable objects specified in *exclusion-file* for code/assertion/group coverage.

Using the HVP_ARCH_OVERRIDE Variable

The `hvp` command detects the current platform and selects the correct binary for that platform. If you want to specify a different platform to select its binary executable, you can set the `HVP_ARCH_OVERRIDE` environment variable to the desired platform:

HVP_ARCH_OVERRIDE = *platform*

[Table 2-2](#) includes a list of the supported Verification Planner platforms.

Table 2-2 Supported Verification Planner Platforms

Platform name	Description
RHEL32	32-bit Red Hat Linux
RHEL64	64-bit Red Hat Linux
suse32	32-bit Suse Linux (Verification Planner delivers RHEL32 binaries)
suse64	64-bit Suse Linux (Verification Planner delivers RHEL64 binaries)
sparcOS5	32-bit Solaris Sparc
sparc64	64-bit Solaris Sparc (Verification Planner delivers sparcOS5 binaries)
rs6000	AIX architecture

Troubleshooting

Perform the following if you do not see coverage information annotated to the spreadsheet after using **-dir** *covdb-path*, where *covdb-path* is your coverage directory:

1. Use the Synopsys Unified Report Generator (URG) to verify that the coverage information exists. For example:

```
%urg -dir wb_dma.vdb
```

2. If the coverage information exists, then contact your Synopsys Applications Consultant for assistance.