10. Constraints and generation

Test generation is a process producing data layouts according to a given specification. The specifications are provided in the form of *type declarations* and *constraints*. Constraints are statements that restrict values assigned to data items by test generation.

A constraint can be viewed as a property of a data item or as a relation between several data items. Therefore, it is natural to express constraints using Boolean expressions. Any valid Boolean expression in ***e*** can be turned into a constraint. Also, there are few special syntactic constructs not based on Boolean expressions for defining constraints.

Constraints can be applied to any data types including user-defined scalar types as well as struct and list types. It is natural to mix data types in one constraint, e.g.,

```
keep my_list.has(it == 0xff) => my_struct1 == my_struct2
```

10.1 Types of constraints

Constraints can be subdivided according to several criteria as follows:

a) Explicit or implicit

   1) *Explicit constraints* are those declared using the **keep** statement or inside **keeping {...}** block.

   2) *Implicit constraints* are those imposed by type definitions and variable declarations.

Examples

```
x : int[1, 3, 5, 10..100];      \\ is the same as:

x : int;
  keep x in [1, 3, 5, 10..100];

l[20] : list of int;            \\ is the same as:

l : list of int;
  keep l.size() == 20
```

b) Hard or soft

   1) *Hard constraints* are honored whenever the constrained data items are generated. A situation when a hard constraint contradicts other hard constraints, and thus cannot be honored, shall result in an error.

   2) *Soft constraints* are honored if they do not contradict hard constraints or soft constraints of the same connected field set honored earlier. If a soft constraint cannot be honored, it is disregarded. (See 10.2.12 for the explanations on how the selection of soft constraints is done.)

c) Simple or compound

   A constraint combining other constraints in a Boolean combination using **not**, **and**, **or**, and => is called *compound*. Otherwise, the constraint is called *simple*.

10.2 Generation concepts

This subclause describes the basic concepts of generation.

10.2.1 Generation action

A generation action is a specific invocation of the generation process, initiated by a `gen` or `do` action. Pre-run generation is also a generation action, and can be considered as an implicit `gen sys action`.

10.2.1.1 Pre-Run Generation Actions

Pre-run generation is initiated before starting the simulation run.

Pre-run generation is the generation of sys, in which sys and all generatable fields within sys, including nested structs, are allocated and generated recursively. Any field prefixed with the do-not-generate character (!) is not generated.

All unit instances must be generated during pre-run generation, so that the unit tree hierarchy is stable for the duration of the run.

10.2.1.2 On-the-Fly Generation Actions

Any field or variable can be generated on-the-fly during a simulation run by executing a `gen` action within a user-defined method.

10.2.2 Generatable variable

A variable that is subject to the generation process and can be constrained is one of the following:

a) Field of a struct or a unit.

b) Local variable that is a parameter of a `gen` action, and its structural descendants, if they exist.

```
var x: int;
gen x;
```

c) List-size of a generatable list.

d) Unit attribute.

10.2.3 Connected Field Sets (CFS)

Within a generation action, constraints create relationships between the fields being generated (or other *generatable variables*). 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:

a) Completeness—every *generatable variable* in a generation action is a member of some CFS. All *generatable variables* that are connected by constraints (directly or indirectly) are placed in the same CFS.

b) Exclusivity—for any given generation action, a *generatable variable* is a member of one and only one CFS.

c) Generation at once—for any given generation action, all *generatable variables* in a CFS are generated at the same time.

d) Unified input state—the same values of the same set of sampled inputs are applied to all fields in a CFS.

e) A building block of a generation action—a generation action consists of the sequential generation of a set of CFSs.

NOTE

— Within a single struct, different fields can belong to different CFSs.
— The same CFS can contain fields from different places in the hierarchy generated by the root gen-action.

10.2.4 Inputs

Expressions in constraints are either generatable or they are inputs to the CFS (that is, they are non-generatable).

Generatable expressions are assigned a value by the current CFS

Input values are not affected by the constraints, but they can affect the values assigned to the generatable items. Thus, the inputs must be evaluated before any generatable items can be generated.

The specific values of a CFS's inputs comprise the *input state*.

An input to a constraint is an expression whose path has one of the following attributes:

— Contains a user-defined method-call

```
keep x == foo(y);
```

— Starts with **sys** (that is, a global or absolute path), for example:

```
keep counter == sys.counter;
```

— Is **me**, for example:

```
keep root_node => parent == me;
```

— Is not in the scope of the current generation action, for example:

```
var my_method_variable:my_struct;
gen x keeping {it == my_method_variable.id};
```

— Contains a call to a predefined unidirectional method or list pseudo-method, for example:

```
keep read_only(z) == p.a.x;
```

For some constraints, it is convenient to assume some of the parameters are always treated as inputs. There are five such kinds of expressions, treating some of their parameters as inputs, even if these parameters represent generatable paths.

— *list segments:* in an expression `l[i..j]`, the segment boundaries *i* and *j* are treated as inputs in the generation of l. Thus, a constraint such as

```
keep ({1; 2; 3; 4; 5})[i..j] in {2; 3};
```

is allowed to cause a contradiction error.

— list in list: The right hand side of a “list in list” or “is a permutation” expression is considered as an input to the constraint.

```
keep list1.is_a_permutation(list2);//list2 is input
```

— soft..select conditions, weights and policies: The condition, weights and policies of a soft…select constraint are inputs, and evaluated before the enforcement of the constraint. The only generatable expression in a soft…select constraint is the expression on which the distribution is applied.

```
keep soft b => x == select {  // b is input, only x is generatable.
      1  : 10;
      y  : z;   //y and z are inputs
};
```

— conditional reset_soft(): The condition of reset_soft() constraint is an input.

— unit instance assignment:

```
u_inst: my_unit is instance;
u_ref:  my_unit;
keep u_ref == u_inst; //u_inst is input
```

10.2.5 Unidirectional and Bidirectional Relations

Bidirectional relations imply that all the generatable fields in a constraint should be solved together in the same CFS. For example:

```
keep x > y;
```

Unidirectional relations on the other hand connect two generatable fields in which there is an implied generation order, for example:

```
keep x == read_only(y);
```

A constraint can have both unidirectional and bidirectional relations. For example, the following constraint contains the unidirectional relations x->y and x ->z and the bidirectional relation y <-> z.

```
keep read_only(x==0) => y==z;
```

In unidirectional relations where there is an implied generation order:

— The field that must be resolved first is called the *determinant*.

— The field that depends on the value of the determinant is called the *dependent*.

For example, in the following example, y is the determinant and x is the dependent:

```
keep x == read_only(y);
```

10.2.6 Inconsistently Connected Field Sets (ICFS)

The generator responds to a generation action by:

1. Partitioning the fields in the gen-action being solved into connected field sets (CFSs)

2. Within each CFS, looping through reduction and assignment until the constraints are solved.

All the fields in a given CFS are solved together.

This process works as long as any two fields are connected only by unidirectional constraints or only by bidirectional constraints. Problems arise when bidirectional constraints directly or indirectly connect two fields that have a unidirectional connection. When this happens, the generator creates *inconsistently connected field sets* (ICFSs), which it might or might not be able to solve.

10.2.7 Order of CFSs

There are two main types of unidirectional connections that imply an order between generatable fields:

a) Structural dependency
   1) Field depends on its containing struct
   2) Field in a subtype depends on its subtype determinant

3) An item in a list depends on the list's size

b) Input dependency

1) Field depends on input (method call, value(), global path) that uses another field as a parameter

10.2.7.1 Structural dependencies

A descendant field is dependent on its structural ancestor (either a struct or a list) because the ancestor (the determinant) needs to be allocated in order to assign a value to the descendant field (the dependent).

For lists, the size of a list must be determined before any list element (or any descendant of that element) is generated.

10.2.7.1.1 when Subtype Dependencies in Constraints

Just like with any structural dependency, any field that is declared under a **when** subtype depends on the value of the **when** determinant. In other words, there is an implicit unidirectional constraint (a subtype dependency) between the **when** determinant and the dependent field:

when-determinant -> dependent-field

If a field is constrained under a **when** subtype, but was declared outside it, the behavior is more robust. If the **when** determinant is also connected to a dependent field directly or indirectly by bidirectional constraints, the **when** determinant is treated as bidirectional, not creating an ICFS. As shown in Example: Subtype Dependency Treated as Bidirectional on page 169, the generator can avoid inconsistent connections by treating the subtype dependency as bidirectional. In these cases, the when determinant and the dependent field remain in the same CFS.

Exception to this rule are cases in which the when determinant is required to be an input of a constraint. Specifically, a **soft…select** or a **reset_soft()** constraint written under a **when** subtype makes the **when** subtype its condition, thus enforcing the **when** subtype to be generated before the constrained field. Another case is of named constraints, where the exact subtype should be determined for the generator to decide which layer of the named constraint should be enforced.

Example: Subtype Dependency Treated as Bidirectional

In this example, the **when** determinant "color" and the dependent field "x" are connected bidirectionally by the `keep color!=YELLOW => x < y;` constraint. Both the when determinant and the dependent field belong to the same CFS, and both constraints are considered bidirectional. No generation order is implied by these constraints.

```
color <-> x <-> y
<'
extend sys {
    p: packet_s
};
struct packet_s {
    color: [RED,BLUE,YELLOW];
    x: uint;
    y: uint;

    keep color!=YELLOW => x < y;
    when RED packet_s {
        keep x < 100
    };
    when BLUE packet_s {
        keep x > 50
```

```
        }
    }
    '>
```

10.2.7.2 Input dependencies

Any constraint containing a unidirectional operator defines an input dependency. For example, in `keep x = value(y)`, `value(y)` is a unidirectional operator. In this example, `y` is the determinant expression and `x` is the dependent expression.

10.2.7.2.1 Dependencies of method-calls

All the parameters of an expression that contains a method-call are inputs to the constraint. A slightly more complex dependency is created for the path of the method-call, thus to the struct or unit that calls the method.

— If the CFS contains no fields belonging to the path, the path should be completely generated, and it's `post_generate()` routine should be called before the CFS that calls the method is solved. This is to enforce that the path is complete and the method-call is evaluated correctly.

— If a CFS contains a descendant of the method-call path, the determinant is the path of the method call, but not its descendents. Thus if the method body uses generatable fields, it is the user's responsibility to pass them as a parameter to the method-call.

For example, for a CFS containing only the following constraint:

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

`p`, and all its fields, and all their fields, will be generated, and `p.post_generate()` will be called before solving the CFS of `y`.

However for the CFS:

```
keep p.x == p.foo();
```

only the `p` object will be generated before the CFS of `p.x` is solved.

10.2.8 Basic flow of generation

Generation can be initiated for any field or variable. For items of struct types, the generation allocates the struct storage and recursively generates all generatable fields of the struct. All fields of a struct are considered generatable, except for the fields prefixed with `!` (see 6.8). There is no specific order in which data items or the fields in a struct hierarchy are generated.

For list items, the generation allocates the list and recursively generates all its elements. There is no specific ordering for whether list items are generated after the size of the list has been fixed or that the items are generated in the order of their indexes. Constraints specified for the items can impose restrictions on the list size or on the items specified earlier in the list.

For scalar types, such as `int`, `uint`, `bool`, etc., the generation only generates the respective value.

The following ordering rules, however, do apply:

a) **pre_generate()** and **post_generate()**

   1) **pre_generate()** of a struct is called after the struct is allocated and initialized using **init()**, but before any of the fields of the struct are generated. In particular, for a struct containing nested

structs, the **pre_generate()** method is called before any of the **pre_generate()** methods of the nested structs.

2) **post_generate()** is called after the generation of all fields of the struct is finished. In particular, for a struct containing nested structs, the **post_generate()** method is called only when all the nested generations are finished.

b) Methods

A method accepting a generatable item as an argument is called after that item is fully generated.

Example

```
struct s {
    a : int;
    b : t;            // 't' is some other struct type
      keep a == f(b)
}
```

The constraint `a==f(b)` implies `b` is fully generated, including the calls to its **pre_generate()** and **post_generate()** before `f` is called on `b`. See also 10.2.9 and 10.2.10.

10.2.9 Using methods in constraints

Constraint paths can include method calls. The syntax is:

[*simple-path.*]*method-name*([*parameter*, ...])[.*trailing-path*]

where *simple-path* does not include method calls and the following restrictions apply:

— If *simple-path* is generatable, then it is fully generated before the method is called.

— Generatable paths used as parameters of the method are fully generated before the method is called.

— For methods returning pointers to structs, the trailing path is sampled after evaluating the method and used as an input of the constraint.

Example

```
struct s {
    x : int[0..5];
    q : t;
      keep x < m(q).y;

    m(param:t): t is {
        result = param
    }
};

struct t {
    y : int[0..5]
}
```

In this example, `q` is generated before `x` and then `q` is used as an input in the constraint `x<m(q).y`. If `q.y` generates to `0`, then the constraint `x<m(q).y` fails.

10.2.9.1 Classification of methods

Methods are classified into the following three categories:

a) Methods that behave like mathematical functions (*pure*). The computed result is entirely determined by the arguments passed to the method. Multiple calls to the method with the same parameters always produce the same result.

   The use of such methods in constraints is safe and unrestricted.

b) Methods that observe the "state of the world," but do not change it. Such method can read fields, signals, global configuration flags, etc., and base the computation on that data. Multiple calls to the method with the same parameters can produce different results.

   When using the methods of this category of constraints the following rules apply:

   1) The method shall not base its computation on the items of the current generatable context, unless such items are passed as parameters to the method.

Example

```
struct packet {
    data     : list of byte;
    checksum : uint;
      keep checksum == calc_checksum(data);

    calc_checksum(data:list of byte): uint is {
        // use 'data' to calculate checksum
    }
}
```

This is correct; `data` is generated before the method is called.

   2) The timing of the call and/or the number of calls to the method cannot be presumed, especially for methods reading values of the real-time or process clocks, operating-system (OS) environment variables, sizes of allocated memory, etc.

Example

```
extend sys {
    l[1000] : list of uint;
      keep for each in l {
        it == read_machine_real_time_clock_msec()
      }
}
```

It is incorrect to assume the method `read_machine_real_time_clock_msec` is called 1000 times, i.e., once for each list element in order (see 10.2.9.2). It is acceptable for the generator to assume this method is a pure function, and thus, call it only once for the list and assign the result to all the list elements. It is also acceptable to assign values to list elements unrelated to their natural order of indexes. Thus (normally in the presence of other constraints), the times read by the method might not be ordered with respect to the list indexes.

c) Methods that observe and change the "state of the world."

   The use of such methods in constraints can create problems. Instead, use the corresponding operations within the **post_generate()** method.

Example

```
struct packet {
    data : list of data_item;

    post_generate() is {
        var id;
```



```
        for each in data do {
            if it.x < 100 then {
                it.id = id;
                id += 1
            }
        }
    }
}
```

In general, it is impossible to classify methods automatically into the preceding three categories. Therefore, the following warnings shall be used if a method calling issue occurs:

— *method call warning #1*: a method used in a constraint contains a non-local path anywhere in its body.

— *method call warning #2*: a method used in a constraint contains an explicit assignment to a non-local path.

10.2.9.2 Number of calls

A method used in constraints can be called zero or more times. The number of calls to a method is irrelevant for the semantics of the constraint if the method behaves as a *pure* function [see 10.2.9.1, category a)]. However, the results of generation can differ depending on the number of calls for the methods with side effects. Therefore, avoid using the methods of category c), and only use methods of category b) with caution.

10.2.10 Generatable paths and the sampling of inputs

The purpose of constraints is to constrain *generatable items*, i.e., those items that can be assigned random values (by the generator) satisfying the constraints. Thus, it is important to define which items are considered generatable and when.

In the context of the initial generation, all fields of **sys** and all fields of nested structs are generatable, except the fields declared as non-generatable (using the `!` prefix).

In the context of a **gen** *item* action (see 10.5.1), *item* is generatable and, if *item* is of a struct type, all its nested fields are generatable—except the fields marked with `!`. If **gen** *item* action applies to a field defined as non-generatable, the *item* becomes generatable; however, any nested non-generatable fields remain non-generatable.

Example

```
struct packet {
    x : int;
   !y : int
};

extend sys {
    p1 : packet;    -- generated during pre-generation
   !p2 : packet;    -- skipped during pre-generation

    post_generate() is also {
        gen p2      -- this allocates p2 and generates p2.x but not p2.y
    }
}
```

Data items in constraints are referenced by using *paths* (see 4.3.4). In generation context, each path is either generatable or non-generatable. *Generatable paths* refer to items that are assigned values during the generation with respect to the corresponding constraints. Each constraint shall have all its inputs sampled before the items referenced by the generatable paths are generated.

Non-generatable paths refer to items that are not affected by generation, but those items might affect generatable items. Thus, non-generatable paths refer to *inputs* of constraints. A path is *non-generatable* if

a) it is an absolute path (e.g., `sys.counter`).
b) it includes method calls (e.g., `x.y.m().z`).
c) it includes *do-not-gen* fields (e.g., `x.y.non_gen_field.z`).
d) the path is **me** (e.g., `keep root_node => parent == me;`).

Otherwise, the path is generatable.

A path that is generatable but is not intended to be generated may be modified by defining it as input to a constraint using the **read_only()** syntax, as in `keep x<read_only(y)`. In this case, the set of values `y` can take is unaffected by the constraints on `x`. The parameter `y` is treated as an input.

Arbitrary expressions can be used as arguments of **read_only()**. For example, in `keep x < read_only(y+z)`, both `y` and `z` become inputs of the constraint. First, `y` and `z` are generated (unaffected by the possible values of x). Then, their sum is computed and used as an input in the constraints.

Semantically, **read_only()** can be viewed as an identity function

read_only(*arg* : TYPE**) is {** *result* = *arg* **}**

defined for each type TYPE known to the generator. The use of **read_only()** in constraints is thus identical to the use of such an identity function.

A constraint that has no generatable paths with respect to the current generation context shall be reported as an error.

10.2.11 Scope of constraints

A constraint can be either applicable or inapplicable depending on the context of generation. There are two basic rules governing that aspect of generation.

a) All constraints defined for **sys** and any of the nested structs are applicable during the initial generation.
b) For generation started by the **gen** *item* action (see 10.5.1), the following are applicable:
    1) The constraints defined within the optional *constraints* block.
    2) All constraints defined in the type of *item*, if *item* is of a struct type.
    3) All constraints referring to *item* in this struct (**me**) and in the struct hierarchy containing **me**.

Example

```
struct packet {
    x : uint;
    y : uint;
      keep x < y
};

extend sys {
```

```
    !p1 : packet;
       keep p1.y == 8;
    !p2 : packet;

     post_generate() is also {
         gen p1 keeping {it.x > 5};
         p2 = new;
         gen p2.x
     }
}
```

The generation of `p1` succeeds. The applicable constraints here are `p1.x>5` (by rule b1), `p1.x<p1.y` (by rule b2), and `p1.y==8` (by rule b3) Thus, `p1.y` becomes `8` and `p1.x` becomes either `6` or `7`.

The generation of `p2.x` fails. For `p2` allocated using `new`, `p2.x=0` and `p2.y=0`. The only applicable constraints in this case is `p2.x<p2.y` (by rule b3). `p2.y` is not a generatable item here in the context of `gen p2.x` (see 10.2.10); it is used as input, so the constraint is equivalent to `p2.x < 0`. Since `x` is a **uint**, the constraint is not satisfiable.

10.2.12 Soft constraints

A constraint can be declared as soft by prefixing it with the **soft** keyword in the declaration. See also 10.4.5.

```
keep soft constraint;
gen item keeping {soft constraint; ...};
keep soft item = select {...}
```

Intuitively, soft constraints are satisfied if possible and otherwise disregarded. Soft constraints suggest default values and relations that can be overridden by hard or other soft constraints. They are considered with respect to the order of importance, which is a reverse of the (textual) order of soft constraints in the model.

The following properties of soft constraints also apply:

a) Assume two soft constraints c_1 and c_2, such that c_1 is more important than c_2. Then the generator shall always produce a solution satisfying c_1, if one exists. It is also required that the generator find a solution satisfying both c_1 and c_2, if it exists.

b) Assume a collection of data items (fields and/or variables) $x_1 \ldots x_n$, a collection of constraints $c_1 \ldots c_k$ linking the data items, and a solution exists satisfying all $c_1 \ldots c_k$. Then a solution needs to be found for `{soft` c_1`;...;soft` c_k`}` such that all soft constraints are satisfied.

Informally, this property means that in the absence of hard constraints, soft constraints act as hard, except for those cases causing contradictions.

Example

```
struct s {
    x : int;
    y : int;
    z : int;
      keep x in [1..100];
      keep x < y or y < z
}
```

is the same as

```
struct s {
```

```
    x : int;
    y : int;
    z : int;
      keep soft x in [1..100];
      keep soft x < y or y < z
}
```

10.2.12.1 keep gen-item.reset_soft()

Purpose	Quit evaluation of soft constraints for a field
Category	Struct member
Syntax	**keep** *gen-item*.**reset_soft()**
Parameters	*gen-item* A generatable item (see 10.4.8).

This causes the program to quit the evaluation of soft value constraints for the specified field. Soft constraints for other fields are still evaluated. Soft constraints are considered in reverse order to the order in which they are defined in the ***e*** code.

The syntax **keep** *gen-item*.**reset_soft()** is used for discarding soft constraints referring to the *gen-item* loaded so far. Soft constraints not referring to *gen-item* or soft constraints referring to *gen-item*, but loaded later, are taken into account by the constraint resolution engine. The main use of this feature is for overloading the default “soft” behavior of a model.

Syntax example:

```
keep c.reset_soft()
```

10.2.12.2 keep soft... select

Purpose	Constrain distribution of values	
Category	Struct member	
Syntax	**keep soft** *gen-item*`==`**select {***weight***:** *value***; ...}**	
Parameters	*gen-item*	A generatable item of type list (see 10.4.8).
	weight	Any **uint** expression. Weights are proportions; they do not have to add up to 100. A relatively higher weight indicates a greater probability that the value is chosen.
	value	*value* is one of the following: a) **set**—An expression of a set type, or a range list such as `[2..7]` or `[a..b]`. A select expression with a **set** as a value, selects the portion of the current range that intersects with the specified set. b) **exp**—Any expression returning the type of the *gen-item*. c) **others**—Selects the portions of the current range that do not intersect with other select expressions in this constraint. Using a weight of `0` for **others** causes the constraint to be ignored, i.e., the effect is the same as if the **others** option were not entered at all. d) **pass**—Ignores this constraint and keeps the current range as is. e) **edges**—Selects the values at the extreme ends of the current range(s). f) **min**—Selects the minimum value of the *gen-item*. g) **max**—Selects the maximum value of the *gen-item*.

This specifies the relative probability that a particular value or set of values is chosen from the current range of legal values. The current range is the range of values as reduced by hard constraints and by soft constraints that have already been applied. A weighted value shall be assigned with the probability of

weight/(*sum of all weights*)

Weights are treated as integers. If an expression is used for a *weight*, the value of the expression shall be smaller than the maximum integer size (`MAX_INT`).

Like other soft constraints, **keep soft select** is order dependent (see 10.2.12) and shall not be met if it conflicts with hard constraints or soft constraints that have already been applied. In those cases where some values conflict with other constraints, **keep soft select** shall bias the distribution based on the remaining permissible values.

Syntax example:

```
keep soft me.opcode == select {
    30 : ADD;
    20 : ADDI;
    10 : [SUB, SUBI]
}
```

10.2.13 Constraining non-scalar data types

This subclause describes constraining structs and lists.

10.2.13.1 Constraining structs

There are two basic constraints that apply to structs: struct equality and struct inequality. Other constraints affecting items of struct types (such as list constraints with structs as list elements) can be equivalently expressed using these basic constraints and Boolean combinators.

10.2.13.1.1 Struct equality

Struct equality constrains two structs to share the same struct layout, i.e., it *aliases* two struct pointers.

Example

```
struct packet {
    x : int;
    y : int
};

extend sys {
    p1 : packet;
    p2 : packet;
      keep p1 == p2;

    post_generate() is also {
        p1.x = 5
    }
}
```

This causes `p1` and `p2` to represent the same struct, i.e., `sys.p1` and `sys.p2` can be viewed as pointers pointing to the same place in memory. Thus, the assignment in `post_generate` has the same effect on both structures, i.e., `sys.p1.x = sys.p2.x = 5`.

In contrast,

```
struct packet {
    x : int;
    y : int
};

extend sys {
    p1 : packet;
    p2 : packet;
      keep p1.x == p2.x;
      keep p1.y == p2.y;
post_generate() is also {
        p1.x = 5
    }
}
```

The first two lines in “extend sys” define two structures with the same contents, `sys.p1` and `sys.p2`. Then the assignment in `post_generate` changes the value of `sys.p1.x`, but not of `sys.s2.x`. Thus, at the end `sys.p1.x=5`, while `sys.p2.x` is set to a random value from the range `[MIN_INT.. MAX_INT]`. Of course, this value could be `5` as well, but the chance for that is 1/(2^32). Thus, most likely at the end `sys.p1.x != sys.p2.x`.

10.2.13.1.2 Struct inequality

Struct inequality states that two struct pointers cannot be aliased, although they can still have the identical contents. Normally, struct inequality only makes sense for structs with a finite set of possible values (see 10.2.13.2).

Example

```
struct packet {
    x : int;
    y : int
};

extend sys {
    p1 : packet;
    p2 : packet;
      keep p1 in sys.list_of_input_packets;
      keep p2 in sys.list_of_input_packets;
      keep p1 != p2;
      keep p1.x == p2.x;
      keep p1.y == p2.y
}
```

This code constrains both `p1` and `p2` to be elements of a (pre-built) list of input packets, such that `p1` and `p2` are distinct packets and have the same contents. The generation succeeds if and only if (**iff**) the list `sys.list_of_ input_packets` contains repetitions. There is no contradiction in the fact `p1` and `p2` are different structs with identical contents.

10.2.13.2 Allocation versus aliasing

By default, a new structure is allocated for each item of a struct type. The only exception to that are the cases when the range of possible structs is limited by constraints to a finite number of choices.

Example

```
p : packet;
  keep (packet == sys.input_packet1) or (packet == sys.input_packet2)
```

In this example, the range of values for `packet` is limited by the values `sys.input_packet1` and `sys.input_packet2`, where both values are pre-built structures, i.e., inputs to the constraint. In contrast,

```
keep packet != sys.input_packet1
```

does not limit the choices of `packet` to a finite set. Here, there are an infinite number of ways to allocate `packet` so that it does not point to `sys.input_packet1`. Thus, the system allocates a `NEW` struct for `packet` in this case. This behavior makes struct inequality redundant for those cases where the set of potential struct values is unlimited.

10.2.13.3 Constraining lists

This subclause describes constraining lists. See also Table 25.

Lists are treated as pointers exactly like structs.

10.2.13.3.1 List equality and inequality

List equality constraint states that two lists are allocated with the same object, and therefore contain the same elements in the same order.

Example

```
extend sys {
    L1 : list of int;
    L2 : list of int;
   !x  : int;
      keep L1 == L2;

    post_generate() is also {
        x = L2.pop()
    }
}
```

This generates two identical lists `L1` and `L2`. Then, `post_generate()` removes the last element of `L2`, which is also the last element of `L1`.

As for the list inequality constraint (`L1 != L2`), it states that the items of list type `L1` and `L2` are not aliased. Still, the lists can have the same number of elements and the same values for their items.

10.2.13.3.2 List item

The syntax *generatable_path_to_list*[*index*] provides a generatable path of a list element. This syntax can be used in constraints as any other generatable path. List item constraints are fully solvable. Thus, the constraint can be used in several different modes.

Examples

```
keep sys.packets[5] == x;      -- element extraction from fixed list
keep l[7] < 25;                -- constraining certain element of list
keep sys.packets[i].id == 10;  -- index look-up for fixed list and value
keep l[i] < x                  -- multi-way constraint
```

10.2.13.3.3 Item in list

The expression *item* **in** *list* states that *item* is an element of the *list*. Note that a constraint such as

```
keep x in l
```

also implies that `l` includes at least one element, i.e., it is non-empty.

10.2.13.3.4 List in list

The syntax *list1* **in** *list2* provides the way of constraining two lists *list1* and *list2* so *list1* is a (possibly permuted) *sublist* of *list2*. *list1* is a possibly permuted sublist of *list2* if for every valid index *i* in *list1* there exists a matching valid index *j* in *list2* such that `list1[i]==list2[j]`. Every index *j* of *list2* is represented at most once in *list1*.

Informally, this definition means *list1* can be obtained from *list2* by a number (possibly zero) of **delete** operations of elements of *list2* and then applying **is_a_permutation(***list2***)**.

Examples

`{1;2;3}` is a sublist of `{0;1;3;2;3}`
`{1;2;3}` is a permuted sublist of `{1;3;2}`
`{1;1;2}` is a sublist of `{1;3;1;4;2}`
`{1;1;2}` is NOT a sublist of `{1;2;2;3}`
`{1;1;2}` is a permuted sublist of `{2;1;1}`

10.2.13.3.5 Permutations

The syntax *list1***.is_a_permutation(***list2***)** states that *list1* is a permutation of *list2*. The lists *list1* and *list2* contain exactly the same elements and the same numbers of repetitions of each element.

Examples

`{2;3;1}` is a permutation of `{1;2;3}`
`{2;3}` is not a permutation of `{1;2;3}`
`{1;2;3}` is a permutation of `{1;2;3}`
`{2;3;2;1}` is NOT a permutation of `{1;2;3}`

is_a_permutation is a symmetric property, i.e., *list1* is a permutation of *list2* **iff** *list2* is a permutation of *list1*. Thus, the following two constraints are equivalent:

```
keep list1.is_a_permutation(list2);
keep list2.is_a_permutation(list1)
```

10.2.13.3.6 List attributes

There are several properties of lists that can be constrained using the *attribute* syntax, **list.***attribute***(...)**.

list.size()—constrains the size of the list, e.g., `keep my_list.size() in [5..8]` `my_list` can have 5,6,7, or 8 elements.

list.count(*exp***)**—counts the number of list elements satisfying *exp* that have a Boolean type, e.g., `keep my_list.count(it == 3) == 5` the number 3 appears exactly five times in `my_list`.

list.has(*exp***)**—verifies at least one item of the list satisfies the Boolean *exp*. This is the same as `list.count(`*exp*`) > 0`.

list.all_different(*exp*[**,***cond_exp*]**)**—returns TRUE if, and only if, evaluation of the expression returns a unique value for each of the list elements, meaning that no two items (or expressions) in the list have the same value. If a *cond_exp* parameter is present, the constraint is applied only to the items with a TRUE *cond_exp*. For example, `keep my_list.all_different(it, index>5)` ensures that there are no duplicate items in indices above 5.

list.sum(*exp***)**—constrains the sum of the list elements satisfying *exp* containing a Boolean type. The attribute applies only to lists of numeric type, e.g., `keep my_list.sum(it) == 100` for the elements of `my_list` in the range `[0..20]` is `100`.

list.and_all(*exp***)**—returns the logical AND of all Boolean expressions. For example, `list.and_all(it>5)` returns TRUE if all the list items are greater than 5.

list.or_all(*exp***)**—verifies at least one item of the list satisfies the Boolean exp. This is the same as **list.has(***exp***)**.

list.max_value(*exp***)**—constrains the maximum exp in a list. For example, `list.max_value(it) == 100` constrains the maximal list item to be 100.

list.min_value(*exp***)**—constrains the minimum exp in a list. For example, `list.min_value(it) == 100` constrains the minimal list item to be 100.

10.2.13.3.7 Constraining all list items: keep for each

Purpose	Constrain list items	
Category	Struct member	
Syntax	**keep for each** [(*item-name*)] [**using** [**index** (*index-name*)] [**prev** (*prev-name*)]] **in** *gen-item* [**do**] **{**(*constraint-bool-exp* | *nested-for-each*)**; ...}**	
Parameters	*item-name*	An optional name used as a local variable referring to the current item in the list. The default is **it**.
	index-name	An optional name referring to the index of the current item in the list. The default is **index**.
	prev-name	An optional name referring to the previous item in the list. The default is **prev**.
	gen-item	A generatable item of type list (see 10.4.8).
	constraint-bool-exp	A simple or a compound Boolean expression (see 10.4.7).
	nested-for-each	A nested **for each** block, with the same syntax as the enclosing **for each** block, except that `keep` is omitted.

This defines a value constraint on multiple list items. The following restrictions also apply:

— **for each** constraints can be nested. The parameters *item-name*, *index-name*, and *prev-name* of a nested **for each** can shadow the names used in the outer **for each** blocks. In particular, if the optional names are unspecified, then the default names **it**, **index**, and **prev** refer to the corresponding details of the innermost **for each** block.

— Within a **for each** constraint, **index** represents a running index in the list, which is treated as a constant with respect to each list item.

— Generated items need to be referenced by using a pathname that starts either with **it**, **prev**, or the optional *item-name* or *prev-name*, respectively. Items whose pathname does not start with **it** can only be sampled; their generated values cannot be constrained.

— If a **for each** constraint is contained in a **gen ... keeping** action, the iterated variable needs to be named first.

Syntax example:

```
keep for each (p) in pkl do {
    soft p.protocol in [atm, eth]
}
```

10.2.13.3.8 All solutions

This feature generates lists of structs covering all possible combinations of values for certain fields. The syntax is *list***.is_all_iterations(.***fieldname***, ...),** where *list* is a list of elements and *fieldname***,** ... are field names of some struct type `T`. The arguments of **is_all_iterations** are unique, i.e., there are no repetitions in the list of fields. All fields shall be defined under the base type `T`, i.e., fields defined in **when** subtypes or **like** successors are not allowed.

Example

```
struct s {
    b1 : bool;
    b2 : bool;
    x  : int
};

extend sys {
    l : list of s;
      keep l.is_all_iterations(.b1, .b2)
}
```

The resulting `sys.l` includes four elements for all four combinations of TRUE/FALSE of `b1` and `b2`. The values of `x` are chosen randomly.

10.3 Type constraints

This subclause describes how to use type constraints to restrict the declared type of a field to one of its **like** or **when** subtypes for a given context. A constraint prefixed with the **type** modifier is both (a) enforced by the generator (like a regular constraint) and (b) presupposed at compile time for purposes of type checking. Expressions for which type constraints apply are automatically downcast to the specified subtype wherever required. This saves explicit downcasting [“is_a()” and “as a” operators] for the expression and lets the downcast expression be used as a generatable term (rather than input) in constraint contexts.

10.3.1 keep type

Purpose	Refine the type of a field to one of its subtypes for the specified context	
Category	Struct member	
Syntax	**keep type** [**me.**]*field-name* **is a** *type* **keep type** [**me.**]*field-name*.*property-name* **==** [**me.**]*my-property-name* **keep for each** [**(***item-name***)**] **in** *list-field-name* **{** … **type** *item-name* **is a** *type***;** … **}** **keep for each** [**(***item-name***)**] **in** *list-field-name* **{** … **type** *item-name*.*property-name* **==** [**me.**]*my-property-name***;** … **}**	
Parameters	*field-name*	The name of a struct field in the enclosing struct.
	type	The name of a struct or unit type.
	property-name	The name of an enumerated or Boolean const field.
	my-property-name	The name of a field of the same type as the *property-name* in this constraint.
	item-name	An optional name used as a local variable referring to the current item in the list. The default is **it**.
	list-field-name	The name of a field of typelist of struct (or unit) in the enclosing struct.

A type constraint can be put either on a field of a struct type or on a list field of a struct type. The declaration is similar to a regular constraint inside a **keep** struct member, or, in the list case, inside a **keep for each** construct, with the **type** keyword prefixing the expression.

The **type** keyword is a constraint modifier syntactically analogous to **soft**. However, unlike **soft**, it can modify only specific constraint expressions and can appear only in restricted contexts.

The type correlation can be fixed or, when the correlated types are **when** subtypes, variable. The former case is expressed using the **is a** operator. In the latter case the determinant property (the **when** determinant) of the referenced struct is equated to a determinant property of the same type in the declaring struct type.

Type constraints affect the static semantics of field-access expressions of the form *instance-expression.field-name* (field-access in which *instance-expression* is omitted is equivalent to one having **me** as the *instance-expression*). Typically the static type of a field-access expression is determined according to the type of the field as it was initially declared in the struct type of *instance-expression* (or in one of its supertypes). Type constraints tying the static type of *instance-expression* with a subtype of the field's declared type can change this rule. If the context in which the field-access occurs requires the subtype, the field-access is automatically downcast. In this case, a runtime check is added to ensure that the casting is justified, and an error is issued if it is not. The runtime check involves a minor overhead, not more than that required by the **as_a()** operator.

NOTE

- In the Boolean expression following **type**, operators other than == and **is a** are not allowed.
  For example, the following is not allowed:
  ```
  keep type TRUE => engine is a FORD engine // not allowed
  ```
- The **for each** clause must occur immediately after **keep**. For example, the following is not allowed:
  ```
  keep my_doors.size() > 4 => for each in my_doors { // not allowed
      type it is a small door
  }
  ```
- Type constraints can equate only constant fields, so the **const** keyword must appear in the declaration of fields involved in equality constraints.
- Type constraints in general affect code from that point onwards. This includes type constraints that appear inside a **for each** clause, in which case other expressions in the same scope after the declaration (but not before it) can assume automatic casting.
- Type constraints cannot appear inside a **gen** action.
- The **soft** keyword cannot be used with type constraints.
- As with non-type constraints, the determinant field of the when subtype is assigned only during generation. Thus the **pre_generate()** method of the type specified in the type constraint is not called during generation.
- A field's type may be restricted by more than one type constraint with respect to different "when" dimensions (determinant fields).

Syntax example:

```
keep type f.p1 == p1;
keep for each in lf {
    type it is a B S1
}
```

10.3.2 Type constraints and struct fields

Automatic casting of a struct-reference field is performed in any context that requires it, including the following:

- Struct-member access
- Assignment

— Parameter passing

10.3.3 Type constraints and list fields

When the type relation is one-to-many, in other words, when a list field is concerned, automatic casting is applied not to the list itself but to its elements. Automatic casting affects list operators whose result type is the element type, such as indexing (the [] operator) and **pop()**. It also affects the iteration variable inside the **for each** construct, both in procedural and in constraint contexts.

10.3.4 Type constraints and like subtypes

Type constraints work just as well for **like** subtypes of the declared type of the field. They apply to the two "**is a**" forms of the **keep type** struct member.

Note that type constraints with **like** subtypes cannot make the actual **like** type of a generated field dependent on a **when** determinant. In other words, they may not figure under a **when** subtype if they affect a field not declared in the same subtype. This is an error: the constraint is unenforceable.

10.4 Defining constraints

This subclause describes the constructs used to define constraints. See also 4.10.

10.4.1 keep

Purpose	Define a hard value constraint	
Category	Struct member	
Syntax	**keep** [*name* **is [only]**] *constraint-bool-exp*	
Parameters	*name*	Optional identifier for constraint overriding and reference by tools.
	constraint-bool-exp	A simple or a compound Boolean expression (see 10.4.7).

This states restrictions on the values generated for fields in the struct or its subtree, or describes required relationships between field values and other items in the struct or its subtree.

Hard constraints are applied whenever the enclosing struct is generated. For any **keep** constraint in a generated struct, the generator either meets the constraint or issues a constraint contradiction message. If the **keep** constraint appears under a **when** construct, the constraint is considered only if the **when** condition is true.

Syntax example (un-named constraint):

```
keep kind != tx or len == 16
```

Syntax example (named constraint):

```
keep address_range is soft addr in [0..9]
```

10.4.1.1 Constraint overriding

Every named constraint must have exactly one actual definition per struct type. An initial definition of a constraint in a struct type may be overridden in **like** and **when** subtypes or in later extensions of the same struct—any number of times—using the **is only** modifier.

The semantics of constraint overriding is identical to that of overriding other extendable struct members, such as methods. A constraint can be redefined in different when subtypes (even if they are not contradictory), and the latest definition that applies to the generated subtype is chosen (for ordering definitions see Annex B).

Example:

```
struct packet {
   size: [big, small];
   data: list of byte;
      keep data_size is undefined; // abstract constraint

   when big packet {
      keep data_size is all of { // concrete definition for big packets
         data.size() > 10;
         data.size() < 20
      }
   }
}
```

10.4.2 keep

Purpose	Define an abstract constraint	
Category	Struct member	
Syntax	**keep** *name* **is [only]**	
Parameters	*name*	Optional identifier for constraint overriding and reference by tools.

10.4.3 keep all of {...}

Purpose	Define a constraint block	
Category	Struct member	
Syntax	**keep** [*name* **is [only]**] **all of {***constraint-bool-exp***; ...}**	
Parameters	*name*	Optional identifier for constraint overriding and reference by tools.
	constraint-bool-exp	A simple or a compound Boolean expression (see 10.4.7).

A **keep** constraint block is exactly equivalent to a **keep** constraint for each constraint Boolean expression in the block. The **all of** block can be used as a constraint Boolean expression itself.

Syntax example:

```
keep all of {
    kind != tx;
    len == 16
}
```

10.4.4 keep struct-list.is_all_iterations()

Purpose	Cause a list of structs to have all iterations of a field	
Category	Constraint-specific list method	
Syntax	**keep** [*name* **is [only]**] *gen-item***.is_all_iterations(.***field-name*: exp**, ...)**	
Parameters	*name*	Optional identifier for constraint overriding and reference by tools.
	gen-item	A generatable item of type list of struct (see 10.4.8).
	field-name	The name of a scalar field of a struct. The field name shall be prefixed by a period (**.**). The order of fields in this list does not affect the order in which they are iterated. The specified field that is defined first in the struct is the one that is iterated first.

This causes a list of structs to have all legal, non-contradicting iterations of the fields specified in the field list. Fields not included in the field list are not iterated; their values can be constrained by other relevant constraints. The highest value always occupies the last element in the list.

Soft constraints on fields specified in the field list are skipped. All other relevant hard constraints on the list and on the struct are applied. If these constraints reduce the ranges of some of the fields in the field list, then the generated list is also reduced.

The following restrictions also apply:

— The number of iterations in a list produced by *list***.is_all_iterations()** is the product of the number of possible values in each field in the list. Use the **absolute_max_list_size** generation configuration option to set the maximum number of iterations allowed in a list (the default is 524 288).

— The *list***.is_all_iterations()** method shall only be used in a constraint Boolean expression.

— The fields to be iterated shall be of a scalar type, not a list or struct type.

Syntax example:

```
keep packets.is_all_iterations(.kind, .protocol)
```

10.4.5 keep soft

Purpose	Define a soft value constraint	
Category	Struct member	
Syntax	**keep** [*name* **is [only]**] **soft** *constraint-bool-exp*	
Parameters	*name*	Optional identifier for constraint overriding and reference by tools.
	constraint-bool-exp	A simple Boolean expression (see 10.4.7).

This suggests default values for fields or variables in the struct or its subtree, or describes suggested relationships between field values and other items in the struct or its subtree. The following restrictions apply:

— Soft constraints are order dependent (see 10.2.12) and shall not be met if they conflict with hard constraints or soft constraints that have already been applied.

— The **soft** keyword shall not be used in compound Boolean expressions.

— Individual constraints inside a constraint block can be soft constraints.

— Because soft constraints only suggest default values, it is better not to use them to define architectural constraints.

Syntax example:

```
keep soft legal
```

10.4.6 read_only()

Purpose	Modify generation sequence	
Category	Pseudo-method	
Syntax	**read_only(***item*: exp**)**	
Parameters	*item*	A legal ***e*** expression.

read_only() computes the value of the expression inside it. It makes the expression an input to the constraint, and if there are generative elements inside the expression, the generation order is enforced so that these elements are generated before the connected field set to which the constraint belongs.

Example

```
keep a == read_only(b + c)
```

This constraint has two results:

— b and c are generated before a.

— The value of a cannot otherwise be constrained in a bidirectional constraint.

Syntax example:

```
keep i < read_only(j)
```

10.4.7 constraint-bool-exp

Purpose	Define a constraint on a generatable item	
Category	Expression	
Syntax	*bool-exp* [**or** | **and** | => *bool-exp*] ...	
Parameters	*bool-exp*	An expression that returns either TRUE or FALSE when evaluated at runtime.

A *constraint Boolean expression* is a simple or compound Boolean expression that describes the legal values for at least one generatable item or constrains the relation of one generatable item with others. A compound Boolean expression is composed of two or more simple expressions joined with the **or**, **and**, or implication (=>) operators. Table 25 shows the ***e*** special constructs that are useful in constraint Boolean expressions.

Table 25—Constraining Boolean expressions

Constraint	Definition
soft	A keyword that indicates the constraint is either a soft value constraint or a soft order constraint. See 10.4 for a definition of these types of constraints.
soft...select	An expression that constrains the distribution of values.
.reset_soft()	A pseudo-method that causes the test generator to quit evaluation of soft constraints for a field, in effect, removing previously defined soft constraints.
.is_all_iterations()	A list method used only within constraint Boolean expressions that causes a list of structs to have all legal, non-contradicting iterations of the specified fields.
.is_a_permutation()	A list method that can be used within constraint Boolean expressions to constrain a list to have the same elements as another list.
[not] in	An operator that can be used within constraint Boolean expressions to constrain an item to a range of values or a list to be a subset of another list; or when used with **not**, to be outside the range or absent from another list.
is [not] a	An operator that checks the subtype of a struct.

The following considerations also apply:

— The **soft** keyword can be used in simple Boolean expressions, but not in compound Boolean expressions.

— The order of precedence for Boolean operators is: **and**, **or**, =>. A compound expression containing multiple Boolean operators of equal precedence is evaluated from left to right, unless parentheses [( )] are used to indicate expressions of higher precedence.

— Any ***e*** operator can be used in a constraint Boolean expression. However, certain operators can affect generation order or can create an constraint that is not enforceable.

— In compound expressions where multiple implication operators are used, the order in which the operations are performed is significant. For example, in the following constraint, the first expression (`a => b`) is evaluated first by default:

```
keep a => b => c;               // is equivalent to:
keep (not a or b) => c;         // is equivalent to:
keep a and (not b) or c
```

However, adding parentheses around the expression (`b => c`) causes it to be evaluated first, with very different results.

```
keep a => (b => c);             // is equivalent to:
keep a => (not b) or c;         // is equivalent to:
keep (not a) or (not b) or c
```

Examples

The following are examples of simple constraint Boolean expressions:

```
not short        // where "short" is of type "bool"
long == TRUE
soft x > y
x + z == y + 7
```

The following are examples of compound constraint Boolean expressions:

```
x > 0 and soft x < y
is_a_good_match(x, y) => z < 1024
color != red or resolution in [900..999]
packet is a good packet => length in [0..1023]
```

See also 5.1.1.

Syntax example:

```
z == x + y
```

10.4.8 gen-item

Purpose	Identifies a generatable item
Category	Expression
Syntax	[**me.**]*field1-name*[**.***field2-name* ...] | **it** | [**it**]**.***field1-name*[**.***field2-name* ...]
Parameters	*field-name* The name of a field in the current struct or struct type.

A *generatable item* is an operand in a Boolean expression that describes the legal values for that generatable item or constrains its relation with another generatable item. Every constraint shall have at least one generatable item or an error shall be issued.

In a **keep** constraint, the syntax for specifying a generatable item is a path starting with **me** of the struct containing the constraint and ending with a field name. In a **gen** action, the syntax for specifying a generatable item is a path starting with **it** of the struct containing the constraint and ending with a field name.

A generatable item cannot have an indexed reference in it, except as the last item in the path. See also 4.3.3.

Syntax example:

```
me.protocol
```


10.5 Invoking generation

There are two ways of invoking generation, as follows:

a) Generation is invoked automatically when generating the tree of structures starting at **sys**.

b) Generation can be called for any data item by using the **gen** action. The scope of this type of generation is restricted (see 10.5.1). The generation order is (recursively):

   1) Allocate the new struct

   2) Call **pre_generate()**

   3) Perform generation

   4) Call **post_generate()**

10.5.1 gen

Purpose	Generate values for an item	
Category	Action	
Syntax	**gen** *gen-item* [**keeping {**[**it**]**.***constraint-bool-exp***; ...}**]	
Parameters	*gen-item*	A generatable item. If the expression is a struct, it is automatically allocated, and all fields under it are generated recursively, in depth-first order.
	constraint-bool-exp	A simple or compound Boolean expression (see 10.4.7).

This generates a random value for the instance of the item specified in the expression and stores the value in that instance, while considering all the constraints specified in the **keeping** block, as well as other relevant constraints at the current scope on that item or its children. Constraints defined at a higher scope than the enclosing struct are not considered.

The following considerations also apply:

— Values for particular struct instances, fields, or variables can be generated during simulation (on-the-fly generation) by using the **gen** action.

— This constraint can also be used to specify constraints that apply only to one instance of the item.

— The **soft** keyword can be used in the list of constraints within a **gen** action.

— The earliest the **gen** action can be called is from a struct's **pre_generate()** method.

— The generatable item for the **gen** action cannot include an index reference.

— If a **gen ... keeping** action contains a **for each** constraint, the iterated variable needs to be named.

Syntax example:

```
gen next_packet keeping {
    .kind in [normal, control]
}
```

10.5.2 pre_generate()

Purpose	Method run before generation of struct
Category	Method of **any_struct**
Syntax	[*struct-exp*.]**pre_generate()**
Parameters	*struct-exp* An expression that returns a struct. The default is the current struct.

The **pre_generate()** method is run automatically after an instance of the enclosing struct is allocated, but before generation is performed. This method is initially empty, but can be extended to apply values procedurally to prepare constraints for generation. It can also be used to simplify constraint expressions before they are analyzed by the constraint resolution engine.

NOTE—Prefix the ! character (see 6.8) to the name of any field whose value is determined by **pre_generate()**. Otherwise, normal generation overwrites this value.

Syntax example:

```
pre_generate() is also {
    m = 7
}
```

10.5.3 post_generate()

Purpose	Method run after generation of struct
Category	Predefined method of **any_struct**
Syntax	[*struct-exp*.]**post_generate()**
Parameters	*struct-exp* An expression that returns a struct. The default is the current struct.

The **post_generate()** method is run automatically after an instance of the enclosing struct is allocated and both pre-generation and generation have been performed. This method can be extended for **any_struct** to manipulate values produced during generation. It can also be used to derive more complex expressions or values from the generated values.

Syntax example:

```
post_generate() is also {
    m = m1 + 1
}
```