8/22/2003

Proposal for VPI model PSL assertion extensions

This proposal has been prepared by Cadence Design Systems, Inc. for consideration by the IEEE 1364 working group for inclusion in the next revision of the IEEE 1364 standard.

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Contents

1 Introduction 4

1.1 Overview 4

1.2 Scope 4

2 VPI model UML notation..5

2.1 UML notation quick reference 5

2.2 VPI interface interpretation of the model 7

3 VPI assertion class diagram..8

4 vpi_user.h header file modifications 9

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1 Introduction

1.1 Overview

[1] It is expected that the Accellera Property Specification Language standard (PSL) [1] will be adopted and included by reference in the Verilog IEEE 1364 standard [2]. This will add assertion verification capabilities to the Verilog Hardware description language. The objective of this proposal is to provide VPI extensions to access the PSL assertions which will be adopted by the Verilog 1364 standard.

[2] The VPI model described in this proposal constitutes an initial subset of the PSL assertion access. We expect that this access will be updated and revised accordingly with the part of the PSL language which will be accepted by the Verilog 1364 committee.

[3] This proposal is divided in four sections. The first section describes the scope and purpose of this proposal. The second section is a quick reference guide to the formal graphical notation used by the VPI diagrams. The third section provides the VPI diagram to access PSL assertions, and the fourth section details the VPI standard header file additions necessary to provide access to PSL assertions.

1.2 Scope

[4] It is assumed that PSL assertions will be part of the Verilog HDL source and would provide the basis of Verilog assertion syntax and semantics. In a similar manner, the VPI model for accessing PSL assertions coexists with the VPI model to access a Verilog instantiated design. The proposed VPI extensions include traversal of all PSL assertions declared within a module instance and queries of certain characteristics of these assertions. This proposal is intended to provide the minimal required access for co-simulation and debugging tools. Access to the full syntax of the PSL assertion is not proposed here but can be donated later as a similar VPI extension.

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2 VPI model UML notation

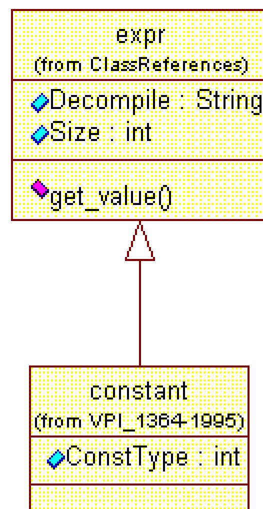
[5] The Unified Modeling Language (UML, [3]) is used to describe the VPI information model. UML is a formal graphical language. It defines a rigorous notation and a meta model of the notation (diagrams) that can be used to describe object-oriented software design. UML is an OMG standard (Object Management Group) which is being proposed to the International Organization for Standards (ISO). The following sub sections provide a quick reference guide to interpret the VPI diagrams. For a more complete specification of the UML notation, consult [3].

2.1 UML notation quick reference

[6] *Class diagrams*

[7] We use the class diagram technique of UML to express the VPI information model. A class diagram specifies the VPI class types and the way classes are connected together. In UML, class inheritance is denoted by a hollow arrow directed towards the parent class.

[8] *A class*



[9] A derived class

[10] An expanded class shows two compartments, the top one displays the **properties** with their names and return type, the bottom one displays the **operations** that are defined for this class. Properties and operations inherited from parent classes may not appear in the compartment boxes of the derived classes but are available for all derived classes. In the example above, two properties are defined for the “expr” class: the “Decompile” and “Size” properties, while a single operation “get_value()” is defined. A additional property “ConstType” is defined for the class “constant” which is not available for the parent class “expr”.

[11] The link between the “constant” class and the “expr” class shows the **inheritance** between a derived class and its parent class. A derived class inherits properties and operations from its parent classes. The hollow arrow points to the parent class.

[12] *Associations*

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

[13] Relationships between classes are called associations and are denoted by straight lines between classes. Associations have descriptive parameters such as multiplicity, navigability and role names.

[14] **Associations** are links between classes that depict their inter-relationships.

[15] Navigability, multiplicity and role names can be used to further describe the relationship.

Navigability expresses the direction of access and is represented by an arrow. An association can be bi-directional in which case arrows may be shown at both ends.

[16] **Multiplicity** expresses the type of relationship between the classes: singular (one, zero or one), multiple (zero or more, one or more) and is represented by numbers at the end of the association to which it applies. It can be one the following:

[17] 1 for access to one object handle (singular relationship)

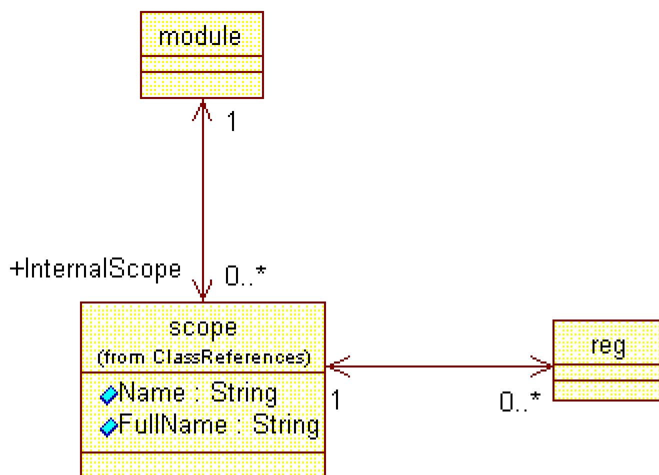
[18] 0..1 for access to zero or one object handle (singular relationship)

[19] 0..* for access to zero or more object handles of the same class (iteration relationship)

[20] 1..* for access to one or more object handles of the same class (iteration relationship)

[21] A **role name** is a tag name on one end of the association. It may be used to indicate more precisely the relationship or to distinguish this relationship from another relationship that leads to an object of the same class. In the figure below, “InternalScope” is the name of the relation that accesses an object of class “scope” from an object of class “module”. The relationship it denotes is an iteration relationship.

[22] In the diagrams, the following convention is used: if a role name is not specified, the method name for accessing the object pointed by the arrow is the target class name. From the scope class, zero or more objects of the reg class can be obtained, the default method name is “reg”.



Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.2 VPI interface interpretation of the model

When interpreting the VPI class diagrams, “vpi” must be added as a prefix to any class, property, method or operation name in order to obtain the standard defined constant listed in the VPI standard header file (vpi_user.h).

A VPI iteration (also called one-to-many method) is modeled by an association with a multiplicity of either zero or more (0..*), or one or more (1..*) to indicate that the iteration may contain zero handles or will contain at least one handle. In order to traverse iteration relationships, use vpi_iterate() and vpi_scan(). The direction or navigability indicates the class of the handles created by the iteration. In the example above, we show that there is a one-to-many relationship between a “module” class and a “scope” class.

A VPI singular (also called one-to-one method) will be represented by a navigable association with a multiplicity of one (1) if the method always returns a handle of the destination class or a multiplicity of zero or one (0..1) if the method may not return a handle. In order to traverse a singular relationship, use vpi_handle(). In the example above, the diagram shows a one-to-one relationship that allows traversal from a “scope” class back to the “module” class.

Note that the diagrams only express the possible access flow and not all access is presented in a single diagram.

A VPI property which appears in the top compartment of a class, can be queried with one of the following VPI interface functions:

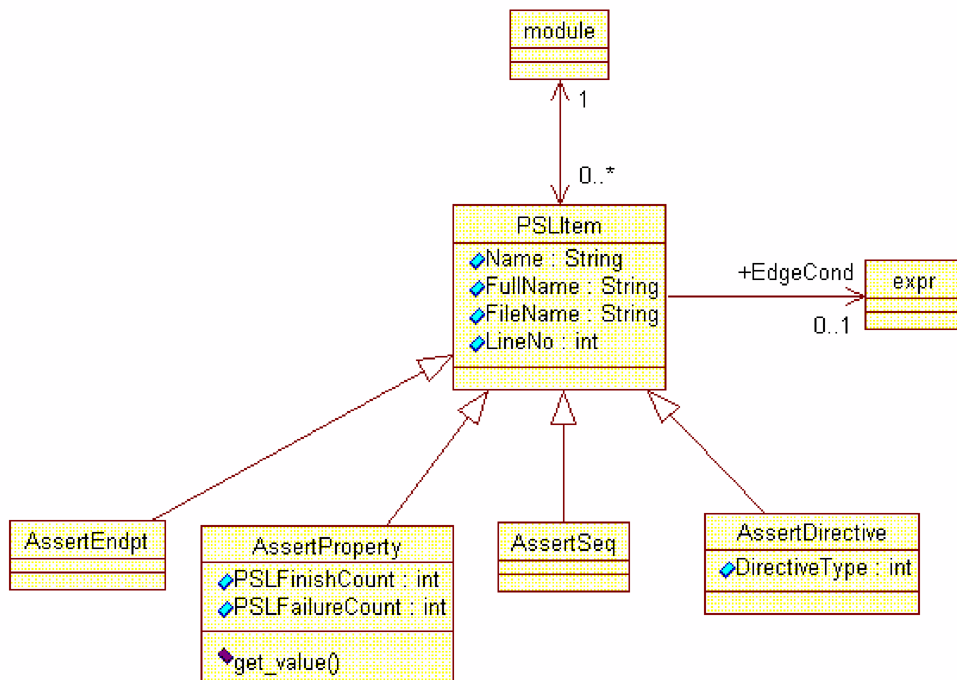
vpi_get() for a boolean or integer property,

vpi_get_str() for a string property.

Additional VPI functions can be available for a certain class and are listed in the bottom compartment of the class. Such functions are for example be vpi_get_value() or vpi_put_value().

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3 VPI assertion class diagram



Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

4 vpi_user.h header file modifications

[23] Below are the additional objects types, relationships, properties and properties constant values which should inserted in the vpi_user.h file in order to provide assertion access. The constant values attributed to the defined constants were chosen to be outside the range used currently by VPI. A range of values should be reserved for the VPI assertion access and that range should be large enough to allow future extensions.

```
/* Object types */
#define vpiAssertEndpt     627   /* PSL Named Endpoint Declaration (LRM
Sect 6.1.3) */
#define vpiAssertProperty  630   /* PSL Named Property Declaration (LRM
Sect 6.2.4) */
#define vpiAssertSeq       631   /* PSL Named Sequence Declaration (LRM
Sect 6.1.2) */
#define vpiAssertDirective 632   /* PSL Verification Directives (LRM
Sect 7.1) */

/* Relationships */

/* One to many relationship from a vpiModule reference handle to
   vpiPSLItems

   vpiModule-->>vpiPSLItems

   Iteration returns handles of type:
   vpiAssertEndpt, vpiAssertProperty,  vpiAssertSeq, vpiAssertDirective
*/
#define vpiALObjects       635


/* One to one relationship from a PSL declaration or directive
   to vpiEdgeCond

    (vpiAssertEndpt, vpiAssertProperty,
     vpiAssertSeq, vpiAssertDirective)  ->vpiEdgeCond

   Returned handle is the HDL clock expression.
*/
#define vpiEdgeCond        638

/* Properties */

/* Integer properties for object of type vpiAssertProperty
   Retrieve with vpi_get()
*/
#define vpiPSLFinishCount   626 /* Number of times a property has
reached the state of vpiAssertFinished */
#define vpiPSLFailureCount  627 /* Number of times a property has
reached the state of vpiAssertFailed */


/* vpiDirectiveType -- Integer and String property for handles of type
vpiAssertDirective.
   Retrieve with vpi_get()/vpi_get_str()
   Integer codes listed below.
```

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
*/
#define vpiDirectiveType   631

/* Directive type codes */
#define vpiDirAssert       1  /* PSL Assert Directive (LRM Sect 7.1.1)
*/
#define vpiDirAssume       2  /* PSL Assume Directive (LRM Sect 7.1.2)
*/
#define vpiDirCover        6  /* PSL Cover  Directive (LRM Sect 7.1.6)
*/


/* Assertion state value encodings
To query, use vpi_get_value() with reference handle of type
vpiAssertProperty and integer or string format (vpiIntVal or
vpiStringVal) */

#define vpiAssertInactive  1  /* There are currently no partial matches
of the sequence of conditions described by the property */

#define vpiAssertActive    2  /* The first term of the enabling
condition is satisfied, and the property has not finished or failed */

#define vpiAssertFinished  3  /* The fulfilling condition has evaluated
to true, or the property has terminated without failing. */

#define vpiAssertFailed    4  /* The fulfilling condition has evaluated
to false. */

#define vpiAssertDisabled  5  /* The property is disabled, and is not
being checked. */
```


Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Annex A: References

[1] Accellera Property Specification Language Reference Manual version 1.01, approved Accellera standard.

[2] IEEE Std 1364-2001, IEEE Standard Verilog Hardware Description Language.

[3] OMG UML Unified Modeling Language v. 1.3, Object Management Group, June 1999.

Copyright (c) 2003 Cadence Design Systems, Inc. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.