

7. Constraints and generation 1

To do list (for Draft 4) — Still a WIP :) 5

- gen_before_subtypes
- gen.before
- reset_soft
- select
- value 10
- read_only
- is a
- as_a

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. 15

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. 20

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., 25

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

7.1 Types of constraints 30

Constraints can be sub-divided according to several criteria.

- a) Explicit or implicit
 - 1) *Explicit constraints* are those declared using the **keep** statement or inside **keeping {...}** block. 35
 - 2) *Implicit constraints* are those imposed by type definitions and variable declarations. Implicit constraints are always hard.

Examples

```
x : int[1,3,5,10..100]; \\ is the same as
x : int; keep x in [1,3,5,10..100]; 40
```

```
l[20] : list of int; \\ is the same as
l : list of int; keep l.size()==20; 45
```

- 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.

<<VIT: maybe we want to link it to a relevant discussion on exceptions and error handling in the e-language chapter>> 50
 - 2) *Soft constraints* are honored if they do not contradict hard constraints or soft constraints honoured earlier. If a soft constraint cannot be honored, it is disregarded. (See ****Semantics of Soft Constraints??** for the explanations on how the selection of soft constraints is done.) 55

- 1 c) Simple or compound
A constraint combining other constraints in a Boolean combination using **not**, **and**, **or**, and \Rightarrow is called *compound*. Otherwise, the constraint is called *simple*.
- 5 d) Fully or partially solvable
- 1) For some constraints, it is always possible to either generate a solution satisfying the constraint or check that no such solution exists. Such constraints are *fully solvable*, e.g. $x < y + z$.
In this example, there is an efficient algorithm that either provides a random solution satisfying the constraint or proves no such solution can be found for any sets of possible values of x , y , and z .
- 10 2) There is a second kind of constraint, which generally cannot be solved by an efficient algorithm. Such constraints are called *partially solvable*. For this kind of constraint, however, some special cases can be efficiently solved, e.g. $x < f(y, z)$.
In this example, the constraint can only be addressed after both y and z are given fixed values. All other possible modes of solving the constraints are either theoretically impossible or intractable, depending on the types of the parameters. Thus, x is constrained by y and z , but not vice-versa.
- 15 < MAYBE LEAVE THIS DISCUSSION TILL LATER >

20 7.2 Generation concepts

This section describes the basic concepts of generation.

25 7.2.1 The basic flow of generation

30 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 4.7). There is no specific order in which data items or the fields in a *struct* hierarchy are generated.

35 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.

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

The following ordering rules, however, do apply.

- 45 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 **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.
- 50 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: 7.2.2 and 7.2.4.*

- c) Determinants of **when** subtypes are generated before any of the fields defined in these subtypes.

7.2.2 Using methods in constraints

Constraint paths can including 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.
- any generatable paths used as parameters of the method are fully generated before the method is called.
- For methods returning pointers to *structs*, the corresponding 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.y` 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.

7.2.2.1 Classification of methods

Methods are classified into three categories.

- a) Methods which 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 which 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, 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 7.2.2.2). 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 which 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 { if (it.x < 100) { it.id = id; id += 1; } };
  };
};

```

In general, it is impossible to classify methods automatically into the above 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.

(VIT: a path is *local* if it starts at a local variable or at one of the method parameters; maybe we can link it with another definition given somewhere else, e.g. Language chapter?)

7.2.2.2 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 7.2.2.1, 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.

7.2.3 Fully/partially solvable constraints

The following constraints are presumed partially solvable:

- method calls; exception: special methods `.size()`, `max()`, `even()`, etc.
- bit slices: indexes are before the rest.
- <<VIT: list constraints? which?>>

7.2.4 Generatable paths and the sampling of inputs

The purpose of *constraints* is to constrain “generatable” items, i.e., those items which 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, <<VIT: do we call it pre-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 7.4.1), *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”. <<VIT: there should be a **definition somewhere**; Language chapter?>> In the context of a constraint, each path is either generatable or non-generatable. *Generatable paths* refer to items which are assigned values during the generation with respect to the corresponding constraints. Each constraint shall have all its inputs are sampled before the items referenced by the generatable paths are generated.

Non-generatable paths refer to items which 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 constraint which has no generatable paths shall be reported as an error.

Example

In a generation context initiated by

```
gen my_item keeping my_constraints
```

the paths in `my_constraints` starting with **it** and satisfying rules b and c above are generatable. All other paths in `my_constraints` are non-generatable.

7.2.5 The 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. (<<VIT: do we call it 'pre-generation'??>>)
- b) For generation started by **gen item** action (see 7.4.1), 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 7.2.4); it is used as input, so the constraint is equivalent to `p2.x < 0`. Since `x` is a **uint**, the constraint is not satisfiable.

7.2.6 Soft constraints

A constraint can be declared as soft by prefixing it with the **soft** keyword (see 7.3) in the declaration.

```
keep soft <constraint>;
gen <item> keeping { soft <constraint>; ... };

keep soft <item> = select { ... }; <<VIT: maybe this one goes>>
```

Intuitively, soft constraints are satisfied if possible and otherwise disregarded. *Soft constraints* suggest default values and relations which 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.

- Assume two soft constraints `c1` and `c2`, such that `c1` is more important than `c2`. Then the generator shall always produce a solution satisfying `c1`, if one exists. It is also ****expected??** (but not required) that the generator can find a solution satisfying both `c1` and `c2`, if it exists.
- Assume a collection of data items (fields and/or variables) `x1...xn`, a collection of constraints `c1...ck` linking the data items, and a solution exists satisfying all `c1...ck`. Then a solution needs to be found for `{soft c1; ...; soft ck}` 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;
};
```

7.2.7 Constraining non-scalar data types

This section describes constraining *structs* and lists.

7.2.7.1 Constraining structs

There are two basic constraints which 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.

7.2.7.1.1 Struct equality

Struct equality constraints 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; };
}

```

First produces “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.p2.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`.

7.2.7.1.2 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 7.2.7.2).

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 constraints 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 **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.

7.2.7.2 Allocation vs. 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 infinitely many 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.

7.2.7.3 List constraints involving structs

In [VIT: Struct Inequality](#) we use the constraint

```

keep p1 in sys.list_of_input_packets;

```

which constrains 'p1' to be a member of the list `sys.list_of_input_packets`. Assuming

```

sys.list_of_input_packets = {r1;r2...;rn}

```

the constraint is semantically equivalent to

```
keep p1==r1 or p1==r2 or ... or p1==rn;
```

Of course, using list constraints (see 7.2.7.4) provides a more concise and expressive syntax, and is also more flexibility, since it works with lists of arbitrary length.

****Given the above statement, I'd rather just remove this entire subsection****

7.2.7.4 Constraining lists

This section describes constraining lists. *See also:* Table 1.

7.2.7.4.1 List equality

List equality constraint states that two lists 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` and preserves it in `x`. `l1` and `l2` are not aliased to the same list by the list equality constraint, they are “copies”. Therefore, `l2.pop()` does not remove the last value of `l1`.

7.2.7.4.2 List inequality

The *list inequality constraint* (`l1 != l2`) states that the items of list type `l1` and `l2` are different. Namely:

- a) the number of elements in the lists is different; or
- b) the number of elements is the same and there is an index i such that

```
l1[i] != l2[i]
```

7.2.7.4.3 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
```

7.2.7.4.4 Item in list 1

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

```
keep x in l;
```

5

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

7.2.7.4.5 List in list 10

The syntax *list1 in list2* provides the way of constraining two lists *list1* and *list2* so *list1* is a “sublist” of “*list2*”. *list1* is a sublist of *list2* if

- a) for every valid index *i* in *list1* there exists a matching valid index *j* in *list2* such that `list1[i]==list2[j]` 15
- b) for two valid indexes *i1* and *i2* in *list1* and the matching indexes *j1* and *j2* in *list2*, if *i1*<*i2* then *j1*<*j2*.

Informally, this definition means *list1* can be obtained from *list2* by a number (possibly zero) of **delete** operations of elements of *l2*. 20

Examples

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

25

7.2.7.4.6 Permutations 30

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. 35

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}
```

40

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);
```

45

7.2.7.4.7 List attributes 50

There are several properties of lists which can be constrained using the “attribute” syntax (<<VIT: terminology??>>) **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. 55

list.count(*exp*) — counts the number of list elements satisfying *exp* which 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.is_unique(*exp*) — verifies the elements satisfying the Boolean *exp* are unique within the list
 e.g., keep my_list.is_unique(it.is_a(RED packet))

all RED packets are unique within the list (of packets) my_list.

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 in [0..20]) == 100

for the elements of my_list in the range [0..20] is 100.

7.2.7.4.8 Constraining all list items

The **for each** constraint constrains list elements (see 7.3.4).

7.2.7.4.9 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.

7.2.8 Constraining bit slices

The bit slice operator can be used 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:

```

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. Or it also can be used to constrain particular bits in relation to each other.

7.2.8.1 Bit slice constraints and generation order

1

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

7.2.8.1.1 Non-constant bit indices

5

Non-constant bit indices shall be generated before other entities in the constraint. For example, the following constraint

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

10

implies

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

15

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;
```

20

7.2.8.1.2 Generation of bit sliced items

By default, bit sliced items are generated after other items in the same constraint. To override this, use the **keep gen** constraint. For example, to meet the following constraints (and x needs to be generated before y):

25

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

Add the constraint:

30

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

or add the **value()** routine to the existing constraint:

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

35

7.2.8.2 Bit slice constraints and signed entities

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

40

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

45

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

- a) The value of the bit slice is treated as an unsigned integer; i.e., 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.
- b) The value of the signed entity is generated as a non-negative. In the example above, y shall always be generated as a non-negative integer.
- c) The value of both the bit slice and the signed entity need to fit into the smaller of
 - 1) the bit width of the bit slice

50

55

- 2) the bit width of the highest possible value of the signed entity (this width excludes one bit used to store the sign).

7.2.8.3 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(0) 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 do not have the desired effect as the soft constraint is always overridden. The only way to achieve this purpose is to apply the soft constraint to each individual bit explicitly, e.g.,

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

7.2.8.4 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 can occur. In the following example, `x` is an argument to the `bit_parity()` function and needs to 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 fails in 50% of the cases. Instead, assign a new virtual field for `x[7:0]`, e.g.,

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

These constraints cause `y` to be generated first, `x[7:0]` to be constrained to have the value of `y` and `x[8:8]` to be constrained to have the return value from the `bit_parity()` method.

7.3 Defining constraints

This section describes the constructs used to define constraints. *See also:* 2.9.

7.3.1 keep

Purpose	Define a hard value constraint
Category	Struct member
Syntax	keep <i>constraint-bool-exp</i>
Parameters	<i>constraint-bool-exp</i> A simple or a compound Boolean expression (see 7.3.13).

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:

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

7.3.2 keep all of {...}

Purpose	Define a constraint block
Category	Struct member
Syntax	keep all of { <i>constraint-bool-exp</i> ; ...}
Parameters	<i>constraint-bool-exp</i> A simple or a compound Boolean expression (see 7.3.13).

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;
};
```

7.3.3 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 <i>gen-item.is_all_iterations</i> (<i>field-name</i> : exp, ...)	
Parameters	<i>gen-item</i>	A generatable item of type list of <i>struct</i> (see 7.3.14).
	<i>field-name</i>	The name of a scalar field of a <i>struct</i> . 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 <i>struct</i> 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 can 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);
```

7.3.4 keep for each

Purpose	Constrain list items	
Category	Struct member	
Syntax	keep for each [(<i>item-name</i>)] [using [index (<i>index-name</i>)] [prev (<i>prev-name</i>))] in <i>gen-item</i> { <i>constraint-bool-exp</i> <i>nested-for-each</i> ; ...}	
Parameters	<i>item-name</i>	An optional name used as a local variable referring to the current item in the list. The default is it .
	<i>index-name</i>	An optional name referring to the index of the current item in the list. The default is index .
	<i>prev-name</i>	An optional name referring to the previous item in the list. The default is prev .
	<i>gen-item</i>	A generatable item of type list (see 7.3.14).
	<i>constraint-bool-exp</i>	A simple or a compound Boolean expression (see 7.3.13).
	<i>nested-for-each</i>	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, **prev** and **index** are predefined constants and cannot be constrained or generated.
- Generated items need to be referenced by using a path name that starts either with **it**, such as `it.pk`, or using the name assigned to that list item (*item-name*). Items whose pathname does not start with **it** can only be sampled; their generated values cannot be constrained.
- Items in lists are generated in ascending order starting with index zero (0). 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 shall cause an error.
- 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 {
    soft p.protocol in [atm, eth];
};
```

7.3.5 keep soft

Purpose	Define a soft value constraint
Category	Struct member
Syntax	keep soft <i>constraint-bool-exp</i>
Parameters	<i>constraint-bool-exp</i> A simple Boolean expression (see 7.3.13).

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 7.2.6) and shall not be met if they conflict with hard constraints or soft constraints that have already been applied.
- The **soft** keyword cannot 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 == TRUE;
```

7.3.6 keep soft... select

Purpose	Constrain distribution of values
Category	Struct member
Syntax	keep soft <i>gen-item</i> == select { <i>weight</i> : <i>value</i> ; ...}
Parameters	<i>gen-item</i> A generatable item of type list (see 7.3.14).
	<i>weight</i> 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.
	<i>value</i> <i>value</i> is one of the following: a) range-list — A range list such as [2 . . 7]. A select expression with a range list selects the portion of the current range that intersects with the specified range list. b) exp — A constant expression. A select expression with a constant expression (usually a single number) selects that number, if it is part of the current range. 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 <i>gen-item</i> . g) max — Selects the maximum value of the <i>gen-item</i> .

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

$$\text{weight}/(\text{sum of all weights})$$

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 7.2.6) and shall not be met if it conflicts with hard constraints or soft constraints that have already been applied.

Syntax example:

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

7.3.7 keep gen-item.reset_soft()

Purpose	Quit evaluation of soft constraints for a field
Category	Struct member
Syntax	keep <i>gen-item.reset_soft()</i>
Parameters	<i>gen-item</i> A generatable item (see 7.3.14).

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.

Syntax example:

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

7.3.8 keep gen ... before

Purpose	Modify the generation order
Category	Struct member
Syntax	keep gen (<i>gen-item: exp, ...</i>) before (<i>gen-item: exp, ...</i>)
Parameters	<i>gen-item</i> An expression that returns a generatable item. The parentheses () are required. <i>See also: 7.3.14.</i>

This requires the generatable items specified in the first list to be generated before the items specified in the second list. This constraint can be used to influence the distribution of values by preventing soft value constraints from being consistently skipped (see 7.2). The following restrictions also apply.

- This constraint itself can cause constraint cycles. If a constraint cycle involving one of the fields in the **keep gen ... before** constraint exists and if the **resolve_cycles** generation configuration option is **TRUE**, the constraint can be ignored if the program cannot satisfy both it and other constraints that conflict with it.
- This constraint cannot appear on the left-hand side of an implication operator (**=>**).

Syntax example:

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

7.3.9 keep soft gen ... before

Purpose	Suggest order of generation
Category	Struct member
Syntax	keep soft gen (<i>gen-item</i> : exp, ...) before (<i>gen-item</i> : exp, ...)
Parameters	<i>gen-item</i> An expression that returns a generatable item. The parentheses () are required. <i>See also</i> : 7.3.14.

This modifies the “soft” generation order by recommending the fields specified in the first field list be generated before the fields specified in the second field list. This soft generation order is second in priority to the hard generation order created by dependencies between parameters and **keep gen before** constraints.

This constraint can be used to suggest a generation order which is later overridden in individual tests with a hard order constraint. This constraint cannot appear on the left-hand side of an implication operator (\Rightarrow).

Syntax example:

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

7.3.10 keep gen_before_subtypes()

Purpose	Specify a when determinant field for deferred generation
Category	Struct member
Syntax	keep gen_before_subtypes (<i>determinant-field</i> : field, ...)
Parameters	<i>determinant-field</i> An expression that evaluates to the name of a field in the <i>struct</i> type. The field shall have at least one value that is used as a when determinant for a subtype definition. If the field is not a when determinant field, a warning is issued and the constraint is ignored. Multiple field expressions can be entered, separated by commas (,).

To speed up generation of *structs* with multiple **when** subtypes, this type of constraint, called a *subtype optimization constraint*, causes the generator engine to wait until a **when** determinant value is generated for a specified field before it analyzes constraints and generates fields under the **when** subtype.

When no subtype optimization constraints are present in a *struct*, the generator analyzes all of the constraints and fields in the *struct* before it generates the *struct*, even those constraints and fields that are defined under **when** subtypes. When a subtype optimization constraint is present, the generator initially analyzes only the constraints and fields of the base *struct* type. When a subtype optimization **when** determinant is encountered, the generator analyzes the associated **when** subtype and then generates it.

The following considerations also apply.

- Subtype optimization can handle multiple determinants. Subtypes are analyzed and generated in the order in which their **when** determinants are encountered.
- If multiple determinants are specified, and some of them are subtype optimization determinants while others are not, then a subtype that is a result of multiple inheritance of a subtype optimization

determinant and a non-subtype optimization determinant shall be treated the same as a other subtype optimization determinant subtype.

- The generator engine’s ability to resolve contradictions is diminished somewhat by subtype optimization constraints. Specifically, the generator might not be able to resolve contradictions arising from constraints under subtypes that involve fields of the base type.
- The analysis and generation is recursive. If a subtype contains another determinant that is specified in a subtype optimization constraint, then that sub-subtype is analyzed and generated as soon as its determinant field is generated under the higher-level subtype.

Syntax example:

```
keep gen_before_subtypes(format);
```

7.3.11 keep reset_gen_before_subtypes()

Purpose	Disable all previous keep gen_before_subtypes() subtype optimization constraints
Category	Struct member
Syntax	keep reset_gen_before_subtypes()

When subtype optimization is turned off by default, this constraint causes the generator to ignore all previously defined **gen_before_subtypes()** constraints for the enclosing *struct* or unit. Any such constraints defined after the reset shall be followed.

When subtype optimization is turned on by default, this constraint turns off subtype optimization for the enclosing struct or unit. When subtype optimization is forced on or off, this constraint has no effect.

Syntax example:

```
keep reset_gen_before_subtypes();
```

7.3.12 value()

Purpose	Modify generation sequence
Category	Pseudo-method
Syntax	value(item: exp)
Parameters	<i>item</i> A legal <i>e</i> expression.

This generates values for any data items that are contained in the expression and returns the value of the expression. This method affects generation order and also makes the constraint unidirectional.

Example

```
keep a == value(b + c);
```

The constraint shown above has two results.

- b and c are generated before a.
- The value of a cannot otherwise be constrained.

Syntax example:

```
keep i < value(j);
```

7.3.13 constraint-bool-exp

Purpose	Define a constraint on a generatable item
Category	Expression
Syntax	<i>bool-exp</i> [or and => <i>bool-exp</i>] ...
Parameters	<i>bool-exp</i> An expression that returns either TRUE or FALSE when evaluated at run time.

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 1 shows the *e* special constructs that are useful in constraint Boolean expressions.

Table 1—Constraining Boolean expression

Constraint	Definition
soft	A keyword that indicates the constraint is either a soft value constraint or a soft order constraint. See 7.3 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 <i>struct</i> .

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 ($a \Rightarrow b$) 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 ($b \Rightarrow c$) 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:

```
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: 3.1.1.

Syntax example:

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

7.3.14 gen-item

Purpose	Identifies a generatable item
Category	Expression
Syntax	<code>[me.]field1-name[field2-name ...]</code> <code>[it] [it].field1-name[field2-name ...]</code>
Parameters	<i>field-name</i> The name of a field in the current <i>struct</i> or <i>struct</i> 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:* 2.3.3.

Syntax example:

```
me.protocol
```

7.4 Invoking generation

There are two ways of invoking generation.

- a) Generation is invoked automatically when generating the tree of structures starting at **sys**.
 <<VIT:: link to the phases of 'test'. Should be explained somewhere>>
- b) Generation can be called for any data item by using the **gen** action. The scope of this type of generation is restricted (see 7.4.1). The generation order is (recursively):
 - 1) Allocate the new *struct*.
 - 2) Call **pre_generate()**.
 - 3) Perform generation
 - 4) Call **post_generate()**.

7.4.1 gen

Purpose	Generate values for an item
Category	Action
Syntax	gen <i>gen-item</i> [keeping {[<i>it</i>]. <i>constraint-bool-exp</i> ; ...}]
Parameters	<i>gen-item</i> A generatable item. If the expression is a <i>struct</i> , it is automatically allocated, and all fields under it are generated recursively, in depth-first order.
	<i>constraint-bool-exp</i> A simple or compound Boolean expression (see 7.3.13).

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:

```

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

```

7.4.2 pre_generate()

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

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 solver.

NOTE—Prefix the ! character (see 4.7) 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;
};

```

7.4.3 post_generate()

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

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;
};

```