cadence®

Improving Verification Productivity with the Dynamic Load and Reseed Methodology

Authored by Corey Goss, Solutions Engineer, Cadence Design Systems

Verification of today's complex ASIC/FPGA designs continues to push the limits of available resources. This causes verification teams to look for new ways to improve productivity of verification tasks. The purpose of this technical paper is to introduce the reader to the new Dynamic Load and Reseed Methodology, and the supporting technologies that can be used to dramatically improve verification productivity.

Contents

Audience

The target audience for this application note is digital verification engineers looking to achieve significant performance improvements in their overall verification flow. In particular, those with very long debug cycles or with verification environments containing common start-up functionality such as link training, long reset sequences, or large register spaces that need to be programmed will find particular benefit. The audience need not be ***e*** or Specman® users to understand the benefits of this methodology and technology.

Introduction and Motivation

Simulation-based functional verification involves a multitude of tests that, together, constitute a regression suite for a given design under test (DUT). Each single simulation test run often involves a DUT start-up phase responsible for performing functions such as reset procedure, link training, register configuration, or bringing the DUT into a state from which specific operations can be executed (e.g., traffic can be sent). In large system-level simulations involving multiple DUTs, several DUTs may need to be brought to a particular state prior to sending anything meaningful. Even though the initialization process is often applicable to large subsets of the regression suite, existing verification practices do not take advantage of this commonality between runs. The same start-up phases are executed over and over for every test. This consumes valuable time and resources while contributing little, if anything, towards coverage.

Moreover, when a test fails during a regression, the debug session must go through the same start-up process yet again, with significant impact on debug cycle efficiency. Debug is especially lengthy if a failure occurs very late in the simulation run (e.g., hours or days in.) Since identifying and isolating a bug correctly can take several iterations, this can lead to many wasted simulation cycles. If the point of failure can be reached significantly faster, bypassing uninteresting functionality along the way, a significant productivity boost can be achieved.

Simulation engines support session persistency in the sense that the entire simulation state can be saved to disk and later resumed from the same point in a different process, possibly multiple times. This means that you could run a simulation up to a certain point, save its state, and later resume it in multiple processes later on.

One approach to take advantage of this is to enable the random seed to be changed after a simulation state is restored. The new seed would cause the test scenarios to change from the restore point forward, due to different results of random generation. This approach is called ***Reseeding***, and can be applied to all verification environments that use random generation for creating test scenarios. Another approach would be to load additional files after the simulation has been restored. The loaded code could contain constraints and additional functionality that could change the behavior of the resulting test scenario from the load point onwards. This approach is called ***Dynamic Loading***.

When Dynamic Load and Reseed approaches are used together, new methodologies can be employed to dramatically improve verification productivity. These new capabilities are only available in the ***e*** language today.

What is Dynamic Load and Reseed Technology?

Simulation engines support session persistency and this functionality has existed within the Specman Elite® tool for many years. Dynamic Load and Reseed technology builds upon the existing save/restore functionality in that, upon restoring the simulation, you can now run a different scenario by:

1. Continuing the same simulation with a new seed value
   a. All subsequent generation actions will be affected by the new random seed
   b. Random stability is preserved for save/restore simulations run with the same seed
2. Loading new ***e*** files and continuing the simulation with the same seed
   a. Loaded files are called dynamically loadable files (DLFs)
   b. Newly loaded files can influence generation actions and add functionality (see "Dynamic Loading of Test Files" for details)
3. Dynamically loading new ***e*** files ***and*** continuing the simulation with a new seed value

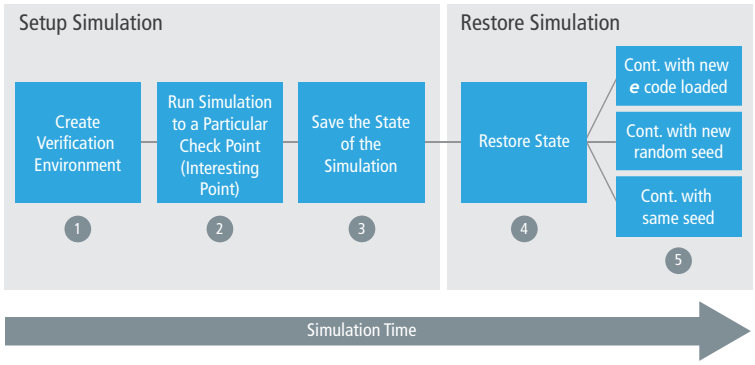


Figure 1: The overall simulation flow

Benefits of Employing the Dynamic Load and Reseed Technology

Dynamic Load and Reseed technology brings many benefits. It allows for new use models to emerge that can significantly enhance productivity. Below, we outline three such models:

1. Faster Test Development/Debug
2. Bug Focusing
3. Improving Regression Throughput

Faster Test Development/Debug

When developing new tests, you can easily bypass un-interesting early phases of the simulation allowing you to see the effects of your tests immediately. This allows tests to be debugged much more quickly than with previous methods as, if a failure occurs, tests can be tweaked and re-launched closer to the point of failure.

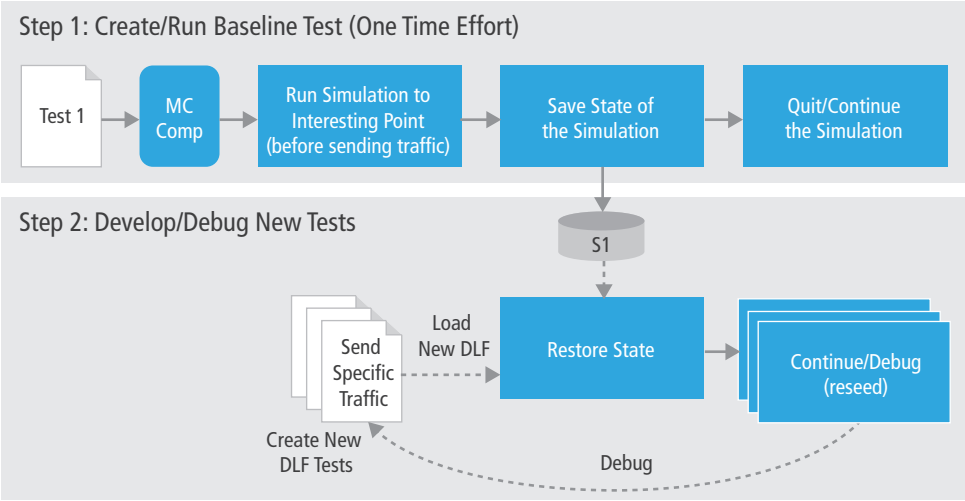


Figure 2: An improved test developoment flow

Bug Focusing

Bugs within a design typically hide in groups. If one bug is found in a particular feature of the DUT, there is a high probability that other bugs are lurking around the same or related features. Since save points can be created at any point in the simulation, one way to shake out additional bugs would be to create a save point shortly before a known bug occurs, then restore and run several simulations that target stimulus at the problem area (through simple reseeding or through dynamically loading additional tests). By significantly varying the stimulus so close to a known failure point, the probability of encountering other related bugs is increased. Since the simulations start right before the bug, such runs will get to the problem area almost immediately, saving valuable simulation debug time and engineering resources. The usage flow is as follows:

1. User encounters a bug in a simulation run
2. User re-runs the failing sim, saving the simulation state at a point just prior to the bug occurring
3. User restores the save point with the failing seed allowing for shortened debug loop
4. User runs random seeds to identify other, related, bugs

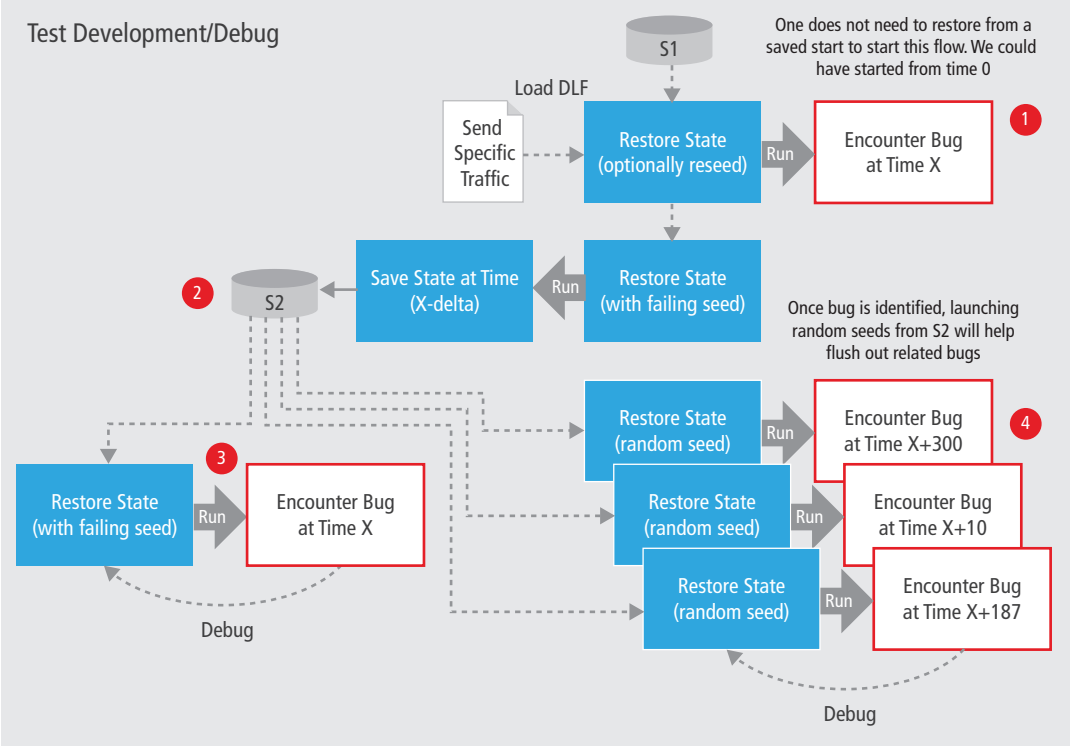


Figure 3: Bug focusing

If a bug is identified to be a register transfer language (RTL) issue, random stability in the verification environment will ensure that the same seed will produce the same stimulus (provided the constraints in the environment are not tied to DUT state that may have been affected by the RTL fix), allowing an engineer to quickly verify that the bug has been fixed correctly. Reseeding again around the previously known bug after the RTL has been fixed will help to shake out any related or newly introduced bugs.

Improving Regression Throughput

In addition to reducing the test development and debug cycle, significant productivity gains can be achieved in regression runs. By identifying groups of tests that share common functionality, you can partition a regression run into groups of tests that should be launched from a common restore point. For example, if every test in the regression proceeds through a common initial start-up sequence, a single set-up run can be launched and the state of the simulation (we'll call this S1) saved after the initial start-up sequence. All other tests in the regression can be loaded and launched after first restoring the S1 state. Coverage can be dramatically increased through launching multiple random seeded runs of each test from state S1.

Furthermore, if a certain group of tests requires that the DUT be set up in a particular mode of operation (e.g., DUT registers need to be programmed or a stream of configuration packets need to be sent), the state of the simulation can again be saved (we'll call this S2) after this secondary phase of set up and all tests related to that particular mode of operation can be launched from state S2. There is no limit to the number of states that can be saved in any given simulation run, paving the way for a single set-up run to save various states of the simulation after certain interesting phases have succeeded (e.g., Reset, DUT Initialization, Link Training, DUT Configuration, etc.).

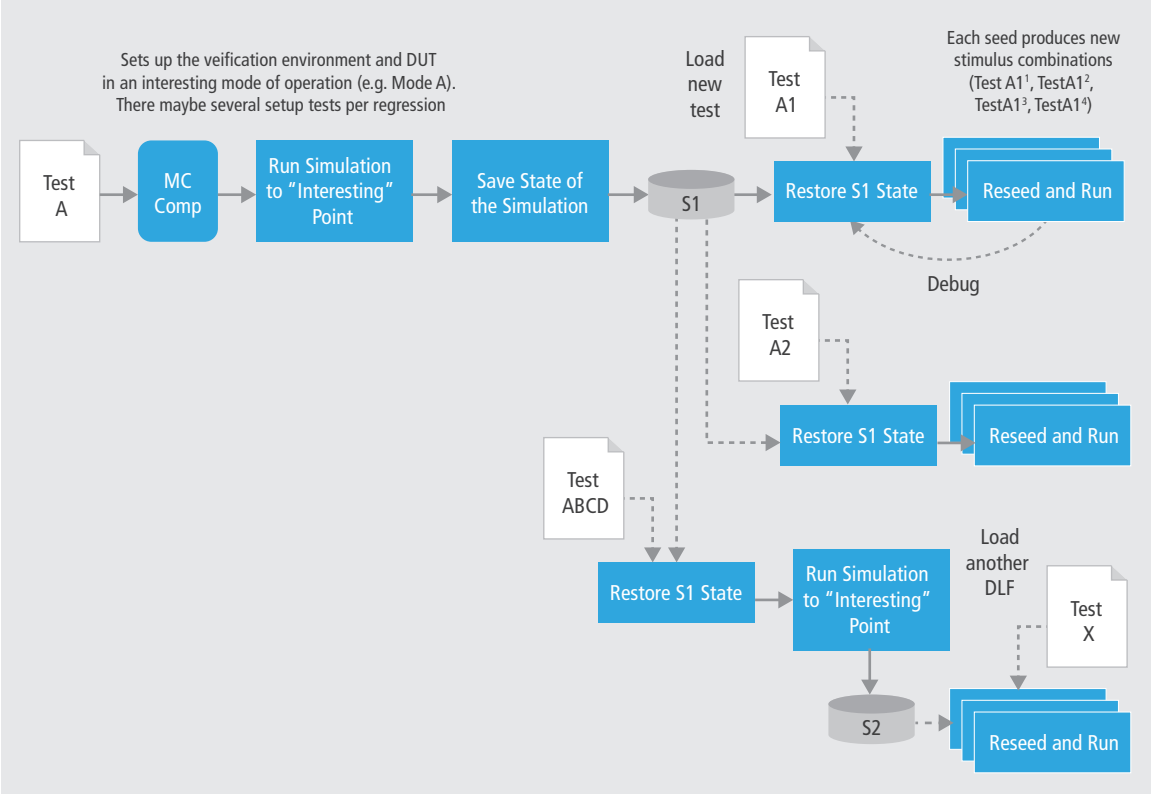


Figure 4: An improved regression flow

Adopting the Dynamic Load and Reseed Technology

Dynamic Load and Reseed technology can be adopted according to your specific needs, allowing for rapid implementation on any existing verification project. Through adopting Dynamic Load and Reseed technology, you can employ new methodologies and flows to your test development and regressions that will allow for significant productivity gains. Early customer feedback has indicated a savings of up to 60% over their current methodologies and flows. For example:

- **Save/Restore + Reseeding:** You can select a key save point(s) in your environment for a test, then run the test to the save point, restore, and reseed the same test multiple times. This is the simplest method of adoption and provides overall reduced simulation time (over running each test from time 0) and faster coverage convergence (due to more exploration of their DUT in a shorter window).
- **Save/Restore + Reseeding + Dynamic Load:** On top of the benefits that Save/Restore + Reseeding offer, you can also benefit by adding functionality after a restore through loading of additional ***e*** files. This adoption method enables you to load and launch many individual tests from a single (or multiple) save point(s), dramatically reducing both test development and regression cycles.
- **Save/Restore + Reseeding + Dynamic Load + UVM *e* Testflow:** Universal Verification Methodology (UVM) ***e*** Testflow provides an ideal simulation framework for use with Dynamic Load and Reseed technology. It breaks the run() phase of participating units and sequences within the verification environment into 8 subphases, each with a clear start/end point, and provides built in hooks that are very useful for saving the state of the simulation. While UVM ***e*** Testflow is not a precondition for Dynamic Load and Reseed usage, it is well suited, easy to integrate into an existing environment, and is recommended. More information on UVM ***e*** Testflow features can be found in "Choosing the Simulation Save Points."

Dynamic Load and Reseed technology is built upon the IntelliGen Generation engine within the Specman Elite tool. A key enabler of future technologies, IntelliGen offers significant benefits to customers over the previous Specman Generation engine (Pgen) in terms of ease of use, solvability, coverage, and performance. All steps required to migrate an existing verification environment to IntelliGen are clearly outlined in the "IntelliGen User Guide" within the Specman Elite documentation.

Choosing the Simulation Save Points

There are several factors to consider when choosing a point to save the simulation state. To minimize redundancy across the test base, a safe point should be chosen just after a group of repetitive actions have completed. In this case a "safe" point means that the particular phase of the simulation that you would like to bypass has completed fully. Any sequences or transactions that are currently "in flight" might be affected after the restore point by DLF content (e.g., new constraints) so it is best to ensure that the state is saved once all transactions for a particular phase of the simulation are completed. Additionally, you should consider creating save points at the beginning of phases where stimulus sent to the DUT varies most (e.g., after the DUT is initialized and actual traffic is being sent), as this allows for the highest amount of coverage to be achieved through dynamically loading tests and reseeding. Because the save points to select are highly DUT dependant, some suggested points are as follows:

- Generally useful save points:
  - After reset has been asserted
  - After programming initial DUT register configuration
  - After clock synchronization
  - After bringing up a system simulation (many DUTs)
  - After all links have been trained
  - At the start of the simulation (to avoid recompiling/reloading files)
- PCI-Express/USB 3.0 protocol specific:
  - After Link Training
  - Once reaching the L0 (PCIE) or U0 (USB 3.0) state
  - After enumeration process has occurred

When saving the state of the simulation, you must save the state of both Specman Elite and the simulator. For users of the Cadence® Incisive® Enterprise Simulator, a synchronized save is easily achieved through either the command line or, preferably, through calling the built-in **sys.simulator_save()** time-consuming method (TCM) embedded within your ***e*** code. An example of this method call is shown in the following code example:

```
extend USER_DEFINED_TEST_FLOW cdn_uart_virt_seq_s {
    do_link_train()@driver.clock is (
        if (save_after_init) then {
            sys.simulator_save("after_link_train",TRUE,TRUE);
        );
    );
);
```

Snapshot name

Overwrite any snapshots with the same name

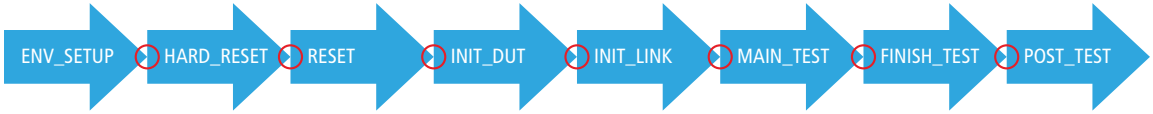
Continue the simulation after saving state

For users of other commercial simulators, the appropriate save command must be executed in addition to the Specman save through either the command line or scripting.

Synchronization of Verification Environment Components

To synchronize verification environment components as the simulation progresses, many users employ some form of structured framework, or phases, within their environment. All simulations proceed through the phases in a sequential manner. This can be in the form of a virtual sequence, event handshaking, or some other synchronization mechanism. An example of an open source framework to perform such a function is UVM ***e*** Testflow (available

within the UVM ***e*** library, which is part of the Specman Elite release). Once implemented in a user environment, the **run()** phase of participating verification environment components (e.g., monitors, sequence drivers, etc.) is broken into 8 subphases as follows:



The phases are executed from left to right and no phase begins until all participating components contributing to the previous phase have completed their tasks. The user is free to extend any of the phases (TCMs and sequences exist for each phase) to add desired functionality and there are a number of built in hooks to tap in to phase execution as needed. The black circles in the diagram above indicate key points in the phase progression for creating save points. An example of how to use UVM ***e*** Testflow built-in features to save the state of the simulation just after the INIT_LINK phase has completed is shown below:

Built-in method tf_phase_ended() is automatically called at the end of each phase

```
extend my_verification_env_u {

    tf_phase_ended(phase: tf_phase_t) is (
        if phase == INIT_LINK {
            start sys.simulator_save("after_link_train",TRUE,TRUE);
        );
    );
);
```

The UVM ***e*** Testflow phases feature is simple to implement into any environment (in case a user does not have a previously existing framework). While the implementation of UVM ***e*** Testflow in a verification environment is very useful, it is not a precondition to make use of the Dynamic Load and Reseed functionality.

Dynamic Loading of Test Files

When the simulation state is restored and the simulation is continued, all ***e*** code executes seamlessly, as though the set-up and restore portions are executed in one continuous simulation. Upon a restore, additional ***e*** files can be loaded that will affect future simulation results. Since the simulation is continuing from a previous state, we must consider that certain tasks have already taken place or are "in progress." Examples are pre-run generation, TCM execution, event triggering, port connections, and external model connections (such as C++/SystemC®). The ***e*** code residing within DLFs must consider this.

One can think of a DLF as a formalized way of defining the contents of a test. As opposed to tests being used to affect pre-run generation (through environment topology constraints) and the functionality/structure of any object in the environment, a DLF is designed to mainly affect or support stimulus generation. This requires a methodology to be employed by verification environment developers where there is clear separation between environment configuration files and test files. Tests contain stimulus control while the environment is written in a manner to support control from the test. For users who have implemented environments based on a coverage-driven verification (CDV) methodology, this separation will be very natural. For directed test users who implement custom checks and/or significantly modify the verification environment (topology, etc.) within each test, a shift in methodology will be required.

DLFs can include code that:

- Modifies constraints on data items so as to inject different traffic scenarios
- Extends objects to add new fields
- Defines and launches new sequences

- Defines and launches new TCMs
- Extends/overrides previously defined empty TCMs
- Extends a previously defined empty TCM
- Creates new checks/expects/events in new struct types
- If using UVM *e* testflow phases, modifies future test phases
- Extends the post run() built-in phase methods: extract(), check(), finalize()
- Procedurally overwrites fields of existing units

An example of a simple DLF is outlined below on the right. The code on the left represents a subset of the base code within the environment:

```
// 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;
    keep plsize is soft payload.size()
    ...
};
extend cdn_uart_seq_kind_t: [INIT];
extend INIT cdn_uart_seq_s {
    !init_sequence: LEGAL cdn_uart_fra
    count: uint;
    body() @driver.clock is only {
        for i from 1 to count do {
            do init_sequence;
    }; // body()@driver.c...
}; // extend INIT cdn...

unit cdn_uart_env_u like uvm_env {
    event clock is rise (clock_p$) @sim;
    tx_agent: TX cdn_uart_agent_u is instance;
(;
```

```
//DSF file
extend cdn_uart_frame_s {
    keep parity_type !=NONE;
    keep plsize is only TRUE;
    my_field: byte;
};
extend INIT cdn_uart_seq_s {
    keep count in range [1..5];
};
extend cdn_uart_env_u {
    !new_field: uint;
    keep new_field < 50;
};
```

New constraints on existing fields

Modify existing constraints

New fileds added to existing structs

New constraints on existing sequences

New fields added to existing units

Note: Must manually generate new fields

Note that in the above example, **new_field** must be manually generated. This is true for all new fields added to units as the unit structure is generated once, at the start of the simulation run.

DLFs cannot include code that:

- Modifies the verification environment topology or coverage model
- Modifies the port bindings: do_bind, .connect(), .disconnect()
- Adds new event, expect,or assume constructs to existing units/structs
  - These **can** be included in new struct type defined in DLF
- Adds new events tied to the @sim sampling event
- Refines a defined, non-empty TCM in an existing type (is first/also/only)
  - Only overriding of empty TCMs is allowed
  - Refinement is allowed in the context of a new type defined in a DLF
- Must be compiled

Interoperability with Incisive Enterprise Manager

The Dynamic Load and Reseed Methodology is fully supported by Incisive Enterprise Manager. Various attributes within a verification session input file (VSIF) manage which tests are executed in a given regression, as well as identify where the results are stored and which scripts to call. The **depends_on** attribute is used to ensure proper ordering between tests. This allows you to code VSIFs such that, first, a set-up run can be launched that saves the

state (or multiple states) of the simulation and, second, any tests that are dependant upon the set-up run will not be launched until it successfully executes. If the set-up run fails for any reason, all of the dependent tests will not be launched and, instead, be marked as failed, with a note pointing you to check the set-up run for issues.

A sample VSIF file is shown below. The **top_files** attribute identifies individual test names within a container. In the example, the test1.*e* file is part of the dynload_test1 container. Each container has its own set of attributes and, since the dynload_test1 container "depends_on" the save_test container, it will not be executed unless save_test has successfully completed.

```
session test_reseed [
    session_name: "Test Reseed and dynamic load";
    top_dir : $SESSIONS_DIR;   //Store all output to this directory
    output_mode: terminal;     //Open an xterm for viewing progress
};
group reseed {

    timeout: 100000;
    //scans all logs for errors
    scan_script: vm_scan.pl ius.flt sv_uvc.flt;

    //default run script
    run_script: "$SCRIPTS_LOCATION/run_test.csh";

    test save_test {
        run_script: "$SCRIPTS_LOCATION/setup_run.csh";
    };
    test dynload_test1 {
        count: 5;
        depends_on: ../save_test;
        top_files: $TESTS_LOCATION/test1.e;
        seed: positive_gen_random
    };
};
```

Restores saved state, loads a new file and runs with new random seed

Runs to a point, then saves the state and continutes and/or exits, seed will be default value of 1

Run dynload_test1 5 times

save_test must complete prior to dynload_test starting

Run with random seed each time

Benefits Summary

As today's devices continue to grow in complexity, verification teams are continually looking for new ways to improve productivity. The Dynamic Load and Reseed Methodology and its underlying technology allow users to dramatically improve verification productivity by building upon the previously existing session persistency features (save/restore) of the Specman Elite tool and the simulator. Simulation states can be restored and reseeded to increase coverage. New files can be dynamically loaded after restoring to guide future generation results.

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

All features are fully compatible with Incisive Enterprise Manager and are available as part of either the Specman Advanced Option or the Incisive Advanced Option feature set available in the 10.2 and later releases. Parallel *e* code compilation, the ability to debug compiled *e* code, and the ability to dynamically switch any/all files from compiled to interpreted mode during a simulation run are also Specman Advanced Option features.

Further Reading

More on Dynamic Load and Reseed technology (and other Specman Advanced Option features):

- "Specman Functional Verification" documentation available in the 10.2 Incisive Enterprise Simulator/Specman release
- Specman Advanced Option workshop
  - 1-day self-paced training workshop that includes slides and several hands-on lab exercises to explore the various features available in the Specman Advanced Option

More on UVM *e* Testflow features:

- "UVM *e* User Guide" documentation available in the 10.2 Incisive Enterprise Simulator/Specman release

More on Specman Elite technology or the *e* language:

- Introduction to the *e* Language workshop
  - 1-day self-paced training workshop including slides and labs
- Introduction to Specman Elite workshop
  - Self-paced training workshop on basic Specman features for loading, launching, running, and analyzing tests
- Generation using IntelliGen workshop
  - 1-day self-paced training workshop including slides and labs

Any of the above can also be obtained by contacting your local Cadence Sales representative.