Improving Verification Productivity

with the

Dynamic Load and Reseed Methodology

Marat Teplitsky
HVC 2012
Agenda

• Advanced Verification Challenges
• Supporting Technology
• Use Modes
• Dynamic Load and Reseed Methodology
  – Saving the State and Reseeding
  – Dynamic Loading of Files
  – Dynamic Load and Reseed using UVM Test Phases
• Summary
The Productivity Gap

Today

- Redundancy
- Hard to get coverage

Regression

- Run entire simulation anew
- Debug requires special configuration

Debug/Test Development

Most interesting for tests

Similar for every test

Our Methodology

- Reuse of simulation phases
- Easy to focus on interesting/hard to cover areas

Regression

- Much shorter simulation
- Much faster simulation

Debug/Test Development

Similar for every test

Today Methodology:

1. Reset
2. Program
3. Link Init
4. Send Traffic
5. Finish

Our Methodology:

1. Reset
2. Program
3. Link Init
4. Send Traffic
5. Finish
Our Methodology Addresses Key Productivity Concerns

- Increases overall verification productivity by 30 – 50%
- In regression throughput
- In simulation time to debug point
Analysis of Today’s Regression

Env

Initialization sequence (All tests, perhaps built into env)

Tests

Test file A (run time configuration)

Test A1
Test A2
Test A3
Test A4

Test file B (run time configuration)

Test B1

Each interesting scenario is a path through design functionality

Each interesting scenario is the combination of a Test file and Seed. Many seeds of a single run can increase coverage dramatically
Analysis of Today’s Regression

Initialization is typically similar/identical for all tests

Initialization can consume many simulation cycles

Many scenarios might also share common test functionality

Env

Initialization sequence
(All tests, perhaps built into env)

Tests

Test file A
(run time configuration)

Test file B
(run time configuration)

Test A1
random seed

Test A2
random seed

Test A3
random seed

Test A4
random seed

Test B1
random seed

Test B1
random seed

Test B1
random seed

Test B1
random seed

Analysis of Today’s Regression

Initialization is typically similar/identical for all tests

Initialization can consume many simulation cycles

Many scenarios might also share common test functionality

Env

Initialization sequence
(All tests, perhaps built into env)

Tests

Test file A
(run time configuration)

Test file B
(run time configuration)

Test A1
random seed

Test A2
random seed

Test A3
random seed

Test A4
random seed

Test B1
random seed

Test B1
random seed

Test B1
random seed

Test B1
random seed
Analysis of Today’s Regression

Env

Initialization sequence
(All tests, perhaps built into env)

It would be useful to start all tests after initialization is complete

Tests

Test file A
(run time configuration)

Test A1
(random seed)
Test A2
(random seed)
Test A3
(random seed)
Test A4
(random seed)

Test file B
(run time configuration)

Test B1
(random seed)
Test B1
(random seed)
Test B1
(random seed)
Test B1
(random seed)

It might also be useful to start tests with common functionality (such as after run time configuration)
Agenda

- Advanced Verification Challenges
- **Supporting Technology**
- Use Modes
- Dynamic Load and Reseed Methodology
  - Saving the State and Reseeding
  - Dynamic Loading of Files
  - Dynamic Load and Reseed using UVM Test Phases
- Summary
Technology Overview Agenda

- Session Persistency
- Dynamic Load and Reseed
- Run-time Optimized/Debug mode Switching
- UVM test phases
- Technology Availability
Session Persistency (Save/Restore)

- Allows saving of the simulation state
  - Both VE and design State
  - Example: S1 S2 S3

Env

Initialization sequence (All tests, perhaps built into env)

It would be useful to start all tests after initialization is complete

Tests

Test file A (run time configuration)

Test A1 random seed
Test A2 random seed
Test A3 random seed
Test A4 random seed

It might also be useful to start tests with common functionality (such as after run time configuration)

Test file B (run time configuration)

Test B1 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed

Test A1 Seed 2
Test A1 Seed 3
Test A2 random seed
Test A3 random seed
Test A4 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed

Test A1 Seed 2
Test A1 Seed 3
Test A2 random seed
Test A3 random seed
Test A4 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed

Test A1 Seed 2
Test A1 Seed 3
Test A2 random seed
Test A3 random seed
Test A4 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed
Test B1 random seed
Technology Overview Agenda

- Session Persistency
- **Dynamic Load and Reseed**
- Run-time Optimized/Debug mode Switching
- UVM test phases
- Technology Availability
Dynamic Load and Reseeds Technology

- Allows a subsequent simulation to start from saved state
  - Can run the remainder of the simulation with a different seed
  - Can load additional code after restoring state
  - Reduces simulation/debug time dramatically

- Saves **considerable** simulation cycles
  - Shortens regression flows dramatically
  - Tests are can be developed/debugged faster
Dynamic Loadable File (DLF) example

// Base code
struct cdn_uart_frame_s like any_sequence_item {
    parity_type : cdn_uart_frame_parity_t;
    %start_bit: bit;
    %payload: list of bit;
    ...
};

// DLF
extend cdn_uart_frame_s {
    keep parity_type != NONE;
    keep plsize is only TRUE;
    my_field: byte;
};
Dynamic Load and Reseed Productivity Gains

- Original regression

<table>
<thead>
<tr>
<th>Test A</th>
<th>Test B</th>
<th>Test C</th>
<th>Test D (reseeded)</th>
</tr>
</thead>
</table>

- Dynamic Load and Reseed regression:

  **Initial setup run (common functionality across Tests A-D)**

<table>
<thead>
<tr>
<th>Test A</th>
<th>Test B</th>
<th>Test C</th>
<th>Test D (reseeded)</th>
</tr>
</thead>
</table>

Start of interesting traffic

**Significant** reduction in overall regression run time

Save State

Restore State
Technology Overview Agenda

- Session Persistency
- Dynamic Load and Reseed
- Run-time Optimized/Debug mode Switching
- UVM test phases
- Technology Availability
Run-time Optimized/Debug mode Switching

• **Motivation:**
  – Users often have to trade off between performance and debug
  – Until now, the decision between compiled and interpreted was done pre-run

• **Debug mode allows extra visibility for debug purposes.**
  – Step by step debug
  – Visibility on internal signals, etc.

• **Debug Switching allows users to realize significant performance gains during debug sessions**
  – Run in Optimized mode all the time
  – Does not introduce any performance penalty on code which is not being debugged
Run-time Optimized/Debug mode Switching

- Automatically enable debug mode when
  - Set breakpoints
  - Step into code
- In run-time, manually enable debug mode for user defined modules/methods

Examples:
- `set_config(debug, linedebug_modules,"*");`
- `set_config(debug, linedebug_modules,"cdn_mipi*");`
- `set_config(debug, linedebug_modules,"");`
Dynamic Switching between Debug and Optimized Mode Use Models

• Regressions:
  – Run in Optimized mode all the time
  – For debug:
    • Restore from last saved state (or beginning of simulation)
    • Begin debug immediately

• Development:
  – Run mature, stable code in Optimized mode
    • Realize significant speed up
  – Load current ‘under development’ code in Debug mode
    • dynamically load it at the interesting save point
  – Can debug all code together while gaining significant speed up
Technology Overview Agenda

- Session Persistency
- Dynamic Load and Reseed
- Run-time Optimized/Debug mode Switching
- **UVM test phases**
- Technology Availability
Save + Reseeding + Dynamic Load + UVM Test Phases

- Though not necessary, we recommend using Dynamic Load and Reseed with UVM Test Phases.
- Test phases slice the built-in `run()` phase into a set of finer grain phases. In ‘e’ they are named as follows:

  - **ENV_SETUP**
  - **HARD_RESET**
  - **RESET**
  - **INIT_DUT**
  - **INIT_LINK**
  - **MAIN_TEST**
  - **FINISH_TEST**
  - **POST_TEST**

Similar for every test

Most interesting for tests

Similar for every test

- Because each of these phases has a well-defined start/end point, this is an ideal starting infrastructure.
Technology Overview Agenda

- Session Persistency
- Dynamic Load and Reseed
- Run-time Optimized/Debug mode Switching
- UVM test phases
- Technology Availability
## Required technology

<table>
<thead>
<tr>
<th>Technology</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>State persistency (save simulation and restore)</td>
<td>Most simulation engines</td>
</tr>
<tr>
<td>Run-time Random Seed Change</td>
<td>Most simulation engines</td>
</tr>
<tr>
<td>Dynamic Load</td>
<td>Full support in Specman.</td>
</tr>
<tr>
<td>Structured run phases</td>
<td>UVM Test Phases</td>
</tr>
<tr>
<td>Run-time Debug mode switch</td>
<td>Full support in Specman.</td>
</tr>
</tbody>
</table>
Technology is Fully Available using Specman Advanced Option
Agenda

• Advanced Verification Challenges
• Supporting Technology

• Use Modes

• Dynamic Load and Reseed Methodology
  – Saving the State and Reseeding
  – Dynamic Loading of Files
  – Dynamic Load and Reseed using UVM Test Phases

• Summary
Dynamic Load and Reseed Use - Regression Flow

Sets up the verification environment and DUT in an interesting mode of operation (e.g. Mode A). There might be several setup tests per regression.

Shorter regressions as majority of tests start from restored state.

Problem: My regressions are taking WAY too long!

Even shorter regressions plus more localized debug.
Integration with Regression Runner

• **Requirements:**
  – Force order between runs
  – Use central location for simulation snapshots
  – Avoid cleaning the snapshot dir

• **Enterprise Manager** example

```bash
#!/bin/csh
irun <workdir>/xor.v -nclibdirpath <some_dir>  
<workdir>/setup.e -input <workdir>/base.tcl -intelligen
base.tcl
------------
runk 500;save s1;exit
```

```bash
#!/bin/csh
irun <workdir>/xor.v -nclibdirpath <some_dir>
<workdir>/setup.e -input <workdir>/base.tcl -intelligen
```

---

**test setup_test** {
  run_script: <workdir>run_base.csh;
}

**test test1** {
  count: 5;
  **depends_on: setup_test**;
  run_script: <workdir>/run_test.csh;
  seed: positive_gen_random;
}

```bash
#!/bin/csh
irun <workdir>/xor.v -nclibdirpath <some_dir> -r s1 -snseed $BRUN_SEED -sndynload <test_dir>/test1.e -exit
```

**Runs to a point, then saves the state and continues and/or exits**

**setup_test must complete prior to test1 starting**

**Restores saved state, loads a new file and runs with new random seed**
Dynamic Load and Reseed Use - Test Development Flow

Problem: My debug cycle is WAY too long!

Create baseline test

Run sim to “interesting” point (before sending traffic)

Save state of the simulation

Quit/continue the simulation

Setup Run (one time effort)

Could be lengthy simulation run to get to interesting point

Test Development/Debug

Bypassing initial setup run allows for more time to write/debug tests

Load new test after state restore

Send specific traffic

Create new tests

Restore state (optionally reseed)

Debug/Opt Mode Switch

Continue/Debug from restored state only
Test Development/Debug

Problem: My debug cycle is WAY too long!

Load DLF

Send specific traffic

Restore state (optionally reseed)

Run

Encounter Bug at time X

Debug/Opt Mode Switch

Save state at time (X-delta)

Run

Restore state with failing seed

Debug/Opt Mode Switch

Restore state (random seed)

Run

Debug/Opt Mode Switch

Encounter Bug at time X

Debug/Opt Mode Switch

Encounter Bug at time X+300

Run

Encounter Bug at time X+10

Run

Encounter Bug at time X+187

Run

Once a bug is identified, launching random seeds from S2 will help flush out related bugs

Restore state with failing seed

Debug/Opt Mode Switch

Encounter Bug at time X

Debug/Opt Mode Switch

Restoring close to bug location permits rapid debug

Debug/Opt Mode Switch

Run

Debug/Opt Mode Switch

Run

Debug/Opt Mode Switch

Run

Debug/Opt Mode Switch

Run
Agenda

- Advanced Verification Challenges
- Supporting Technology
- Use Modes
- Dynamic Load and Reseed Methodology
  - Saving the State and Reseeding
  - Dynamic Loading of Files
  - Dynamic Load and Reseed using UVM Test Phases
- Summary
Dynamic Load and Reseed Methodology

Agenda

• Saving State
• Reseeding
• Dynamic Load
• Saving State using UVM Test Phases
• Coding Considerations
Saving the State of the Simulation

- Considerations when deciding on save point:
  - Want to capture all repetitive actions to minimize redundancy
  - Want to maximize the coverage achieved through testcases
  - Want to choose a clean point in time to restart new sims from
  - Interesting or hard-to-reach DUT states
  - Error prone DUT functionality
    » Perhaps previous bugs have been encountered

- Some examples:
  - After programming a DUT (clock synch, reset, register programming)
  - After protocol initialization (e.g. link training)
  - Just before reaching an error/bug situation requiring debug
  - Hard to reach DUT states (to launch many tests in this state)
Test bench considerations – Choosing ‘safe’ Save Points

- Consider a save point selected midway through a sequence execution
  - Sequence should send 6 RED packets
  - Save point occurs after transmitting only 4

- Original intent of 6 RED packets is lost
- This might or might not be an issue for the verification environment or DUT, but should be considered

```plaintext
extend RED seq_s {
  for i from 1 to 6 {
    do packet keeping {
      soft .colour == RED;
    };}
};
```

//DLF loaded at restore point
```plaintext
extend packet_s {
  keep colour == BLUE;
};
```

5th and 6th packets must consider the new constraints in DLF
There are several ways to save the state of the simulation:

- Interactively through commands
  - Pros: Can select exact save point manually
  - Cons: Lack of automation

- Passing tcl scripts on the command line
  - Pros: Automated, no need to modify base code
  - Cons: Depends on base code not changing as save point might be based on breakpoints and/or particular point in time during the sim

- Embed the simulator-save command in code
  (example `sys.simulator_save()` in Specman-e)
  - Pros: Automated, embedded within the environment code (will always be at the correct point). Can be controlled through config flags and or other guard conditions in the code.
  - Cons: Must modify/extend the base code
Example: Using `sys.simulator_save()` to Save State

- Let’s say we have a virtual sequence driver controlling all tests in a particular environment through a sequence called `USER_DEFINED_TEST_FLOW`
  - `body()` executes three methods
    - `do_initialization()`
    - `do_link_training()`
    - `do_test()`
  - In this verification environment, users typically extend the `do_test()` method to implement their tests
  - Environment developer would like to create a save point after link training is completed
Example: Saving State After Link Training – Virtual Sequence

```
extend USER_DEFINED_TEST_FLOW cdn_uart_virt_seq_s {
...

  body()@driver.clock is {
    do_initialization();
    ...
    do_link_train();
    ...
    do_test();
  };  

do_initialization()@driver.clock is { do initialization; };  
do_link_train()@driver.clock is {
  do training;
  if (save_after_init) then {
    sys.simulator_save(“after_link_train”,TRUE,TRUE);
  };
}

do_test()@driver.clock is {do test_sequence};
};
```

User hooks for extension within tests

Save state after the DUT has been initialized, and the links trained but before the start of the interesting test
Dynamic Load and Reseed Methodology

Agenda

• Saving State
• Reseeding
• Dynamic Load
• Saving State using UVM Test Phases
• Coding Considerations
Restore and Reseed Use Models Flow

• Bugs are flushed out of the DUT quicker as reseeding from a restore point allows users to run more meaningful scenarios in a shorter amount of time
• Changes in regression methodology will be required
  – Interesting save points embedded in the environment
  – Regressions must handle dependencies between save/restore runs
• Coverage is increased as more random seeds are run
Restore and Reseed Use – Regression Flow

Tests saves state at different interesting points

Each seed produces new stimulus combinations (Test1\textsuperscript{1}, Test1\textsuperscript{2}, Test1\textsuperscript{3}, Test1\textsuperscript{4})

- **Run Test1**
  - Run to “interesting” point
  - Save state of the simulation
  - Reset Complete
  - Restore R1 state
  - Reseed and run

- **Run Test2**
  - Run to “interesting” point
  - Save state of the simulation
  - DUT programmed
  - Restore P1 state
  - Reseed and run

- **Run Test3**
  - Run to “interesting” point
  - Save state of the simulation
  - Links Initialized
  - Restore L1 state
  - Reseed and run
Restore and Reseed Use – Improved DUT Coverage

- Single Setup Run
  - Launching with many seeds from a single state improves coverage
  - Can be random seeded
  - Multiple saves allows for reseeding from each state. If bug is found, only need to revert back to last saved state, saving simulation cycles.

DUT Functionality
Dynamic Load and Reseed Methodology

Agenda

• Saving State
• Reseeding
• Dynamic Load
• Saving State using UVM Test Phases
• Coding Considerations
Dynamic Load Overview

• Dynamic Load builds on the save/restore features
• After restoring to a particular saved state, users can load additional code to influence future stimulus
• Because we are restoring from a previously saved state, we must consider the following in dynamically loaded files:
  – Component topology and coverage model have been built
  – Events are already triggering
  – TCMs/Tasks are already running
  – Compiled external models are already connected
• The new files we load are called Dynamic Loadable Files (DLFs) and will only affect the simulation from the point of loading onwards
Dynamic Loadable Files

- DLFs can contain code to modify the verification environment moving forward
  - We must consider that the simulation is already underway
- Analogy: Jumping onto a moving bus

**Cannot** change the number of windows on the bus (topology)

**Cannot** change the engine of the bus (events, currently running TCMs/Tasks, etc.)

**Can** change the scheduled stops the bus makes (sequences sent)

**Can** change what goes on/off the bus at various stops (transfers/transactions)

**Can** change the speed, beep the horn (constraints)
Dynamic Loadable File (DLF) Contents - Procedurally Overwriting Existing Fields

// Base code
...

```cpp
// DLF
extend cdn_uart_agent_u {  
  my_method() is also {  
    if (some_condition) then {  
      speed = M200;  
      gen clock_freq;  
    }  
  }  
};
```

Can set new field values procedurally.

**WARNING:**

- clock_freq will not be updated automatically.
- clock_freq must be generated on-the-fly for constraint to be enforced and value set correctly.

```cpp
// Base code
...

```
• DLF tests should not depend on a save point occurring at a particular instance in time
  – Save point might change

• Ensure DLF tests are robust:
  – Any needed guard conditions should be in place
    • Ensures that stimulus is sent at the right time
    • Example: Test should send particular traffic after reset

• Use environment ‘hook’ methods to launch TCMs/Tasks
• Should not depend on built-in hooks associated with restoring/dynamic loading (pre_save/pre/post_restore)
Dynamic Load and Reseed Methodology

Agenda

• Saving State
• Reseeding
• Dynamic Load

**Dynamic Load with UVM Test Phases**

• Coding Considerations
UVM Test Phases

- Environments integrate multiple verification components from heterogeneous sources, which might be
  - Internal company sources (multiple sites on one project)
  - Commercially available VIP products

- Integrated components need to be in sync with respect to each stage of the simulation
  - Example: Cannot program DUT registers until reset is complete

- Integrators and test writers need a standard scheme and API to orchestrate system scenarios
  - Multiple sub-environments (domains) might need to be managed
    - Might have different clocks to each domain
Each component in the environment performs actions throughout the simulation over time.
Built-in phases address some of the needs of components.

The run() phase could benefit from more granularity though.

Components need synchronization to ensure that all actions are executed in order throughout the system.

The run() phase could benefit from more granularity though.

Components need synchronization to ensure that all actions are executed in order throughout the system.

The run() phase could benefit from more granularity though.

Components need synchronization to ensure that all actions are executed in order throughout the system.

The run() phase could benefit from more granularity though.
Intro to UVM Test Phases Phases

Testbench components

- **A**: HARD_RESET
  - reset
- **B**: RESET
  - reset
  - register
  - config
- **C**: INIT_DUT
  - init
- **D**: INIT_LINK
  - monitoring
  - traffic
  - protocol
  - training
- **E**: MAIN_TEST
  - traffic
  - post test
  - checking
- **F**: FINISH_TEST
  - post test
  - checking

All functionality related to a particular phase is executed together.

Simulation time:
- generate()
- connect_pointers()
- setup()
- run()
- stop_run()
- extract()
- check()
- finalize()
UVM Test Phases Semantics

- A component declares its participation in the scheme explicitly
- One task per phase for each test phase component
  - Automatically started at the beginning of the phase
  - Next phase does not start before previous one returns
  - Automatically terminated upon phase reset or time-out
- User might choose to define activity for any given phase, or leave it empty

```verilog
extend xyz_master_bfm_u {
  tf_testflow_unit;
  event tf_phase_clock is only @clk;
  tf_main_test() @tf_phase_clock is {
    send_frames();
  };
}
```

- Macro inclusion declares participation in test phase
- tf_phase_clock must be defined for each participating component
- tf_main_test() will not start before all previous phases are complete
tf_phase_started/ended/stopped Methods

In addition to built-in phase TCMs/Tasks, each component has three built-in phase hook methods

- any_unit.tf_phase_started(phase: tf_phase_t)
- any_unit.tf_phase Ended(phase: tf_phase_t)
- any_unit.tf_phase_stopped(phase: tf_phase_t)

- phase – the phase which is currently starting/ending/stopping

User can tap into any phase as needed:

```pascal
unit xyz_master_bfm like uvm_bfm {
    tf_phase_ended(phase: tf_phase_t) is {
        if phase == ENV_SET UP {
            start sys.simulator_save(“after_env_setup”);
        }
    }
};
```
UVM Test Phases Sequences

- A sequence library might declare participation in the test phase scheme
- One MAIN sequence generated and started per phase
  - Its body is yet another phase TCM
- Enumerated field associates sequences with phases
  - Orthogonal to kind
  - Can be used to classify sequence kinds into phases
  - Can be used to constrain sequence generation to relevant kinds

```verbatim
sequence xyz_sequence using testflow = TRUE, item = xyz_packet;
extend MAIN INIT_LINK xyz_sequence {
  body() @driver.clock is {
    do TOKEN pkt;
    do HANDSHAKE pkt;
  };
}
```
UVM Test Phases and Dynamic Load and Reseed

Example: Saving state after link initialization

- **Without test phases**, users would have to carefully manage when to save the state of the simulation
  - Example: DUT has multiple links that require initialization

```verbatim
// Base code for initialization sequence
extend INIT my_seq_s {
  ...
  body()@driver.clock is only {
    do init_sequence;
    driver.env.agent[channel].link_trained = TRUE;
  };
};

// Base code to perform save
extend my_env_u {
  ...
  !num_links_trained: uint;
  save_after_links_trained()@clock is {
    repeat {
      !num_links_trained == 0;
      for each in agents {
        if it.link_trained {num_links_trained += 1;};
      }
      wait cycle;
    } until num_links_trained == agents.size();
    sys.simulator_save("post_link_train");
  }
};
```

Initialization sequence is running on multiple agents within the env

Only after all links have completed their training can we save simulation state
UVM Test Phases and Dynamic Load and Reseed

Example: Saving state after link initialization

- **With test phases**, we can tap into the built-in methods to know when a particular phase is done
  - Example: DUT has multiple links that require initialization

```vhd
// Base code
extend MAIN INIT_LINK my_seq_s {
  !init_sequence: TRAINING my_seq_s;
  count: uint;
  body()@driver.clock is only {
    for i from 1 to count do {
      do init_sequence;
    }
  }
};

// File: save_states.e
extend my_env_u {
  tf_phase Ended(phase: tf_phase_t) is {
    if phase == INIT_LINK{
        start sys.simulator_save("post_link_train");
    }
  }
};
```

`tf_phase Ended` is called automatically, so no need to synchronize across all agents.
Dynamic Load and Reseed Methodology

Agenda

• Saving State
• Reseeding
• Dynamic Load
• Saving State using UVM Test Phases
• Coding Considerations
Coding Considerations When Using Dynamic Load and Reseed Methodology

- Environment Architects
  1. Maximize control of the environment through constraints
     - Allows for greater DLF control
     - Named constraints can be overwritten in DLFs
  2. Implement empty methods as callbacks within the environment
     - Users can extend these in tests to start TCMs/Tasks and add functionality at key points in the simulation
  3. Implement a well thought out save/restore architecture
     - Let test writers know what the save/restore points are
     - Consider using standard naming convention for state saves
     - Place guard conditions around save points to allow for better control (same code used for both regular and save/restore sims)
  4. Implement UVM Test Phases to take advantage of the built-in features
Coding Considerations When Using Dynamic Load and Reseed Methodology

• Test writers
  1. Ensure that their DLF tests are independent of load time
     • Environment developer could move the state save a few cycles
  2. When procedurally overwriting fields in existing structs/components, ensure that other related fields (connected via constraints) are still valid
Agenda

- Advanced Verification Challenges
- Supporting Technology
- Use Modes
- Dynamic Load and Reseed Methodology
  - Saving the State and Reseeding
  - Dynamic Loading of Files
  - Dynamic Load and Reseed using UVM Test Phases
- Summary
Dynamic Load and Reseed Methodology Summary

Through bypassing initial (and often lengthy) start-up functionality, users can:

• Get to the more meaningful portion of their simulations faster
• Achieve higher degrees of functional coverage
• Reduce test development cycles/flows
• Reduce debug cycles/flows
• Reduce regression runs
• Save hundreds of simulation hours
Dynamic Load and Reseed Methodology Summary

Technology is Fully Available with eSpecman Advanced Option