3 Generation Concepts

This chapter describes the basic concepts of generation and the terminology used in this document:

- "Generation Terminology" on page 3-1
- "Specifying IntelliGen as the Default Generator" on page 3-15
- "Generative list Pseudo-Methods" on page 3-19
- "Generation and Test Phases" on page 3-21
- "Real Number Generation with IntelliGen" on page 3-23

3.1 Generation Terminology

3.1.1 Constraints

Constraints are rules that define the values that can be assigned to a field, for example:

```
keep x == 5;
```

3.1.2 Constrained Random Generation

Generation is the Specman process that allocates memory for all scalar, struct, unit and list fields defined in **sys**, solves user-defined constraints on those fields, and assigns values to them. You can restrict the possible values for a data element, the size of a list, and other generated items by specifying constraints on the items.

The degree to which generated values are random for an item depends on constraints specified on the item. With no constraints for an item, the value for the item is completely random. The more constraints there are on an item, the less random its values will be. Some items can be fully random (no constraints), some can be fully constrained (with constraints like keep x == 5), and some can be constrained random (with constraints like "keep x in [2..8]").

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

3.1.3 Generation Action

The value for any particular item can be generated at the beginning of the test (pre-run generation) or during the test (run-time generation, usually called on-the-fly generation). A generation action is a specific invocation of the generation process, initiated by a **gen** or **do** action. Pre-run generation, because it is an implicit **gen sys**, is also a generation action.

3.1.4 Generation Context

Within IntelliGen, a generation (or gen) context is a set of CFSs which represent a complete and independent generation tree. There are two types of gen contexts:

- **gen** and **do** actions (including **sys**)
- struct and unit types

For example, after executing IntelliGen's **gen lint -i** command, the data displayed in the Gen Debugger's ICFS window is presented in the scope of gen contexts (either **gen** actions or struct/unit types).

3.1.5 Reduction

An activity of the generator that narrows the possible values for a field based on the constraints applied to that field.

3.1.6 Assignment

An activity of the generator that assigns a value to a field.

3.1.7 Distribution Policy

An activity that assigns weights to values in the range of possible values for a field before the assignment is made. These weights affect the chance of those values being assigned to the field. Distribution policies are determined by **soft select** constraints.

3.1.8 Connected Field Set (CFS)

Within a generation action, constraints create relationships between the ranges of legal values of some of the fields being generated. A set of fields in a generation action that is connected by a set of constraints is called a *connected field set (CFS)*.

A CFS has the following attributes:

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

- Exclusivity— for any given generation action, a generatable field is a member of one and only one CFS.
- Completeness—all fields that are generated by the same action and connected by constraints (directly or indirectly) are placed in the same CFS.
- Generation at once— for any given generation action, all fields in a CFS are generated at the same time.
- Unified input state—the same values of the same set of sampled fields is applied to all fields in a CFS.
- A building block of a generation action— a generation action consists of the sequential generation of a set of CFSs.

Notes

- There are cases where the fields in a CFS are not generated all at once. For more information, see Chapter 5 “Inconsistently Connected Field Sets (ICFSs)”.
- A CFS must contain at least one field. If no constraints are applied to a field, and the field is generated randomly based on its type alone, it can be in a CFS by itself. An example of this type of CFS is a CFS containing a pointer.

Example

In the following example, there are two CFSs, as shown in Figure 3-1:

- CFS #1 consists of kind, color, data, and header.color.
- CFS #2 consists of cell_size and header.addr.

Notes

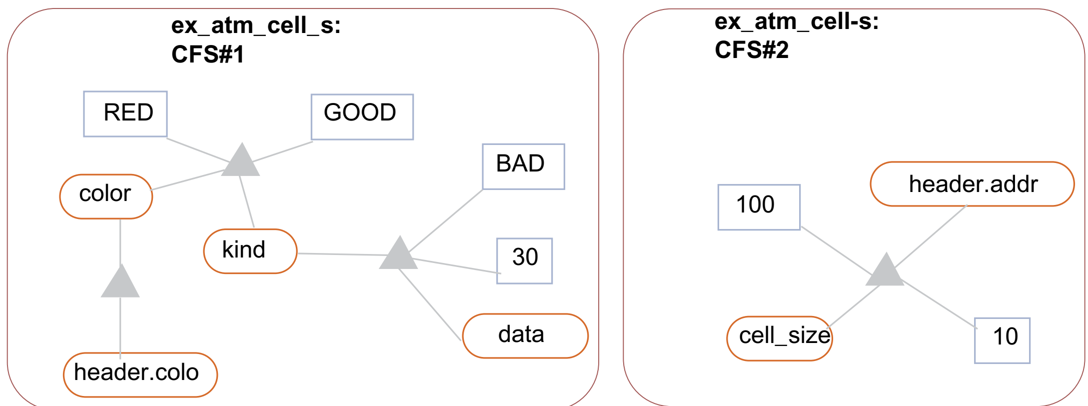
- Within a single struct, different fields can belong to different CFSs. In this example, the fields “cell_size” and “kind” belong to two different CFSs.
- The same CFS can contain fields from different places in the hierarchy generated by the action. In this example, both CFSs contain fields from ex_atm_cell_s and from ex_atm_cell_s.header — two structs located differently in the hierarchy.
- The fields in each one of these CFSs are generated at the same time. Therefore, the generation order of “color” and “kind”, for example, is not a consideration.

```
/// test1.e
<'
type ex_atm_color_t: [RED, GREEN];
type ex_atm_kind_t: [GOOD, BAD];
struct data_item {};
struct ex_atm_cell_s {
  data: list of data_item;
  color: ex_atm_color_t;
    keep color == RED => kind == GOOD;     // kind and color in the same set
  kind: ex_atm_kind_t;
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        keep kind != BAD => data.size() > 30; // data joins the set of kind
      header: ex_atm_header_s;
      cell_size: uint[0..1000];
        keep cell_size < 100 =>
                header.addr.size() < 10;      // cell_size and header.addr in
                                              // a new field set
        keep header.color == color;          // header.color joins set of color
    };
    struct ex_atm_header_s {
        addr: list of bit;
        color: ex_atm_color_t;
    };
    '>
```

Figure 3-1 Example of Connected Field Sets for ex_atm_cell_s



Legend

Constraint

Generatable field

Input value (constant, method, global, or do-not-generate field)

3.1.9 Relationships between Fields in a Generation Action

For any generation action, the relative position in the environment hierarchy of the generated or referenced entities helps to determine whether it is safe to sample or modify an entity during the generation process from user procedural code. The coding guidelines in this document are articulated in terms of the relative position between items in a generation action.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

Example

This example illustrates the terms used to define the relative position of fields in a generation action.

In the following example, **sys** has two unit instances, “config” and “env”, which are created during pre-run generation. The “env” unit has a nested struct, “next_item”, which is marked do-not-generate. It is generated on-the-fly by the drive_item() TCM, and thus comprises a separate generation action.

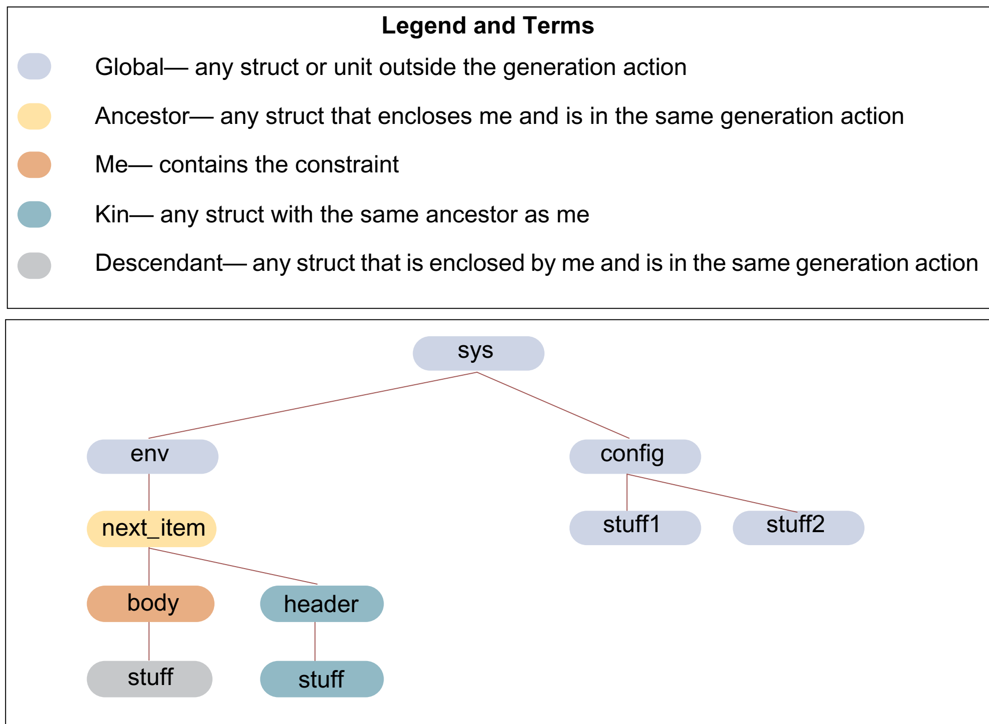
```
/// test2.e
<'
type env_mode_t: [yours, mine];
extend sys {
  config: config_u is instance;
  env: env_u is instance;
  event clk;
};
unit config_u {
  mode : env_mode_t;
  bus_width : uint;
  data_width : uint;
    keep data_width == calc_width(mode, bus_width);

  calc_width(m:env_mode_t, bw:uint) : uint is {
    return bw * m.as_a(uint);
  };
  stuff1: stuff_s;
  stuff2: stuff_s;
};
unit env_u {
  num_items: int [0..20];
  !next_item: data_s;
  drive_item()@sys.clk is {
    for i from 0 to num_items -1 do {
      wait ([100]*cycle);
      gen next_item;
    };
  };
  run() is also {
    start drive_item();
  };
};
struct data_s {
  header: header_s;
  body: body_s;
};
struct body_s {
  dwidth: uint;
--line below causes WARN_GEN_LINTER_G1
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        keep dwidth == sys.config.data_width;
      stuff: stuff_s;
    };
    struct header_s {
      h_field: uint;
      stuff: stuff_s;
    };
    struct stuff_s {
      s_field: uint;
    };
    '>
```

Figure 3-2 shows the terms used in this document for fields generated or referenced in the **gen** action of drive_item().

Figure 3-2 Terms for Fields in a Generation Action



© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

Notes

- The global field “sys.config.data_width” is part of the input state of the CFS that includes “body.dwidth”. It is safe to assume that an appropriate value has been applied to the global field before the constraint in “body” is applied, because “config” and its fields are generated during pre-run generation.
- If you make the ancestor struct (“next_item”) generatable during pre-run generation, then “sys.config.data_width” and “body.dwidth” are generated in the same action. To ensure that calc_width() is called to calculate the width in this case, you might need to modify the constraint in “body” to “keep dwidth == read_only(sys.config.data_width)”.
- Any field enclosed in a struct is considered a descendant of that struct, unless it is generated in a separate **gen** action.

3.1.10 Input State

An input to a generation action is an expression that affects the generated results but is not affected by them. Examples of input to a generation action include:

- Expressions containing a unidirectional operator, such as a user-defined method call or a call to certain predefined methods, for example, **read_only()** and **value()**.
- Expressions containing a global path (a field whose pathname starts with “sys”).
- Local fields and variables marked as “do not generate”.

A generation action is dependent on its inputs. In other words, before any variables can be generated, the inputs must be sampled. The specific values of a generation action’s inputs comprise the input state.

3.1.11 Input Sampling

Having a well-defined semantic for when inputs are sampled makes it easier for you to predict how constraints will behave. It is particularly important to understand input sampling when you are writing procedural code to be called during the generation process.

1. In general, IntelliGen samples inputs to a CFS just before solving it.
2. If IntelliGen determines that the value of an input does not affect the solving of a constraint, it does not sample that input.

   For example, if “read_only(a>0)” returns FALSE, then “f()” is not sampled.

   ```
   keep read_only(a>0) => x == f();
   ```

   If it returns TRUE, then “f()” is sampled.

   Note All logic expressions are treated in this manner, not only the expressions in implication constraints.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

3. The sampling order within a constraint is from left to right.

   For example, the following constraints are logically equivalent but behave differently.

```
keep x==0; keep read_only(x>0) => y == f(x) // no error
keep x==0; keep y==f(x) || !read_only(x>0)  // assertion error

with f(a:uint):uint {assert a>0; return a-1;};
```

   For the first constraint, no error is issued. The input “read_only(x>0)” is sampled first. Because its value is FALSE, “f(x)” is not sampled. For the second constraint, “f(x)” is sampled first, resulting in an assertion.

The input sampling process is complex and flexible. A decision not to sample an input can be made based on analysis, not only on the sampling of another input as shown above. In addition, the sampling order is not strictly left to right. For example, given the constraints below, IntelliGen first reduces the legal range of values for “a” and “b”. Because these variables are fully constrained, and because they are the only variables in the right-hand-side expression of the implication constraint, that expression is evaluated. IntelliGen then knows that the sampling of f() is unnecessary:

```
keep x==f() => a < b;
keep a == 10;
keep b == 20;
```

3.1.12 The Semantics of Input Dependency

Open Issue: This section reads like an academic spec or a functional spec. We need to clarify how the user needs to apply this information.

When an input to a generation action is a struct, there is a semantic issue of whether the struct must be fully generated or whether it is sufficient simply to allocate the struct. Whenever possible, IntelliGen chooses to simply allocate the struct because this enhances solvability. Understanding this semantic issue helps you to understand how to sample the descendant fields or call the methods of a struct that is input to a generation action.

The generation flow of a struct field

The generation of a struct field comprises three steps:

1. The allocation of the struct (and calling its pre_generate() method)
2. The generation of its fields
3. Calling its post_generate() method

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

In IntelliGen, once generation of **me** begins, this process can be invoked for fields that are not descendants of **me** before calling **me.post_generate()**, in order to generate fields in an order that suits the user-defined constraints. (In Pgen, once generation of **me** begins, this process is only invoked for descendant fields of **me** before **me.post_generate()** is called.)

Therefore, there are two possible meanings for defining the dependency on a struct:

- Simple dependency means only that the struct's allocation has been performed, possibly together with the allocation of a set of relevant fields.
- Post_generate dependency means dependency not only on the struct's allocation, but also on its full generation, including the call to **post_generate()**.

Simple Dependency

In most cases, dependency on a struct implies a dependency on its allocation. For example, in a constraint such as:

```
keep read_only(p)!=NULL => y == p.x
```

The input (or *determinant*) is the struct field “p”, while the generatable elements (or *dependents*) are “y” and “p.x”. Since the constraint only tests whether or not “p” is NULL, only the allocation of “p” is needed prior to the solving of the generatable elements.

Such inputs can appear also in struct assignments, such as:

```
keep p1 == read_only(p2)
keep x>0 =>  p1 == value(p2)
keep p == lp.first(it!=NULL)
```

In all these cases the dependency is simple, in the sense that the assignment must be done after the allocation of the determinant field (and not after it was entirely generated). In addition, there is an implicit dependency between the allocated struct's fields and their mirror fields in the assigned struct. Thus, in such an example:

```
keep p1 == read_only(p2)
keep y == p1.x
```

When the time comes to generate the field “y”, “p2.x” has already been generated, so that “y” gets the correct value.

Simple dependency allows IntelliGen much more flexibility to handle cases where Pgen fails on an order cycle. The following example is such a case:

```
/// test3.e
<'
struct packet {
    x: uint;
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
    y: uint;
    keep x<1000;
};

extend sys {
    p1: packet;
    p2: packet;
    y: uint;
      keep p2 == read_only(p1);

    top_value: uint;
      keep top_value == 2*p2.x;

      keep p1.y + y < top_value;
};
'>
```

Pgen ends with an order cycle in this example, because it solves the whole tree under a struct field. Because the **read_only()** requires that “p1” has to be generated before “p2”, Pgen cannot solve “p2.x” before or together with “p1.y”.

Post_generate Dependency

While simple dependency enhances solvability, some cases require that dependency on a struct means dependency on its full generation. These cases are identified automatically by IntelliGen, and result from the fact that they depend on operations that are performed at the time of the struct’s generation, and not only at its allocation.

The following two testcases present examples that require post_generate dependency:

Example 1 Method call

In this example, the field “z” is dependent not only on the struct “p”, but also on a call to its method. This call can access any of the struct’s fields, and can have different results each time it is called. This means that the struct “p” must be fully generated before “z” is assigned.

```
/// test4.e
<'
struct packet {
    x: uint;
    y: uint;

    my_sum():uint is {
--line below causes WARN_GEN_LINTER_G1
        result = x + y;
    };
};
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
extend sys {
    p: packet;
    z: uint;

    keep z == p.my_sum();
};
'>
```

Example 2 The dependency of do-not-generate fields

Here the field “sys.z” is a do-not-generate field of the struct “p”. This field can be generated at any time during the struct's generation, including in the post_generate() method itself.

```
/// test5.e
<'
struct packet {
    !x: uint;

    post_generate() is also {
        gen x keeping {
            it < 1000;
        };
    };
};

extend sys {
    p: packet;
    z: uint;

    keep z == p.x;
};
'>
```

Note The post_generate() dependency applies only if the generated field can be generated after the struct’s full generation. If this is impossible (for example, when the generated fields are fields of the dependent struct) the dependency becomes simple.

See Also

- “Passing Fields as Arguments” on page 4-8
- “Input Dependencies” on page 5-6

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

3.1.13 Generation of Subtypes

IntelliGen's approach to generating subtypes is similar to the **gen_before_subtypes(***when-determinant***)** feature of Pgen:

1. Solve the determinant for the when subtype.
2. Solve all the constraints under the when subtype.

Note Even if the **when** determinant is in a separate CFS, IntelliGen performs inferences between the CFS containing the **when** determinant and other CFSs containing the dependent fields.

See "When Subtype Dependencies" on page 5-12 for more information.

3.1.14 Static Analysis

The term *static analysis* refers to the processing of constraints that needs to be performed only once. For example, if a field is generated multiple times, the results of any initial range reduction steps can be reused each time the field is generated.

Unlike Pgen, IntelliGen performs static analysis when necessary as an integral part of the generation process. It is completely transparent to the user.

3.1.15 Generation Errors

A *contradiction* error occurs when the range of legal values for an item is reduced to 0. In the following example, there is no value that satisfies the constraints, and a contradiction error is issued:

```
keep size > 10;
keep size < 5;
```

Order cycle errors occur when the constraint solver determines two conflicting order rules, for example, field A must be generated before field B and field B must be generated before field A. A simple example of an order cycle is the following pair of constraints:

```
keep size == get_size(kind);
keep kind == get_kind(size);
```

The first constraint requires that kind be generated first, in order to determine the correct value for size; the second constraint implies the reverse order.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

3.1.16 Struct Equality and Struct Assignment

Struct equality and struct assignment are two basic, but very different, concepts in the generation of struct fields. The fundamental approach to struct generation, like the procedural approach, treats structs as pointers. Thus, two struct fields are identical if they point to the same location in memory, not necessarily if they have the same value. This principle applies to the operations of struct equality and struct assignment, but these operations differ in other respects.

Struct Equality

Struct equality applies when two struct fields (or list fields) are made equal through a bi-directional constraint. Examples for this syntax are:

```
keep p1 == p2
keep p1 == p2 || p1 == p3
keep read_only(lp.size()) > 0 => p1 in lp
```

The two (or more) fields that are made equal are fully generated, and they represent different paths to the same actual location. This means that constraints on fields of both structs are fully enforced. For example, in the last constraint above, assuming that “p1” is a packet and “lp” is a list of packets, “p1” and one of the structs in “lp” both point to the same struct instance, which is generated while considering any additional constraints on “p1” and the list element.

```
/// test6.e
<'
struct packet_s {
    x:uint;

    post_generate() is also {
        out("generated packet", me," with x= ", x);
    };
};

extend sys {
    p1: packet_s;
    p2: packet_s;

    keep p1 == p2;
    keep p1.x < 101;
    keep p2.x > 99;

    post_generate() is also {
        print p1, p2;
    };
};
'>
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

In the example above, the field p1.x (which is equivalent to p2.x) is generated to 100, as expected. In addition, the post_generate() method is called once, since only one struct has been generated, despite its dual representation.

Struct Assignment

In struct assignment, the struct field is assigned a value of a pointer through a unidirectional relation, such as the value of another generatable field or a new struct created by a method. Examples for constraints that use struct assignment are:

```
keep p1 == read_only(p2)
keep p == foo();
keep p == f(p1) || p == f(p2)
```

Unlike struct equality, once a struct is assigned, the “structural tree” underneath it is not actually generated. This has two main consequences:

- The post_generate() and pre_generate() methods of the struct field that is assigned are not called. (Of course, if it is assigned a value of another generatable field, these methods have been called for the original field.)
- Fields of the assigned struct field are not generated, so constraints on them are not enforced, as shown in the example below. If these fields are in the same CFS as other generatable fields, they are treated as inputs to the CFS.

Example

```
<'
struct packet_s {
    x: uint;
    y: uint;

    keep x == 10;
};

extend sys {
    p: packet_s;
    z: uint;

    keep p.y == 20;
    keep p == foo();

    keep z == p.y;

    foo(): packet_s is {
        result = new packet_s with {
            .x = 5;
            .y = 5;
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
            };
        };
    };
    '>
```

Because “sys.p” is assigned, the constraints on its fields are not enforced. For information on how to require IntelliGen to enforce these constraints, see “Overridden Constraints in Struct Assignment” on page 6-12.

See Also

- “Convert struct equality to struct assignment” on page 10-4 in Chapter 10 “Performance Guidelines”

3.2 Specifying IntelliGen as the Default Generator

Before loading any Specman modules, specify that IntelliGen is the default generator using the following Specman command:

```
cmd-prompt> config gen -default_generator=IntelliGen
```

Note You cannot specify the default generator using the **set_config(gen**, **default_generator**, *value***)** routine in an ***e*** file.

If you are using **irun**, use the **-snset** option:

```
% irun -snset "config gen -default_generator=IntelliGen" ...
```

See Also

- **configure gen** on page 6-23 in the *Specman Command Reference*
- “Configuring Specman Before Loading or Compiling e Code” on page 2-2 in *Running Specman*

3.3 Pseudo-Randomness and Random Stability

IntelliGen test data generation is *pseudo-random*, which means that the random values are repeatable.

IntelliGen uses a seed number to start its random value production. *Random stability* is a Specman feature, enabling users to modify tests written in ***e***, re-generate the tests, and still get the same generated values for unmodified fields (that is, fields unrelated to the modifications).

The **test** command uses its **-seed** option to change the seed. The default seed is 1.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

To specify a specific seed number:

```
test -seed = number
```

To specify a random seed number:

```
test -seed = random
```

Random stability supports coverage driven methodology in two aspects:

- It is essential during debugging that a constrained-random generator reproduces similar results each time the test is run using a given seed.
- A test exhibiting good coverage in one aspect can be extended in another aspect without losing its good coverage.

3.3.1 Random Stability in IntelliGen

Random stability applies to **gen** actions, CFSs, and variables.

A **gen** action generates an instance of an ***e*** type. The fields in the structural tree that this type defines are grouped into CFSs. When the **gen** action is invoked, the CFSs are sorted in a topological order according to the dependencies between them, and are then solved one after the other. Each CFS is solved once per invocation of the **gen** action.

Over a simulation, a **gen** action can be executed several times, creating a sequence of the generated results. A result is the tree generated for this **gen** action. Two trees T1 and T2 are equal if, and only if, the structures of T1 and T2 are the same, and for each two corresponding leaves (for example, the right-most leaf in T1 and the right-most leaf in T2), the values of the leaves are equal.

Between simulations that run with the same seed, Specman guarantees CFS-level random stability for the I’th element in the sequence. (See “CFS-Level Random Stability” on page 3-17 for more information.)

gen actions are identified by their syntactic properties. These properties are the name of the containing type and the name of the containing method (the scope), and the generated path. In a case where the same path is generated several times inside a specific method, the index of the **gen** action is also added to the identifier. See the following example:

Example 1

```
struct packet {
    !d: data;

    run() is also {
        gen d.address; //#1
        ..
        ..
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
            gen d.address; //#2
        };
    };
```

The identifiers of these **gen** actions are packet.run.d.address.0, packet.run.d.address.1.

The I'th invocation is a run-time property. Adding a call to a **gen** action shifts the index in the sequence of all the future calls to this **gen** action.

See Also

- "Random Stability within Method Extensions" on page 3-17
- "Per Instance Random Stability" on page 3-18

3.3.2 CFS-Level Random Stability

For a given CFS, random stability guarantees that the I'th generation of the CFS generates the same values, under the following restrictions:

- No constraints were added to the CFS. This also means that the CFS does not contain new variables.
- In this generation, and in the previous (0…i-1) generations of this CFS, the CFS was input stable.

  Input stable for the i'th generation means that all the inputs that the CFS depends upon got the same values as they got in the I'th generation of this CFS in the previous run. For CFSs that contain constraints under when subtypes, the values of the corresponding when subtypes are considered inputs. For CFSs that contain list elements, the list size is considered an input to the CFS.
- The names of the variables in the CFS were not changed.

3.3.3 Random Stability within Method Extensions

A method is composed of all of its extensions. Extending a method and adding a **gen** action affects **gen** actions with the same generated path in other layers of this method. See the following example:

```
    struct packet {
        !x: uint

        foo() is {
            gen x; //1
        };
    };

    extend packet {
        foo() is first {
        ..
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        ..
        gen x //#2
        };
    };
```

With the extension, the **gen** action in the extension gets index 0. This means that if an extension is added between two simulations, during the second simulation **gen** action #2 gets the random sequence of **gen** action #1 in the original sequence.

3.3.4 Per Instance Random Stability

Random stability is maintained per instance. However, since part of the identifier of the **gen** action is its scope, and the scope is shared between all instances of the containing struct, instances can sometime influence each other in terms of random stability. See the following example:

```
    struct s {
        !p:packet;

        gen_p() is {
            gen p;
        };
    };

    extend sys {
        s1: s;
        s2: s;
        s3: s;

        run() is also {
            s1.gen_p();
            s2.gen_p();   //#line2
            s3.gen_p();
        };
    };
```

In the example above, all calls to gen_p() refer to the same **gen** action. If line #2 is commented out between two simulations, the tree generated for s3.p in the second simulation is identical to the tree generated for s2.p in the first simulation.

3.3.5 Random Stability Notes

Consider the following issues in regards to random stability:

- Random stability is maintained within a major Specman release, including updates and hotfixes. When simulations are run in different major Specman releases (for example, 8.1 and 8.2), random stability can not be guaranteed.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

- Using the Gen Debugger does not affect random stability.
- The Gen Debugger config flags do not affect random stability.
- Some generation config flags can affect random stability. If one of these config flags is changed between generations, random stability is not guaranteed.
- Non-generation config flags (for example, memory config flags) do not affect random stability.

3.4 Generative list Pseudo-Methods

The **list.sum()** pseudo-method can be used in a constraint. It is handled as a generative, bi-directional expression. For example:

```
keep my_list.sum() < 124;
```

Solving **list.sum()** as a bi-directional expression requires IntelliGen to solve the list as a **lace** (see "Performance Guidelines" on page 10-1), so the usage can have a considerable performance impact if used with large lists.

To detect whether a specific environment contains **list.sum()** constraints that could now be solved in a bi-directional way, Specman uses a parse-time notification (GEN_BI_DIRECTIONAL_LIST_SUM). The default setting of this notification is IGNORE. Also, a bi-directional **list.sum()** constraint is printed by the **show lace** command.

A second problem with **list.sum()** is that the distribution is biased, often resulting in lists containing many items with the value 0. To address this problem, you can add a constraint that directs the generation of the list's items towards the average value of the list items.

Example 1 list size and sum are static constants

```
extend sys {
    l[10]:list of uint;
    keep l.sum(it)==100;
    keep for each in l {
        soft it == select {
            90: [5..15];  //direct generation to the average.
            10: others;
        };
    };
};
```

Example 2 list size and sum are dynamic

```
extend sys {
    l1:list of uint;
    S:uint;
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        keep S == 100;
        keep l1.sum(it)==S;

        keep for each in l1 {
        //direct generation to the area around the average:
            soft it >= 0 ;
            soft it <= (2*S)/value(l1.size()) ;
        };
    };
```

3.4.1 Input Rules

The following input rules reflect IntelliGen semantic or performance-related semantic decisions. In these cases, **list.sum()** is not treated as a bi-directional expression:

- Regular input. For example,
  `read_only(list.sum(it)).`
- When there is no generatable path. For example,
  `pkts.sum(value (.x)+value(.y)).`
- When there is more than one list.sum() in a single expression. For example,
  `pkts.sum(.l.sum(it)).`
- When used under keep for each. For example,
  `keep for each in list {it==other_list.sum(it)}.`

3.4.2 Limitations

- **list.sum()** does not work under a **lace** of depth 2, and is not treated as a bi-directional expression in that case. Consider the following example:

```
struct S {
    x:uint;
    lu:list uint;
    keep lu.sum(it)==x;
    keep x == 100;
};

extend sys {
    ls: list of S;
    y: uint;
    keep for each (s) in ls {
        for each in s.lu {
            it < y;
        };
    };
};
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

3.5 Generation and Test Phases

IntelliGen performs a generation action as follows:

1. Partition the fields of the generation action into a set of CFSs.

   This step occurs once at any time after the start of the generation action and before the actual generation of any field in the generation action. The timing is internal to the generator and is not guaranteed in any way.

2. Solve the CFSs one by one, using the following steps:

   a. Perform any static reductions that are valid regardless of the sampled values of the inputs to the CFS.

   b. Compute the input state of the CFS by sampling all the input variables in all the constraints of that CFS.

   c. Compute the valid ranges for all the variables in the CFS, given all the constraints of the CFS.

      This step is called *reduction*, and can be performed either as a single step for all the constraints, or as a set of consecutive steps involving one or more constraints at a time. At each step, a new set of ranges is calculated for the set of fields. This set of ranges is a refinement of the set of previous ranges.

      IntelliGen decides which constraints to reduce and in what order. This decision is internal to the generator and is not guaranteed in any way.

   d. From the valid range computed in the previous step, assign a random value to each variable using a *distribution policy*.

      The default distribution policy is a uniform distribution, unless defined otherwise. You can define a different policy using **soft...select**.

3. Once a CFS is solved, all of its variables are assigned and are accessible to the verification environment.

   The order in which the CFSs are solved is internal to the generator and is not guaranteed in any way.

Timing of Calls to pre_generate() and post_generate()

- **pre_generate()** is called for a struct or a unit, as soon as memory is allocated and the new pointer is assigned.

  A struct's **pre_generate()** runs before the **pre_generate()** of its descendant struct fields, after the **pre_generate()** of its ancestors, and before any of its generatable fields, in any CFS, are solved.

  The order of execution of **pre_generate()** between kin fields is undefined.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

- **post_generate()** is called for a struct or a unit, as soon as all its fields (recursively) are assigned.

  A struct's **post_generate()** runs after all of its generatable fields, in all CFSs, are solved and assigned, after the **post_generate()** of its descendant struct fields and before the **post_generate()** of its ancestors.

  The order of execution of **post_generate()** between kin fields is undefined.

Notes

- As with Pgen, the IntelliGen generation process applies to most ***e*** types, including scalars, units, structs, lists and strings. It does not apply to keyed lists.
- Generating a field of type struct or unit entails one of the following:
  - Creating a new instance of it in memory and assigning a new reference to it
  - Assigning a new reference to an already existing struct of that type
- Generating a list includes
  - Generating its size (handled as a constrained-random variable)
  - Allocating it accordingly
  - Generating each of the list items
- Unconstrained strings are generated as empty strings. IntelliGen can solve constraints such as

```
keep string1 == "aaa";
keep string2 == string1;
```

Example

The following example code has two CFSs.

```
struct packet {
  x: int;
  y: int;
    keep x > y;
};
extend sys {
  p1: packet;
};
```

The CFSs are:

CFS #1 contains "sys.p1"

CFS #2 contains "sys.p1.x" and "sys.p1.y"

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

The example code is generated in the following way:

1. A new struct for **sys** is allocated and its pointer is assigned.
2. The **pre-generate()** of **sys** is called.
3. CFS #1 is solved by allocating a new struct for "sys.p1" and assigning its pointer.
4. The **pre-generate()** of "sys.p1" is called.
5. CFS #2 is solved by:
   a. Computing the input state of the CFS by sampling all the input variables in all the constraints of that CFS. (There are none in this example.)
   b. Computing the valid ranges for "sys.p1.x" and "sys.p1.y", given the "keep x > y" constraint.
   c. From the valid range computed in the previous step, assigning a random value to "sys.p1.x" and "sys.p1.y" using the current distribution policy.

3.6 Real Number Generation with IntelliGen

IntelliGen supports generation of items of type **real**. Only the following Boolean expressions are allowed in constraints on generatable items of type **real**:

```
real-exp == real-exp
real-exp != real-exp
real-exp in list-exp
```

Note

- No other operator is allowed in a Boolean expression involving items of type **real**. For example, the following constraints are illegal:

  ```
  keep my_real1 > my_real2;       // results in an error
  keep my_real1 == my_real2 + 5; // results in an error
  ```

  If you want IntelliGen to treat a real expression as an input, you must use **value()** to identify that expression. For example:

  ```
  keep my_real1 == value(my_real2 + 5); // no error
  ```

  See "Example 1" on page 3-24.

- Automatic casting is not performed. For example, the following constraint is illegal:

  ```
  keep my_real1 == my_int1; // results in an error
  ```

  Explicit casting with **as_a()** is allowed:

  ```
  keep my_real1 == my_int1.as_a(real); // no error
  ```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

- You can use any of the **real** routines in a real expression. For example:

```
keep my_real1 == rdist_uniform(-99.9 , 99.9);
keep my_real1 == floor(real2);
```

Example 1 Identifying a Real Expression as an Input

In the following example, the intention is to generate "my_real2" first and then generate "my_real1". However, IntelliGen does not recognize "my_real2 + 5" as an input expression.

```
<'
extend sys {
    my_real1: real;
    my_real2: real;
    keep my_real1 == my_real2 + 5; // unsupported
};
'>
```

The following error message is issued:

```
***Error: Unsupported generation of 'real' type.
   Only '==', '!=' and 'in <list>' operators can be used on real type during
    generation.
```

To eliminate this error, use **value()** to identify the input expression:

```
keep my_real1 == value(my_real2 + 5); // no error
```

Example 2 Generating Real Numbers in a Sequence

This example shows how to generate and constrain real numbers in a sequence using the **do keeping** action:

```
<'
// BOUNDED SEQ: Using rdist_uniform() to generate items between
// low and high limits.

extend seq_kind :[BOUNDED];
extend BOUNDED seq {
    !pkt:packet_item;
    count:int;
    keep soft count==3;
    low:real;
    keep soft low==2.5;
    high:real;
    keep soft high==7.5;
    body()@driver.clock is {
        for i from 1 to count {
            do pkt keeping {it.data==rdist_uniform(low,high)};
```


© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

```
        };
    };
};
'>


<'
// Invoke the BOUNDED sequence from the MAIN sequence.

extend MAIN seq {
    !my_seq:BOUNDED seq;
    body()@driver.clock is only {
        do my_seq keeping {it.count==10;it.low==1.5;it.high==5.5;};
    };
};
'>
```

See Also

- "Real Number Generation with Pgen" on page 2-15 in the *Specman Pgen User Guide*
- "The real Type" on page 3-16 in the *Specman **e** Language Reference*
- "Routines Supporting the real Type" on page 26-26 in the *Specman **e** Language Reference*
- Chapter 15 "Specman/AMS Integration" in the *Specman Usage and Concepts Guide for **e** Testbenches*

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.

© 2009 Cadence Design Systems, Inc. All rights reserved worldwide.