

# **TestBuilder-to-TestBuilder-SC Interoperability Guide**

**Product Version 4.1  
October 2002**

© 2002 Cadence Design Systems, Inc. All rights reserved.  
Printed in the United States of America.

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522.

All other trademarks are the property of their respective holders.

**Restricted Print Permission:** This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used solely for personal, informational, and noncommercial purposes;
2. The publication may not be modified in any way;
3. Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and
4. Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

---

# Contents

---

<u>1</u>	
<u>Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC</u>	5
<u>Overview</u>	5
<u>HDL Connection</u>	6
<u>\$tbv tvn connect</u>	6
<u>\$tbv main</u>	9
<u>Direct HDL Signal Connection</u>	10
<u>Other Entry Points</u>	11
<u>Currently Missing Pieces</u>	11
<u>Arbitrary-Width Signals (tbvSignalT)</u>	12
<u>User Data Types (tbvSmartRecordT)</u>	12
<u>Concurrency (tbvTaskT and spawn)</u>	14
<u>Running or Spawning a Task</u>	14
<u>Example TestBuilder-SC Class to Mimic tbvTaskT</u>	16
<u>Spawning Arbitrary Methods and Functions</u>	17
<u>Concurrency Objects</u>	18
<u>Transactors (tbvTvmT)</u>	19
<u>Randomization and Constraint Solving</u>	20
<u>Differences in Randomization</u>	20
<u>Differences in Constraint Solving</u>	21
<u>Transaction Recording</u>	21
<u>Verification Data Structures</u>	23
<u>TestBuilder-1.3-to-TestBuilder-SC Mapping</u>	24

# TestBuilder-to-TestBuilder-SC Interoperability Guide

---

---

# Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

## Overview

TestBuilder-SC is a verification library built on top of the SystemC modeling library. It is very similar to TestBuilder 1.3, but there are differences:

- TestBuilder-SC uses naming conventions that are consistent with SystemC.
- TestBuilder-SC uses SystemC as the threading kernel.
- TestBuilder-SC needs to communicate to host simulators using the host simulator's native SystemC communication mechanism to improve performance and reliability.

In most cases, there is a one-to-one correspondence between the TestBuilder 1.3 API and the TestBuilder-SC API. However, some functionality that exists in TestBuilder 1.3 is not yet available in TestBuilder 1.3, and in some cases the way things are done in TestBuilder-SC is significantly different from TestBuilder 1.3.

This document describes how you can move a TestBuilder 1.3 test environment to TestBuilder-SC. The areas of concentration are:

- HDL Connection (`$tbv_tvm_connect`, `$tbv_main`)
- Arbitrary-Width Signals (`tbvSignalT`)
- User Data Types (`tbvSmartRecordT`)
- Concurrency (`tbvTaskT` and `spawn`)
- Transactors (`tbvTvmT`)
- Randomization and Constraint Solving
- Transaction Recording
- Verification Data Structures

## ■ Utility functions

For a mapping between commonly used TestBuilder 1.3 functions and methods and the TestBuilder-SC equivalents, see [“TestBuilder-1.3-to-TestBuilder-SC Mapping”](#) on page 24.

## HDL Connection

### **\$tbv\_tvm\_connect**

The Verilog `$tbv_tvm_connect` user task and VHDL `tbv_tvm_connect` foreign model have been removed. They are replaced by a vendor-specific connection methodology. If you are using the SC-HDL SystemC simulator provided with TestBuilder-SC, the connection methodology is a Verilog user task called `$sc_cosim_init` or a VHDL foreign model called `sclib:scmod`.

In TestBuilder 1.3, you used the `tbvTvmTypes` array or the `tbvTvmT::registerTvm()` function to register your TVMs in the test. In TestBuilder-SC, there is a macro called `SC_MODULE_EXPORT` that you use to register SystemC modules for communication with your HDL.

The following example shows a TestBuilder 1.3 TVM connection:

```
//mytvm.v
module mytvm (clk, out1, out2);
    input clk;
    output out1;
    output [7:0] out2;

    reg out1;
    reg [7:0] out2;

    initial $tbv_tvm_connect;
endmodule

--mytvm.vhd
architecture tb of mytvm is
    attribute foreign of tb:architecture is "FMI tbvlib:tbvlib";
begin
end tb;

//mytvm.tw
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
class mytvm : public tbvTvmT {
public:
    TBV_HDL_SIGNALS_BEGIN
        tbvSignalHdlT clk(clk, tbvEnumsT::READ_ONLY);
        tbvSignalHdlT out1(out1, tbvEnumsT::WRITE_ONLY);
        tbvSignalHdlT out2(out2, tbvEnumsT::WRITE_ONLY);
    TBV_HDL_SIGNALS_END
};
```

```
//tbvMain.cc
#include "mytvm.h"
tbvTvmTypeT tbvTvmTypes[] = {
    {"mytvm", mytvm::create, 0 },
    { 0 }
}
```

The following example shows a TestBuilder-SC transactor connection:

```
//mytvm.v
module mytvm (clk, out1, out2);
    input clk;
    output out1;
    output [7:0] out2;

    reg out1;
    reg [7:0] out2;

    initial $sc_cosim_init;
endmodule

--mytvm.vhd
entity mytvm is port
    (clk : in std_logic;
     out1: out std_logic;
     out2: out std_ulogic_vector(7 downto 0) );
end mytvm;

architecture schdl of mytvm is
    attribute foreign of schdl:architecture is "FMI sclib:scmod";
begin
end schdl;
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
//mytvm.h
SC_MODULE(mytvm) {
    sc_in<bool> clk;
    sc_out<bool> out1;
    sc_out<sc_uint<8> > out2;
    SC_CTOR(mytvm) : clk("clk"), out1("out1"), out2("out2")
    {}
};
```

```
//tbvMain.cc
#include "mytvm.h"
SC_MODULE_EXPORT(mytvm);
```

The next example shows how to code the HDL connection when you will be simulating with NC-SystemC:

```
// The C++ is exactly the same as SC-HDL, the only difference is in the HDL wrapper module.
```

```
//mytvm.v
module mytvm (clk, out1, out2)
    (* const integer foreign = "SystemC mytvm"; *)

    input clk;
    output out1;
    output [7:0] out2;
endmodule
```

```
--mytvm.vhd
entity mytvm is port
    (clk : in std_logic;
    out1: out std_logic;
    out2: out std_ulogic_vector(7 downto 0) );
end mytvm;

architecture ncsc of mytvm is
    attribute foreign of ncsc:architecture is "SystemC";
begin
end ncsc;
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

#### **\$tbv\_main**

The `tbv_main` Verilog system task and VHDL foreign procedure provided the main test entry point in TestBuilder 1.3. In TestBuilder-SC, no test entry point is required. For TestBuilder 1.3, you must create a dummy SystemC module that starts the main test thread manually.

**Note:** When the `sc_main()` entry point is supported, you will be able to instantiate the main in SystemC without the need for the extra HDL layer.

The following example shows the TestBuilder 1.3 main test entry point:

```
//top.v
module top;
    // Do normal connections, etc
    ...

    initial $tbv_main;
endmodule
```

The following example show the TestBuilder-SC main test entry point:

```
//top.v
module top;
    // Do normal connections, etc.

    // Instantiate a dummy test module.
    tb_main test();
endmodule
module tb_main;
    iniital $sc_cosim_init; // Using SC-HDL
endmodule

--top.vhd for vhdl connection
entity top is
end top;

entity tb_main is
end tb_main;

architecture al of top is
    component tb_main is
    end component;
begin
    test: tb_main;
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
end a1;

architecture ncsc of tb_main is
  attribute foreign of ncsc:architecture is "SystemC";
begin
end ncsc;

//tbvMain.cc
void tbvMain() {
  // Do the test.
  ...
}

//tbvMain.cc
SC_MODULE(tb_main) {
  SC_CTOR(tb_main) {
    TB_THREAD(tbvMain);
  }
  void tbvMain();
};
SC_MODULE_EXPORT(tb_main);

void tb_main::tbvMain() {
  // Do the test.
  ...
}
```

## Direct HDL Signal Connection

TestBuilder 1.3 has three HDL signal classes: `tbvSignalHdlT`, `tbvSignalHdl2StateT`, and `tbvSignalArrayHdlT`.

SystemC has an `sc_signal<>` channel that can be an arbitrary type. If the type is two-state (`sc_uint<>` for example), a connection to HDL will be two-state. If a signal is four-state (`sc_logic`), the connection to the HDL will be four-state.

For direct HDL connection, `sc_signal` has two methods:

- `sc_signal<>::observe_foreign_signal`
- `sc_signal<>::control_foreign_signal`

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

For example:

```
tbvSignalHdl2StateT in1 ("top.in1", tbvEnumsT::READ_ONLY); // tb 1.3
sc_signal<bool> in1; in1.observe_foreign_signal("top.in1"); // tbsc

tbvSignalT out1("top.out1", tbvEnumsT::WRITE_ONLY); // tb 1.3
sc_signal<sc_lv<8> > out1; out1.control_foreign_signal("top.out1"); // tbsc
```

## Other Entry Points

TestBuilder 1.3 provides a `tbvInit()` entry point that occurs before simulation. There is no corresponding entry point in TestBuilder-SC. However, when `sc_main()` is supported, it will occur before anything else, so you will be able to use it like `tbvInit()`.

NC-SystemC and SC-HDL provide other useful entry points inside of `sc_module()`. They are virtual methods:

- `virtual void sc_module::end_of_construction()`

This method is called just before the end of elaboration. It gives a module an opportunity to do special port bindings.

- `virtual void sc_module::end_of_elaboration()`

This method is called after elaboration, but before simulation starts. You cannot do any port binding at this point, but you can do other initialization.

- `virtual void sc_module::end_of_simulation()`

This method is called when the simulation ends, which occurs before the destructor is called, so the design hierarchy is still intact at the end of simulation.

## Currently Missing Pieces

- No connection to HDL memory (no equivalent to `tbvSignalArrayHdlT`).
- No force/release of signals in HDL.
- No delayed signal assignments.

## Arbitrary-Width Signals (tbvSignalT)

TestBuilder 1.3 defines the `tbvSignal2StateT` and `tbvSignal4StateT` classes to implement arbitrary-width signals. TestBuilder-SC does not define an equivalent set of classes, but instead uses the SystemC `sc_signal` channel for this type of class.

An `sc_signal` can be associated with any data type, so you can use the following mappings between the TestBuilder 1.3 signal classes and the SystemC signal channels:

**Table 1-1**

TestBuilder 1.3	TestBuilder-SC
<code>tbvSignalT (tbvSignal2StateT)</code>	<code>sc_signal&lt;int&gt;</code> <code>sc_signal&lt;uint&gt;</code> <code>sc_signal&lt;sc_int&lt;N&gt; &gt;</code> <code>sc_signal&lt;sc_uint&lt;N&gt; &gt;</code> <code>sc_signal&lt;sc_bigint&lt;N&gt; &gt;</code> <code>sc_signal&lt;sc_biguint&lt;N&gt; &gt;</code> <code>sc_signal&lt;sc_bit&gt;</code> <code>sc_signal&lt;sc_bv&gt;</code>
<code>tbvSignal4StateT</code>	<code>sc_signal&lt;sc_logic&gt;</code> <code>sc_signal&lt;sc_lv&lt;N&gt; &gt;</code> <code>sc_signal_rv&lt;sc_logic&gt;</code> <code>sc_signal_rv&lt;sc_lv&lt;N&gt; &gt;</code>
<code>tbvSignalArrayT</code>	<code>sc_signal&lt;int&gt;[N], etc.</code>
<code>tbvSignalArray4StateT</code>	<code>sc_signal&lt;sc_logic&gt;[N], etc.</code>

## User Data Types (tbvSmartRecordT)

In TestBuilder 1.3, the `tbvSmartDataT` base class includes a number of verification capabilities, such as randomization, value callbacks, and transaction recording.

In TestBuilder-SC, those same capabilities are available by using the TestBuilder-SC data extensions interface. You create a normal C/C++ user-defined type, then create a `tb_extensions<>` version of the type. You can randomize an extended object, apply value change callbacks to it, and use it for attributes in transaction recording.

The following example shows a TestBuilder 1.3 smart record:

```
// This example needs to be run through tbvWizard to generate code:  
class myrecordT : public tbvSmartRecordT {
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
TBV_SMART_FIELDS_BEGIN
    tbvSmartIntT i1;
    tbvSmartUnsignedT i2;
TBV_SMART_FIELDS_END
};

// Using the object
void use_data() {
    myrecordT *data_p = new myrecordT;
    data_p->randomize();
    tbvOut << "data = " << *data_p << endl;
}
```

The next example shows a corresponding TestBuilder-SC extended data type:

```
// Create a normal struct:
struct myrecordT {
    int i1;
    unsigned i2;
};

// Run the struct through tb_wizard_ext to automatically generate
// the following code:
template<>
class tb_extensions<myrecordT> : public tb_extensions_base<myrecordT> {
public:
    tb_extensions<int> i1;
    tb_extensions<unsigned> i2;
    TB_EXTENSIONS_CTOR(myrecordT) {
        TB_FIELD(i1);
        TB_FIELD(i2);
    }
};

// Using the object
void use_data() {
    tb_smart_ptr<myrecordT> data_p;
    data_p->next();
    tb_out << "data = " << *data_p << endl;
}
```

## Concurrency (tbvTaskT and spawn)

In TestBuilder-SC, there is no direct equivalent to a `tbvTaskT` object type. The convenience operations done in TestBuilder 1.3, specifically automatic transaction recording, must be done manually. The TestBuilder-SC API for spawning and running tasks is much more flexible in TestBuilder-SC, however. You can spawn or run any arbitrary function or method using TestBuilder-SC.

### Running or Spawning a Task

To spawn a task in TestBuilder 1.3, you create a `tbvSmartDataT` argument and call the `spawn()` or `run()` method of the task passing a pointer to the argument, as shown here:

```
// Definition of a task
class mytask : public tbvTaskTypeSafeT<myrecordT> {
    ...
    virtual void body (myrecordT*);
};
class mytvm : public tbvTvmT {
    ...

    mytask t1;
    ...
};

void tbvMain() {
    mytvm *tvm = tbvTvmT::getTvmByInstanceNameP("tvm1");
    myrecordT data;

    mytvm->t1.run(&data); // Blocking call
    tbvThreadT t;
    t = mytvm->t1.spawn(&data); // Non-blocking call
    ...
    t.wait(); // Wait for the spawned task.
}
```

In TestBuilder-SC, the signature of the method to call is arbitrary. It does not have to be a method named `body()` that takes exactly one argument. There is no task base class in TestBuilder-SC. Instead, you need to create an interface class that can be attached to an `sc_module` using port binding, as shown here:

```
// Definition of task a interface
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
class mytask_if : virtual public sc_interface {
    public:
        virtual void mytask(myrecordT) = 0;
};
class mytvm : public mytask_if {
    public:
        void mytask(myrecordT*);
};

void test::tbvMain() { // A test thread
    mytvm *tvm = (mytvm*)sim_context()->find_object("top.tvm1");
    myrecordT data;
    tvml->mytask(&data); // A blocking call

    tb_thread t;
    t = tb_spawn("mythread", tvml, &mytvm::mytask, &data);

    tb_wait(t);
}
```

## Monitor Threads

In TestBuilder 1.3, you typically start a monitor thread as follows:

```
class mytvmT : public tbvTvmT {
    public:
        monitor_task monitor;
        ...
        void setup() {
            monitor.spawn()
        }
};
```

where the monitor body function is implemented like this:

```
void monitor_task::body(tbvSmartDataT*) {
    while(1) {
        // Do monitor tasks
    }
}
```

In TestBuilder-SC, you can do a soperation by using the SC\_THREAD or TB\_THREAD macros to start a monitor thread.

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
class mytvmT : public sc_module {
    void monitor();
    ..
    SC_CTOR(mytvmT) {
        TB_THREAD(monitor);
    }
};
void mytvmT::monitor() {
    while(1) {
        // Do monitor tasks
    }
}
```

### Example TestBuilder-SC Class to Mimic tbvTaskT

You can mimic the `tbvTaskT` class by using the following class definition.

```
template<class T>
class tb_task_t {
protected:
    sc_module *parent_tvm;
    tb_tr_stream fiber;
    tb_tr_generator<T,T> gen;
    tb_mutex mutex;
    string name;

public:
    virtual void body(T*) = 0;
    void run(T* datap) { body_wrapper(datap); }
    void spawn(T* datap)
        { tb_spawn(name, this, &tb_task_t<T>::body_wrapper, datap); }

public:
    void body_wrapper(T* datap) {
        mutex.lock();
        datap->handle = gen.begin_transaction(name, *datap);
        body(datap);
        gen.end_transaction(*datap, datap->handle);
        mutex.unlock();
    }
};
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

The task concept use model in TestBuilder 1.3 is slightly different than what is generally recommended for TestBuilder-SC. In TestBuilder-SC, it is best to use an interface class that defines tasks. You can still use the TestBuilder 1.3-style task within the TestBuilder-SC methodology, as shown in this sample TestBuilder-SC task interface:

```
// Interface class
class mybus_task_if() : virtual public sc_interface {
    virtual void do_mytask(dataT) = 0;
};

// TestBuilder 1.3-style task using the task wrapper
class mytask_t : public tb_task_t<dataT> {
public:
    mytask_t(sc_module *tvm) : tb_task_t<dataT>(tvm, "mytask_t", "data") {}
    virtual void body(dataT* data) {
        ... // Do functionality
    }
};

// TestBuilder-SC-style transactor using an interface class and task
class mytvm : public mytvm_module_if, mybus_task_if {
private:
    mytask_t mytask;
public:
    virtual void do_mytask(dataT* data) { mytask.run(data); }

    mytvm(sc_module_name nm) : mytvm_module_if(nm), mytask(this) {}
};
```

## Spawning Arbitrary Methods and Functions

In TestBuilder 1.3, you can use the `tbvSpawnNArgMethodT<>` and `tbvSpawnNArgFunctionT` classes to spawn functions and methods. For TestBuilder 1.3, this works as long as  $N$  is between 0 and 2, and the function or method returns void. In TestBuilder-SC, you can use the `tb_spawn` function, which spawns any function or method. The function or method can take up to 10 arguments and can have a return value.

The following is a TestBuilder 1.3 example of spawning a function:

```
void myfunction(int arg) {
    ...
}

void tbvMain() {
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
tbvSpawnOneArgMethodT<int>::spawn(myfunction, 10);
tbvThreadT::waitDescendents();
}
```

The next example shows corresponding TestBuilder-SC example that use `tb_spawn`:

```
void myfunction(int arg) {
    ...
}
int myfunction_ret(int arg) {
    ...
}
int sc_main(int argc, char** argv) {
    int retval;
    tb_spawn("t1", myfunction, 10);
    tb_spawn("t2", myfunction_ret, &retval, 20);
    tb_thread::wait_descendents();
}
```

## Concurrency Objects

TestBuilder-SC uses the SystemC channel concept to define concurrency objects such as semaphores. Following is a list of TestBuilder 1.3 concurrency classes, how they map to TestBuilder-SC concurrency channels, and the channel interfaces in TestBuilder-SC.

**Table 1-2 Concurrency Object Mapping**

---

TestBuilder 1.3	TestBuilder-SC	Channel Interface
<code>tbvSemT</code>	<code>tb_semaphore</code> <code>sc_semaphore</code>	<code>tb_semaphore_if</code> <code>sc_semaphore_if</code>
<code>tbvMutexT</code>	<code>tb_mutex</code>	<code>tb_mutex_if</code>
<code>tbvBarrierT</code>	<code>tb_barrier</code>	<code>tb_barrier_if</code>
<code>tbvMailboxT</code>	<code>tb_sync_list</code>	<code>tb_list_if</code>
<code>tbvThreadT</code>	<code>tb_thread</code>	
<code>tbvSignalHdlT</code>	<code>sc_signal&lt;T&gt;</code>	<code>sc_signal_in_if</code>
<code>tbvEventExprT</code>	<code>tb_sync_expr</code>	

---

## Transactors (tbvTvmT)

The concept of a transactor is the same in TestBuilder-SC as it was in TestBuilder 1.3. The primary difference is in the task interface of the transactor, which primarily affects master transactors.

In both TestBuilder 1.3 and TestBuilder-SC, a master transactor consists of three sections:

- The interface section, which defines how the transactor connects to the device under test.
- The task section, which defines how a test requests transaction activity.
- The local transactor component information.

The following example shows a TestBuilder 1.3 master transactor definition that will be generated by using tbvWizard macros:

```
class myTvmT : public tbvTvmT {
    // Device interface
    TBV_HDL_SIGNALS_BEGIN
        tbvSignalHdlT clk("clk", tbvEnumsT::READ_ONLY);
        tbvSignalHdlT data("data", tbvEnumsT::WRITE_ONLY);
    TBV_HDL_SIGNALS_END
    // Task interface section
    TBV_TASKS_BEGIN
        doWriteT do_write;
    TBV_TASKS_END
};
```

This example shows a corresponding TestBuilder-SC master transactor definition:

```
// Device interface class
struct myTvm_duv_if : public sc_module {
    sc_in<bool> clk;
    sc_out<sc_uint<16> > data;
    myTvm_duv_if(sc_module_name nm) : sc_module(nm), clk("clk"), data("data") {}
};

// The task interface; refer to the Concurrency section to see how tasks are
// mapped between TestBuilder 1.3 and TestBuilder-SC.
struct myTvm_task_if : public virtual sc_interface {
    virtual void do_write(sc_uint<16> addr, sc_uint<16> data) = 0;
};
```

# TestBuilder-to-TestBuilder-SC Interoperability Guide

## Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
// Transactor declaration
struct myTvmT : public myTvm_duv_if, public myTvm_task_if {
    void do_write(sc_uint<16> addr, sc_uint<16> data);
    myTvmT(sc_module_name nm) : myTvm_duv_if(nm) {}
};
```

## Randomization and Constraint Solving

Randomization and constraint solving are essentially the same between TestBuilder 1.3 and TestBuilder-SC. Some functionality has been added to TestBuilder-SC, and some changes have been made to the way randomization and constraint solving are done.

### Differences in Randomization

The following lists the primary differences in randomization. Some of the differences are minor and some are significant.

#### ■ Minor differences

- ❑ Name changes to conform with SystemC naming conventions:

```
tbvRandomT          tb_random
tbvBagT             tb_bag
tbvSmartDataT::randomize()  tb_smart_ptr<>::next()
```

- ❑ A global seed allows repeatability with a single simple statement (`tb_random::set_global_seed()`).
- ❑ Randomization is done using extended types (`tb_extension_if` through the `tb_smart_ptr<>` class) instead of in a special class (`tbvSmartDataT` for TestBuilder 1.3).

#### ■ Major differences

- ❑ Seed names are generated for each thread, so that recreation of seeds using seed names will be independent of thread ordering.
- ❑ Distribution ranges are available using `tb_bag<pair<T,T> >`.
- ❑ Enumerated data types that have been extended can be randomized. For example:

```
enum myenum { a, b, c };
void doit() {
```

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
tb_smart_ptr<myenum> e;  
e->next();  
}
```

- ❑ Value generation classes have been removed. There is no direct replacement for classes such as `tbvIntGeneratorT`.

## Differences in Constraint Solving

- A minor difference is that constraint equations are set up by using `tb_smart_ptr<>` objects instead of `tbvSmartDataT` objects.
- Major differences
  - ❑ Constraints are class-based, which means that all constraints must be created at elaboration time.  
**Note:** It is possible to dynamically modify values within a constraint, but the constraint equation is static. In TestBuilder 1.3, constraints are completely dynamic.
  - ❑ You can use distribution constraints in complex constraints.

## Transaction Recording

Transaction recording has changed quite a bit between TestBuilder 1.3 and TestBuilder-SC. It is possible to get precisely the same functionality, but in some cases you have to manage more things. Following is a list of the major and minor differences between TestBuilder-SC and TestBuilder 1.3 transaction recording.

- Minor Differences:
  - ❑ Name changes in the components of a transaction.

<code>tbvFiberT</code>	<code>tb_tr_stream</code> with <code>tb_tr_generator&lt;T,T&gt;</code>
<code>tbvTransactionHandleT</code>	<code>tb_tr_handle</code>

- ❑ In TestBuilder-SC, relationships are set between two transaction handles, instead of between a transaction handle and a fiber as in TestBuilder 1.3.
- ❑ In TestBuilder-SC, transaction attributes are extended datatypes (`tb_extensions_if<>`) instead of `tbvSmartData` objects. This is the same as for randomization.

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

#### ■ Major differences

- ❑ There is no automatic transaction recording in TestBuilder-SC. Refer to [“Concurrency \(tbvTaskT and spawn\)”](#) on page 14 for information about how a task class similar to the TestBuilder 1.3 task class can be used to get this type of functionality, if desired.
- ❑ In TestBuilder-SC, there is no mutex in the fiber equivalent (`tb_tr_stream`). Transactions are allowed to overlap in TestBuilder-SC.
- ❑ In TestBuilder-SC, parent-child transaction relationships must be made explicitly. There is no longer an implicit type of parent-child relationship.
- ❑ You can write to multiple databases simultaneously in TestBuilder-SC, for example, to both text and SST2 databases.
- ❑ TestBuilder-SC includes a callback mechanism in the transaction recording library. This allows applications to setup up begin/end transaction callbacks, among others.

The following is an example of TestBuilder 1.3 transaction code.

```
void do_tx() {
    tbvSmartIntT data;
    tbvFiberT fiber("my_tx_fiber");
    tbvTransactionHandleT handles[5];

    for(int i=0; i<5; ++i) {
        handles[i] = fiber.beginTransactionH("tx_type1"); // Implicit mutex lock
        data = i;
        fiber.recordAttribute(data, "data");
        tbvWait(100);
        fiber.endTransaction(handles[i]);
    }
    for(int i=0; i<5; ++i) {
        fiber.beginTransactionH("tx_type2"); // Implicit mutex lock
        fiber.setPredecessor(handles[i]);
        data = 5-i;
        fiber.recordAttribute(data, "data");
        tbvWait(100);
        fiber.endTransaction();
    }
}
```

The next example shows the exactly equivalent TestBuilder-SC code. The mutex is optional, but it is needed to be equivalent to 1.3.

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

```
void do_tx() {
    tb_tr_stream fiber("my_tx_fiber");
    tb_tr_generator<int> gen_tx1("tx_type1", fiber, "data");
    tb_tr_generator<int> gen_tx2("tx_type2", fiber, "data");
    tb_tr_handle handles[5];

    tb_mutex mutex("m"); // Need a mutex if do_tx can be spawned

    // Note: you can use record_attribute, but it is faster if you use the begin
    // and/or end attributes in the generator.
    for(int i=0; i<5; ++i) {
        mutex.lock();
        handles[i] = gen_tx1.begin_transaction(i);
        tb_wait(100, SC_NS);
        gen_tx1.end_transaction(handles[i]);
        mutex.unlock();
    }
    for(int i=0; i<5; ++i) {
        tb_tr_handle current;
        mutex.lock();
        current = gen_tx2.begin_transaction(5-i);
        current.add_relation("predecessor", handles[i]);
        tb_wait(100, SC_NS);
        gen_tx1.end_transaction(current);
        mutex.unlock();
    }
}
```

## Verification Data Structures

The data structures in TestBuilder-SC are generally the same as in TestBuilder 1.3. However, some data structures have been removed because STL provides exact versions. Following is a list of the verification structures in TestBuilder 1.3 and their new names in TestBuilder-SC.

# TestBuilder-to-TestBuilder-SC Interoperability Guide

## Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

**Table 1-3 Data Structures**

TestBuilder 1.3	TestBuilder-SC
<code>tbvListT</code>	Removed  <b>Note:</b> <code>tb_list</code> is a special channel in TestBuilder-SC.
<code>tbvPriorityListT</code>	Removed
<code>tbvStackT</code>	Removed
<code>tbvQueueT</code>	Removed
<code>tbvPriorityQueueT</code>	Removed
<code>tbvSparseArrayT</code>	<code>tb_sparse_array</code>
<code>tbvAssociativeArrayT</code>	<code>tb_associative_array</code>
<code>tbvSmartQueueT</code>	<code>tb_smart_queue</code>
<code>tbvBagT</code>	<code>tb_bag</code>
<code>tbvSafeHandleT</code>	<code>tb_safe_handle</code>

---

## TestBuilder-1.3-to-TestBuilder-SC Mapping

This section lists the TestBuilder 1.3 classes, functions, and methods, and their equivalents in TestBuilder-SC, by category:

- [HDL](#)
- [Entry Points](#)
- [HDL Mapped Signals](#)
- [Signals \(includes HDL mapped signals\)](#)
- [Exception Handling](#)
- [Time-Oriented Methods](#)
- [Thread Wakeups \(waiting\)](#)

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

**Table 1-4 HDL**

TestBuilder 1.3	HDL	TestBuilder-SC
\$tbv_tvm_connect	Verilog	\$sc_cosim_init (schdl) (*const integer foreign = "SystemC"; *) (ncsc)
tbv_tvm_connect	VHDL	attribute foreign of tb:architecture is "FMI sclib:scmod" (schdl) attribute foreign of tb:architecture is "SystemC" (ncsc)
\$tbv_main	Verilog	N/A Use a model connection with no ports.
tbv_main	VHDL	N/A Use a model connection.

**Table 1-5 Entry Points**

TestBuilder 1.3	TestBuilder-SC
tbvInit()	sc_main() Not implemented yet.
tbvTvmTypes[] tbvTvmT::registerTvm()	SC_EXPORT_MODULE()
tbvMain()	N/A Use a model connection with no ports.
tbvTvmT::tbvTvmT()	sc_module::sc_module()
N/A	void sc_module::end_of_construction()
N/A	void sc_module::end_of_elaboration()
N/A	void sc_module::end_of_simulation()

**Table 1-6 HDL Mapped Signals**

TestBuilder 1.3	TestBuilder-SC
tbvSignalHdlT::tbvSignalHdlT()	sc_signal<T>::control_foreign_signal sc_signal<T>::observe_foreign_signal
tbvSignalHdlT::getHdlNameP()	sc_signal<T>::name()

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

**Table 1-6 HDL Mapped Signals**

TestBuilder 1.3	TestBuilder-SC
<code>tbvSignalHdlT::isForced()</code>	N/A
<code>tbvSignalArrayHdlT::tbvSignalArrayHdlT</code>	N/A

**Table 1-7 Signals (includes HDL mapped signals)**

TestBuilder 1.3	TestBuilder-SC
<code>tbvSignalT(msb, lsb)</code>	<code>sc_signal&lt;sc_logic&lt;size&gt; &gt;</code>
<code>tbvSignal2StateT(msb, lsb)</code>	<code>sc_signal&lt;sc_uint&lt;size&gt; &gt;</code> Other types are supported for two-state logic.
<code>tbvSignalT::msb()</code>	N/A
<code>tbvSignalT::lsb()</code>	N/A
<code>tbvSignalT::size()</code>	<code>sc_logic&lt;&gt;::size()</code>
<code>tbvSignalT::getValue()</code>	<code>sc_signal&lt;&gt;::read()</code>

**Table 1-8 Exception Handling**

TestBuilder 1.3	TestBuilder-SC
<code>tbvExceptionT</code>	<code>tb_exception</code>

**Table 1-9 Time-Oriented Methods**

TestBuilder 1.3	TestBuilder-SC
<code>tbvUInt64T tbvScaleSimTime ()</code>	N/A
<code>tbvUInt64T tbvGetInt64Time ()</code>	<code>sc_time&amp; sc_time_stamp()</code>
<code>double tbvGetRealTime()</code>	<code>sc_time&amp; sc_time_stamp()</code>
<code>const char* tbvGetSimTimeUnitP()</code>	<code>sc_time_unit sc_get_default_time_unit()</code>
<code>tbvUInt64T tbvconvertTime()</code>	N/A

## TestBuilder-to-TestBuilder-SC Interoperability Guide

### Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---

**Table 1-10 Thread Wakeups (waiting)**

---

<b>TestBuilder 1.3</b>	<b>TestBuilder-SC</b>
<code>tbvWaitCycle()</code>	<code>wait(sc_event&amp;)</code>
<code>tbvWaitEvent()</code>	No equivalent; all waits have a delta cycle.
<code>tbvWait()</code>	<code>wait(sc_time&amp;)</code>

---

**TestBuilder-to-TestBuilder-SC Interoperability Guide**  
Converting Testbenches From TestBuilder 1.3 to TestBuilder-SC

---