New and Active Ways to Bind to Your Designs

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

***Abstract*—Verification engineers have long known of the advantages of using the SystemVerilog bind construct to tie their verification code to design code. Sets of assertions, wrapped in modules or interfaces, can be bound to design modules in a reusable, hierarchically independent manner. Similarly, functional coverage monitors can be bound to the DUT without having to modify the design RTL code. These and other current use models for binds are passive. We use the term passive, since once the code to be bound has been sent into the design hierarchy, the testbench code loses all contact with it.**

This paper introduces a new technique where binds are active. The testbench code can actively manage and control, at runtime, the code that was bound into the design. By combining the use of packages as well as abstract base classes with the bind construct, this paper details how *active binding* can be realized. The solution uses basic SystemVerilog constructs defined in IEEE 1800-2009 [3], and works on the simulators from all major vendors. It is also independent of methodologies such as UVM, OVM or VMM, allowing the testbench architect to deploy these techniques regardless of the chosen methodology.

With active binding, new use models for using the bind construct are available, which this paper will describe. Several examples from recent projects will be provided as illustrations.

I. Introduction & Motivation

The SystemVerilog `bind` construct was originally envisioned to allow verification engineers to insert assertions into design RTL in an unobtrusive way. Before its introduction, such verification code was typically appended to the end of the designer's RTL module, protected by pre-processor defines so they could be easily removed. Since the verification code was not synthesizable, it was surrounded by synthesis pragmas to avoid problems in synthesis. This approach raised a number of methodological and administrative challenges stemming from the fact that design and verification teams ought to operate independently, with their own databases. Design engineers don't like anybody else touching their code.

```
// Example showing direct inclusion of verification
// code in DUT
module design (...);
  //              ...
  // synthesizable, RTL description of something
  //              ...

// synopsys translate_off
`ifdef EMBED_VERIFICATION_CODE
    assert (blah_blah);
    internal_bus_mon u_mon(
      .clk(clk),
      .signal(signal)
    );
`endif
// synopsys translate_on

endmodule
```
Listing 1

An alternative approach was to place assertions inside a module that was instantiated in the verification environment, and not in the DUT. Here, connections had to be made between the module and signals buried within the DUT, as shown in Figure 1 and Listing 2. Hierarchical path names to the internal DUT signals need to be used, making the connection process error-prone and dependent on the design hierarchy, which often changes late in the design cycle, and without advance notice.

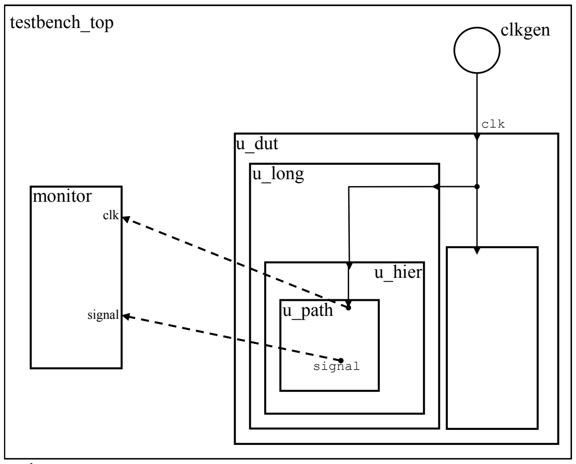


Figure 1

```
// Example showing hookup using hier. paths into DUT

module testbench_top;
   DUT u_dut(...);

   assert (u_dut.u_long.u_hier.u_path.blah_blah);

   internal_bus_mon u_mon(
      .clk(u_dut.u_long.u_hier.u_path.clk),
      .signal(u_dut.u_long.u_hier.u_path.signal)
   );
endmodule
```
Listing 2

The `bind` construct elegantly solves both problems, specifying code to be inserted into the target modules/instances at elaboration time before simulation starts. The DUT is unmodified and thus neither the designer nor the backend team are affected. The SystemVerilog LRM [3] defines two variants of the bind directive, shown below:

- a module-based form:
```
bind dut_module module_to_bind
       bind_inst(...);
```

- an instanced-based form:
```
bind dut_module:hier_path
       module_to_bind
       bind_inst(...);
```

The module-based form of the construct inserts into every instance of the target module in the design, regardless of hierarchical path, and thus is robust to hierarchy changes. The instance-based form can target specific instances of a target module, but requires that the hierarchical path to that instance be specified. This dependence on hierarchical paths negates much of the benefit of using binds. The rest of this paper will only focus on the module-based form.

To avoid confusion, some terminology needs to be defined. In every bind directive, there are three elements:

- the verification module (e.g., `module_to_bind`) that is to be bound, and the resulting instance(s), which we will refer to as 'bound instance', or 'bound code'. [parasite]
- the target module (e.g., `dut_module`) where the 'bound instance' is instantiated. This is called the 'bind target'. [host]
- the scope (typically the top level verification module) containing the bind directive, also called the 'bind instantiation'. [where the host got sick]

In addition to binding assertions, several papers [1][4] in previous years have discussed the possibility of binding functional coverage objects into design code. We consider both these use cases, the binding of assertions or coverage objects, to be *passive binding*. Once bound into the DUT, the verification environment has no ability to interact with the bound code at run-time. The code operates autonomously, observing the surroundings it has been bound into, collecting information or triggering assertions as necessary.

More complex use cases are possible if it were possible for the verification environment to communicate with the bound code in a dynamic manner at run-time. This paper describes a technique to achieve this communication, which we shall refer to as *active binding*. Section II describes how the mechanism works as well as the necessary preliminaries. a simple example is described in Section III, while Section IV illustrates an advanced example. The use case in this advanced example was the motivating factor behind this work.

II. How Active Binding Works

Consider a transactor, loosely defined as something that has a signal level interface and a control API. Interaction with the DUT is through its signal level interface, and the control API is used by the testbench. A transactor, implemented as a module, can be bound deep inside a DUT, with the bind instantiation describing how the signal level interface is connected. With active binding, the control API can be used by the testbench to interact with the transactor.

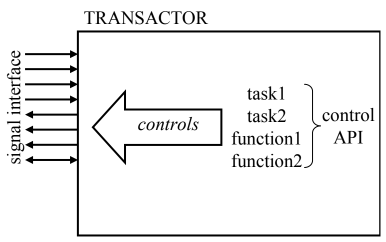


Figure 2

A control API consisting of a set of task/function prototypes can be specified as an abstract base class with pure virtual methods. This base class is defined in a package, making it globally visible, accessible from any part of the testbench. Listing 3 illustrates this.

Inside the transactor, a concrete class derived from the abstract base is defined, and implements each of the methods in the API, as shown in Listing 4. Thus, the manner in which the API relates to the signal interface is defined. A single instance of the concrete class, also defined in the transactor, serves as an object that other testbench components can use to interact with the transactor.

```
package axi_xactor_api_pkg;
virtual class axi_xactor_api;
  pure virtual task
       SEND_TO_DUT(input axi_transaction t_in);
  pure virtual task
       WAIT_FOR_TRANSACTION(output axi_transaction
                                   t_out);
endclass
endpackage
```
Listing 3

This elegant technique of connecting testbench to DUT has been discussed in [2], [5] and [6]. By building on it with the bind construct, the need for hierarchical references is eliminated. At time 0, the transactor API object described above is created and assigned into a symbol table implemented in a package (`bind_dropbox`). Once the object is registered into the symbol table, other testbench components recover it to gain access to the transactor. Since a full testbench may contain many bind directives, each of which potentially yielding multiple objects, a unique identifying string is required as the index into the table. The %m operator returns the hierarchical path in which it is used and is thus unique and ideal for this identifying string. It is important to note that the instance name used in the bind directive is the last part of the dotted name string returned by %m. The internal operation of the `bind_dropbox`, described in Section V, takes advantage of this fact.

```
module axi_xactor (
    <... signal interface ...>
);
import axi_xactor_api_pkg::axi_xactor_api;
event got_transaction;
axi_transaction saved_t;
always @(posedge clk)
  begin
   // ... monitor signal interface ...
   // ... save info in saved_t ...
   if (transaction_done)
     ->got_transaction;
  end

  // concrete implementation of axi_xactor_api
class my_axi_xactor_api extends axi_xactor_api;
  task SEND_TO_DUT(input axi_transaction t_in);
    if (!passive)
       // ... wiggle signal interface ...
  endtask

  task WAIT_FOR_TRANSACTION(output axi_transaction
                                   t_out);
    @(got_transaction) t_out = saved_t;
  endtask
endclass

  // one instance of concrete implementation,
  // automatically constructed at time 0 (before all
  // initial blocks)
my_axi_xactor_api _my_api = new;

  // register above instance in the dropbox
initial bind_dropbox::register(
                        $psprintf("%m"), _my_api);

endmodule
```

Listing 4

Combining binds with a package based symbol table was also described in [5]. Astute reads may observe that UVM provides the same mechanism through its resources database. The underlying SystemVerilog constructs used to implement the UVM resources database is similar to our approach, which is methodology neutral.

The technique of using abstract base classes to connect to a DUT was also discussed in [6]. There, SystemVerilog interfaces, rather than Verilog modules, were used in the context of a UVM-based testbench. The bind directives there were instance-based and suffered from explicit hierarchical paths, which our method does not.

Once the entry in the symbol table has been set by the transactor, the testbench can retrieve the API object, allowing it to actively interact with it. Since the module-based variant of the bind directive instantiates the bound code into all instances of the target module, there is potentially a one-to-many relationship between bind directive and API objects. The recovery mechanism returns an array of object/string pairs, where the string defines the full hierarchical path to the object. In the case of multiply instantiated target modules, the testbench must determine which instance it wishes to interact with.

```
module testbench;
import axi_xactor_api_pkg::axi_xactor_api;

    DUT u_dut(...);
```

```
bind cpu_with_axi axi_xactor bind1
                     (... signal connections ...);
bind dma_with_axi axi_xactor bind2
                     (... signal connections ...);
initial
  begin
  axi_xactor_api my_apis[$];
  string hier_paths[$];

  bind_dropbox::recover("bind1",
                        hier_paths, my_apis);

  assert (hier_paths.size()==1) else $fatal;

  monitor_loop(my_apis[0], "CPU");
  end

initial
  begin
  axi_xactor_api my_apis[$];
  string hier_paths[$];

  bind_dropbox::recover("bind2",
                         hier_paths, my_apis);

  fork
    monitor_loop(my_apis[0], "DMA0");
    monitor_loop(my_apis[1], "DMA1");
  join_none
  end

task monitor_loop(axi_xactor_api api, string ID);
  axi_transaction t;
  while(1)
    begin
    api.WAIT_FOR_TRANSACTION(t);
    $display ("detected AXI transaction at %s: %s",
                    ID, t.to_string());
    end
endtask

endmodule
```

Listing 5

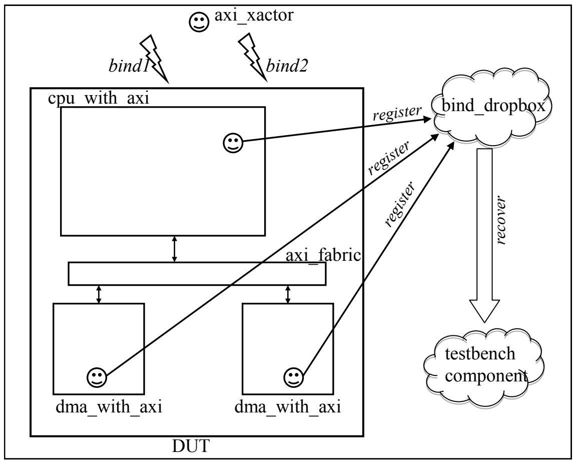


Figure 3

A DUT with an internal AXI-based bus fabric is shown in Figure 3. There is a CPU and two instances of a DMA, each with an AXI interface. Listing 5 shows how an AXI transactor can be bound into each of these for the purposes of monitoring internal bus activity. The bind directive, `bind1`, goes into CPU, while `bind2` results in two instances of the transactor being inserted. At the top level testbench, the handle to each of the API objects is recovered and passed down to other testbench code (e.g. monitor_loop), which uses it.

III. A Simple Example

The following example illustrates the concepts described in the previous section. Figure 4 shows a DUT with a register block, which is further sub-divided into configuration registers and FIFO registers.

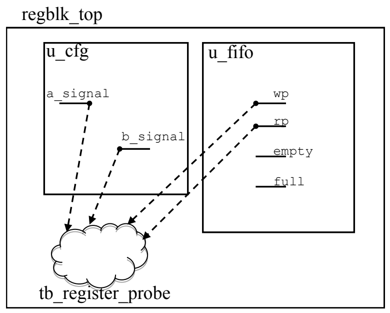


Figure 4

A testbench containing a register model (such as the UVM register layer) may wish to track and check the DUT's registers. Some registers, such as FIFO registers, are not predictable since they operate autonomously, and thus white-box testing is required. In other words, internal probing of the DUT is used to detect register state changes, which are then reflected in the testbench's class-based register model. A solution to this problem is given in [7], but relies on the VPI mechanism, as well as using hard-coded hierarchical paths.

Using binds to insert an observer module (shown as a cloud in Figure 4), provides a better solution. Note that the bind need not target the lowest leaf-level module. The port connections in the bind instance may use downward paths, since all signals visible at the targeted module are available for use. The bind directive used is shown in Listing 6.

```
bind reg_top tb_register_probe bind_inst(
  .a_signal_from_cfg_regs(u_cfg.a_signal),
  .b_signal_from_cfg_regs(u_cfg.b_signal),
  .fifo_wp_from_fifo_regs(u_fifo.wp),
  .fifo_rp_from_fifo_regs(u_fifo.rp)
);
```
Listing 6

The object API of the observer module embedded in the DUT, when recovered by the testbench, is the mechanism that bridges the DUT to class-based verification components, such as the UVM register layer.

Signals from multiple register sub-blocks may be aggregated into a larger monitor module, simplifying the interface between the DUT and the class-based world. The bind target is chosen high enough in the hierarchy such that aggregation is possible, but low enough to be resilient to changes in hierarchy. As subsystems are built from IP blocks, and SoCs are built from subsystems, these higher layers of design hierarchy are more likely to change as opposed to hierarchy within the IP block.

This example serves to illustrate a solution to the more general problem of connecting legacy (or module-based) transactors into a class-based (e.g. UVM) methodology. The preferred use of virtual interfaces by UVM does not work well here. The 2009 revision of the SystemVerilog LRM has closed a loophole, no longer allowing interfaces that have external hierarchical references to be made virtual. The use of abstract classes in our example eliminates the use of virtual interfaces, and the use of binds eliminates the need for hierarchical references.

IV. An Advanced Example

The advanced example presented in this section demonstrates how, when up-module references are added to active binding, the testbench can intrusively manipulate memories internal to the DUT. This is achieved without explicit knowledge of where in the design hierarchy the memory lies, and without the use of force statements.

Modern SoCs containing embedded CPUs have numerous internal memories that are required by the CPU to operate. Key among them are the memories used to hold the code and data binaries which represent the program the CPU is to execute. In a real system, these may be ROMs, or more typically, RAMs which are loaded from external non-volatile store before the CPU is released from reset. This is a time-consuming operation that is not practical (and not interesting) to simulate, and thus is often skipped and replaced with a backdoor, zero-time initialization mechanism.

The internal memories may be modelled abstractly using behavioural models, or more accurately using technology specific simulation models. In both cases, a 2-dimensional array representing the memory elements is at the heart of it, surrounded by a signal interface, timing elements and timing checks. We use a technology-neutral wrapper with standardized signal names to facilitate easy switching of technologies. This wrapper also supplies backdoor read/write functions and implements the necessary, wrapper-specific accesses to the 2-dimentional array.

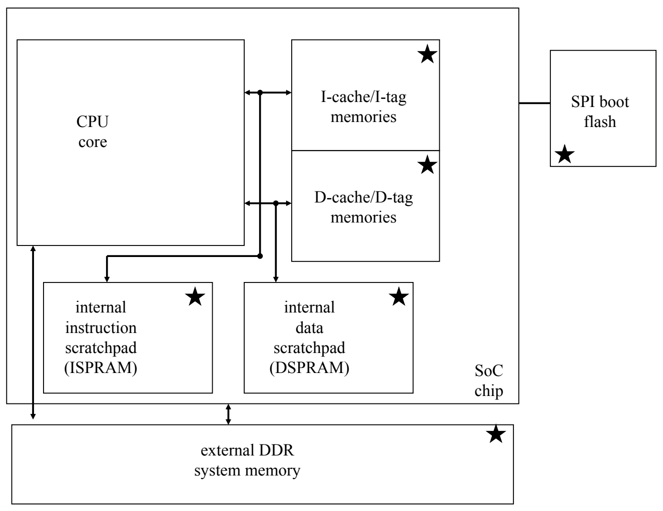


Figure 5

The design described above is illustrated in Figure 5. Blocks marked with a star are memories (with their wrappers) and candidate bind targets.

The bind directive targets the memory wrapper module. The bound transactor module being inserted differs slightly from the standard one shown in Figure 2. While it keeps its control API interface, no signal interface is required; i.e.: a module with no ports.

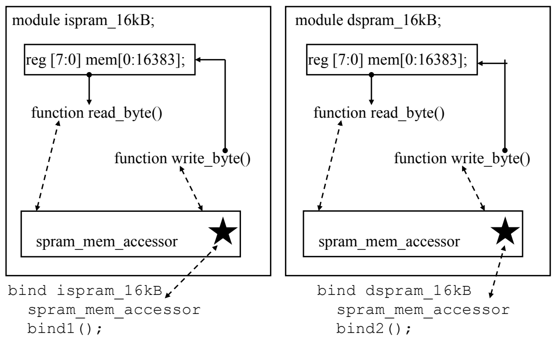
The bound code interacts with its DUT through function calls to the read/write functions in the wrapper, which does the rest of the work. From the scope of the bound transactor, the wrapper functions are accessible through up-module reference (see section 23.8.1 of [3]). An elaboration-time error occurs if the name resolution mechanism fails.

This up-module reference serves to replace the traditional signal-level interface, and imposes the same strict elaboration-time check. In other words, bound code that takes advantage of up-module references assume that they will be instantiated in a scope where the reference will eventually resolve. A module which instantiates a sub-module, but uses incorrect port names causes the same elaboration-time error, for the same reason.

```
( 1) module spram_mem_accessor();
( 2) import mem_accessor_pkg::mem_accessor_base;
( 3)
( 4) function void _write_byte(input [31:0] a,
( 5)                           input  [7:0] b);
( 6)   write_byte(a,b);        // up-module reference
( 7) endfunction
( 8) function reg [7:0] _read_byte(input [31:0] a);
( 9)   return read_byte(a);    // up-module reference
(10) endfunction
(11)
(12) class my_mem_accessor extends
(13)                         mem_accessor_base;
(14)   virtual function void write_byte
(15)                      (input [31:0] a, [7:0] b);
(16)     _write_byte(a,b);
(17)   endfunction
(18)   virtual function reg [7:0] read_byte
(19)                      (input [31:0] a);
(20)     return _read_byte(a);
(21)   endfunction
(22) endclass
(23)
(24) my_mem_accessor _mem_a = new;
(25) initial bind_dropbox::register
(26)                     ($psprintf("%m"), _mem_a);
(27)
(28) endmodule
```

Listing 7

Listing 7 shows a module which expects to be bound into a target module which implements the `write_byte()` and `read_byte()` functions. Note the similarity to Listing 4. It is irrelevant what the name of this module is, provided these two functions exist.



```
bind ispram_16kB
  spram_mem_accessor
  bind1();
```

```
bind dspram_16kB
  spram_mem_accessor
  bind2();
```

Figure 6

This is illustrated in Figure 6, which shows how the `spram_mem_accessor` module successfully binds to both `ispram_16kB` and `dspram_32kB`, since both implement the `write_byte()` and `read_byte()` functions.

Lines 12-22 in Listing 7 define the concrete implementation of the abstract base class, defining the `write_byte()` and `read_byte()` functions required by the base. An instance of this is created at time 0 on line 24. Line 25 registers this into the bind dropbox, where the testbench can recover it.

When the testbench uses the API object, calls to `write_byte()` will map to the function on lines 14-17, which will call a secondary function on lines 4-7. Note that the name has been slightly changed, to avoid name clashing. Continuing on, line 6 refers to the `write_byte()` function, which does not resolve to anything in this module. The enclosing scope (i.e., the bind target module) is searched, where we expect the name to be found.

Note that the up-module references on lines 6 and 9 work only because the identifier is a task/function. Direct access to `mem[]` does not work since it is not a scope that the up-module reference can resolve, and thus we rely on a wrapper function to create such a scope. An alternative way of writing these references, shown in Listing 8a, works without the wrapper. The disadvantage here is that the name of the module needs to be hard-coded, since it provides the encompassing scope the name-resolution mechanism needs. The accessor transactor module will need to be duplicated and the bind directive (Listing 8b) must match each to the appropriate target.

```
module spram_mem_accessor_ispram_16kB;

function void _write_byte(input [31:0] a,
                          input [7:0] b);
  ispram_16kB.mem[a] = b;      // up-module reference
endfunction
              . . .          . . .
```

Listing 8a

```
bind ispram_16kB
        spram_mem_accessor_ispram_16kB bind1();
bind dspram_32kB
        spram_mem_accessor_dspram_32kB bind2();
```

Listing 8b

By using this mechanism, a chip-level testbench can pre-initialize internal memories with the contents required for the CPU to operate. Such a system was implemented in the testbench for a recent SoC project. From a command-line argument, the ELF program file can be specified from which S-records are formed, translated into byte-wise memory writes, decoded through an address map, and initialized into the appropriate memory models. The state of the system after this process closely matches that of a real system after a JTAG debugger has loaded a program into a target. The processor is then released from reset, happily running the loaded program, oblivious to how it got into memory.

V. Implementation Details of bind_dropbox

The register and recover mechanism key to active binding are implemented in a SystemVerilog package, called `bind_dropbox`. Some specific points to how the two functions operate are described here.

In the `bind_dropbox` package is an associative array, which serves as a lookup table. The table elements are objects of a control API, and are indexed by string. The index strings must be unique.

Each instance that results from a bind directive needs to call the `register` function in the dropbox, providing a unique string as well as the control API object. The %m mechanism is used in the call to `bind_dropbox::register`, automatically providing a unique string. Since this string is a hierarchical path, it is in the form of a dotted name. The last component of the dotted name is the instance name of the bind directive. Listing 9 shows the code for the associative array and the `register` function.

```
package bind_dropbox;
import mem_accessor_pkg::*;
typedef struct {
  string hier_path;
  mem_accessor_base api_obj;
} dropbox_t;
typedef dropbox_t dropbox_t_list[$];

// define an assoc. array for the dropbox
dropbox_t_list lookup[string];

function void register(string hier_path,
                       mem_accessor_base obj);
  dropbox_t a;
  string l[$];
    // split the hier_path (a dotted name)
    // into its parts.
  string_pkg::string_split(hier_path, l, ".");
    // glue all but the last one back together.
    // This is basically basename(hier_path);
  a.hier_path = string_pkg::string_join
                       (l[0:$-1], ".");
  a.api_obj = obj;

    // insert entry into list, using the last part
    // as an index.  There may be multiple entries
    // per index.
  lookup[l[$]].push_back(a);
endfunction

endpackage
```

Listing 9

A verification testbench using active binding will contain one or more bind directives. The instance name of each of these directives must be unique, since this used to recover the control API object. Bind directives are elaboration time constructs and thus take effect before time 0. Registration into the dropbox happens in an initial block, at time 0. At run-time, presumably in an initialization phase, the objects need to be recovered, using the unique bind instance names. The implementation for recover is shown in Listing 10.

```
function void recover(input string ref_string,
                  output string hier_paths[$],
                  output mem_accessor_base objs[$]);
  dropbox_t_list list;
  hier_paths = {}; objs = {};
  if (!lookup.exists(ref_string)) return;

  list = lookup[ref_string];
  foreach(list[i])
    begin
    hier_paths.push_back(list[i].hier_path);
    objs.push_back(list[i].api_obj);
    end
endfunction
```

Listing 10

This method uses module-based binding, specifying the modules to be targeted and not hierarchical paths to individual instances. Therefore, each bind directive may result in multiple instantiations. For a given call to `recover`, multiple matches may be returned, which the verification testbench needs to be aware of. The list of unique hierarchical paths returned aids the testbench in determining which instantiation is interesting.

Not all control APIs are identical — transactors of different types have different control mechanisms. The example above shows a lookup table with elements of identical type (`mem_accessor_base`), for use in the environment described in Section IV. As written, all bind directives must bind things that have the same control API, a very limiting restriction. For the bind dropbox to support heterogeneous APIs, an umbrella base class from which all APIs are derived must be defined. It contains no objects or methods, serving only the purpose of parenting the various differing control APIs. This extra level means that `$cast` must be used to regain the proper type of the objects after they have been recovered from the dropbox.

Conclusion

By combining the technique of using abstract base classes for transactors with the bind construct, this paper has shown how these transactors can be instantiated inside a DUT without hierarchical references, and controlled dynamically at runtime. The technique described here is methodology neutral.

We have deployed this technique in the full-chip verification environment for a recent SoC. Active binding was used to place transactors inside the SoC's internal memories, which the testbench subsequently uses to backdoor initialize with program code to be executed by the embedded CPU.

Acknowledgment

The author wishes to thank J. Bromley and S. Bresticker for their invaluable comments during the development of this paper.

References

[1] B. Bailey and P. Marriott, “Functional coverage using SystemVerilog,” *Design & Verification Conference*, Feb. 2006, San Jose, CA.

[2] J. Bromley and D. Rich, “Abstract BFMs Outshine Virtual Interfaces for Advanced SystemVerilog Testbenches,” *Design & Verification Conference*, Feb. 2008, San Jose, CA.

[3] IEEE, “Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language,” IEEE Std. 1800-2009.

[4] M. Baird, “Coverage Driven Verification of an Unmodified DUT within an OVM Testbench,” *Design & Verification Conference*, Feb. 2010, San Jose, CA.

[5] K. Ho, “Using SystemVerilog Packages in Real Verification Projects,” *Design & Verification Conference*, Feb. 2010, San Jose, CA.

[6] D. Rich, “The Missing Link: The Testbench to DUT Connection,” *Design & Verification Conference*, Feb. 2011, San Jose, CA.

[7] J. Bromley, “I Spy with My VPI: Monitoring Signals by Name, for the UVM Register Package and More,” *SNUG* 2012, Munich, Germany.