8/22/2003

Proposal for VPI model data type 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.

Contents

1 Introduction

1.1 Overview

[1] This proposal complements the Cadence data types donation [1] to the Verilog 1364-2001 standard [2] by providing the specification of the VPI model enhancements for accessing the new data types and declared objects of these data types. The proposal uses a formal graphical language called UML to describe the VPI information model. UML stands for "Unified Modeling Language" [3] and is typically used for describing object oriented software. We believe that using a formal language to represent the VPI information provides many advantages; this is currently considered by the 1364 PLI task force and is not the direct object of this proposal.

1.2 Scope

[2] In the data type proposal [1], the existing concept of variable and net objects was enhanced to create a type system that is orthogonal to the simulation semantic of the object itself. The type system was extended beyond that of the current language definition by adding user-defined types for enumerations, multi-dimensional arrays, structures, and dynamically allocated objects. These data type extensions allowed variables, nets, parameters, ports, and arguments to tasks and functions to be of any type.

[3] The data type proposal made the distinction between the kind of an object (net, variable, event) and the type of an object. The *kind* of an object establishes its simulation semantics. The *data type* of an object specifies the set of values that the object can hold.

[4] Similarly, the VPI information model is extended to introduce the concept of a data type. Note that the vpiType of a vpi handle is not the same as the data type of an object. In the remaining of this document, we will talk about the vpiDataType of an object to denote the set of values that an object is allowed to have, and we will talk about the vpiType of an object handle when referring to the vpi property which returns the handle type constant as defined in the vpi_user.h standard header file.

[5] When extending the VPI information model, we took into consideration several requirements to maintain backward compatibility.

[6] Requirement #1: Old VPI applications should continue to work on designs which do not involve the use of the new data types. That is, an old application recompiled with the new version of the VPI standard header file (vpi_user.h) should work exactly the same as before.

[7] Requirement #2: A reasonably written old VPI application should work (and not crash) on designs involving new data types. That old application may need to be modified in order to access the new data types objects but the application code contains enough error checking that it can prevent itself from accessing incorrectly new types of objects. Example of such coding practices is to check for the vpiType of an object before proceeding, or having a switch on the vpiType of an object and a default case for unexpected vpiTypes.

2 VPI model UML notation

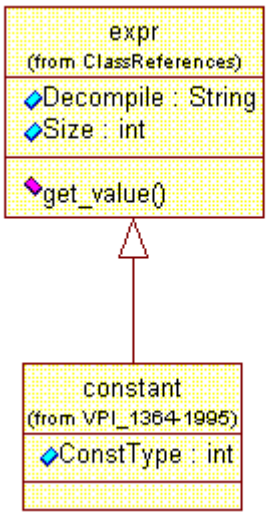
[8] The Unified Modeling Language (UML, [3]) is used to describe the VPI information model. UML is a formal graphical language. It defines a rigorous notation and a meta model of the notation (diagrams) that can be used to describe object-oriented software design. UML is an OMG standard (Object Management 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

[9] *Class diagrams*

[10] 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.

[11] *A class*



[12] A derived class

[13]

[14] 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”.

[15] 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.

[16] *Associations*

[17] 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.

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

[19] 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.

[20] **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:

[21] 1 for access to one object handle (singular relationship)

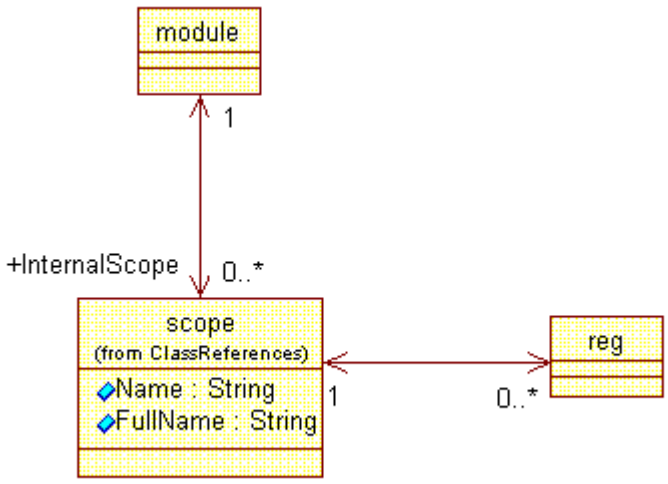
[22] 0..1 for access to zero or one object handle (singular relationship)

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

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

[25] 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.

[26] 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”.



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.

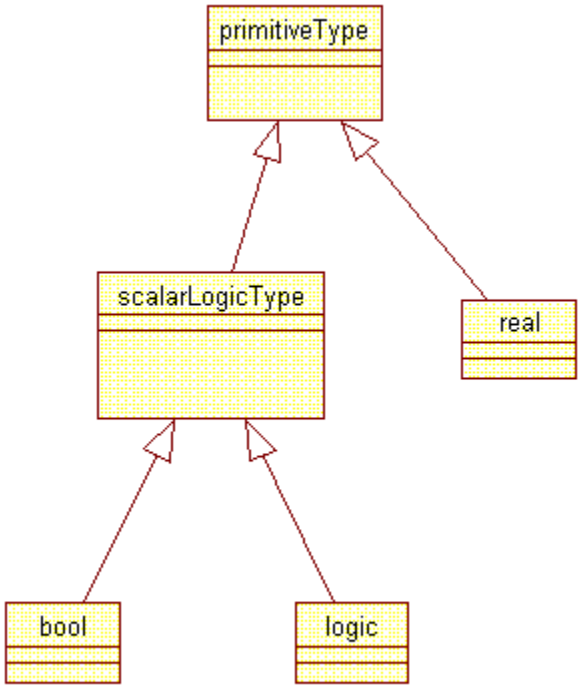
Additional VPI functions can be available for a certain class and are listed in the bottom compartment of the class. Such functions are for example be vpi_get_value() or vpi_put_value().

3 VPI class diagrams

[27] Diagrams 3.1 to 3.7 are new diagrams which depict the Verilog data type system. Diagrams 3.8 to 3.20 are either existing VPI diagrams of the Verilog 1364-2001 which have been modified in order to support the new data types or new diagrams.

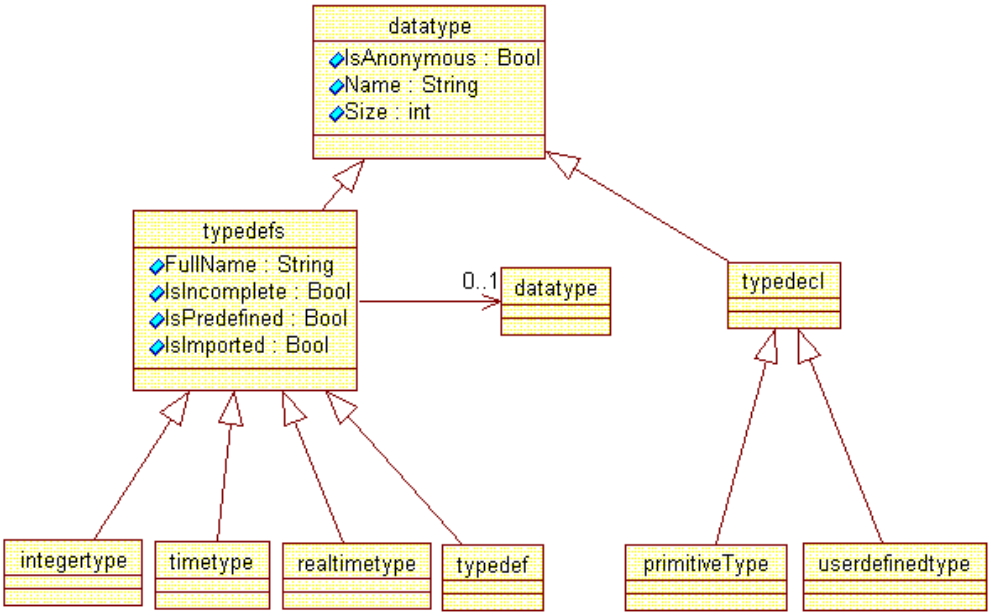
3.1 Primitive data types

[28] Refer to section 2.1 of the data type donation [1].



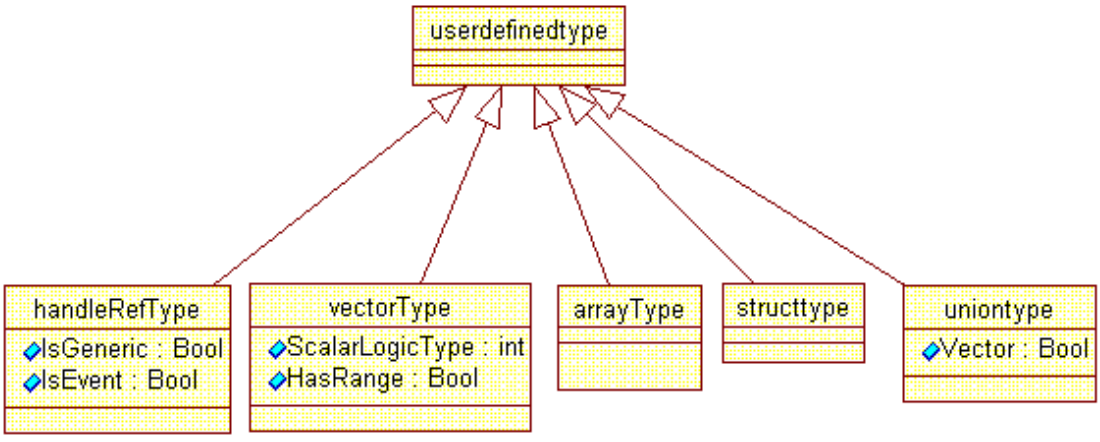
3.2 Type definitions and declarations

[29] Refer to sections 2.2, 2.9 and 2.10 of the data type donation [1].



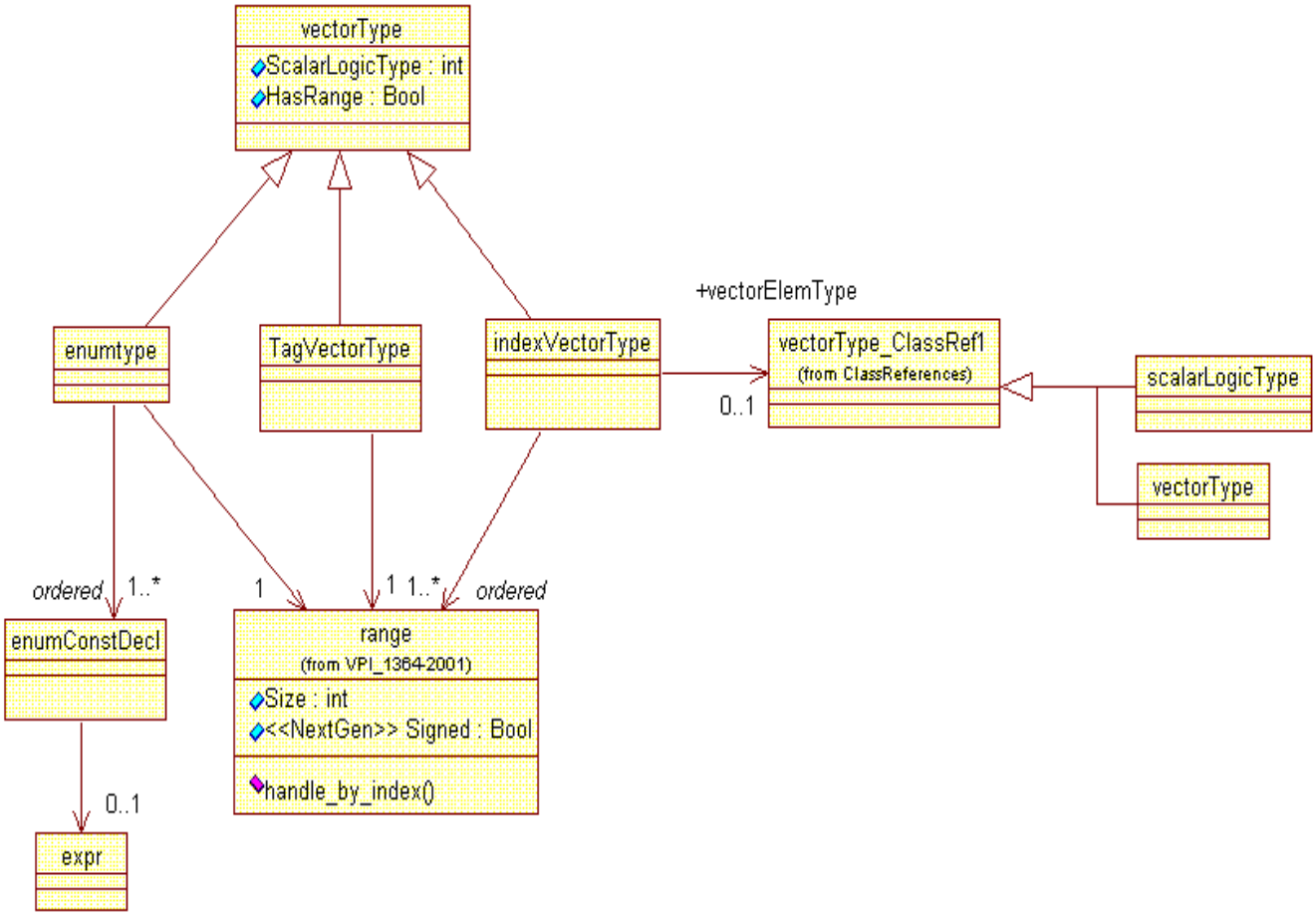
3.3 User-defined data types

[30] Refer to section 2.3 of the data type donation [1].



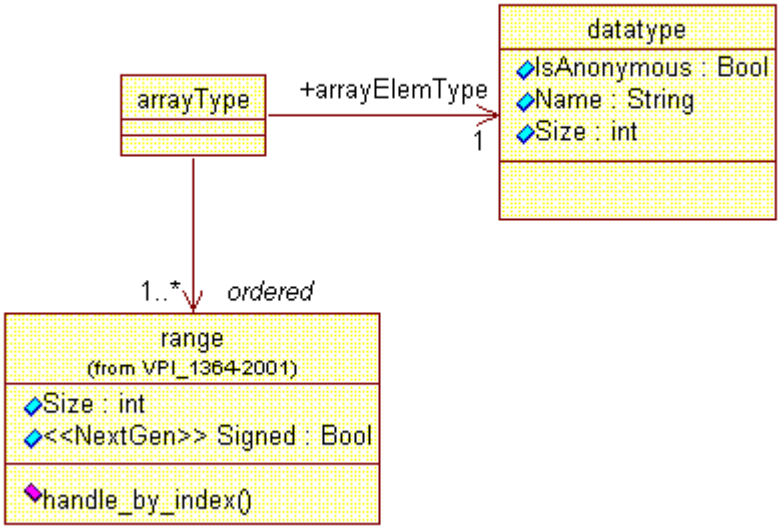
3.4 Vector data types

[31] Refer to section 2.4 of the data type donation [1].



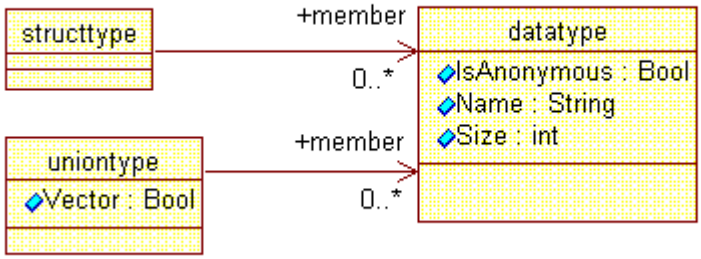
3.5 Array data types

[32] Refer to section 2.5 of the data type donation [1].



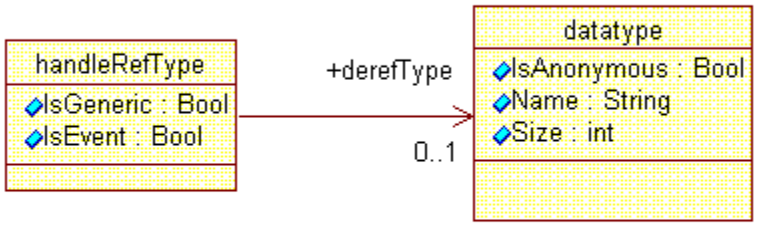
3.6 Struct and Union data types

[33] Refer to sections 2.6 and 2.7 of the data type donation [1].



3.7 Reference data types

[34] Refer to section 2.8 of the data type donation [1].



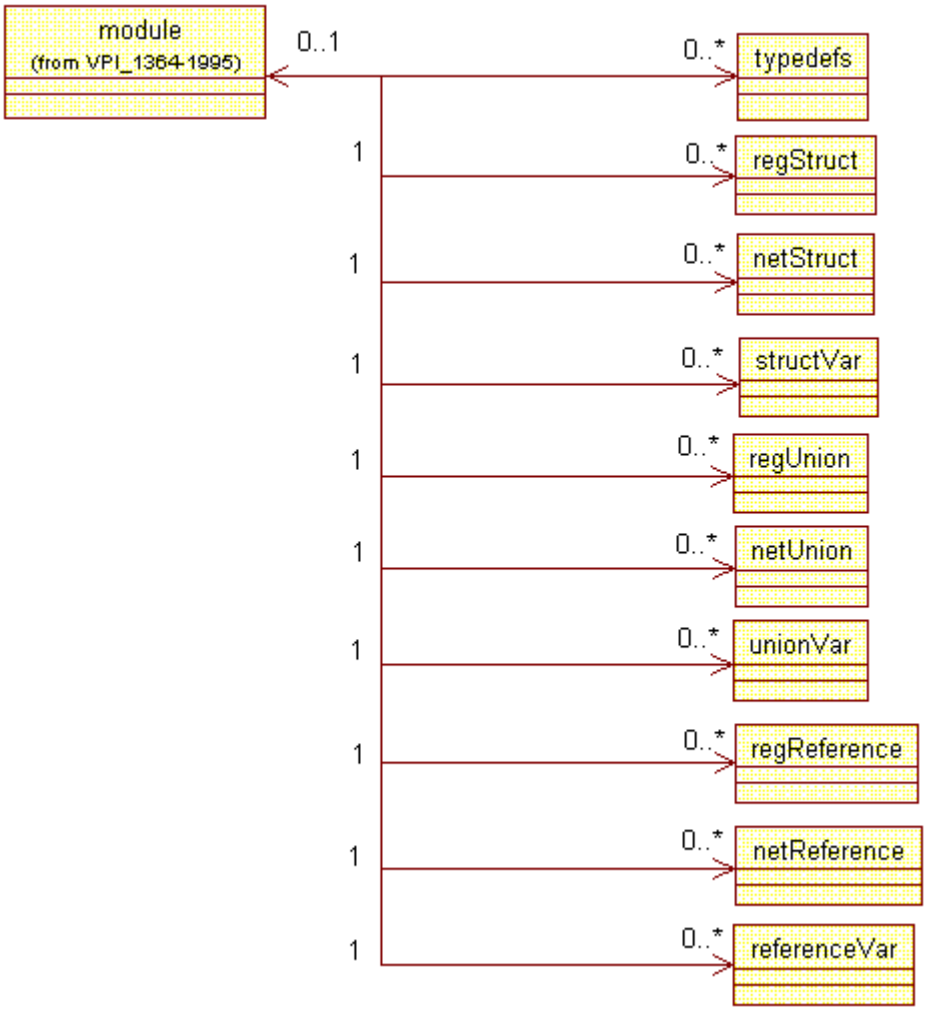
3.8 Module

[35] Additional iterations and singular relationships are added in order to access new vpiTypes of objects: vpiTypedef, vpiRegStruct, vpiRegUnion, vpiRegRef, etc from the module class. The diagram below depicts the additional traversals to the currently existing VPI module diagram in section 26.6.1 of the Verilog 1364-2001 LRM.

[36] Notes:

[37] 1. Iterating on vpiTypedefs from a vpiModule handle type returns all the visible typedefs, i.e. declared outside the module or inside the module. If a typedef is declared outside the module, the boolean property vpiIsImported shall return TRUE. If a typedef is declared outside a module, the vpiModule method shall return NULL.

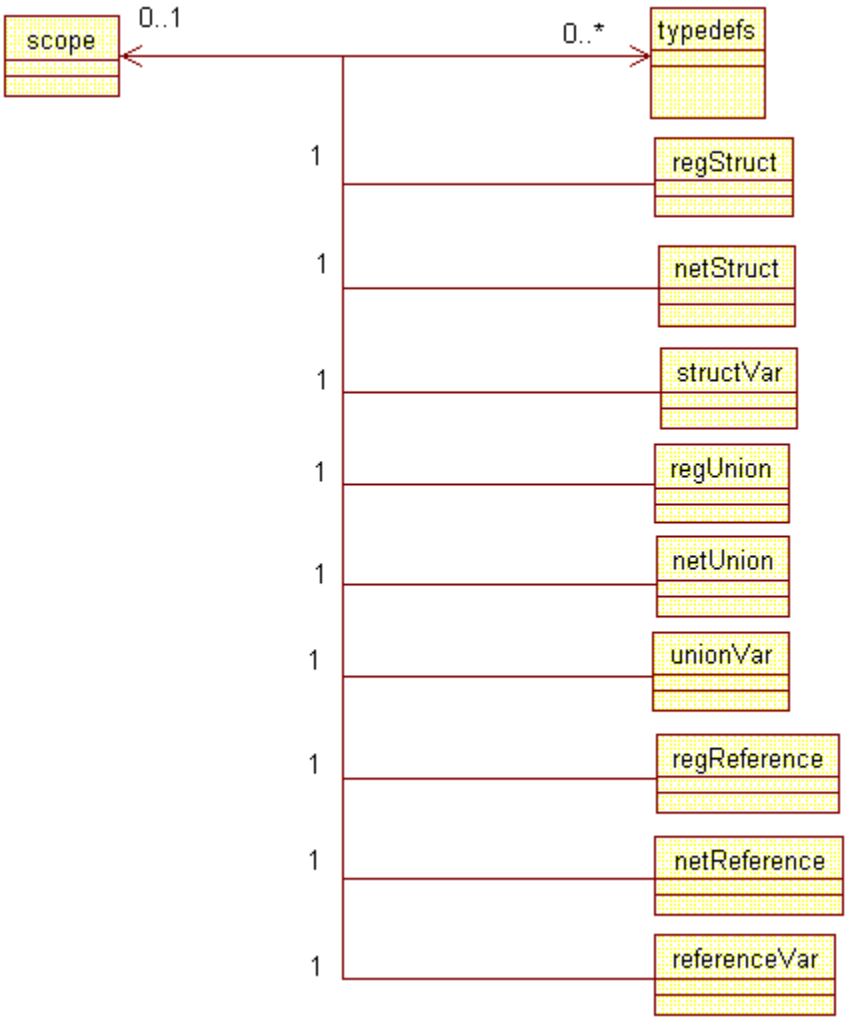
[38] 2. Specific iterations vpiStructVar, vpiUnionVar, vpiReferenceVar are provided. Itetation on variables is already available in 1364-2001 and is a more general iteration.



3.9 Scope

[39] Similar additional iterations and singular relationships are added in order to access new vpiTypes of objects from the scope class. The diagram below depicts the additional traversals to the currently existing VPI scope diagram in section 26.6.3 of the Verilog 1364-2001 LRM.

[40] Notes: See Notes 1 and 2 in section 3.8.



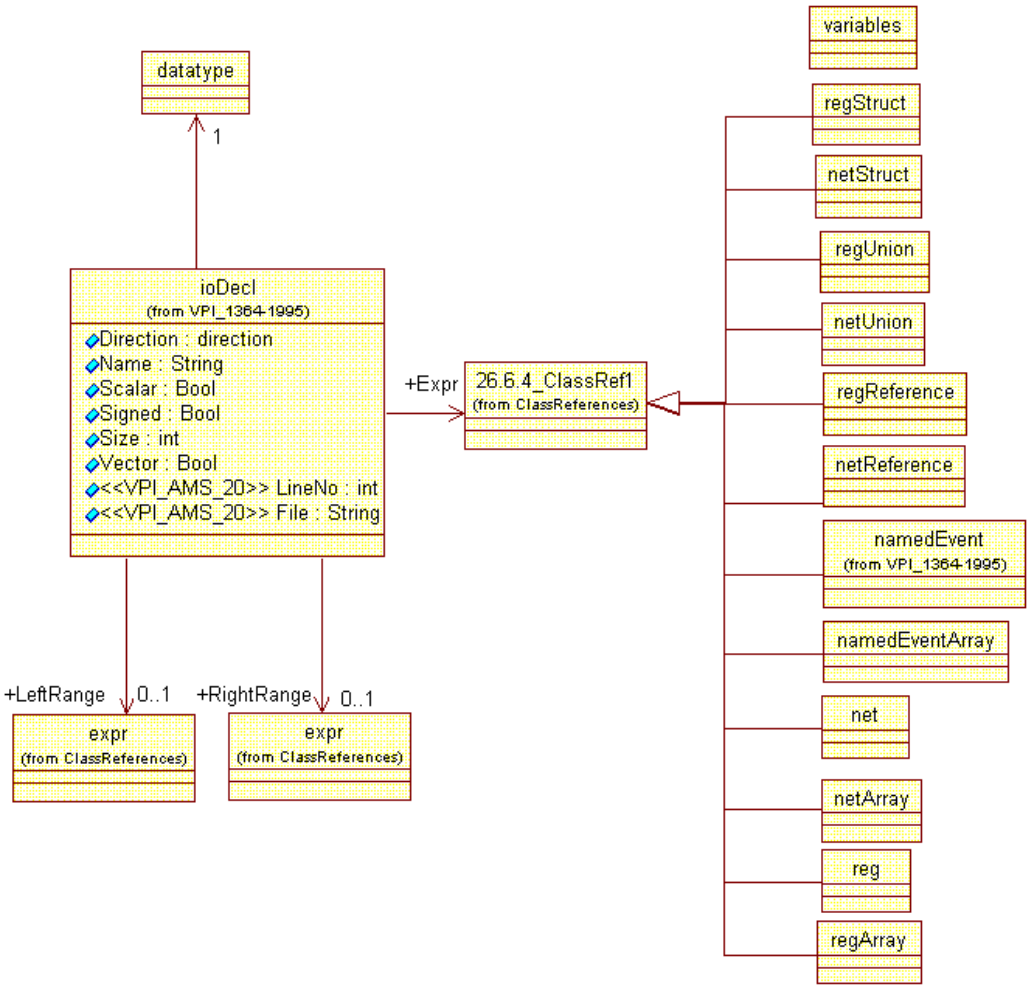
3.10 IO declaration

[41] An IO declaration can be of any data type. The diagram below shows the additional classes which can be obtained from a reference handle of vpiType vpiIODdecl when traversing the vpiExpr relationship. Refer to the VPI diagram in section 26.6.4 of the Verilog 1364-2001 LRM. Instead of adding boolean properties similar to vpiScalar and vpiVector to indicate whether the iodecl is an array, struct, union or reference type, a relationship to the data type (vpiDataType) of the iodecl is provided.

[42] Notes:

[43] 1. The vpiDirection property can return an additional value constant: vpiRef to represent a reference io declaration.

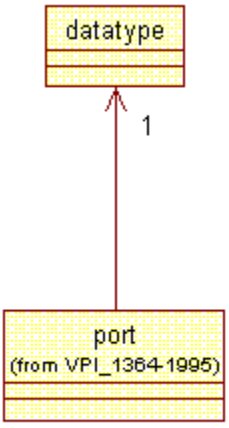
[44] 2. The existing vpiLeftRange and vpiRightRange methods shall return NULL handles if the iodecl is not of a vector data type.



[45]

3.11 Ports

[46] An additional one to one method is added to obtain the vpiDatatype of a vpiPort handle. This complements the existing VPI diagram in section 26.6.5 5 of the Verilog 1364-2001 LRM.

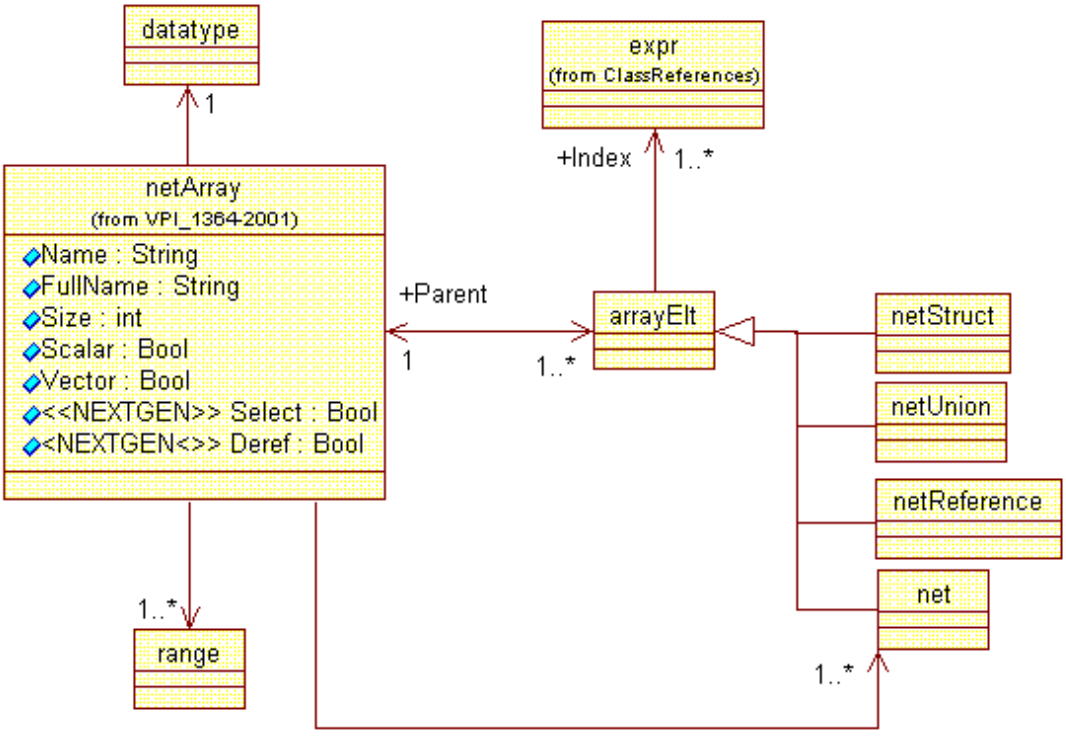


3.12 Nets and net arrays

[47] This diagram replaces the existing VPI diagram for net arrays in section 26.6.6 of the Verilog 1364-2001 LRM.

[48] Notes:

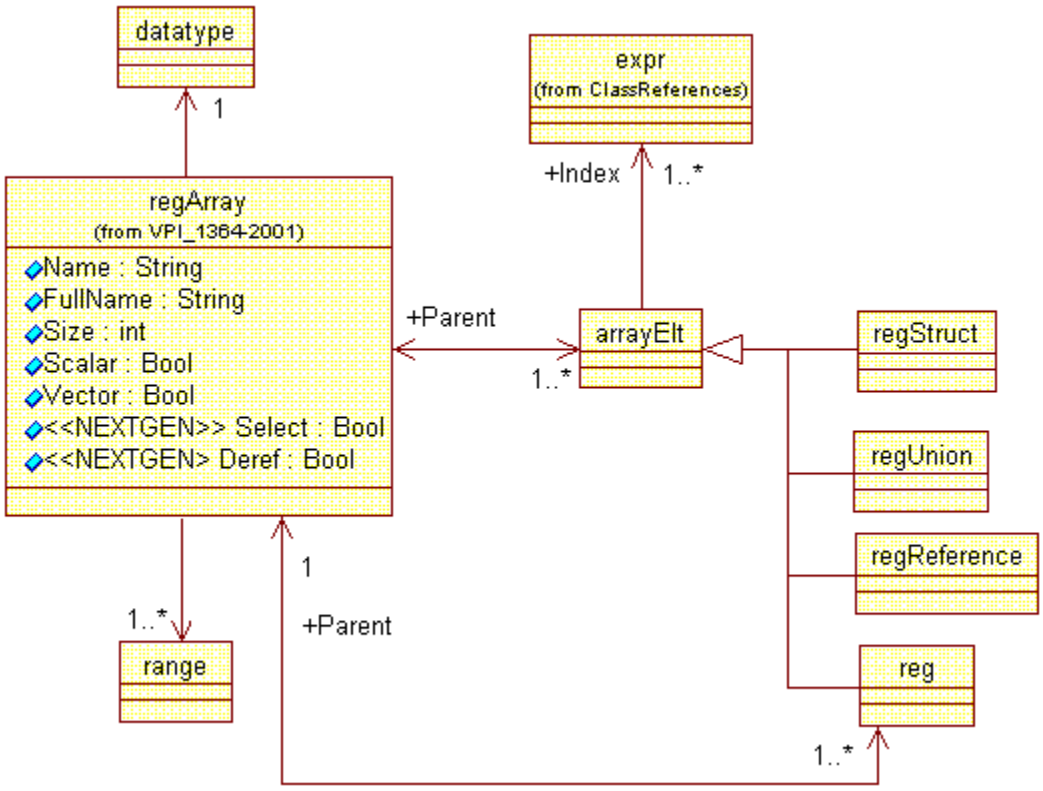
[49] 1. The vpiNetType property for a vpiNet or vpiNetBit can return the additional constant value of vpiWone.



[50]

3.13 Reg and reg arrays

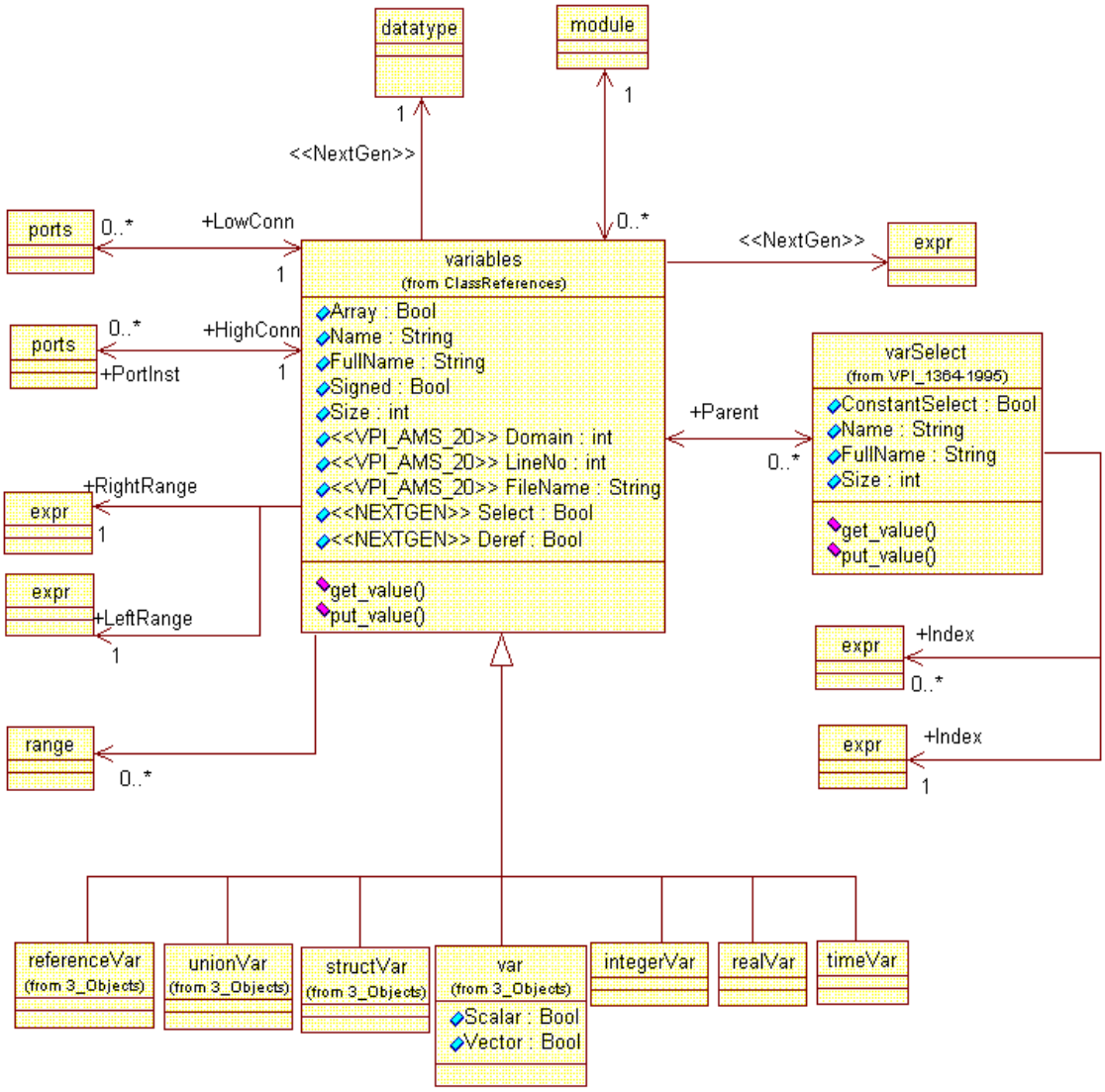
[51] This diagram replaces the existing VPI diagram for reg arrays in section 26.6.7 of the Verilog 1364-2001 LRM.



[52]

3.14 Variables

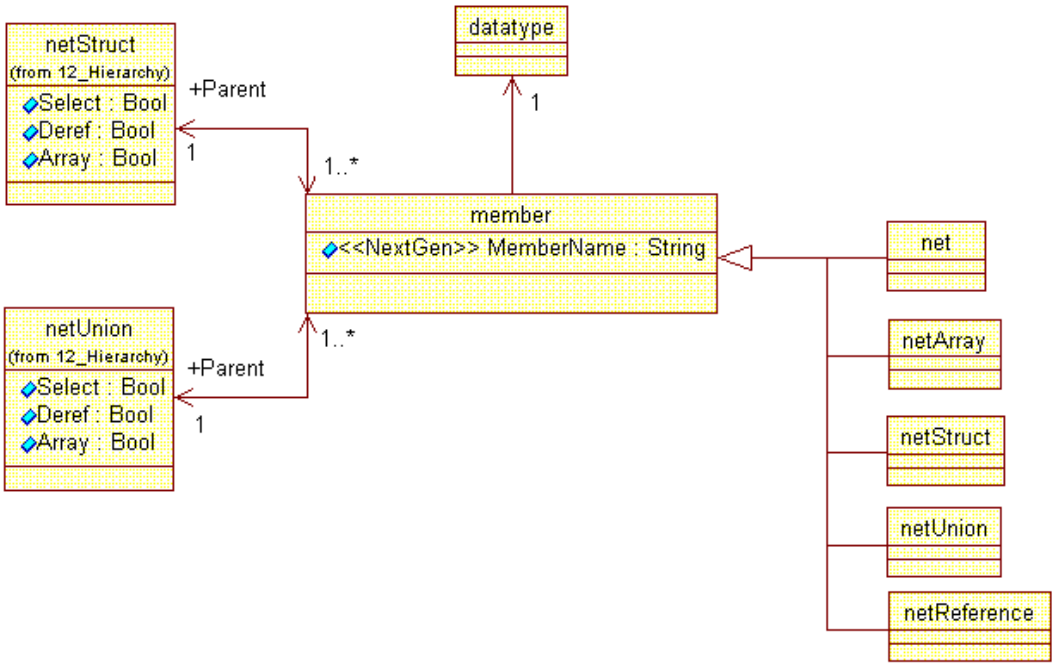
[53] This diagram replaces the existing VPI diagram in section 26.6.8 of the Verilog 1364-2001 LRM.



[54]

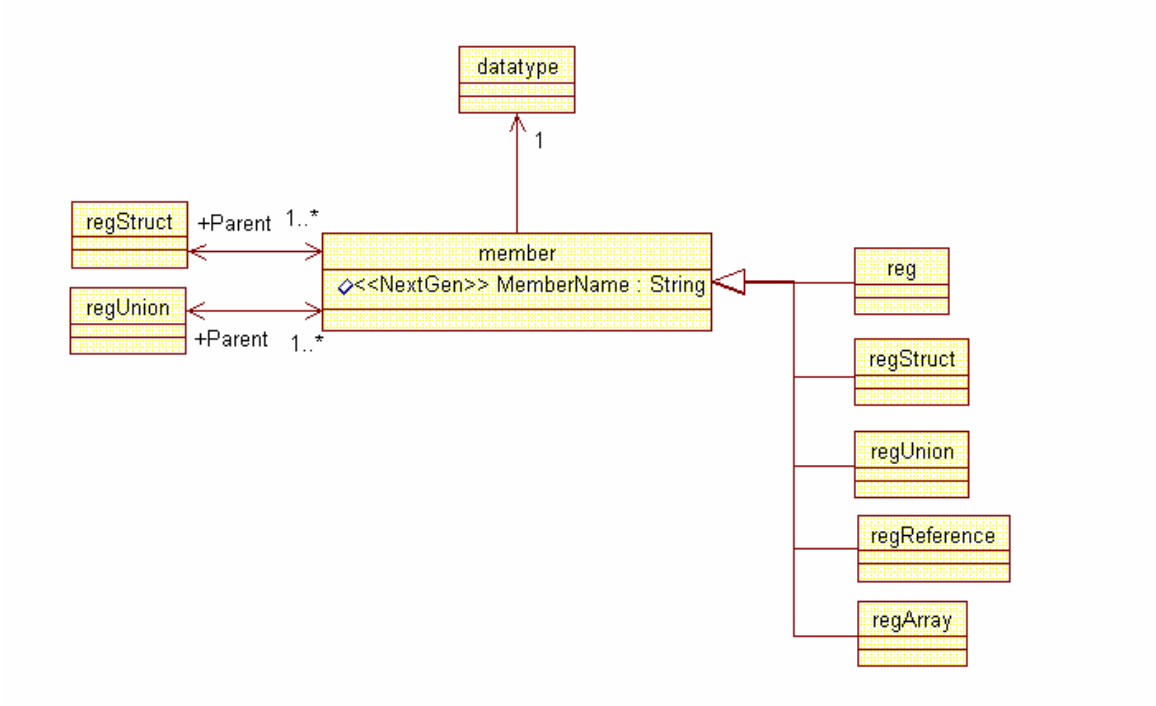
3.15 Net structs and unions

This is a new diagram.



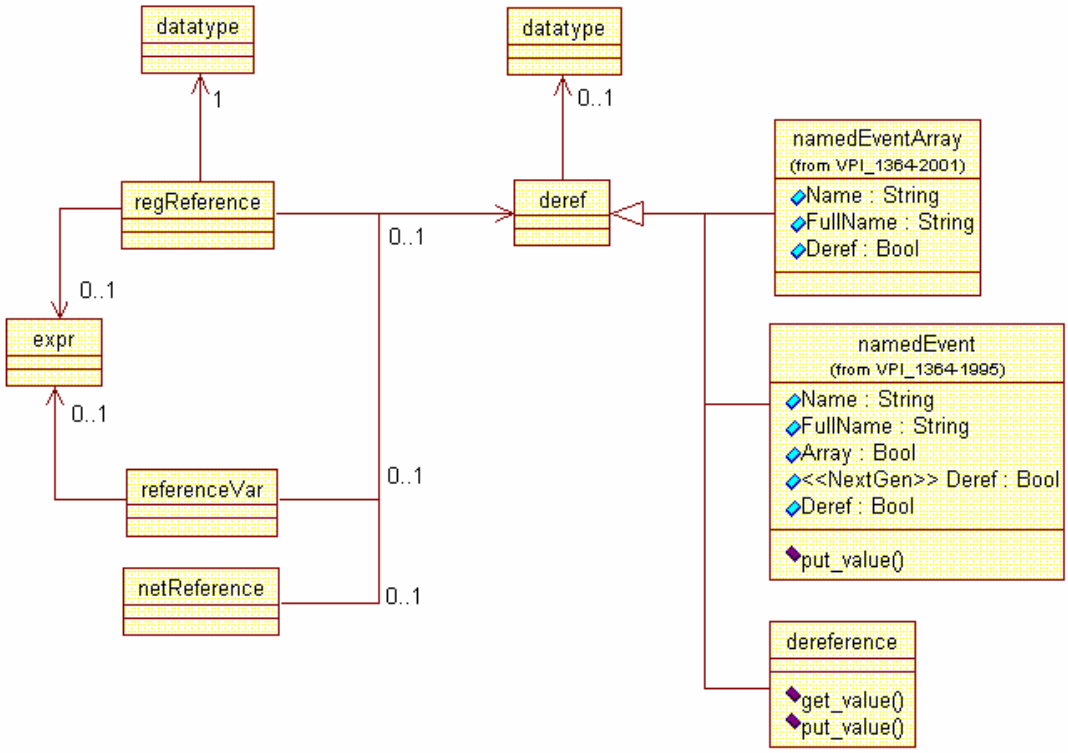
3.16 Reg structs and unions

This is a new diagram.



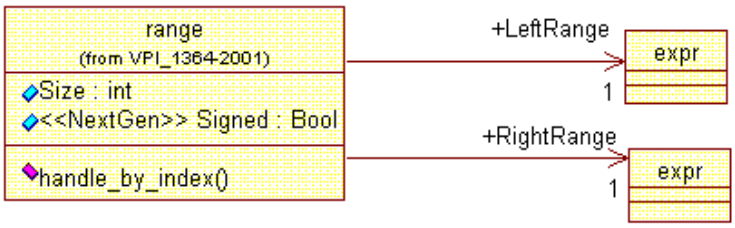
3.17 References

This is a new diagram.



3.18 Object range

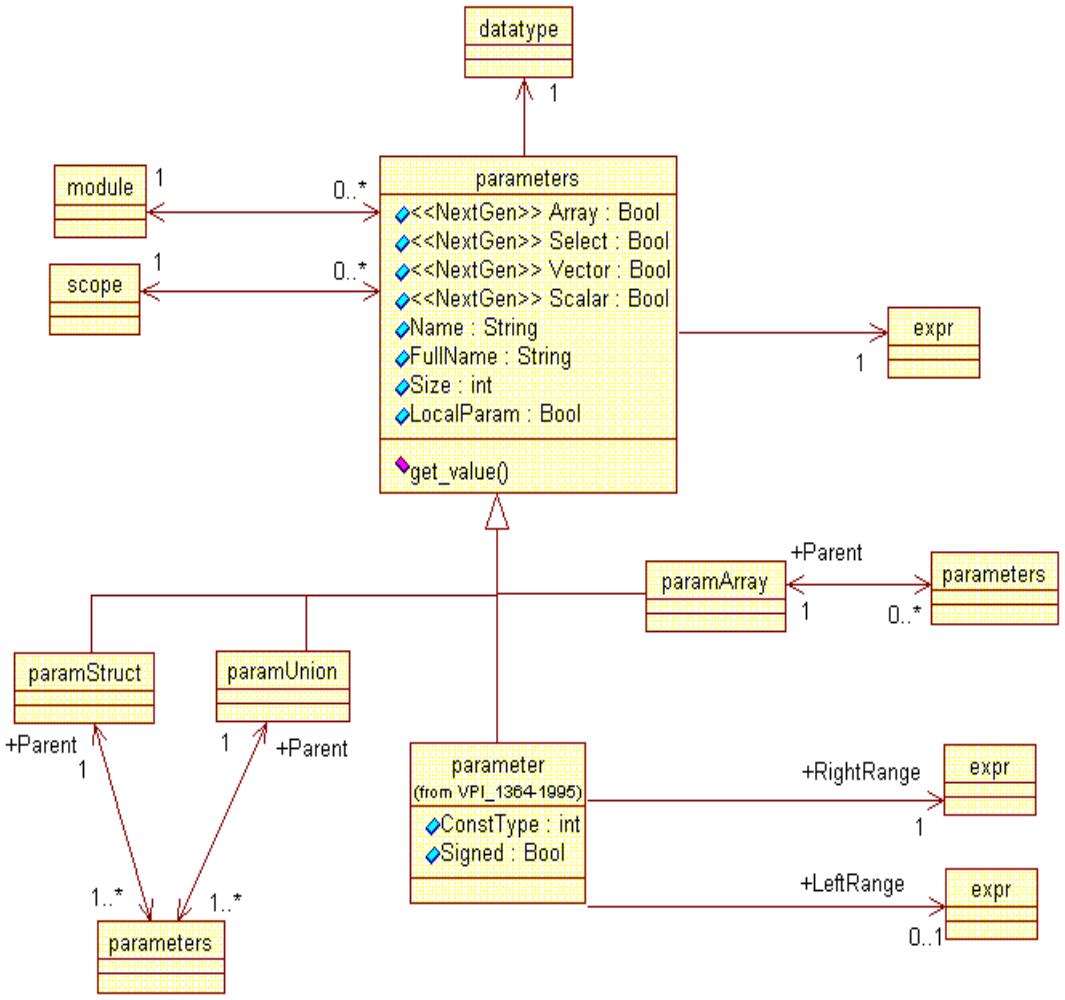
[55] This diagram replaces the existing VPI diagram in section 26.6.10 of the Verilog 1364-2001 LRM.



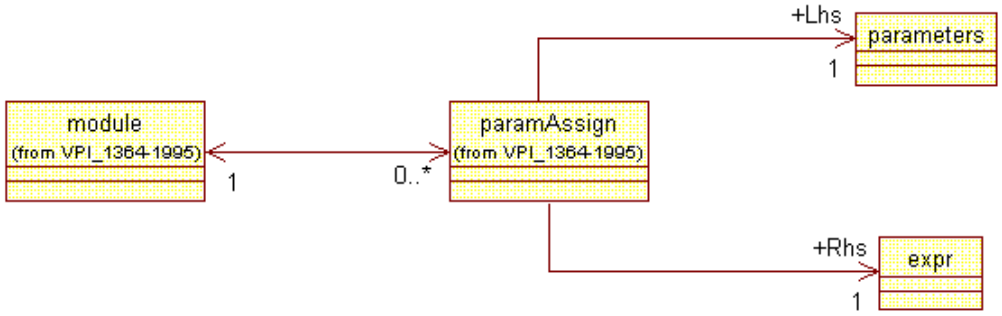
[56]

3.19 Parameter

[57] The following diagrams replace the existing parameter and param assign diagrams in section 26.6.12 of the Verilog 1364-2001 LRM.



[58]



[59]

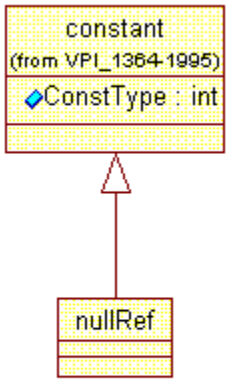
3.20 Expressions

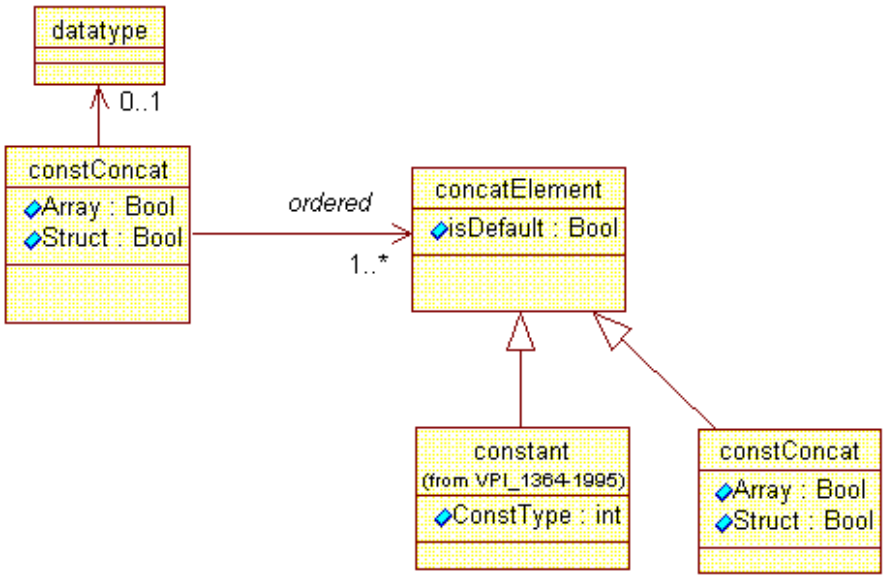
[60] The following diagrams supplements the existing VPI diagram in section 26.6.26 of the Verilog 1364-2001 LRM. The vpiNullRef type represents the null value of a reference type. The vpiConstConcat type represents constant array and struct concatenation expressions. See section 4.3 in the Cadence data type proposal [1]. The vpiConstConcat class inherits from the expr class.

[61] Notes:

[62] 1. The vpiConstType property shall return an additional defined constant vpiNullConst which represents the null value of a reference type object.

[63] 2. The vpiOptType property of a vpiOperation type handle shall return additional defined constants vpiArrayConcat and vpiStructConcat. Refer to section 4.3 of the Cadence data type donation proposal [1].





Annex A: References

[1] Cadence proposal for extending Verilog data types.

[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.