Using SystemVerilog Packages in Real Verification Projects

Kaiming Ho
Fraunhofer IIS
Erlangen, Germany
kaiming.ho@iis.fraunhofer.de

This paper details some of the key features and frustrations of using the package construct in SystemVerilog. The package construct is compared to similar features in other languages such as the identically named construct in VHDL and namespaces in C++. Valuable lessons learned over the course of multiple projects in the development of verification environments are described, and the paper makes recommendations for basic DOs and DONTs for SystemVerilog package use. The theme of code reusability is always important, and tips on how packages can be used to achieve this are discussed.

Users of languages such as VERA, which does not have the package construct, are more accustomed to using include files, which provide some but not all the features packages provide. The paper discusses why the continuance of this approach, while possible in SystemVerilog, is not recommended and why the package construct is superior.

Finally, the paper discusses how SystemVerilog allows packages and classes to be mixed in interesting ways not seen in other languages.

I. Introduction

The package construct is one of the many incremental improvements added to SystemVerilog that Verilog notably lacked. With its introduction, all users of SystemVerilog, from RTL designers to verification engineers, now have an encapsulation mechanism that VHDL users have had for many years. Since SystemVerilog also added the class data type many interesting usage models for packages are possible as a consequence, some of which are not applicable to VHDL since VHDL does not provide classes.

This paper is divided into three sections. The first summarizes the properties of the package construct, highlighting important features that underpin the rationale for using packages. Similar constructs in other languages, such as VHDL packages and C++ namespaces are discussed and compared. Important changes to the semantics of the construct made between IEEE 1800-2005 [1] and 1800-2009 [2] that are of interest to verification engineers are highlighted.

The second section describes practical issues that arise from the deployment of SystemVerilog verification environments. Problems that we have encountered in recent projects are described and the pros and cons of various solutions discussed. The last section explores advanced things one can achieve with packages. The discussion centres on how packages and classes can be used together to implement common design patterns, solving problems such as bridging hierarchical boundaries.

II. Properties of SystemVerilog packages

SystemVerilog (SV) packages are a top-level design element that provides an encapsulation mechanism for grouping together data types (including classes), tasks/functions, constants and variables. Additionally, assertion related constructs such as sequences and properties can be encapsulated, which is of particular interest to verification engineers. Once encapsulated into a named package, the contents are available for use in other design elements (such as modules, programs, interfaces or other packages) irrespective of module or class hierarchy.

Surprisingly, this construct is unavailable to Verilog (1364-1995 and 1364-2001) users, who often resort to using modules to emulate package behaviour. Modules also serve as an encapsulation mechanism, and when left uninstantiated, become a top-level module whose contents are accessible through hierarchical reference. It is common in FPGA libraries to have global definitions and/or variables, and it is informative to note that the VHDL version of these libraries use the package construct while the equivalent library in Verilog uses modules [6]. Caution must be exercised to ensure that the module must never be instantiated more than once since variables encapsulated inside will then exist multiple times. The use of packages avoids this problem, since packages are not instantiated, thereby guaranteeing that all variables inside are singletons (with exactly one instance in existence).

Packages can be considered as stand-alone elements, dependent only on other packages and not on anything in the context they are used. Thus, they can be compiled separately into libraries of functionality, pulled in only when required. One can view this to be conceptually equivalent to how ‘C’ libraries are

organised and used. This stand-alone property means that code inside packages cannot contain hierarchical references to anything outside the package, including the compilation unit. Other encapsulation mechanisms such as modules and classes do not require this, so a module/class meant to be reusable must rely on the discipline of the writer to avoid these external dependencies. Thus packages represent a much better mechanism for encouraging reuse, since external dependencies are explicitly disallowed and checked at compile-time. Users of purchased verification IP should insist that their vendors provide the IP in the form of a package.

While the stand-alone property of packages may make it seem that it is impossible to tightly integrate outside code with package code, this is not the case. By using callbacks and abstract base classes, this is possible. A subsequent section describes this further.

SV packages were inspired by and share many commonalities with similar constructs in other languages. Namespaces in C++ are similar, providing a scope for encapsulating identifiers. Explicit access is given through the :: scope operator, identical in both SV and C++. Importation of namespace symbols in C++ with the *using* keyword mirrors the use of the *import* keyword in SV. A notable difference is that nested C++ namespaces can be hierarchically referenced using multiple :: operators. A user which imports a SV package, *top_pkg*, which imports another package, *bottom_pkg*, does not automatically have access to any of the symbols in *bottom_pkg*, and access through multiple :: operators is not possible. This lack of package "chaining", and the closely related *export* statement, is described in more detail in a subsequent section.

VHDL packages also share many common features with SV packages. One notable feature of packages in VHDL absent in SV is the explicit separation of the package header and body. The header represents the publically visible interface to the package, giving access to type definitions and function prototypes. The body implements the various functions described in the header, and are invisible and irrelevant to the user.

SystemVerilog allows the collection of files defining a simulation to be broken into compilation units, the definition of which is implementation dependent. This is often a function of the compile strategy implemented by the tool, with an "all-in-one" command line defining one large compilation unit for all files, and an "incremental-compile" strategy defining compilation units on a per file basis. Various issues ranging from visibility to type compatibility are linked to compilation units, leading to unpleasant surprises when switching compile strategies. Using packages avoids this problem, since the rules of visibility and type compatibility surrounding package items are independent of compilation unit. The problems described later in III-E do not occur when packages are used.

III. PRACTICALITIES IN PACKAGE USE

Over the past several years, we have deployed SV environments using packages in multiple projects. With each passing project, lessons learned from the previous mistakes refine the implementation choices made going forward. This section discusses some selected lessons from this experience.

A. COMPILE PACKAGES BEFORE USE. While this may sound obvious, it is sometimes not as trivial as it seems. Nicely written packages are good reuse candidates and may be referred to by other packages, bringing up the issue of inter-package dependencies. Large projects typically have many packages with complex inter-dependencies, with some packages reused from other projects and others reused from block to cluster to chip level environments. The mechanism which controls how all the files for a particular simulation are compiled, be it one file at a time, or all files together, must infer the dependency structure of the packages and generate an appropriate compile order. This is most easily done by a tree graph with each node representing a package. From the resultant tree, the packages furthest away from the root must be compiled first. As the project evolves, careless modifications can lead to the formation of circular dependencies, resulting in no suitable compile order. This is most likely to occur in projects with a large set of packages and multiple environments that use different subsets of packages.

The following guidelines are suggested to minimize the chance of circular dependencies among packages as well as promote better reuse.

- Prefer smaller packages to larger ones.
- Don't let the entire team modify packages.
- Adequately document the contents of every package and what each package item does. It is also important to document which projects and environments use a particular package.
- Categorize packages into two types: those reused from other projects, and those reused from block to cluster to chip levels.

Finer grained package structuring reduces the possibility of unintended dependencies. Packages designed to be reused over multiple projects should be as flat and dependency free as possible. This allows re-users of the package to not pull in additional dependencies, which may cause problems. A typical package structure involves separate packages for each block level, which are brought together to form packages for higher-level environments. The direction of reuse should start from block environments and move upwards. Monolithic environments are particularly at risk of introducing downward dependencies as code evolves, creating potential circular dependencies.

When circular compile dependencies do occur, they can be resolved by repartitioning code between packages. The extreme solution of creating one huge package, guaranteed to have no circular compile dependencies, is always an option if one gets desperate.

B. IMPORTING PACKAGES. The easiest way to use packages is through a wildcard import, with the "import pkg::*" statement. This gives the importing scope access to all identifiers inside the package without having to use explicit imports for each desired identifier, or prefixing each identifier with the package name. While the latter two methods of package use are legal, they can be overly verbose and unpractical. Prefixing at each use has the added disadvantage of making it difficult to quickly determine package dependencies. Thus, using wildcard imports while understanding their disadvantages is the most practical strategy. These disadvantages are discussed below.

When a package is wildcard imported, the importer's namespace is widened to include every possible symbol from the package, even the undesired ones, or the ones that the package designer had no intention of making externally visible. Noteworthy is that the labels of enumerated types are also added. Thus, special care must be made to avoid naming conflicts, when possible. This is sometimes difficult with packages containing a lot of code, or where the code has been split out into multiple sub-files.

Purchased IP may be in package form, but encrypted, meaning that the user has no way of knowing what a wildcard import will bring. When the user imports multiple packages, the risk of naming conflicts between the various packages or with the importing scope is even higher. While naming conflicts are legal in some situations, the rules defining how these are resolved are lengthy and complex. Following a naming convention using reasonable and relatively unique names can greatly reduce the changes of naming conflicts, thus avoiding the task of having to learn SV's name resolution rules.

The question of whether package symbols are chained is specifically addressed in SV-2009, but not mentioned in the earlier SV-2005 LRM. This has led to the unfortunate situation of diverging behaviour between different simulation tool implementations. One tool disallows package chaining by default, consistent with SV-2009 rules, while another tool automatically chains all symbols. The *export* statement added in SV-2009 was meant to give package designers explicit control on which symbols are chained and visible to importers of the higher-level package. The code example below illustrates how the two tools described above are divergent.

```
package bottom_pkg; int foo; endpackage

package top_pkg;
  import bottom_pkg::*;
`ifndef VCS
  export bottom_pkg::*;
`endif
  int bar;
endpackage

module importer;
import top_pkg::*;
  initial
    begin
    foo = 1;          // from bottom_pkg
    bar = foo + 2;
    end
endmodule
```

Note that the tool that automatically chains all the symbols from *bottom_pkg* also does not support the export statement. In effect, the export is automatically implied, whereas the LRM requires the statement to be explicitly present for chaining to occur.

The export statement, together with the use of a wrapper package, can fine tune and control exactly what is to be externally visible. This technique further mitigates the side effects of wildcard imports. SV packages have no explicit mechanism to define certain symbols as private (e.g., through the use of the *local* keyword as a modifier), and the wrapper technique is an alternative to overcome this fact. These wrappers can be tailored to each use case, on a per project or per environment basis. The use of wrappers on unfamiliar or encrypted package contents can be viewed as a safety precaution.

C. USING SUB-INCLUDES. When many large classes are defined in a package, the sheer amount of code can lead to very large and difficult to maintain files. It is tempting to separate the package contents into multiple files, then have the package definition consist simply of a list of `include statements. This solution is seen often, but several dangers need to be managed, discussed below.

By separating package code out into other files, the original context can be easily forgotten. Allowed dependencies owing to the fact that multiple files make up the compilation unit, as well as disallowed ones are not readily evident. A further problem is that the file could be included in multiple places, resulting in several packages with the same definition. These packages could then be used together, causing problems at import. Specifically, identical type definitions included into two different packages may not be compatible. One must remember that with packages, it is no longer required or appropriate to implement reuse at the file level using `include statements.

The context loss problem can be easily addressed by having a clear warning comment at the top of the file indicating that only the intended package may include the file. An example is shown below.

file: my_huge_pkg.sv

```
package my_huge_pkg;
  `include "my_class1.svh"
  `include "my_class2.svh"
endpackage
```

file: my_class1.svh

```
// WARNING:
// This file is meant to be used only by
// "my_huge_pkg.sv".  DO NOT directly include
// in any other context.
class my_class1;
  ...
endclass
```

A more robust mechanism, for people who don't read comments, is to use an #ifdef check with an #error clause to trigger an immediate compilation error in cases of unintended inclusion. Modelled after the mechanism used by 'C' include files, the main package file would define a unique pre-processor symbol, then include the various sub-files. Each included file would check that the symbol is defined and trigger an error if it is not. The previous example, modified to incorporate this, is shown below.

file: my_huge_pkg.sv

```
package my_huge_pkg;
  `define _IN_MY_HUGE_PKG_
  `include "my_class1.svh"
endpackage
```

file: my_class1.svh

```
`ifndef _IN_MY_HUGE_PKG_
** ERROR     ERROR      ERROR
** This file is meant to be used only by
** "my_huge_pkg.sv".  DO NOT directly include
** in any other context.
`error "SV doesn't have this"
`endif
class my_class1;
   ...
endclass
```

Since the SV pre-processor does not have the `error directive, inserting text which will cause a compile syntax error can be used to do the same thing.

D. THE PARAMETER PROBLEM. Parameters can be used to define constants. They can also be used to facilitate generic programming, where the parameterized values can be varied. The usage of constant parameters in packages is problem free and a recommended replacement for pre-processor `defines for constants. This effectively gives a namespace to constants and avoids the potential problem of multiple (and/or conflicting) pre-processor symbols in the same compilation unit.

The second usage, for generic programming, causes a serious problem when used in the context of a package. When a function defined in a package uses a parameter, one might think a template function is defined. However, since packages are not instantiated, there is no way to vary the parameter to create different specializations of the function. The example below shows the problem for both type and value parameters. The same function defined in a module does not suffer this problem, since many instances of the modules may be created, with varying parameterizations.

```
package utils_pkg;
  parameter type T = int;
  typedef T T_list[];

  // extend 'n' samples right
  // extend 'm' samples left
  function T_list dwt_extend(T sin[],int n,m);
    T sout[$] = sin;
    int unsigned size = sin.size();
    for (int i=0; i<m; i++)
      sout = {sout[2*i+1], sout};
    for (int i=1; i<=n; i++)
      sout = {sout, sout[$-(2*i)+1]};
    return sout;
  endfunction

parameter win = 8;
localparam wout = win+1;
  function void do_rct(
    input bit signed[win:1] rgb[3],
    output bit signed[wout:1] ycbcr[3]);
  endfunction
endpackage
```

To overcome this problem, a static class can be used to wrap the function. The class can be parameterized, and access to the function is through the class resolution operator along with the parameterization. This, however, leads to unsynthesizable code, a problem if the code is to be used for RTL design. We have found that this problem occurs often in modelling mathematical algorithms meant for a DSP where the bit-depth of the operands is parameterized. An example of the solution is shown below.

```
package utils_pkg;
  virtual class colour_trans#(int win=8);
  localparam wout = win+1;
  static function void do_rct(
    input bit signed[win:1] rgb[3],
    output bit signed[wout:1] ycbcr[3]);
  endfunction
  endclass
endpackage
```

```
import utils_pkg::*;
initial
  begin
  bit signed [18:1] rgb[3];
  bit signed [19:1] ycbcr[3];
  colour_trans#(18)::do_rct(rgb, ycbcr);
  end
```

E. DEFINING CLASSES AT TOP-LEVEL. Class definitions may appear in various design elements, but packages remain by far the best place for classes. Alternatives such as modules or program blocks suffer from problems such as poor accessibility or reusability issues due to hierarchical references.

Users with a VERA background often do not appreciate the multitude of choices where classes may be defined. In VERA, all classes are typically defined in separate files, included when required and exist in a single global scope — in other words, "floating" at top-level. The code example below illustrates this, with each box representing a separate file and compile.

```
class logger {
  integer curr_sev;
  task put_msg(integer lvl, string msg);
}
task logger::put_msg(integer lvl, string msg)
{  ...  }
```

```
#include "logger.vrh"
class ahb_trans {
  logger log_obj;
  task new(logger l) { log_obj = l; }
}
```

```
#include "logger.vrh"
#include "ahb_trans.vrh"
class ahb_write_trans extends ahb_trans {
  task new(logger l) { super.new(l); }
}
```

```
#include "logger.vrh"
#include "ahb_trans.vrh"
#include "ahb_write_trans.vrh"
program top {
  logger log_obj;
  ahb_trans       a1;
  ahb_write_trans a2;
  log_obj = new;
  a1 = new(log_obj);
  a2 = new(log_obj);
}
```

While an equivalent structure is possible in SV, this usage style is not recommended. Not only are these potentially non-

reusable, the rules governing such structures (compilation-units) have changed between SV-2005 and SV-2009.

When class (or other) definitions do not appear in a module, package or program block scope, these "floating" definitions are part of the compilation unit scope (also called $unit). SV-2005 specifies that $unit behaves as an anonymous package. The consequences of this are significant and negative. Since the package is unnamed, there is no way to refer to any of its contents outside the compilation unit. Additionally, having to adhere to the rules governing packages means the code in $unit may not have hierarchical references. Unable to enjoy the advantages of package membership but still subject to its restrictions, the anonymous package concept is overall a bad idea and should be avoided.

SV-2009 has completely eliminated the term "anonymous package" from the LRM and changes the semantics of compilation-units to allow hierarchical references. The reasoning behind this is that compilation-units are not considered stand-alone, but rather always considered within some other context. This allows for the use of top-level classes with hierarchical references (consistent with the VERA usage described above), but the code cannot be reasonably considered reusable.

Notwithstanding the relaxation of rules in SV-2009, we recommend against the use of "floating" code in compilation-unit scopes. As previously mentioned, situations may arise where the definition of compilation unit boundaries is dependent not only on the way the source files are specified on the command-line to the compiler, but also compiler implementation decisions allowed by the LRM and outside the control of the user.

Further complicating the issue is the inconsistent application of the rules among different simulators. One product strictly enforces the SV-2005 hierarchical reference rule for compilation units even as the LRM has changed to allow for it. Surveying the releases over the past 3 years of another product shows that early versions falsely allowed hierarchical references in packages, with later versions corrected to produce a compile error, compliant with SV-2005. The latest revision adopts SV-2009 rules, reallowing hierarchical references in compilation units.

Another important issue is the type compatibility rules in SV (both 2005 and 2009 versions) surrounding compilation units. User-defined types and classes residing in the compilation-unit scope, as will be the case when top-level include files are used, are not equivalent to another type with the identical name and contents in another compilation-unit. Using the same include file for both compiles, ensuring that the type's name and contents are identical, does not make the types equivalent. An "all-in-one" compilation strategy with one large compilation-unit solves this problem, but this precludes the advantages of using separate compile, including the creation of libraries of reusable code. Using packages for these user-defined types is a superior approach, independent of compilation strategy adopted or how any tool implements compilation-units.

The type checking that an SV simulator performs occurs after all the source files are compiled, at the elaboration (linking) stage. In other words, the problem described above passes compile, but fails to link. One may wonder why the same approach in "C" does not encounter this problem. The key difference lies in the nature of the object file, which is low-level assembly code for the case of a "C" compiler. The type information is long gone, and the linker resolves symbols, reporting an error when symbols are not found or duplicated.

IV. Advanced Use Cases (classes and packages)

Packages and classes may be mixed together to implement interesting and useful things.

A. Singleton Implementation. The pure nature of its specification means that packages are singletons, or objects with exactly one instantiation. One can use classes with a private constructor to also implement the singleton design pattern and both approaches are equally effective.

Singletons find several uses in testbenches, from the encapsulation of global variables to the creation of testbench services such as logging objects and abstract factories. While ultimately a question of style, the author has the flexibility to choose between package- and class-based implementations. We find the package-based approach more lightweight, suitable for global variables such as error counters. The class-based approach is more suitable when the singleton is used as part of another design pattern, such as factories.

B. Callbacks and Abstract Base Classes. The value of packages being standalone is its reusability. However, each reuse situation might have slightly different requirements in its interaction with package code. Hierarchical references from package code are not allowed and a workaround using DPI and strings with paths, suggested in [5], violates the spirit of the rule. We strongly recommend against it. A better solution, using callbacks, is recommended.

Well-defined and placed callbacks provide a mechanism for customization while at the same time keeping the package code closed. This technique is well proven in multiple verification methodologies and found in software libraries such as the standard C library. It is instructive to illustrate from there the `signal()` function, shown below.

```
signal(int sig, void (*func)(int));
```

This allows the user to register a callback, `func`, to be called when the named event occurs. Here, the callback is a function pointer, reminding us that an object-oriented language is not required for implementations. SV has no function pointers, so implementations using abstract base classes are used. The example below illustrates this.

```
package ebcot_pkg;

  // define callback function interface that
  // 'ebcot_enc' will use.
  // (pure not in SV-2005)
  virtual class ebcot_encoder_cb;
    pure virtual task push(...);
  endclass
```

```
  function ebcot_enc(data, state,
                ebcot_encoder_cb cb=null);
    // iterate over rows/cols, calling 'doit'
    for (int r=0; r<nrows; r+=4)
      for (int c=0; c<ncols; c++)
        begin
        partial = doit(data,state);
          // execute callback, if it exists
        if (cb!=null) cb.push(partial);
        end
  endfunction
endpackage
```

```
// Application which uses ebcot_pkg::ebcot_enc
module enc(clk, data, data_valid);
import ebcot_pkg::*;
  // customize callback for this application
class my_enc_cb extends ebcot_encoder_cb;
   task push(...); ... endtask
endclass

my_enc_cb cb = new;
always @(posedge clk)
    // call encoder, passing in callback
  if (data_valid) ebcot_enc(data,state,cb);

endmodule
```

An abstract base class with a pure virtual method is defined in the package alongside all the other contents. In each use situation, this base class is extended to concretely implement what the function is to do. An object of this extension is then indicated when the package code is used. The example above provides the callback procedurally as part of the entry point to the package code. Many other techniques of "registering" the callback are possible.

C. CONNECTING TESTBENCH COMPONENTS WITH ABSTRACT BASE CLASSES. The SV language unifies a hardware description language, with its statically elaborated module hierarchy with features from object-oriented programming, with dynamic elements such as class objects. It is sometimes necessary to merge the two worlds together in a testbench. The combination of classes and packages, along with abstract base classes is one way to achieve this.

When testbench components (such as transactors), written as SV modules need to interface to other class-based testbench components, a bridge needs to be created. The interface that the module wishes to expose needs to be written as a set of tasks/functions forming an API. The class-based component may assume this API in its abstract base class form. The module-based component implements the concrete class extended from this virtual base. The abstract base class needs to be globally visible and thus must be implemented in a package. The concrete extension is normally local and defined in the module, since access to the variables/ports in the module's scope is required. A handle to an instance of the concrete extension class is obtained through an accessor function, which can then be bound to the class-based world.

This technique, an extension of the one described in [4], allows testbench components, regardless of their hierarchical relationship, to communicate with each other. This is done without the use of hard-coded XMRs (cross-module references), or virtual interfaces. While the motivation in [4] centred around BFMs, our treatment is more general, abstracting the communication with an API embodied in an abstract base class. Not only a class object and module instance can be bridged, but also two modules can also be bridged. One recent project uses this technique to embody the API of a module-based testbench-top (test harness), of which there were several varieties including multiple block levels to chip level harnesses. This API was then passed to a series of testcases (scenarios), which could be implemented either as top-level modules or classes.

An example of this technique is shown below. A module-based memory with a set of backdoor tasks exists in the statically elaborated world. The API for these tasks can be exported and connected to any other component, be it another module (as shown) or another class (not shown). All components are independent, with only a single place (in 'harness') where everything is tied together.

The package that holds the abstract base class representing the API is shown below:

```
package mem_access_pkg;
  virtual class mem_access;
  pure virtual function bit [7:0]
      backdoor_read(bit [31:0] addr);
  pure virtual function void
      backdoor_write(bit[31:0] a, bit[7:0] d);
  endclass
endpackage
```

The module based memory model, 'ddr', implements backdoor memory access functions. The module-based version of the functions may be called using hierarchical reference. The class-based version may be used by any component regardless of hierarchy, once a handle to the API object has been obtained.

```
module ddr;
  bit [7:0] mem_array[bit[31:0]];
    // backdoor memory access functions
  function bit [7:0] backdoor_read(
                            bit [31:0] addr);
    return mem_array[addr];
  endfunction
  function void backdoor_write(
               bit [7:0] d, bit[31:0] addr);
    mem_array[addr] = d;
  endfunction

    // implement class-based version
  import mem_access_pkg::*;
  class my_mem_access extends mem_access;
    function bit[7:0] backdoor_read(
                             bit[31:0] addr);
      return ddr.backdoor_read(addr);
    endfunction
      // NB:arguments swapped for illustration
    function void backdoor_write(
                  bit [31:0] a, bit [7:0] d);
      ddr.backdoor_write(d,a);
    endfunction
  endclass

  // definition of object, with accessor
my_mem_access _obj;
  function mem_access get_mem_access();
    if (_obj==null) _obj=new;
    return _obj;
  endfunction
endmodule
```

The module below shows how the class-based API enables the backdoor access functions to be used, without knowledge of the hierarchical relationship between the two modules. Only the package holding the abstract base class is required.

```
module sister_module;
  import mem_access_pkg::*;
  mem_access ma_handle;
  function void put_mem_access(mem_access a);
    ma_handle = a;
  endfunction

initial
  begin
  wait (ma_handle != null);
  ma_handle.backdoor_write(100, 8'h2b);
  $display ("read=%x",
              ma_handle.backdoor_read(100));
  end
```

The top level testbench module ties everything together and is the only place where the hierarchical relationships (u_ddr and u_oth) are used.

```
module harness;
  ddr u_ddr();
  sister_module u_oth();

initial
  begin
    // this triggers u_oth to do mem accesses
  u_oth.put_mem_access(u_ddr.get_mem_access);
  #10;
    // again, but with hierarchical reference
    // to functions in u_ddr
  u_ddr.backdoor_write(8'h45, 100);
  $display ("read=%x",
                u_ddr.backdoor_read(100));
  end

endmodule
```

Without the use of packages to store the abstract base class, this technique becomes hard to implement. One can use an include file for the class, including it in each place that requires it. However, this runs into the type compatibility problems described previously.

Alternatives to this approach include using hard-coded XMRs from the class to module in question. Not only is this not reusable due to the hard-coded XMRs, this is not even legal when the class is defined in a package or program block scope.

D. BINDS, PACKAGES, AND WHITE-BOX TESTING. The combination of the SV bind construct along with a package implementing a global symbol-table allows verification code to be deeply embedded in a DUT with no hard-coded hierarchical references. A module or interface with the verification code, be it assertions, a monitor, or coverage collector is bound to the DUT module in question. Access to the results of the monitor or coverage collector is normally problematic, requiring hierarchical references through the DUT module hierarchy to reach the target.

By using packages, each monitor can define an API and register it in a global symbol table implemented in the package. The end-user of the monitor/coverage result can access the API through the package. The symbol table acts as a drop-box and avoids the need for hierarchical references.

E. POOR MAN'S INHERITANCE. Packages containing variables and tasks/functions can be compared to classes with data and methods. However support for inheritance of packages is not as flexible as that in classes. A so-called poor man's inheritance mechanism is possible, allowing for static (compile-time) polymorphism but not the dynamic polymorphism that classes can implement. A wrapper package can be created which redefines some of the functions in the underlying package, provided the prototypes are identical. In the extreme case where all functions are redefined a complete substitute package can be made, with a different implementation of all functions provided by the package.

It is interesting to note that VHDL, a non object-oriented language, is capable of this by strictly separating the package implementation from its declaration. Modules from ordinary Verilog can be said to have the same capability.

F. MIXED USE OF VHDL AND SV PACKAGES. Mixed language simulation is sometimes a necessary evil. The combination that we see most often is an SV testbench verifying a VHDL design. Often, a rich set of records, types and functions on the VHDL side is defined in packages. Unfortunately, neither SV nor VHDL LRMs specify how these definitions can be mapped across the language boundary, even though most package items have exact parallels in SV. Tool specific implementations, often as simple as adding an additional compile-line switch, are available.

V. CONCLUSION

We have given an overview of the SystemVerilog package construct, from its motivation to the characteristics that make it an important feature of the language.

Practical issues that arose when using packages in real projects were described. Suggestions to avoid or overcome these issues were made. We further discussed how packages and classes could be used together to implement interesting constructs.

REFERENCES

[1] "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2005, 2005.

[2] "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language," IEEE Std 1800-2009, 2009.

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

[4] D. Rich, J. Bromley. "Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches". DVCon 2008.

[5] "XMR in Testbench-to-DUT or Top-Module Tasks." $VCS_HOME/doc/UserGuide/pdf/VCSLCAFeatures.pdf, p179. Version C-2009.06. June 2009.

[6] Xilinx library code. $XILINX/vhdl/src/simprims/simprim_Vcomponents.vhd and $XILINX/vlog/src/glbl.v. Release v11.1i. Apr. 2009.