Sugar 2.0
An Introduction

Cindy Eisner[1] Dana Fisman[1,2]

[1] IBM Haifa Research Laboratory
[2] Weizmann Institute of Science, Rehovot, Israel
{eisner,danaf}@il.ibm.com

1 Introduction

Sugar is a language for the formal specification of hardware. It is used to describe properties that are required to hold of the design under verification. For instance, it might be required that "if a request is made, it must stay asserted until a grant is received". This simple English specification leaves a lot to be desired: Which signals indicate a request, and which indicate a grant? By "until" do we mean that the request signal must stay asserted only up to, but not including the cycle in which a grant is received, or must they also stay asserted the cycle of the grant? These questions could of course be answered easily in English. But for more complicated specifications, using English to explain exactly what is meant can be difficult. Sugar provides a means to write specifications which are both easy to read and mathematically precise. Thus, one important use of Sugar is for documentation, either in place of or in conjunction with an English specification.

Another important use of Sugar is as input to verification tools. A Sugar specification can be used as input to a formal verification tool, such as a model checker or a theorem prover, which can automatically determine whether a design obeys its Sugar specification. A Sugar specification can also be used to automatically generate simulation checkers, which can then be used to check the design "informally" using simulation.

Sugar has been selected by the Accellera Formal Verification Technical Committee for standardization by the IEEE. This article provides a short introduction to Sugar. It describes the basic structure of the language, and some of the major operators. For the full definition of the language, including more examples, formal semantics, implementation issues and a dicussion of design decisions, see the complete Sugar definition at
`www.haifa.il.ibm.com/projects/verification/sugar/literature.html`.

2 An overview of Sugar

Sugar consists of four layers. The *boolean layer* is comprised of boolean expressions. For instance, `a` is a boolean expression, having the value `1` when signal `a` is high, and `0` when signal `a` is low. Sugar interprets a high signal as `1`, and a low signal as `0`, independent of whether the signal is active-high or active-low.

In the remainder of this tutorial, we will assume that all signals are active-high, so we will say that `a` is a boolean expression having the value `1` when signal `a` is asserted, and `0` otherwise.

The *temporal layer* consists of temporal properties which describe the relationships between boolean expressions over time. For instance, `always (req -> next ack)` is a temporal property expressing the fact that whenever (`always`) signal `req` is asserted, then (`->`) at the next cycle (`next`), signal `ack` is asserted.

The *verification layer* consists of directives which describe how the temporal properties should be used by verification tools. For instance, `assert always (req -> next ack)` is a verification directive that tells the tools to verify that the property `always (req -> next ack)` holds. Other verification directives include an instruction to assume, rather than verify, that a particular temporal property holds, or to specify coverage criteria for a simulation tool. The verification layer also provides a means to group Sugar statements into *verification units*.

Finally, the *modeling layer* provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer is also used to give names to properties and other entities from the temporal layer.

Sugar comes in three flavors, corresponding to the hardware description languages Verilog and VHDL, and to the language EDL, the environment description language of IBM's RuleBase model checker. The flavor determines the syntax of the boolean and modeling layers. In all three flavors, the temporal layer and the verification layer are identical.

3 The boolean layer

The boolean layer consists of boolean expressions, in a syntax determined by the flavor of Sugar being used. For instance, in the Verilog flavor, the boolean expression `a[0:3] & b[0:3]` represents the bit-wise "and" of the most significant four bits of vectors `a` and `b`. In the VHDL flavor, this is written using VHDL syntax as `a(0 to 3) and b(0 to 3)`. In the EDL flavor, this is `a(0..3) & b(0..3)`.

In the remainder of this tutorial, only the Verilog flavor is used for examples.

4 The temporal layer

The temporal layer consists of properties describing the relationships between boolean expressions over time. Sugar can be used to reason about synchronous designs with a single clock or with multiple clocks, as well as about asynchronous designs; it can be used with cycle-based simulators or event-based simulators, as well as formal verification tools (which are usually, although not necessarily, cycle-based). Since the simplest case is a synchronous, singly-clocked design, that is the model we will assume in this tutorial. In addition, we will assume that the single clock has been declared as a default clock, so that our examples will

ignore the clock altogether. The clock operator and its use is described in the full Sugar definition at
`www.haifa.il.ibm.com/projects/verification/sugar/literature.html`.

4.1 Some basic temporal operators

`always` and `never`

One of the simplest temporal properties states that some boolean expression holds at all times. For instance, the following property expresses the fact that signals `ena` and `enb` should always be mutually exclusive:

```
always !(ena & enb)
```
(1)

The use of the temporal operator `always` indicates that the boolean expression `!(ena & enb)` must hold at every cycle. It is important not to forget the `always` operator. Leaving it out gives the property

```
!(ena & enb)
```
(2)

which states that `ena` and `enb` are mutually exclusive at the initial cycle only. Think of it this way: Sugar properties reason about sequences of cycles. Every Sugar property starts reasoning at the beginning of the sequence, and progresses from there only by means of a temporal operator. Thus, Property 2 reasons about only the initial cycle, because it has no temporal operator to "push it forward". In Property 1, the use of the temporal operator `always` causes the boolean expression `!(ena & enb)` to be applied to all cycles in the sequence, rather than just the initial cycle.

The `never` operator allows us to specify conditions that should never hold. Thus, to state that `ena` and `enb` are mutually exclusive, we could have used the following

```
never (ena & enb)
```
(3)

instead of Property 1.

`next`

The `always` and `never` operators allow us to specify a condition that holds or does not hold at *all* cycles. Other operators allow us to be more specific about the timing. The operator `next` takes us forward one clock cycle. Thus, the property

```
always (req -> next ack)
```
(4)

says that whenever signal `req` is asserted, signal `ack` is asserted the next cycle. Property 4 uses the implication operator (`->`) which can be read as "if-then". Thus, `(req -> next ack)` can be read as "if `req` is asserted, then at the next cycle `ack` is asserted". By enclosing this within the `always` operator in Property 4 we get "at every cycle, if `req` is asserted, then at the next cycle `ack` is asserted",

or simply "whenever signal `req` is asserted, signal `ack` is asserted at the next cycle".

Sugar also provides a shorthand for repetitions of `next`, so that, for instance, the following:

```
always (req -> next[4] ack)
```
(5)

specifies that `ack` is required to occur four cycles after `req`, rather than the very next cycle.

`eventually!`

The `next` operator moves us forward exactly one cycle. The `eventually!` operator allows us to move forward without specifying exactly when to stop. The property

```
always (req -> eventually! ack)
```
(6)

states that whenever signal `req` is asserted, signal `ack` should be asserted some time in the inclusive future. Anytime will do: either now, or at the next cycle, or the one after that, and so on. The exclamation point (!) which is part of the name of the `eventually!` operator indicates that it is a *strong* operator. For more on weak and strong operators, see below.

The until operators

The until operators provide another way to move forward, this time while putting a requirement on the cycles in which we are moving. For instance, the property

```
always (req -> next (busy until end))
```
(7)

states that whenever signal `req` is asserted, then, starting at the next cycle, signal `busy` will be asserted up until signal `end` is asserted. Sub-property `busy until end` specifies that `busy` will be asserted up to, but not necessarily including, the cycle where `end` is asserted. Thus, if signal `end` is asserted the cycle after signal `req` is asserted, Property 7 does not require that signal `busy` be asserted at all. In order to include the cycle where `end` is asserted, use the operator `until_`. The underscore (`_`) is intended to represent the extra cycle in which we require that busy should stay asserted,

The `until` and `until_` operators are *weak* operators, in that they do not require that the terminating condition eventually occur. Thus, in Property 7, signal `end` is not required to eventually occur. If it does not, then signal `busy` must stay asserted forever. Both `until` and `until_` come in *strong* forms as well, which are `until!` and `until!_`. The strong forms require that the terminating condition eventually occur. The use of the exclamation point (!) in `until!` and `until!_` is intended to represent the strength of the operator.

The before operators

The before operators provide an easy way to state that we require some signal to be asserted before some other signal. For instance, suppose that we have a pulsed signal called `req`, and we have the requirement that before we can make a second request, the first must be acknowledged. We can express this in Sugar as follows:

```
always (req -> next (ack before req))
```
(8)

We need the `next` to take us forward one cycle so that the `req` in `ack before req` is sure to refer to some other `req`, and not the one we have just seen. To understand this, let us examine a flawed version of the same specification:

```
always (req -> (ack before req))
```
(9)

Consider Figure 1. Property 9 requires that `(ack before req)` holds at every

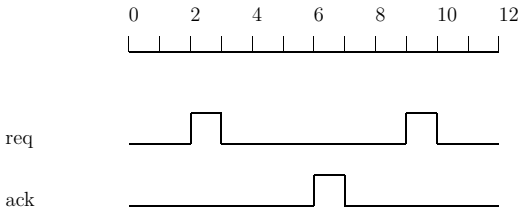

Fig. 1. Behavior described by Property 8, and not described correctly by Property 9

cycle in which `req` holds. Consider, for example, cycle 2. Signal `req` is asserted. Therefore, `(ack before req)` must hold at cycle 2. However, it does not, because starting at cycle 2 and looking forward, we first see an assertion of signal `req` (at cycle 2), and only afterwards an assertion of signal `ack` (at cycle 6), so `req` is asserted before `ack`, and not the other way around. Property 8, on the other hand, states what we want: at cycle 2, for example, we require `next(ack before req)` to hold. Therefore, we require that `(ack before req)` holds at cycle 3. Starting at cycle 3 and looking forward, we first see an assertion of `ack` (at cycle 6), and only afterwards an assertion of `req` (at cycle 9).

The `before` operator requires that its first operand happen strictly before its second. In order to specify that something must happen before or at the same cycle as something else, use `before_`. The underscore (`_`) is intended to represent the cycle in which we allow an overlap between the left and right sides.

The operators `before` and `before_` are weak operators. Thus, in Property 8, an assertion of `ack` is not required to occur. If it does not, then `req` must stay deasserted forever. The strong forms, `before!` and `before!_`, do require that `ack` eventually occur.

4.2 Sugar Extended Regular Expressions (SEREs)

The basic temporal operators presented above can be combined to give quite complicated properties. However, writing such properties is sometimes cumbersome and reading them can be difficult. For instance, the following property

```
always (reqin -> next(ackout ->
 next(!abortin -> (ackin & next ackin))))          (10)
```

states that if signal `reqin` is asserted, then if in the next cycle signal `ackout` is asserted, then if in the following cycle signal `abortin` is not asserted, then starting at that cycle, signal `ackin` is asserted for two consecutive cycles. Sugar provides an alternative way to reason about sequences of values which is in many cases more concise and easier to read and write. It is based on an extension of regular expressions, called Sugar Extended Regular Expressions, or SEREs.

Building SEREs using ";"

A Sugar Extended Regular Expression (SERE) provides an easy way to string together sequences of boolean expressions over time. The simplest SERE is built from a series of boolean expressions, separated by semi-colons (;). Thus, the SERE {`reqin;ackout;!abortin`} describes a sequence in which `reqin` is asserted the first cycle, `ackout` the second, and `abortin` is not asserted the third.

NOTE: SEREs are grouped using curly braces ({}) as opposed to boolean expressions, which are grouped using parentheses (()).

Weak suffix implication {} `|->` {}

A SERE is not a Sugar property in and of itself. In order to use a SERE to build a Sugar property, we link a SERE with another Sugar property or with another SERE. For instance, in order to express Property 10 using SEREs, we could write the following:

```
always {reqin;ackout;!abortin} |-> {ackin;ackin}          (11)
```

Property 11 can be read as "whenever we have a sequence of `reqin` followed by `ackout` followed by `!abortin`, then (at the cycle of `!abortin`) we expect to see `ackin`, followed by another cycle of `ackin`".

Notice the suffix implication operator `|->`, which takes two SEREs as its operands, and compare it to the implication operator `->`, which takes two properties. There is an important difference in how they work. When an implication operator works on two properties, it means that the property on the right-hand-side should hold at the same cycle as that on the left-hand-side. So the property

```
always ((special_req & next ack) -> en_special)           (12)
```

states that if `special_req` is asserted and the following cycle `ack` is asserted, then `en_special` must be asserted during the cycle in which `special_req` was

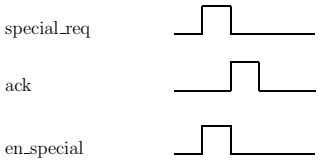

Fig. 2. Behavior described by Property 12

asserted. This is illustrated by Figure 2 below. Signal `special_req` is asserted at cycle 2, and signal `ack` is asserted at cycle 3. Therefore, the sub-property `(special_req & next ack)` holds at cycle 2. The use of the implication operator (`->`) between `(special_req & next ack)` and `en_special` means that we want the two sub-properties to hold at the same cycle. Therefore, `en_special` must hold at cycle 2, the cycle when `(special_req & next ack)` holds.

The suffix implication operator `|->` has a different meaning: it means that the SERE on the right-hand-side should hold at the cycle in which the SERE on the left-hand-side completed. Thus, in Property 11, we expect to see a sequence of two assertions of signal `ack` starting at the cycle in which we finished seeing the sequence {`reqin;ackout;!abortin`}. The use of the pipe (`|`) symbol in the `|->` operator is intended to convey the intuition that we stop, draw a line at the cycle we have reached, and start evaluating there.

Repetitions and other SERE operators

Sugar provides a shorthand form for expressing repetitions of sub-sequences, and other common forms. For instance, the consecutive repetition operator `[*i..j]` lets us easily express consecutive repetitions of a sub-sequence. Suppose we want to express the following property: "Whenever we have a sequence of `req` followed by `ack`, we should see a full transaction starting the following cycle. A full transaction starts with an assertion of signal `start_trans`, followed by one to eight consecutive data transfers, followed by the assertion of signal `end_trans`. A data transfer is indicated by the assertion of signal `data`." Using the consecutive repetition operator `[*i..j]`, we express this as follows:

```
always {req;ack} |=> {start_trans;data[*1..8];end_trans}      (13)
```

The `|=>` operator is similar to the `|->` operator, except that it takes us forward one cycle, so that `start_trans` is expected the cycle after `ack` is asserted. The SERE `data[*1..8]` means either one assertion of `data`, or two consecutive assertions ({`data;data`}), etc., up to eight consecutive assertions ({`data;data;data;data;data;data;data;data`}). For instance, behavior like that shown in Figure 3 is legal according to Figure 13.

Sugar also provides an easy way to specify non-consectuive SERE repetitions. Consider Property 13, which was illustrated by Figure 3. Suppose that we want to specify instead that the data transfers are not necessarily consecutive. In

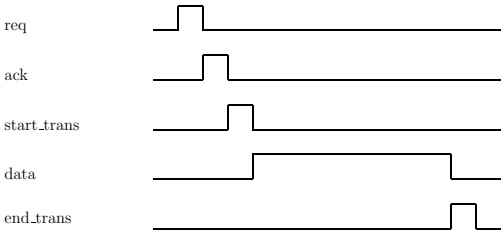

Fig. 3. Example of legal behavior according to Property 13

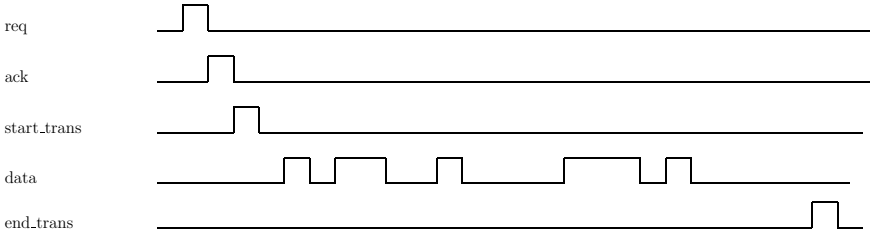

Fig. 4. Alternative behavior described by Property 14

other words, suppose that the behavior shown in Figure 4 is legal. We can code `data[=1..8]` to say that we expect to see any sequence in which the number of assertions of signal `data` is equal to eight. Thus, we can code our modified specification as follows:

```
always {req;ack} |=> {start_trans;data[=1..8];end_trans}      (14)
```

Sugar also provides shorthand for jumping to the next occurence of a boolean expression, and "anding" and "oring" two SEREs.

Strong suffix implication {} |-> {}!

When the SERE on the right-hand-side of a suffix implication is of unbounded length, the right-hand-side is not required to "reach its end". For instance, consider Property 15.

```
always {req;ack} |=> {start_trans;data[*];end_trans}      (15)
```

It is similar to Property 13, except that any number of consecutive assertions of `data` are allowed. A trace on which an assertion of `req` followed by an assertion of `ack` is followed by an assertion of `start_trans` after which `end_trans` never occurs is legal according to Property 15 as long as `data` is asserted forever, because {} |-> {} and {} |=> {} are weak properties.

If we want to require the right-hand-side of Property 15 to "reach its end", we can use strong suffix implication({} |-> {}! or {} |=> {}!), as follows:

```
always {req;ack} |=> {start_trans;data[*];end_trans}!      (16)
```

Property 16 is identical to Property 15, except that the strength of the suffix implication (indicated by the "!" at the end) requires that `end_trans` must eventually occur.

4.3 `forall`

In the previous sections, we have seen some basic temporal operators, and also seen how to build a Sugar formula using SEREs. The `forall` operator is a different kind of operator. Rather than have some temporal meaning, it provides an easy shorthand for expressing a group of related properties. For instance, suppose that we want to specify that whenever a read is received (signal `read_start` is asserted), it will eventually complete (signal `read_complete` will be asserted). The catch is that the reads are tagged, so that in order to know whether or not an assertion of `read_complete` belongs to an assertion of `read_start`, we must compare the tags. Suppose that their are eight possible tags, indicated by `tag[2:0]`. Instead of writing out eight very similar properties, we can use the `forall` operator to get a single property as follows:

```
forall i in 0..7:
 always((read_start & tag[2:0]==i) ->
 eventually! (read_complete & tag[2:0]==i))          (17)
```

4.4 More temporal operators

The next_event operators

The next_event operators are a conceptual extension of the next operators. While `next` refers to the next cycle, `next_event` refers to the next cycle in which some boolean condition holds. For instance, the Sugar formula

```
always (high_pri_req -> next_event(grant)(dst==high_pri))      (18)
```

is used to expressed the requirement that whenever a high priority request is received (signal `high_pri_req` is asserted), then the next grant (assertion of signal `grant`) must be to a high priority requester (signal `dst` has the value `high_pri`). The `next_event` operator is a weak operator. Thus, in Property 18, `grant` is not required to occur. The strong version,

```
always (high_pri_req -> next_event!(grant)(dst==high_pri))     (19)
```

requires, in addition, that `grant` eventually be asserted.

As with `next`, Sugar also provides a shorthand for repetitions of `next_event` so that, for instance, the following:

```
always (req -> next_event(ready)[4](last_ready))          (20)
```

specifies that every time a request is issued (signal `req` is asserted), signal `last_ready` must be asserted on the fourth assertion of signal `ready`, instead of on the very next one.

`abort`

The `abort` operator provides a way to lift any future obligations of a property when some boolean condition is observed. For instance, the following property:

```
always (start ->
  ((always (req -> eventually! ack)) abort interrupt))    (21)
```

says that after an assertion of `start`, every request (assertion of `req`) should receive an acknowledge (assertion of `ack`), except that if an interrupt is received, all future obligations to acknowledge a request are lifted.

5 The verification layer

We have seen how to express properties using the temporal layer of Sugar. The verification layer provides a way to instruct the verification tools on what to do with these properties via directives to the verification tools. Although not every verification tool will support every verification directive, the verification layer provides a standard way to communicate with all tools. In addition, it provides a way to group related directives and other Sugar statements into *verification units*.

The basic verification directive is `assert`, which instructs the verification tool to verify that the asserted property holds. The verification directive `assume` instructs the tool to assume, rather than verify, that a property holds. An assumption can be considered a constraint on design inputs. In other words, an asserted property is required to hold only along those paths which obey the assumption.

Other verification directives work with SEREs rather than with complete Sugar properties. The verification directive `restrict` is a constraint that every trace match a specified SERE. A restriction can be used to specify that an assertion should hold only for specific scenarios. For instance, a restriction can be used to specify a reset sequence. The verification direction `cover` directs the verification tool to check that a path matching the specified SERE was covered by a simulation test suite.

A *verification unit* is a group of verification directives or other Sugar statements that are grouped together, given a name, and possibly bound to a design module or module instance (an unbound verification unit refers to the top level module). For instance, the following groups an assumption and an assertion together into a verification unit called `non_decreasing`, which is bound to module `my_fifo`.

```
vunit non_decreasing(my_fifo) {
    assume forall n in 0..15:
        always ((en_in & data_in[0:3]==n) ->
          always (en_in -> (data_in[0:3] >= n)));
    assert forall n in 0..15:
```

```
        always ((en_out & data_out[0:3]==n) ->
          always (en_out -> data_out[0:3] >= n))
}
```

A verification unit is the only entity that can be directly accessed by a verification tool.

6 The modeling layer

The modeling layer provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer is also used to give names to properties and other entities from the temporal layer. The modeling layer, like the boolean layer, comes in three flavors, corresponding to Verilog, VHDL, and EDL. As stated previously, this tutorial uses only the Verilog flavor.

As an example of the use of the modeling layer, consider a design for which the specification is the one we considered at the start of this tutorial: "if a request is made, it must stay asserted until a grant is received". We can make this precise in Sugar with the following verification unit:

```
vunit every_request_gets_a_grant {
  assert always ((req_en & req_out) ->
              ((req_en & req_out) until_ gnt));
}
```

If we would like to make this a little more concise, we can use the following instead:

```
vunit getgrant {
  wire request;
  assign request = req_en & req_out;
  assert always (request -> (request until_ gnt));
}
```

Verification unit `getgrant` expresses the same specification as verification unit `every_request_gets_a_grant`, but more concisely. Signal `request` is an auxiliary signal, not present in the design, but declared solely for the purpose of verification unit `getgrant`.

7 Conclusion

This short introduction gave a taste of Sugar. For the full definition of the language, including more examples, formal semantics, implementation issues and a dicussion of design decisions, see the complete Sugar definition at `www.haifa.il.ibm.com/projects/verification/sugar/literature.html`.