Using Portable Stimulus to Verify an LTE Base-Station Switch

Adnan Hamid (adnan@brekersystems.com)
Breker Verification Systems, 1879 Lundy Avenue, Ste 126, San Jose, CA 95131
www.brekersystems.com

Objective

Generate software driven verification (SDV) test cases in "C" for an **LTE Base Station Switch**

- □ **Complex** design supporting many possible use models
- □ Particular focus on **dependency management** across tasks and resources
- □ Create **single verification model** that can scale from simple to complex tests
- □ Crate **portable stimulus** model that can generate tests for simulation, emulation and post-silicon
- □ Crate **self-checking** test cases that are easy to **debug**
- □ Prove **coverage closure** on system use cases

LTE Base Station Switch Design

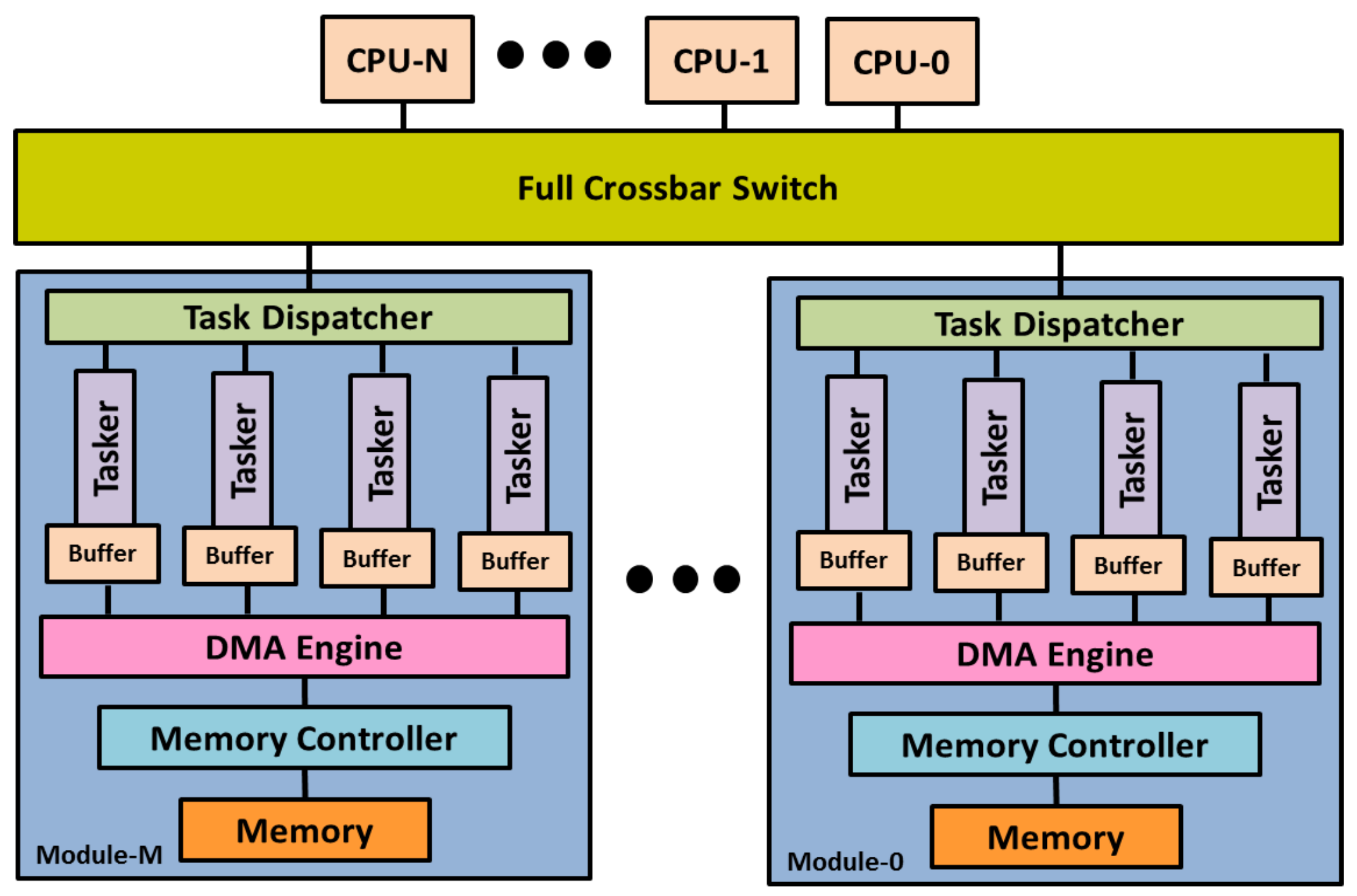


Figure 1: LTE Base Station Switch

- □ **CPUs** configure Task Dispatcher with 1000's of tasks
- □ **Task Dispatcher** schedules tasks on **Taskers** as resources become available
- □ Each task has **many possible formats** with different paths through system
- □ Each task many have **complex dependencies** on other tasks and resources
- □ Task Dispatcher must **manage dependencies** across tasks and resources
- □ Task Dispatcher must track **progress and completion** of tasks to free resources
- □ One Task Dispatcher and multiple Taskers in each **cluster**
- □ Multiple clusters in **full chip**

Verification Requirements

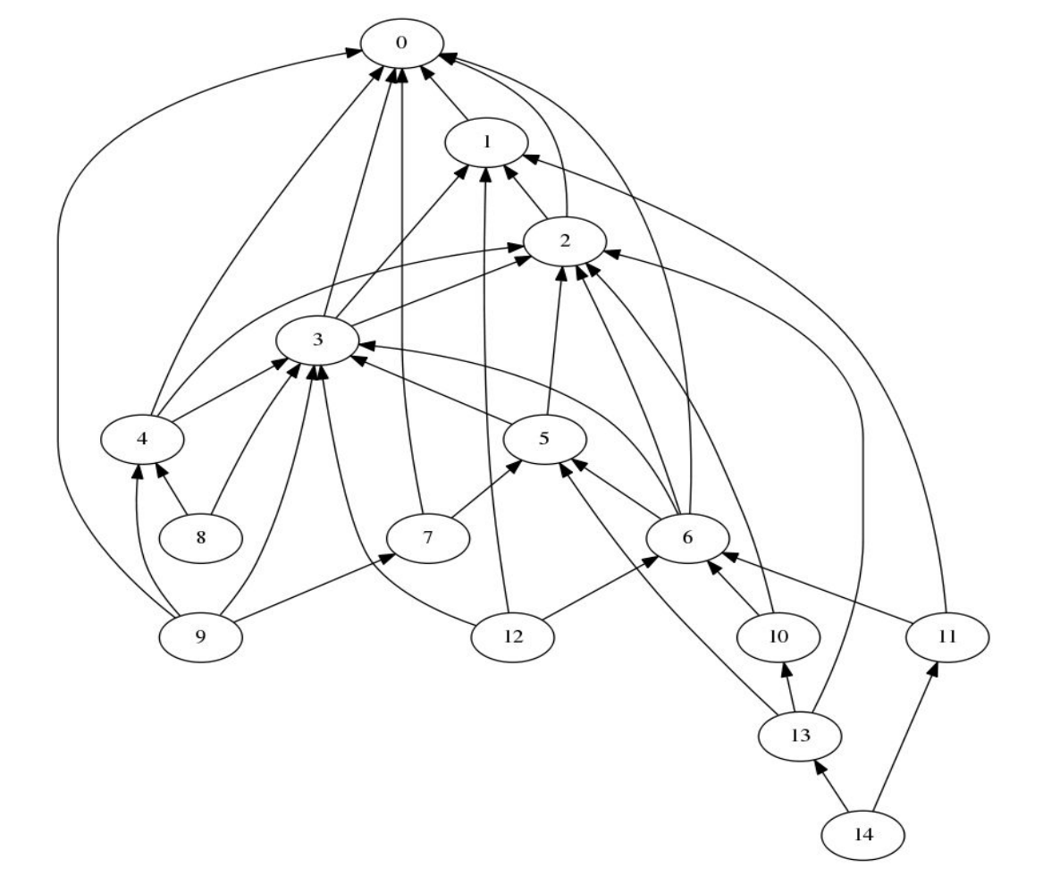


Figure 2: Task dependency graph generated from TrekSoC

Complexity Requirements

- □ Individual tasks have **many possible formats** and many possible **resource dependencies**
- □ Need to test **complex dependency scenarios** across tasks, task dispatchers and taskers (Figure 2).
- □ Need to generate complex **data structures** in memory
- □ Need to **manage memory** across multiple regions
- □ Task Dispatchers have **complex configurations** with many variations

Scalability Requirements

- □ Scaling of generated tests from **simple to com**plex
- □ Scaling of system under test from **sub-system** to **full-chip**

Portability Requirements

- □ Single source of tests must run on **simulation**, **emulation**, **post-silicon**
- □ Support **different test case mechanics** for each platform
  - e.g. back-door byte-by-byte memory checking in simulation, but front-door checksum-based checking in post-silicon

Checking and Debug Requirements

- □ **Automatic generation** of self-checking tests
- □ Track progress of each task through **life-cycle**
  - All tasks completed
  - Tasks completed in dependency order
- □ **Predict and check** results in memories and registers
- □ **Interactive debug** to identify and root-cause errors
  - must work in simulation / emulation / post-silicon

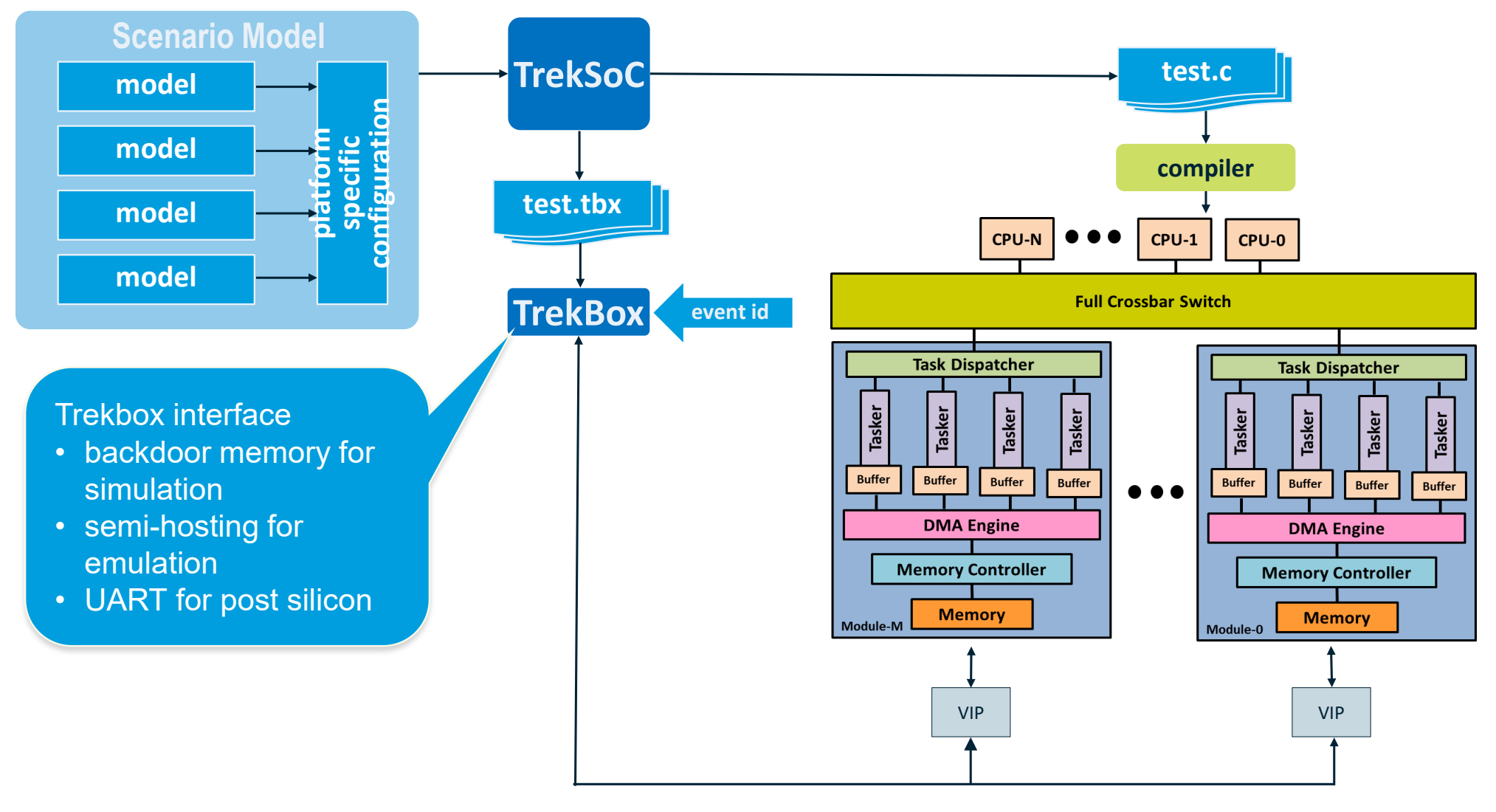
Coverage Requirements

- □ Prove **coverage closure** on all use cases

Method

Test Generation Tool

- □ Use **Breker TrekSoC** product as tool of choice for scalable portable stimulus (Figure 3)
- □ **algorithmic graph-based scenario models** to capture verification space



Trekbox interface
- backdoor memory for simulation
- semi-hosting for emulation
- UART for post silicon

Figure 3: TrekSoC Software Driven Verification Flow

Portable Stimulus

- □ Common **scenario model** used at all stages of verification (Figure 4)
- □ TrekSoC tool **configured** for each environment
  - complexity of test to generate
  - number of CPUs, Task Dispatchers and Taskers
  - Number and size of memory regions
  - Communication mechanism between test and TrekBox

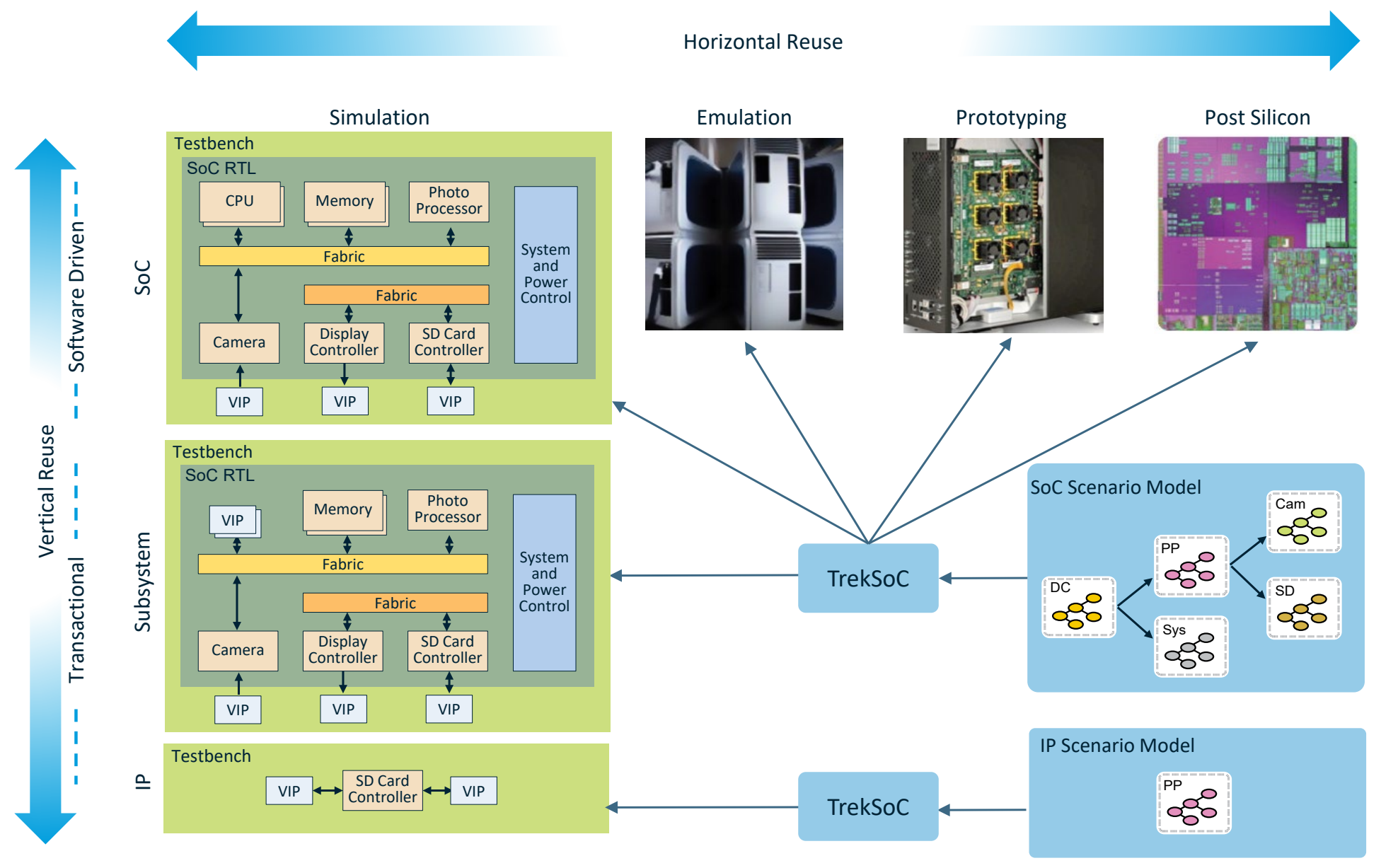
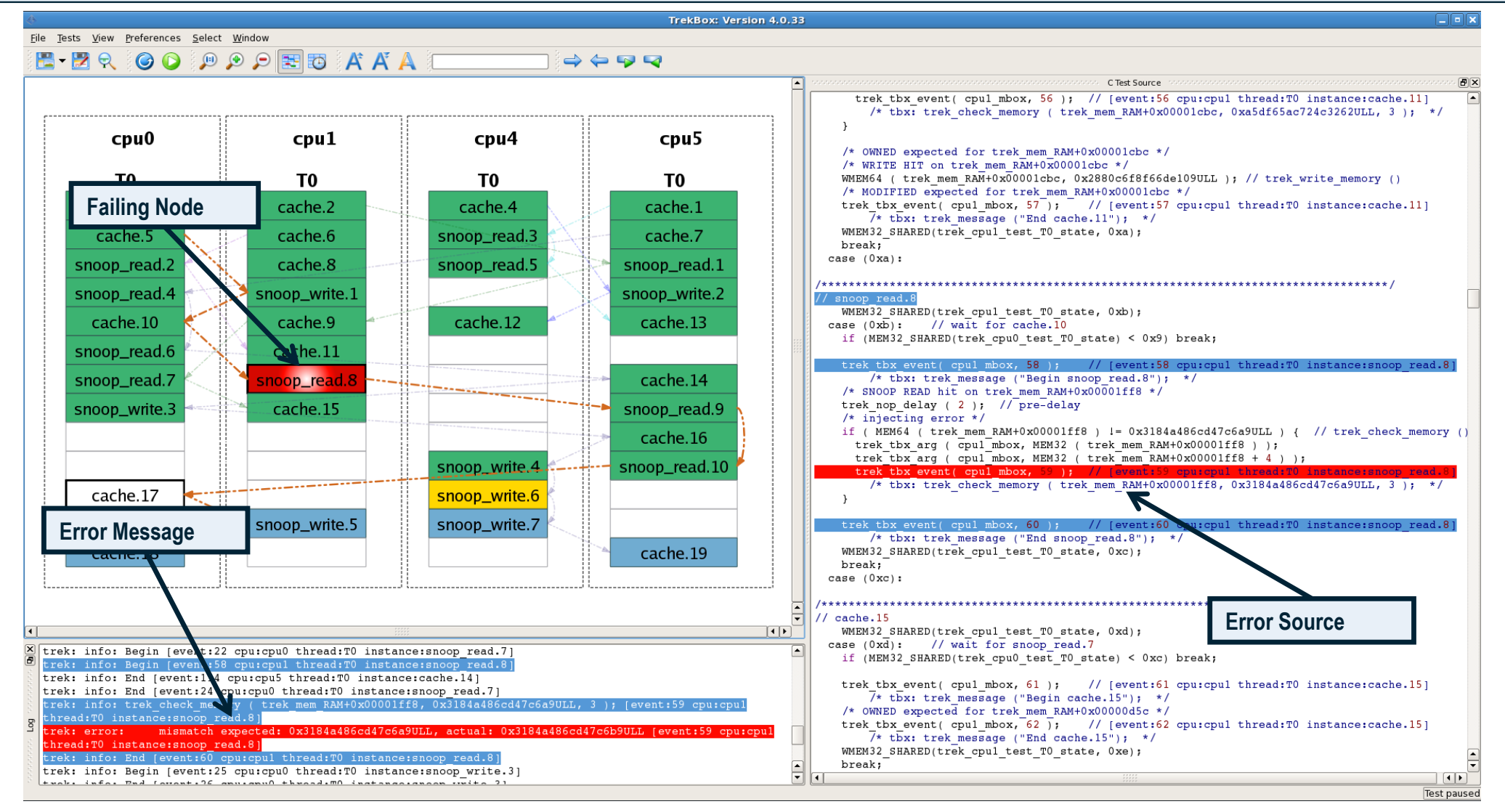


Figure 4: TrekSoC Target Specific Test Case Synthesis

Checking and Debug

- □ TrekSoC generated **self-checking test cases**
- □ Interactive Test Map view to **identify** and **root-cause** errors (Figure 5)



```
trek: info: Begin [event:22 cpu:cpu0 thread:T0 instance:snoop_read.7]
trek: info: Begin [event:58 cpu:cpu1 thread:T0 instance:snoop_read.8]
trek: info: End [event:14 cpu:cpu5 thread:T0 instance:cache.14]
trek: info: End [event:24 cpu:cpu0 thread:T0 instance:snoop_read.7]
trek: info: trek_check_memory ( trek_mem_RAM+0x00001ff8, 0x3184a486cd47c6a9ULL, 3 ); [event:59 cpu:cpu1
thread:T0 instance:snoop_read.8]
trek: error:     mismatch expected: 0x3184a486cd47c6a9ULL, actual: 0x3184a486cd47c6b9ULL [event:59 cpu:cpu1
thread:T0 instance:snoop_read.8]
trek: info: End [event:60 cpu:cpu1 thread:T0 instance:snoop_read.8]
trek: info: Begin [event:25 cpu:cpu0 thread:T0 instance:snoop_write.3]
```

```
      trek_tbx_event( cpu1_mbox, 56 );  // [event:56 cpu:cpu1 thread:T0 instance:cache.11]
        /* tbx: trek_check_memory ( trek_mem_RAM+0x00001cbc, 0xa5df65ac724c3262ULL, 3 );  */
    }

    /* OWNED expected for trek_mem_RAM+0x00001cbc */
    /* WRITE HIT on trek_mem_RAM+0x00001cbc */
    WMEM64 ( trek_mem_RAM+0x00001cbc, 0x2880c6f8f66de109ULL ); // trek_write_memory ()
    /* MODIFIED expected for trek_mem_RAM+0x00001cbc */
    trek_tbx_event( cpu1_mbox, 57 );    // [event:57 cpu:cpu1 thread:T0 instance:cache.11]
      /* tbx: trek_message ("End cache.11");  */
    WMEM32_SHARED(trek_cpu1_test_T0_state, 0xa);
    break;
  case (0xa):

/****************************************************************************/
// snoop_read.8
    WMEM32_SHARED(trek_cpu1_test_T0_state, 0xb);
  case (0xb):    // wait for cache.10
    if (MEM32_SHARED(trek_cpu0_test_T0_state) < 0x9) break;

    trek_tbx_event( cpu1_mbox, 58 );    // [event:58 cpu:cpu1 thread:T0 instance:snoop_read.8]
      /* tbx: trek_message ("Begin snoop_read.8");  */
    /* SNOOP READ hit on trek_mem_RAM+0x00001ff8 */
    trek_nop_delay ( 2 );  // pre-delay
    /* injecting error */
    if ( MEM64 ( trek_mem_RAM+0x00001ff8 ) != 0x3184a486cd47c6a9ULL ) {  // trek_check_memory ()
      trek_tbx_arg ( cpu1_mbox, MEM32 ( trek_mem_RAM+0x00001ff8 ) );
      trek_tbx_arg ( cpu1_mbox, MEM32 ( trek_mem_RAM+0x00001ff8 + 4 ) );
      trek_tbx_event( cpu1_mbox, 59 );  // [event:59 cpu:cpu1 thread:T0 instance:snoop_read.8]
        /* tbx: trek_check_memory ( trek_mem_RAM+0x00001ff8, 0x3184a486cd47c6a9ULL, 3 );  */
    }

    trek_tbx_event( cpu1_mbox, 60 );    // [event:60 cpu:cpu1 thread:T0 instance:snoop_read.8]
      /* tbx: trek_message ("End snoop_read.8");  */
    WMEM32_SHARED(trek_cpu1_test_T0_state, 0xc);
    break;
  case (0xc):

/****************************************************************************/
// cache.15
    WMEM32_SHARED(trek_cpu1_test_T0_state, 0xd);
  case (0xd):    // wait for snoop_read.7
    if (MEM32_SHARED(trek_cpu0_test_T0_state) < 0xc) break;

    trek_tbx_event( cpu1_mbox, 61 );    // [event:61 cpu:cpu1 thread:T0 instance:cache.15]
      /* tbx: trek_message ("Begin cache.15");  */
    /* OWNED expected for trek_mem_RAM+0x00000d5c */
    trek_tbx_event( cpu1_mbox, 62 );    // [event:62 cpu:cpu1 thread:T0 instance:cache.15]
      /* tbx: trek_message ("End cache.15");  */
    WMEM32_SHARED(trek_cpu1_test_T0_state, 0xe);
    break;
```

Figure 5: TrekSoC Test Map Debug View

Results

Scenario Model Construction

- □ Algorithmic graph-based portable stimulus scenario model "**blind-built**" from spec before simulation available
- □ First generated test running **within hours** of initial bring-up!
  - graph-based models allow simple creation of complex test scenarios
  - graph-based models allow incremental build out of scenario model
  - test cases correct by construction, even with 1000's of interacting tasks and complex Task Dispatcher programming cases

Test Complexity Scaling

- □ Seamless scaling of test complexity from **single task to 1000's of inter-dependent tasks**, and from **single** CPU/Task Dispatcher/Tasker to **multiple** CPUs/Task Dispatchers/Taskers across multiple clusters

Portable Stimulus

- □ **Target specific** self-checking tests generated for simulation and emulation with different test case mechanics
  - Easy to replay failing emulator cases on simulation
  - **Waiting** for silicon to generate post-silicon tests

Checking and Debug

- □ **Task dependency graph** generated from tool (Figure 2) used to help follow test case intent
- □ **Design errors** rapidly identification and root-caused, with **common debug view** across all platforms

Coverage Closure

- □ **Path coverage** and **automatic coverage closure** of graph-based models provide metric for use case coverage
  - cannot be measured with RTL functional coverage

Realized Benefits

Bugs Found

- □ **Hardware, software,** and **environment bugs** found and fixed at each level of application

Effort & Cost Savings

- □ **Automated generation** of 1000's of tests for emulation/Si
  - Could not achieve this **coverage** with manually written tests

For more information on this verification process, see www.brekersystems.com