Soft Constraints in SystemVerilog
Semantics and Challenges

Mark Strickland, Joseph Hanli Zhang
Cisco Systems
{mastrick, jhzhang}@cisco.com

Jason Chen, Dhiraj Goswami, Alex Wakefield
Synopsys Inc.
{jasonc, dhiraj, alexw}@synopsys.com

Abstract—**This paper introduces SystemVerilog soft constraints, describing their syntax and semantics. We then examine how soft constraints can be applied to verification, how they can be used to manage constraint complexity and used with VIP. Lastly we discuss tool requirements including debug features.**

Keywords-SystemVerilog, constraint solver, soft constraint

I. INTRODUCTION

Constrained random verification is the leading methodology for verification of chip designs today. As this methodology has grown in popularity, so has the size and complexity of the environments used to verify increasingly complex System-on-Chip (SoC) designs. One of the challenges facing verification engineers is the number of constraints present in the verification environment, and how these constraints are reused between projects or can be leveraged from existing Verification IP (VIP) components.

The IEEE P1800 (SystemVerilog) [1] standard provides two mechanisms to modify constraints – (1) overriding by defining a constraint block with the same name, and (2) constraint mode control to enable and disable an existing constraint block.

The IEEE P1647 (*e*) standard [2] defines soft constraints as a mechanism to allow constraints to be overridden by other constraints. A proposal (Mantis 2987 [3]) was recently approved by the P1800 working group to add soft constrains to the 2012 revision of the SystemVerilog standard. Later in this paper we will details the similarities and differences between *e* and SystemVerilog soft constraint semantics.

We believe both of these concepts provide complementary ways of managing constraints in a large verification environment. This paper will describe the semantics for soft constraints, discuss methodology for using soft constraints effectively, and investigate some additional SystemVerilog specific issues.

II. SOFT CONSTRAINT

A. What Are Soft Constraints?

SystemVerilog and *e* simulators both use a constraint solver to create random stimulus. The constraint solver examines all the active constraints to generate a legal solution where all constraints are satisfied. Sometimes it would be desirable to have the constraint solver disregard certain constraints if (and only if) no solution is possible when they are included.

This requirement is more pronounced when VIP is used, as is typical in today's SoC environment. Here the test writer is often not familiar with all the constraint details of the VIP and simply wants to override some of the constraints to generate an error condition or direct the distribution of specific variables. Soft constraints allow the VIP creator to specify rules that can be easily overridden by the test writer. In other words, soft constraints provide more scope for relaxing these rules.

Soft constraints are the constraints that need to be satisfied if possible; otherwise, they are disregarded. If no solution is possible when all the hard and soft constraints are considered, individual soft constraints are iteratively disregarded based on a priority scheme (described in section III.A). Soft is applied to individual constraint expressions, not the entire constraint block, providing fine grain control for each constraint expression.

B. Soft Constraint Examples

Figure 1 shows a simple example of a soft constraint being honored.

```
class A;
  rand int x;
  constraint A1 {
     soft x == 10 ;
  }
endclass

A obj = new ();
obj.randomize() with {x inside {[8:12]}; };

=> Result:  solver generates x == 10
```

Figure 1 – Soft Constraint Honored Example

There is a soft constraint (x == 10) and a hard constraint (x inside { [8:12] }). The solver will apply maximum satisfiability on constraints and since the soft constraint does not cause a constraint conflict in the presence of the hard constraint (i.e. 10 is inside the range between 8 and 12), the soft constraint is honored. The solver generates a value of 10 for the random variable x.

Figure **2** shows a slightly different example with a dropped soft constraint.

```
class A;
  rand int x;
  constraint A1 {
     soft x == 10 ;
  }
endclass

A obj = new ();
obj.randomize() with {x inside {[5:9]};};

=> Result: solver generates x==5,6,7,8 or 9
```

Figure 2 – Soft Constraint Dropped Example

If the constraint expression inside the constraint block "A::A1" were not declared as soft, the call to randomize would have failed as there would be no solution that would satisfy both x == 10 and x inside the range of [5:9]. To resolve the solver failure, the user would need to introduce some additional procedural code to turn off the constraint block "A1" or extend class A to override the constraint block "A1" with additional constraints. Either way, it makes the test more complicated.

With the constraint expression (x == 10) declared as a soft constraint, this soft constraint automatically gets turned off by the solver because if it were honored, it would have been in conflict with the hard constraint (x inside { [5:9] }). The outcome is a simpler test.

III. SOFT CONSTRAINT SEMANTICS

Similar to the soft constraint definition in the *e* verification language, SystemVerilog soft constraints are often used to specify default values and relations that can be disregarded in the presence of conflicting hard or other soft constraints. However there is one area where we must extend the semantics for SystemVerilog. *e* always uses the file order to determine priority when multiple soft constraints conflict. This is not possible in SystemVerilog. While SystemVerilog requires a limited file order dependency (base class must be compiled before a derived class), it does allow packages and units to be compiled in an arbitrary order to a certain extent. For this reason, SystemVerilog soft constraints must have a well-defined priority scheme for determining which constraints are disabled if there are conflicts between two or more expressions.

A. Soft Constraint Priorities

Soft constraints are assigned priorities and this is an important concept to understand when and how a soft constraint is honored or dropped. Obviously, the hard constraints must always be satisfied. If there are two soft constraints, SC1 and SC2, and SC2 is a higher priority constraint than SC1, and if both constraints SC1 and SC2 can be satisfied, then they will both be honored. Otherwise, if SC2 can be satisfied, SC1 will be dropped. If the presence of SC2 will cause a constraint conflict, SC2 will be dropped and SC1 be honored, if possible. Otherwise both SC1 and SC2 will be dropped.

Mantis 2987 proposal [3] to SV-EC IEEE 1800 committee defines the priorities of soft constraints:

- Constraint expressions that appear later in the same construct (constraint block, class, or struct) have higher priority.
- Constraint expressions in out-of-body constraint blocks whose prototypes appear later in the class have higher priority.
- Constraints in contained objects (rand class handles) have lower priority than all constraints in the container object (class or struct).
- Constraints in objects whose handles appear later in the container object have higher priority
- Constraints in derived classes have higher priority than all constraints in their super classes
- Constraints within inline constraint blocks have higher priority than constraints in the class being randomized.
- Latter iterations within a foreach constraint have higher priority than former iterations

Consider the following example in Figure 3:

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10;
}
endclass

class N extends M;
  constraint b;
  constraint c { soft x == 5; }
endclass

constraint N::b { soft x == 9; }

class Q ;
  rand N n;
  constraint d { soft n.x inside {[5:8]};
}
endclass

Q obj = new();
obj.randomize() with { soft n.x >= 7; };
=> Result: solver generates x = 7 or 8
```

Figure 3 – Soft Constraint Priority Example

All constraints in the above example are soft constraints. The priorities for the soft constraints are as follows:

Highest

x >= 7	inline constraint
x inside { [5:8] }	Q::d in the container class
x == 5	N::c (appear later)
x == 9	N::b, out of body (appear earlier)
x < 10;	M::a (appear later)
x > 2;	M::a (appear earlier)

Lowest

Not all soft constraints above can be satisfied as it is clear that to satisfy (x >= 7), the highest priority soft constraint in this example, (x == 5) and (x == 9) constraints could not have been satisfied. For this reason, these two soft constraints (x == 5) and (x == 9) are dropped; the other soft constraints are honored.

The solution for x, considering all the honored soft constraints, is { [7:8] }.

B. Discarding Soft Constraints

Sometimes a test writer would like to disable or turn off soft constraints on a variable. This can be accomplished using the '*disable* soft' construct as shown in Figure 4. The disable soft construct removes any existing soft constraints on a variable so new constraints can be applied. This is important when a soft constraint restricts variable values to a range and the test level constraints want to expand on this range.

```
class M;
  rand int x;
  constraint a { soft x > 2; soft x < 10; }
endclass

M obj = new();

obj.randomize() with { x inside {[0:20]};};
=> Result: x == 3...9

// disable soft constraints on 'x'
obj.randomize() with {
  disable soft x;
  x inside {[0:20]};
};
=> Result: x == 0...20
```

Figure 4 – Disable Soft Example

IV. METHODOLOGY

A. Stimulus Solution Space

Constraints are declarative in nature and specify a set of equations or rules that the solver will use to generate test stimulus. It is useful to draw a diagram of the possible solution space to describe how the VIP creator and test writer will modify this space for their needs as shown in Figure 5.

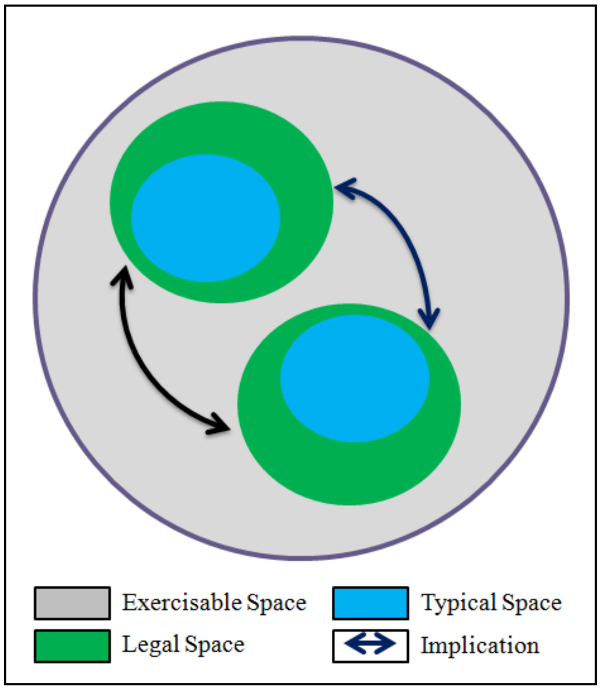

Figure 5 – Solution Space

We have split the diagram into several areas to describe the test space. (These areas are not part of the SystemVerilog standard, but are useful to describe soft constraint methodology.)

a) The entire solution space possible for the random variables with no constraints present. This is the set of stimulus that would be created if no restrictions were placed on the stimulus class being randomized. This range is often not useful at all, as predictable behavior from the DUT is not defined. Examples are CPU stimulus with an infinite loop, corrupt configuration registers or packets of almost infinite length. This is represented by the outer box in Figure 5.

b) The VIP code typically contains a set of "exercisable space" constraints, which define the stimulus where a predictable response from the DUT is defined. This range will contain many cases where the stimulus violates the protocol, yet a well-defined error response is produced. Using these constraints the testbench will create the legal and error stimulus that is possible for the protocol.

c) A set of "legal space" constraints will generate valid stimulus, without any error injection. A default test with only these constraints would eventually generate all legal input stimuli.

d) The VIP code also constrains a set of "typical space" constraints. These will cause the stimulus to be more useful and coverage for the protocol or the design will be hit more quickly using these constraints.

e) Variables in the constraint space are often related to other variables via implication. This is shown in the diagram by a bidirectional arrow linking the two variable range spaces.

The test writer typically defines two sets of constraints at the test-level.

f) Test specific constraints that further restrict the range of values generated. These are biases or directives for a specific test to increase the chance of hitting a coverage point or corner case. These are the typical constrained-random tests that are part of a level-1 or level-2 test suite.

g) Test specific constraints that change or expand the range of values generated to some range outside the "typical space" to hit a corner case. These are typically constrained-random tests created during the coverage closure process, where corner cases are targeted by adding constraints.

B. Comparison of Constraint Modification Mechanisms

The SystemVerilog language provides several mechanisms for modifying, overriding or disabling constraints. We will review each of these constructs and then discuss issues with each in relation to the (f) and (g) requirements above.

i. *Inheritance with different constraint block name.*

If constraints are specified in the derived class in a constraint block whose name is different than any constraint block in the base classes, then the new constraints add to the existing constraint set and all are solved when the class is randomized, as shown in the long_packet class in Figure 6.

ii. *Inheritance with the same constraint block name.*

If the constraint block name in the derived class is the same as a block name in a base class, then the constraints in the base block are replaced by those in the derived, as shown in the error_packet class in Figure 6. To use this strategy to accomplish overriding, one must know the name of the original constraint block, and that original block must not contain other constraints that are not intended to be overridden.

```
class packet;
  rand bit [15:0] len;
  constraint valid_len {
     len inside {[0:1500]};
  }
endclass

class long_packet extends packet;
  constraint long_len {
    len > 1000; // used with valid_len
  }
endclass;

class error_packet extends packet;
  constraint valid_len { //overrides
    len inside {[1501:1600]};
  }
endclass
```

Figure 6 – Parent/Child Class Constraints

iii. *Procedural calls to constraint_mode()*

As constraint blocks are named in SystemVerilog, each block can be enabled or disabled using procedural code. The test writer can simply disable any constraint block explicitly in the test, as shown in Figure 7. To use this strategy, one must know the name of the original constraint block and that the original block must not contain other constraints that are not intended to be overridden, and because constraint_mode() is set on an instance, it must be repeated for all instances where the override is required.

```
class packet;
  rand bit [15:0] len;
  constraint valid_len {
     len inside {[0:1500]};
  }
endclass

// Test invalid length 1500-1600.
// Disable valid_len cst (0-1500)
// Gen packets with len 1501-1600

pkt.valid_len.constraint_mode(0);
pkt.randomize() with {
   len inside {[1501:1600]};
};
```

Figure 7 – constraint_mode()

iv. Dist to approximate soft constraints.

A distribution constraint with extreme bias will be honored if possible yet will not cause a conflict even if as few as one of the distribution alternatives overlaps with other constraints. If the desired typical space is given a very large weight and the exercisable space is given a very small weight, then the test writer can in effect override the constraint specified as dist without disabling or re-defining the constraint block, as shown in Figure 8. There is, of course, a small chance that the typical space would not be chosen.

```
class packet;
  rand bit [15:0] len;
  constraint valid_len {
    len dist {[0:1500] := 10000,
              [0:1600] := 1};
  };
endclass

packet pkt = new();
pkt.randomize();
=> Results: len very likely in 0:1500

// overlapping range
pkt.randomize() with {
  len inside {[1400:1600]};
};
=> Results: len very likely in 1400:1500

// conflict range
pkt.randomize() with {
  len inside {[1501:1600]};
};
=> Results: len in 1501:1600
```

Figure 8 – *dist* Approximating Soft Constraint Example

To satisfy the requirements for the test to further restrict range (f), the inheritance with a differently named constraint block can be used. This simply adds additional constraints and further restricts the solution space.

The requirements for (g) are quite different – here we want to disable one or more VIP constraints to allow the test constraints to define the space. We are effectively enlarging the space in some area above and beyond what was specified as typical by the VIP developer.

There are several problems with the existing solutions:

a) The user must determine which constraints need to be overridden or disabled, which is often not easy to calculate or intuitive to code. In many cases the constraints dropped will depend on state variables so cannot be calculated statically and coded into the test.

b) The user must know the name of the constraint block in order to override it or disable it. That puts a burden on documentation, especially if the block is in encrypted code.

c) If a constraint block is overridden or disabled to remove a particular constraint, any other constraints in that block need to be re-specified. This suggests a methodology of always putting constraints in their own block if they are known to be subject to being disabled. Such a methodology can be hard to enforce and maintain (a later decision that a constraint should be possible to disable means blocks would be subsequently partitioned).

d) For the override or disable solutions, even though the original intent may have been to allow the test to not use the constraint, the test still has to write extra code to make that happen.

e) If a conflicting constraint is presented in the "with" clause of a randomize call, there is no derived class within which to apply the override.

f) The solution of using a skewed dist constraint can fail with a finite probability. Distribution issues can arise if a second dist is specified (e.g. in the test) for the same variable. The constraint solver will determine some dist variables to relax, however there is no control over the priority and no way to specify which dist variables should be relaxed in the test.

C. Soft Constraint Methodology/Solution

Soft constraints solve all of the issues listed above when a test writer needs to expand the solution space.

Soft constraints that do not conflict are simply additive, and follow the same rules are regular constraints, satisfying requirement (f).

When a soft constraint is present but a conflict occurs, then the soft constraint may be disabled by a regular (hard) constraint or another soft constraint with higher importance.

This allows the VIP creator to specify some constraints as soft, knowing that these may then be disabled by the test writer using either a hard constraint or another soft constraint with higher priority.

The VIP creator also knows that the soft constraint will be disabled only if a conflict occurs, which is determined at runtime based on the state variables.

No false failures due to a small but finite probability of a dist choosing a heavily biased value can occur. The constraints either conflict and are disabled or do not conflict and remain. Results are predictable and failures do not occur due to specific seeds being used.

Finally, if a soft-constraint in the VIP must be disabled, this can be accomplished using the "disable soft" construct. This causes the soft constraints on a specified variable to be disabled across all constraint blocks (as was shown in Figure 4).

D. Soft Constraint Examples

The concepts discussed above will now be explained with a set of simple examples, to illustrate the points in the context of a verification environment.

Each of the requirements from the VIP and test-writer needs to be mapped to these language features in a well-defined manner so the test-writer can understand how the test will affect the underlying VIP code.

i. Exercisable Space

The exercisable space is the stimulus the DUT will return a predictable result. This space should be defined using regular (hard) constraints. These constraints will not be disabled by the tool unless an explicit call to *constraint_mode*() is made to turn them off for some design specific reason.

Soft constraints should not be used for this purpose, as it is too easy to accidentally override them and cause illegal stimulus to be injected into the design, wasting considerable time debugging false failures.

An example is shown in Figure 9 where the packet length is a 16-bit field. We want to inject packets larger than 1500 allowed by the protocol but do not want to inject packets of very large length that would be slow to simulate or hang the test.

```
class packet;
   rand bit [15:0] len;
   // Eth protocol has len 0-1500
   // DUT timeout at 5000 bytes
   // Exercisable 0..6000 bytes
   constraint exercisable_len {
     len inside {[0:6000]};
   }
endclass

packet pkt = new();

// Test valid/invalid lengths
// Packets with length 1400-1600
pkt.randomize() with {
    len inside {[1400:1600]};
};
=>  Results: len in 1400:1600
```

Figure 9 –Exercisable (Hard) Space Constraints

ii. Legal and Typical Space

The legal and typical space should be defined using a mix of hard and soft constraints. The soft constraints should be used for areas where you want the test writer to easily override the settings to expand the solution space.

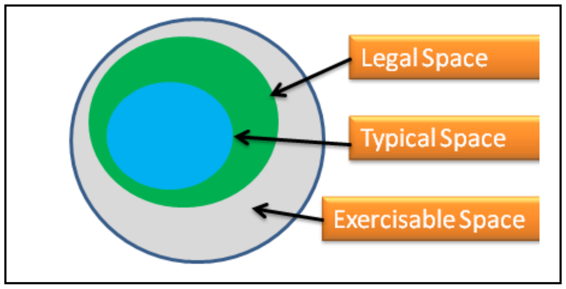

Figure 10 – Legal and Typical Space

```
class packet;
  rand bit [15:0] len;
  // Protocol len 0..1500
  // Use 0..1600 for testing
  // of valid + error traffic
  constraint legal_len {
    len inside {[0:1600]};
   }
endclass

packet pkt = new();

pkt.randomize();
=>  Results: len in 0:1600
```

Figure 11 – Legal Space Constraints

```
class typ_packet extends packet;
  // Bias towards interesting len
  constraint typical_len {
    soft len dist {
      [   1:   5] :/ 30,  // short
      [   2:1498] :/ 40,  // med
      [1495:1500] :/ 30}; // long
  }
endclass

typ_packet typ_pkt = new();

typ_pkt.randomize();
=> Results: distribution shown
```

Figure 12 – Typical Space Constraints

iii. Corner Case Space

To close coverage, additional stimulus values need to be injected into the DUT to hit specific corner cases. This usually requires some slight expanding of the typical space as shown in Figure 13.

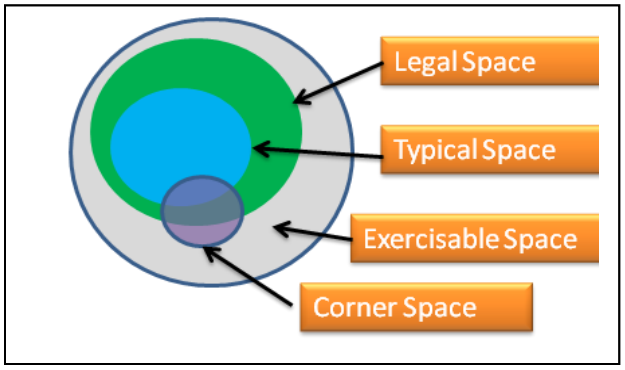

Figure 13 – Corner Case Space

Simply adding a constraint with the new extended range will not work as the solver can find a valid solution in the intersection of typical and corner space. This is shown in Figure 14 where the interesting packet length range of 1490-1510 does not conflict with the typical space constraints in Figure 12.

```
class err_packet extends typ_packet;
  // Generate error lengths
  constraint error_len {
    len inside {[1490:1510]};
  }
endclass

err_packet err_pkt = new();
err_pkt.randomize();

=> Results: len = 1490:1500
```

Figure 14 – Corner Case Constraints

As the hard constraint to generate packets of length 1490-1510 does not conflict with the earlier soft constraints nothing is dropped. The solver can find a solution using all constraints, and returns length 1490-1500. This may not be the desired test behavior. Functional coverage provides an important check that the full range of desired stimulus was applied.

To generate the error lengths, where the error space partially overlaps with the previous ranges, we need to remove the soft constraint. The *disable soft* function can be used to perform this action. Calling *disable soft* on a variable causes all soft constraints on that variable to be dropped. This allows the new constraint to now define the required space as shown in Figure 15.

.

```
class err_packet extends typ_packet;

  // Generate error lengths
  constraint error_len {
    disable soft len;
    len inside {[1490:1510]};
   }
endclass

err_packet err_pkt = new();
err_pkt.randomize();

=> Results: len = 1490:1510
```

Figure 15 – Overlapping Error Solution Constraints

iv. Implication Constraints

One cannot apply all of the necessary solution space shaping using the relational and distribution constraints shown so far. A very important category of constraints is the implication constraint, which can be implemented with either the "->" operator or an "if-else" constraint. Let's look at how the concept of soft constraints works in conjunction with implication constraints.

One permutation is that there is a hard implication constraint and a soft constraint referencing the same variable. In this situation, it does not matter whether the variable in question is in the left or right side of the implication; if the possible range of the variable after consideration of the implication constraint allows the soft constraint to be satisfied, the soft constraint will be considered, and otherwise it will be ignored. This is illustrated in Figure 16. So any critical implication constraint applied by the test or by the environment is certain to be honored, and the typical space specified by soft constraints will be honored if possible. Be aware that a reduction in the range of the other variable (not the soft constrained one) in the implication may be necessary to find a solution (see the value of y in Figure 16).

```
class tst_packet extends typ_packet;

  constraint x_y_len {
    x < 2 -> len inside {[1490:1510]};
    y < 2 -> len > 1500 ;
   }
endclass

tst_packet tst_pkt = new();
tst_pkt.randomize();

=> Results: len in 1490:1500 if x < 2
or len uses distribution from typ_packet
otherwise; y >= 2 (assuming no other
hard constraint on y)
```

Figure 16 – Hard Implication Constraints

A variation which is likely less common is that there is an implication constraint that itself should be considered a soft constraint. Perhaps a variable x has a typical space only when another variable y has certain values. In that case, you can create a constraint of the form "y inside {[…]} -> soft x inside {[…]}". Now if the left-hand side expression is true, the constraint on the right-hand side will be honored if possible. Unlike the hard implication constraint, if there are other constraints that make the right-hand side impossible to satisfy, values can still be chosen that cause the left-hand side to be true. A soft implication constraint is illustrated in Figure 17.

```
class typ2_packet extends packet;
  constraint typical_len {
    x < 2 -> soft len inside {[0:100]};
   }
endclass

typ2_packet typ_pkt = new();

typ_pkt.randomize() ;
=> Results: len in 0:100 if x < 2,
otherwise unconstrained

typ_pkt.randomize() with
   {len > 100; x < 2};
=> Results: len > 100 and x < 2, soft
constraint ignored
```

Figure 17 – Soft Implication Constraints

V. Debug

Debugging soft constraints introduces some additional complexity. Previously all constraints were solved correctly or a conflict was reported. The conflict could then be debugged and changes made to the source code.

With soft constraints, some of the constraints may be dropped due to a conflict, and this can then affect the resulting distribution or solution set. A tool supporting soft constraints will need to have a capability to debug this functionality.

The debug functionality must show which constraints are dropped and which are honored. This can be done in a similar manner to showing when a constraint block is overridden due to inheritance. One key difference is that an overridden block due to inheritance disabled the entire block, while soft constraints are disabled at the individual equation level.

The tool must report which soft constraints are dropped, and which other conflicting constraints caused this behavior. Some graphical way of displaying this information is preferred.

Finally, it is useful to interactively debug soft constraints by stopping in the simulator GUI, and then be able to re-randomize the result with some soft constraints converted into hard constraints. This allows conflicts to be reported and show why a soft constraint could not be satisfied.

All of these features should be available with links to source code, waveforms, class browser, object browser and local watch panes.

VI. Conclusion

The addition of soft constraint to the SystemVerilog language provides an effective and efficient way to specify constraints in the VIP layer that can easily be overridden by the a test writer. The priority of which constraints are dropped is controlled in a well defined manner and allows the VIP creator and test writer to determine which items will be dropped in a predictable way. The software tools can assist with debug of soft constraints by providing various GUI and interactive features.

VII. References

[1] "IEEE Standard for SystemVerilog – Unified Hardware Design, Specification and Verification Language" IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2009
http://standards.ieee.org/findstds/standard/1800-2009.html

[2] "IEEE Standard for the Functional Verification Language e" IEEE Computer Society, IEEE New York, NY. IEEE Std 1647-2008
http://standards.ieee.org/findstds/standard/1647-2008.html

[3] "Mantis 2987 [proposal to SV-EC IEEE 1800 committee] P1800-2012",
http://www.eda.org/mantis/file_download.php?file_id=5478&type=bug