

2 Generation Semantics

This chapter describes Pgen semantics, including:

- “Soft Constraints” on page 2-1
- “Constraining Struct Instances” on page 2-2
- “Constraining Lists” on page 2-2
- “Constraining Bit Slices” on page 2-7
- ● “Constraints with “or” Expressions” on page 2-12
- “Implication Constraints” on page 2-14
- ● “Constraints under “when” Struct Members” on page 2-15
- “Real Number Generation with Pgen” on page 2-15

Generation tracing and debugging facilities are described later in this guide.

2.1 Soft Constraints

For Pgen, **soft** is an attribute of a simple constraint. In compound constraints (constraints with **or**, **and**, or imply (\Rightarrow)), the **soft** modifier cannot be located in the initial expression of the constraint. Thus the first constraint below is valid, but the second generates a compile-time error:

```
keep x > 0 => soft y < 0;  
keep soft x > 0 => y < 0; // Syntax error in Pgen
```

In contrast, **soft...select** and **reset_soft()** constraints can be used anywhere in a compound constraint.

If a **soft** constraint is defined first on a field and then a **soft..select** constraint is added, the **soft..select** always overrides the previously defined **soft** constraint.

If you specify two contiguous value ranges with **edges**, they are treated as a single range. For example, the generator treats only 1 and 6 as edges in the following code; 3 and 4 are not treated as edges.

```
extend transaction {
  keep address in [1..3,4..6];
  keep soft address == select {
    50: edges;
    50: others;
  };
};
```

2.2 Constraining Struct Instances

You can constrain two instances of the same struct type to have the same contents. Struct equality causes the two struct instances to refer to the same memory location.

Pgen enforces constraints that specify equality between two fields of struct type by generating one of the fields and constraining the items in the second field to be equal to those in the first. Because of this approach, any constraints on items in the second field are ignored. For example, if `packet1` is generated before `packet2`, the following constraint causes any additional constraints on `packet2`'s items to be ignored.

```
keep packet1 == packet2;
```

2.3 Constraining Lists

Since Pgen generates lists as empty by default, it is recommended to either define a list size in the list declaration or define a `list.size()` constraint.

```
keep [soft] my_list.size() == 10;
```

2.3.1 List Equality Constraints

A list equality constraint requires that two lists be generated to contain the same elements in the same order.

However, unlike struct equality, because the two lists are allocated and stored in separate memory locations, subsequent changes to one list do not affect the other list.

In the following example, “list1” and “list2” are first generated to have the same list elements in the same order. However, the `post_generate()` method removes the last element of “list2” and stores it in the field “x”. “list1” remains unchanged.

```
/// test1.e
<'
extend sys {
  list1[5]: list of int;
  list2[5]: list of int;
```

```
!x: int;

keep list1 == list2;

post_generate() is also {
    x = list2.pop();
};
};
'>
```

Note Assigning one list to the other causes the references to the two lists to point to the same memory location. Subsequent changes to either one of the lists are reflected in the other list.

```
/// test2.e
<'
extend sys {
data: list of byte;
!copy_of_data: list of byte;
post_generate() is also {
    copy_of_data = data;
};
};
'>
```

2.3.2 Constraining a List Item

Neither multiple list indexing nor index expressions can be used in constraints without the use of **value()**. For example, to use expressions such as `top[0].dstruct[0].data` in a constraint, you must enclose it with **value()** as follows:

```
keep value(top[0].dstruct[0].data) == 0xff;
```

Unidirectional and Bidirectional List Constraints

Constraints of the following type are bidirectional by default:

```
keep l[i].z == x;
```

This means that the value can propagate from the list item to `x` and vice versa (from `x` to the list item). Such constraints are equivalent to **keep for each** constraints. For example, the above constraint is equivalent to:

```
keep for each in l { index == i => x == it.z};
```

You can require that such constraints be unidirectional by setting the **config gen -list_constraint_is_bidir** option to `FALSE`. In this case, the value can propagate from `l[i].z` to `x` only.

Bound Lists of Port Are Always Bidirectional

The **bind()** attribute for ports is equivalent to a bidirectional constraint. For example, the final line of code in the following example propagates (binds) the “llp” inputs to the “alp” outputs:

```
/// test3.e
<'
unit left_hand {
    llp: list of in simple_port of bit is instance;
    keep llp.size() == 10;
};
extend sys {
    alp: list of out simple_port of bit is instance;
    keep alp.size() == 10;

    left_hand is instance;

    keep for each (p) using index (pi) in alp {
        bind(p, left_hand.llp[pi]);
    };
};
'>
```

Note Bound lists of port are always bidirectional, regardless of the setting of **config gen -list_constraint_is_bidir**.

2.3.3 Constraining a List to Contain an Item

You can use the **in** operator in a constraint to require a list to contain a specified element or elements.

The following constraint requires “list1” to have an element whose value is equal to the value of “x”.

```
keep x in list1;
```

This constraint is bidirectional, meaning that it does not imply a generation order for the item and list. However, the item is always at the last place in the list, regardless of which is generated first, the item or the list.

In this example, x is generated before “list1” and therefore the last item in “list1” is the value of “x”.

Therefore, the following code results in a contradiction with Pgen:

```
/// test4.e
<'
extend sys {
    x: uint;
    y: uint;
```

```
lu: list of uint;
keep x in lu; // last item in lu is x
keep y in lu; // last item in lu is y
keep x != y;
};
'>
```

2.3.4 Constraining All Items in a List

If a constraint in **keep for each** calls a user-defined method, Pgen calls the method for each item in the list.

```
keep for each in data {
    it.x < 100 => it.id == get_id(); // Use of method in a constraint
};
```

Pgen enforces **keep for each** constraints by first generating any generatable items that are referenced under the constraint and then generating the list item. For example, in the following constraint, Pgen generates *x* and *y* and then generates the list item.

```
keep for each in mylist {
    it < x + y;
};
```

This means that any additional constraints on the list item, such as

```
keep index == 0 => it = f();
```

may be ignored.

Notes

- You cannot change the constraint rule during the execution of the loop.
- Items in lists are generated in ascending order starting with index zero. Constraints that use an index expression to refer to other items in a list can only refer to items with lower index values.
- Referencing **prev** while in the first item of the list causes an error.

2.3.5 is_all_iterations()

The **-absolute_max_list_size** configuration option specifies a limit on the number of items created by **is_all_iterations()**. If the number of iterations is greater than this limit, Pgen issues an error, **ERR_GEN_ALL_ITER_SPACE_TOO_LARGE**.

Pgen creates iterations with **is_all_iterations()** starting with the last field in the struct's list of field definitions.

2.3.6 Limitations

Be aware of the following limitations related to constraining lists:

2.3.6.1 Complex Constraints on List Items

Some compound constraints on list items may lead to an undue contradiction.

Examples

```
keep l[0] == 5 or l[0] == 6
```

Also, the combination of soft constraints and list-item constraints may also lead to undue contradiction. For example:

```
keep l[0] == 45;  
keep soft l == {1;2;3;4};
```

Finally, the combination of for each constraints and list-item constraints may also lead to undue contradiction. For example:

```
keep soft l[0] == 1;  
keep for each in l {  
    index == 0 => it == 0;  
};
```

Workaround

In the first example, use **keep for each** in combination with the imply operator (\Rightarrow).

In the second example, constrain the list size to ensure the existence of the constrained item:

```
/// test5.e  
<  
  extend sys {  
    l: list of uint;  
    keep l.size() >= 4;  
  };  
>
```

In the third example, use **for each in** in combination with the imply operator instead of the list constraint:

```
keep for each in l {  
    index == 0 => soft it == 1;  
};  
keep for each in l {  
    index == 0 => it == 0;
```

```
};
```

2.3.6.2 Multiple List-in-List Constraints on the Same List

You cannot use the list-in-list constraint to keep a list in more than one (other) list. This situation is reported as a contradiction.

Example

```
keep l in {1;2;2}; keep l in {2;2;3}; -- causes contradiction
```

Note This situation can be caused indirectly using list equality constraints:

```
keep l1 in l2;  
keep l3 in l4;  
keep l1 == l3; -- contradiction since l1 is in both l2 and l4
```

Workaround

Compute procedurally the intersection of all containing lists and use one list-in-list constraint to keep the targeted list in the intersection.

2.4 Constraining Bit Slices

You can use the bit slice operator in constraints to achieve a variety of purposes. A simple example is using the bit slice operator to constrain the fields of a CPU instruction:

```
/// test6.e  
<'<br>struct cpu_env {  
    instr: uint (bits: 16);  
  
    keep instr[15:13] == 0b100;  
    keep instr[12:8] == 0b11001;  
    keep instr[7:0] == 0b00001111;  
};  
>'>
```

Another simple but useful application of the bit slice constraint is to generate a list of even integers:

```
/// test7.e  
<'<br>struct cpu_env {  
    lint: list of int;  
  
    keep for each in lint {
```

```
        it[0:0] == 0;
    };
};
'>
```

Note Using “it%2 == 0” to generate a list of even integers does not work. Since the “%” operator makes the constraint unidirectional, “it” is generated before the constraint is checked, and a contradiction occurs about 50% of the time.

You can also use a bit constraint to constrain particular bits in relation to each other. For example, the following constraint ensures that only one of the lower four bits of “x” is 1:

```
keep x[3:0] in [1,2,4,8];
```

You can use non-constant bit indices in bit slice constraints, as in the following example, which generates a 4-bit integer with 1s in two consecutive bits:

```
/// test8.e
<'
struct cpu_env {
i: int [0..3];
j: int [0..3];
l: int (bits: 4);

keep j - i == 1;
keep l[j:i] == 0b11;
};
```

Note The ability of Pgen to solve bit-slice constraints is limited. If you do not see the results you expect, for example, if the generated results are not totally random, make the constraint unidirectional by enclosing the bit-slice expression in **value()**:

```
keep for each in list1 {
    it[31:0] != value(pattern[31:0]);
};
'>
```

See Also

- “Bit Slice Constraints and Generation Order” on page 2-9
- “Bit Slice Constraints and Signed Items” on page 2-10
- “Bit Slice Constraints and Soft Constraints” on page 2-11
- “Limitations of Bit Slice Constraints” on page 2-11
- “Debugging Bit Slice Constraints” on page 2-11

2.4.1 Bit Slice Constraints and Generation Order

A generatable item can contain a bit slice reference; however, there are implications for generation order:

Non-constant Bit Indices

Non-constant bit indices must be generated before other items in the constraint. You cannot override this order.

For example, the following constraint

```
keep x[j:i] == y;
```

implies

```
keep gen (j, i) before (x, y);
```

Note A further implication is that constraints like the following, where the bit indices are non-constant and the other items are constant, cannot be solved.

```
keep 125[j:i] == 0b101;
```

Generation of Bit Sliced Items

By default, bit sliced items are generated after other items in the same constraint. You can override this default with a **keep gen** constraint.

For example, the following constraint

```
keep x[j:i] == y;
```

implies

```
keep soft gen (y) before (x);
```

There can be cases where you need to override this default generation order with a **keep gen** constraint. For example, to meet the following constraints, “x” must be generated before “y”:

```
keep y == x[1:0];  
keep x in [1,2,5,6];
```

In order to make this happen, you can add the constraint:

```
keep gen (x) before (y);
```

or you can add the value() routine to the existing constraint:

```
keep y == value(x[1:0])
```

2.4.2 Bit Slice Constraints and Signed Items

Bit slices in *e* are treated as unsigned. It is possible, however, to constrain the value of a bit slice (or any unsigned item) relative to a signed item. In the example below, a bit slice of “x” is constrained by a signed item, “y”:

```
x: int;
y: int (bits:5);
keep x[4:0] == y;
```

There are several implications of constraints that relate a bit slice to a signed item:

- The value of the bit slice is treated as an unsigned integer; in other words, none of the bits in the slice is treated as a sign bit. In the example above, although “x” can be a negative number, x[4:0] is treated as a positive value.
- The value of the signed item is generated as a non-negative. In the example above, “y” will always be generated as a non-negative integer.
- The value of both the bit slice and the signed item must fit into the smaller of
 - The bit width of the bit slice
 - The bit width of the highest possible value of the signed item (This width excludes one bit used to store the sign.)

Example

Given the following integers, “x” and “y”,

```
x: int;
y: int (bits:5);
```

any one of the following constraints requires the value of “y” to be a non-negative number no larger than four bits (the bit width of “y”, minus one bit to store the sign). In other words, the value of both “y” and the specified bit slice of “x” is generated in the range [0..15]. Any upper bits of the bit slice not required to store the value are set to 0:

```
keep x[7:0] == y;    // x[7:4] is 0
keep x[4:0] == y;    // x[4] is 0
keep x[3:0] == y;
```

By contrast, the value of “y” in the following constraint must fit into only three bits (the bit width of the bit slice), so “y” and “x[2:0]” are generated in the range [0..7]:

```
keep x[2:0] == y;
```

2.4.3 Bit Slice Constraints and Soft Constraints

A hard constraint on a bit slice of a variable always overrides a soft constraint on that variable. For example, the intention of the following constraints is to make all the bits of a scalar be zero by default, then set individual bits with bit slice constraints:

```
keep soft x == 0;
keep x[7:7] == 1;    // Doesn't have desired effect
```

These constraints will not have the desired effect as the soft constraint will always be overridden. The only way to achieve this purpose is to apply the soft constraint to each individual bit explicitly:

```
keep soft x[0:0] == 0;
...
keep soft x[31:31] == 0;
keep x[7:7] == 1;
```

2.4.4 Limitations of Bit Slice Constraints

If a bit slice is a function of another bit slice of the same field or variable, in many cases a contradiction occurs.

In the following example, “x” is an argument to the “bit_parity()” function and must be generated before the function is called:

```
keep x[8:8] == bit_parity(x[7:0]); // Usually a contradiction error
```

The result of the function call is then compared to “x[8:8]” and will fail in 50% of the cases.

The workaround is to assign a new virtual field for “x[7:0]”.

```
y: uint (bits:8);
keep y == x[7:0];
keep x[8:8] == bit_parity(y);
```

These constraints cause Specman to generate “y” first, constrain “x[7:0]” to have the value of “y” and constrain “x[8:8]” to have the return value from the bit_parity() method.

2.4.5 Debugging Bit Slice Constraints

For bit slice constraints, the **collect gen** command displays the item’s range list (enclosed in square brackets) together with the item’s bit value representation (enclosed in angle brackets) as shown below:

```
[range-list]: <bit-value-representation>
```

The bit value representation has a single character, either 0, 1, or X, that represents each bit. The characters 0 and 1 indicate that a particular bit must be a 0 or a 1, respectively. The character X indicates that a bit can be either 0 or 1.

For example, the following display describes an 8-bit odd integer within the range 10 to 20 or 50 to 60:

```
[11..19, 51..59] : <00XXXXX1>
```

2.5 Constraints with “or” Expressions

Constraints using **or** consist of multiple alternatives from which Pgen must satisfy at least one. When one of the alternatives of an **or** expression becomes true, the constraint is satisfied, and the other alternatives have no impact on the generation.

Since Pgen cannot always determine which alternative to satisfy for an **or** constraint, contradictions might arise when multiple constraints are specified. For example, consider the following two **or** constraints involving the same two fields:

```
keep x == 1 or y == 1;  
keep x == 2 or y == 2;
```

Pgen cannot induce that both x and y must be in the range [1..2] for these constraints to be met. To help Pgen solve the constraints, you must provide this information, as follows:

```
keep x in [1..2] and y in [1..2];
```

2.5.1 or Constraint Limitations

If the same field appears in all of the alternatives of an **or** constraint, and if at least one of those alternatives relate the field to non-literals, then the generator may not infer the implied constraints and ranges for that field.

Example

```
x == 1 or x in [20..220];
```

is translated correctly into

```
x in [1,20..220]
```

but

```
x == i or x == j
```

depends on the order of generation for i, j, and x. It might lead to a contradiction. For example, if i and j are generated before x, then:

- If the value of *i* is NOT in the currently known range of values for *x*, then $x == j$ is enforced.
- If the value of *j* is NOT in the currently known range of values for *x*, then $x == i$ is enforced.
- If the values of *i* and *j* are both in the currently known range of values for *x*, then the generator does NOT infer the resulting range for *x*, that is $[i,j]$. *x* is generated from its range, which will probably lead to a contradiction.

Note If *i* is generated before *x*, and *x* is generated before *j*, then if *i* happens to be $\neq x$, then $x == j$ (using generated value for *x*) is enforced when *j* is generated.

Workaround

There are several ways to work around this problem. You can define a Boolean field and use that field in implication constraints to control the generation of *x*.

Note You must generate the Boolean field before generating *x*. For example:

```
/// test9.e
<'
  struct packet {
    x_is_i: bool;
    i: uint;
    j: uint;
    x: uint;

    keep x_is_i => x == i;
    keep !x_is_i => x == j;
  };

  extend sys {
    plist: list of packet;
  };
'>
```

For generating on the fly, you can also use a Boolean variable, for example:

```
/// test10.e
<'
  struct packet {
    i: uint;
    j: uint;
    x: uint;

  };

  extend sys {

    run() is also {
```

```
var x_is_i: bool;
var p: packet;
gen x_is_i;
case x_is_i {
  TRUE: {gen p keeping {.x == .i}};
  FALSE: {gen p keeping {.x == .j}};
};
};
};
'>
```

Another solution is to put the **or** alternatives under different when subtypes, for example:

```
/// test11.e
<'
struct packet {
  x_is_i: bool;
  i: uint;
  j: uint;
  x: uint;

  when x_is_i packet {
    keep x == i;
  };

  when FALSE'x_is_i packet {
    keep x == j;
  };
};

extend sys {
  plist: list of packet;
};
'>
```

These workarounds have the advantage of letting you control distribution between **or** alternatives.

2.6 Implication Constraints

Constraints can be written in the form of implications, using the `=>` implication operator. The following constraint means “if color is red, resolution must be in the range 900 to 999”:

```
keep color == red => resolution in [900..999];
```

Pgen translates implication constraints into **or** constraints. The above example becomes:

```
keep color != red or resolution in [900..999];
```

This makes it possible for Pgen to treat implication constraints symmetrically. If Pgen concludes that the expression on the right side of the **or** is false, it makes the expression on the left side true, or if the left side is false, Pgen makes the right side true.

For example, suppose an additional constraint is applied to resolution:

```
keep resolution < 512;
```

Then the expression “resolution in [900..999]” is false, and Pgen treats the remaining alternative, the left expression, as a simple constraint:

```
keep color != red;
```

2.7 Constraints under “when” Struct Members

A constraint under a **when** struct member is considered only if the **when** condition is true. Consider the following example:

```
kind: [good, bad];  
when good pkt {  
    keep size < 64; -- a constraint under when  
};
```

The constraint on the size is considered only if `good pkt` is true (that is, if `pkt.kind == good`). That is, the constraint has the form of an implication, “`good pkt => size < 64`”.

As explained in “Constraints with “or” Expressions” on page 2-12, this is translated into an **or** constraint:

```
keep pkt.kind != good or size < 64;
```

Handling of constraints under **when** is discussed in more detail in Chapter 3 “Generation Order and Architectural Dependencies”.

2.8 Real Number Generation with Pgen

Pgen does not support generation of items of type **real**. When Pgen is used, you must mark fields of type **real** as **non-generatable**. You can use the routines that support real numbers to procedurally generate random numbers for these fields.

Example 1 Generating Real Numbers with Pgen

```
<  
extend sys {  
    !real1: real; // not generatable  
    !real2: real; // not generatable  
};
```

```
post_generate() is also {
    real1 = rdist_uniform(-99.9 , 99.9);
    real2 = floor(real1);
    print sys.real1;
    print sys.real2;
};
'>
```

Results

```
sys.real1 = -82.923
sys.real2 = -83
```

See Also

- “Real Number Generation with IntelliGen” on page 3-23 in the *Specman IntelliGen 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*