Assertion-Based Verification (ABV)

- It has been claimed that assertion based verification:

  "*is likely to be the next revolution in hardware design verification*"

- Basic idea:
  - document designs with formal properties
  - use simulation (dynamic) and model checking (static)

- Problem: too many languages
  - academic logics: LTL, CTL
  - tool-specific industrial versions:
    Intel, Cadence, Motorola, IBM, Synopsys

- What to do? Solution: a competition!
  - run by Accellera organisation
  - results standardised by IEEE
  - lots of politics

IBM's *Sugar* and Accellera's PSL

- *Sugar 1*: property language of IBM RuleBase checker
  - CTL plus *Sugar Extended Regular Expressions* (SEREs)
- Competition finalists: IBM's *Sugar 2* and Motorola's *CBV*
  - Intel/Synopsys ForSpec eliminated earlier (apparently industry politics involved)
- *Sugar 2* is based on LTL rather than CTL
  - has CTL constructs: "Optional Branching Extension" (OBE)
  - has clocking constructs for temporal abstraction
- Accellera purged "Sugar" from it property language
  - the word "Sugar" was too associated with IBM
  - language renamed to PSL
  - SEREs now *Sequential Extended Regular Expressions*
- Lobbying to make PSL more like ForSpec (align with SVA)

SEREs: Sequential Extended Regular Expressions

- SEREs are from the industrial PSL (more on PSL later)
- Syntax :

$$
\begin{array}{rcll}
r & ::= & p & \text{(Atomic formula } p \in AP\text{)} \\
 & | & !\,p & \text{(Negated atomic formula } p \in AP\text{)} \\
 & | & r_1 \mid r_2 & \text{(Disjunction)} \\
 & | & r_1 \,\&\&\, r_2 & \text{(Conjunction)} \\
 & | & r_1 \,;\, r_2 & \text{(Concatenation)} \\
 & | & r_1 : r_2 & \text{(Fusion)} \\
 & | & r[*] & \text{(Repeat)}
\end{array}
$$

- Semantics:
  (w ranges over finite lists of states s; $|w|$ is length of w;
  $w_1.w_2$ is concatenation; **head** w is head; $\langle\rangle$ is empty word)

$$
\begin{array}{ll}
[\![p]\!](w) & = p \in L(\textbf{head } w) \wedge |w| = 1 \\
[\![\,!\,p]\!](w) & = \neg(p \in L(\textbf{head } w)) \wedge |w| = 1 \\
[\![r_1 \mid r_2]\!](w) & = [\![r_1]\!](w) \ \vee \ [\![r_2]\!](w) \\
[\![r_1 \,\&\&\, r_2]\!](w) & = [\![r_1]\!](w) \ \wedge \ [\![r_2]\!](w) \\
[\![r_1 \,;\, r_2]\!](w) & = \exists w_1\ w_2.\ w = w_1.w_2 \ \wedge \ [\![r_1]\!](w_1) \ \wedge \ [\![r_2]\!](w_2) \\
[\![r_1 : r_2]\!](w) & = \exists w_1\ s\ w_2.\ w = w_1.s.w_2 \ \wedge \ [\![r_1]\!](w_1.s) \wedge \ [\![r_2]\!](s.w_2) \\
[\![r[*]]\!](w) & = w{=}\langle\rangle \vee \exists w_1 \cdots w_l.\ w{=}w_1.\cdots.w_l \wedge [\![r]\!](w_1) \wedge \cdots \wedge [\![r]\!](w_l)
\end{array}
$$

Example SERE

- Example

  A sequence in which `req` is asserted, followed four cycles later by an assertion of `grant`, followed by a cycle in which `abortin` is not asserted.

- Define p`[*3]` $= p;p;p$
- Then the example above can be represented by the SERE:

  `req;T[*3];grant;!abortin`

- In PSL this could be written as:

  `req;[*3];grant;!abortin`

  - where `[*3]` abbreviates `T[*3]`
  - more 'syntactic sugar' later
  - e.g. `true`, `false` for `T`, `F`

PSL Foundation Language (FL is LTL + SEREs)

- Syntax:

$$
\begin{array}{llll}
f & ::= & p & \text{(Atomic formula - } p \in AP\text{)} \\
 & \mid & !\,f & \text{(Negation)} \\
 & \mid & f_1 \texttt{ or } f_2 & \text{(Disjunction)} \\
 & \mid & \texttt{next } f & \text{(Successor)} \\
 & \mid & \{r\}(f) & \text{(Suffix implication: } r \text{ a SERE)} \\
 & \mid & \{r_1\} \texttt{ |-> } \{r_2\} & \text{(Suffix next implication: } r_1,\ r_2 \text{ SEREs)} \\
 & \mid & [f_1 \texttt{ until } f_2] & \text{(Until)}
\end{array}
$$

- Semantics (omits clocking, weak/strong distinction)

$$
\begin{array}{ll}
[\![p]\!]_M(\pi) & = p \in L(\pi(0)) \\
[\![!\,f]\!]_M(\pi) & = \neg([\![f]\!]_M(\pi)) \\
[\![f_1 \texttt{ or } f_2]\!]_M(\pi) & = [\![f_1]\!]_M(\pi) \ \vee \ [\![f_2]\!]_M(\pi) \\
[\![\texttt{next } f]\!]_M(\pi) & = [\![f]\!]_M(\pi{\downarrow}1) \\
[\![\{r\}(f)]\!]_M(\pi) & = \forall \pi'\ w.\ (\pi = w.\pi' \ \wedge \ [\![r]\!]_M(w)) \ \Rightarrow \ [\![f]\!]_M(\pi') \\
[\![\{r_1\}\texttt{|->}\{r_2\}]\!]_M(\pi) & = \forall \pi'\ w_1\ s.\ (\pi = w_1.s.\pi' \wedge [\![r_1]\!]_M(w_1.s)) \\
 & \qquad \Rightarrow \exists \pi''\ w_2.\ \pi' = w_2.\pi'' \wedge [\![r_2]\!]_M(s.w_2) \\
[\![[f_1 \texttt{ until } f_2]]\!]_M(\pi) & = \exists i.\ [\![f_2]\!]_M(\pi{\downarrow}i) \ \wedge \ \forall j.\ j<i \ \Rightarrow \ [\![f_1]\!]_M(\pi{\downarrow}j)
\end{array}
$$

- There is also an Optional Branching Extension (OBE)
  - completely standard CTL: **EX**, **E**[$-$ $-$ **U** $-$ $-$], **EG** etc.

Combining SEREs with LTL formulae

- Formula $\{r\}f$ means LTL formula f true after SERE r
- Example

  After a sequence in which `req` *is asserted, followed four cycles later by an assertion of* `grant`*, followed by a cycle in which* `abortin` *is not asserted, we expect to see an assertion of* `ack` *some time in the future.*

- Can represent by

```
always {req;[*3];grant;!abortin}(eventually ack)
```

- where `eventually` and `always` are defined by:

```
eventually f  =  [true until f]
always f      = !(eventually !f)
```

- N.B. Ignoring strong/weak distinction
  - strong/weak distinction important for dynamic checking
  - semantics when simulator halts before expected event
  - strictly should write `until!`, `eventually!`

SERE examples

- How can we modify

  `always reqin;ackout;!abortin |-> ackin;ackin`

  so that the two cycles of `ackin` start the cycle after `!abortin`

- Two ways of doing this

  ```
  always{reqin;ackout;!abortin}|->{true;ackin;ackin}
  always{reqin;ackout;!abortin}|=>{ackin;ackin}
  ```

- `|=>` is a defined operator

  `{r1}|=>{r2}  =  {r1}|->{true;r2}`

- Note: `true` and `T` are synonyms

Examples of defined notations: consecutive repetition

- Define

```
r[+]       = r;r[*]
             __
             |  false[*]  if i=0
r[*i]      = |
             |  r;...;r otherwise (i repetitions)
             __

r[*i..j] = r[*i] | r[*(i+1)] | ... | r[*j]
[+]        = true[+]
[*]        = true[*]
```

- Example

  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 the 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`

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

Fixed number of non-consecutive repetitions

- Example

  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 the signal `start_trans`, followed by eight not necessarily consecutive data transfers, followed by the assertion of signal `end_trans`. A data transfer is indicated by the assertion of signal `data`

- Can represent by

```
always
 {req;ack} |=>
 {start_trans;
  {{!data[*];data}[*8];!data[*]};
  end_trans}
```

- Define: `b[= i] = {!b[*];b}[*i];!b[*]`
- Then have a nicer representation

```
always{req;ack}|=>{start_trans;data[= 8];end_trans}
```

Variable number of non-consecutive repetitions

- Example

  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 the signal `start_trans`, followed by $\boxed{\text{one to eight}}$ not necessarily consecutive data transfers, followed by the assertion of signal `end_trans`. A data transfer is indicated by the assertion of signal `data`

- Define

```
b[= i..j] = {b[= i]} | {b[= (i+1)]} | ... | {b[= j]}
```

- Then

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

- These examples are meant to illustrate how PSL/Sugar is much more readable than raw CTL or LTL

Clocking

- Basic idea: `b@clk` samples `b` on rising edges of `clk`
- Can clock SEREs (`r@clk`) and formulae (`f@clk`)
- Can have several clocks
- Official semantics messy due to clocking
- Can 'translate away' clocks by pushing `@clk` inwards
  - rules given in PSL manual
  - roughly: `b@clk` $\rightsquigarrow$ `{!clk[*];clk & b}`

Model checking PSL (outline)

- SEREs checked by generating a finite automaton
  - recognise regular expressions
  - these automata are called “satellites”
- FL checked using standard LTL methods
- OBE checked by standard CTL methods
- Can also check formula for runs of a simulator
  - this is dynamic verification
  - semantics handles possibility of finite paths – messy!
- Commercial checkers only handle a subset of PSL

PSL layer structure

- Boolean layer has atomic predicates
- Temporal layer has LTL (FL) and CTL (OBE) properties
- Verification layer has commands for how to use properties
  - e.g. `assert`, `assume`

```
assert always (!en1 & en2))
   |      |          |
   |      |          |--- Boolean layer
   |      |
   |      |--------------- temporal layer
   |
   |--------------------- verification layer
```

- Modelling layer: HDL specification of e.g. inputs, checkers
  - e.g. augment `always(Req  ->  eventually! Ack)`
  - add counter to keep track of numbers of `Req` and `Ack`

PSL/Sugar summary

- Combines together LTL and CTL
- Regular expressions – SEREs
- LTL – Foundation Language formulae
- CTL – Optional Branching Extension
- Relatively simple set of primitives + definitional extension
- Boolean, temporal, verification, modelling layers
- Semantics for static and dynamic verification
  (needs strong/weak distinction)