Agenda

Introduction: SystemVerilog Motivation Vassilios Gerousis, Infineon Technologies Accellera Technical Committee Chair	**Session 3: SystemVerilog Assertions** **Language Tutorial** Bassam Tabbara, Novas Software **Technology and User Experience** Alon Flaisher, Intel **Using SystemVerilog Assertions and Testbench Together** Jon Michelson, Verification Central
Session 1: SystemVerilog for Design **Language Tutorial** Johny Srouji, Intel **User Experience** Matt Maidment, Intel	**Session 4: SystemVerilog APIs** Doug Warmke, Model Technology
Session 2: SystemVerilog for Verification **Language Tutorial** Tom Fitzpatrick, Synopsys **User Experience** Faisal Haque, Verification Central	**Session 5: SystemVerilog Momentum** **Verilog2001 to SystemVerilog** Stuart Sutherland, Sutherland HDL **SystemVerilog Industry Support** Vassilios Gerousis, Infineon
Lunch: 12:15 – 1:00pm	**End: 5:00pm**

Integrating Assertion and Testbench DV Methodologies

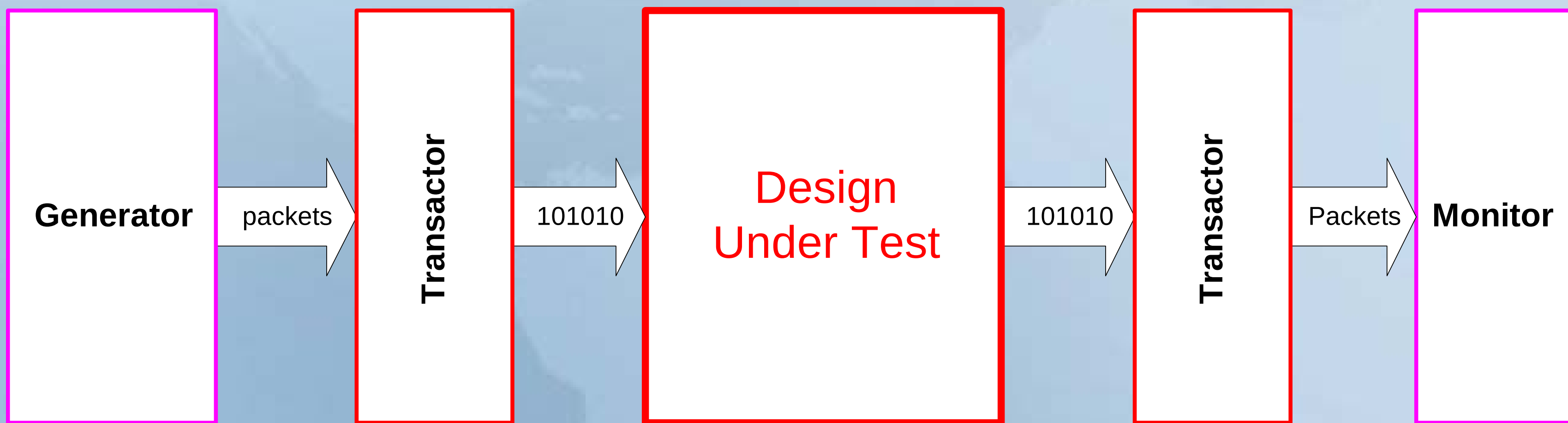
Jon Michelson
Verification Central

DAC 2003

Outline

- Typical functional DV testbench
- Brief words about assertions
- Step-by-step integration of assertions into testbench methodologies

Typical Testbench From
The Art of Verification with Vera



Operates at a higher level of abstraction

Converts stimulus to binary vectors and injects into RTL

Converts response in binary vectors to higher level data

Checks response to determine pass/fail. Operates at a higher level of abstraction

Modularized verification: Complete separation of stimulus generation and results checking enables reuse

Reuse Details

- Generators and their transactors do not directly communicate with monitors
- Monitors and their transactors watch DUT inputs, model (or obtain) correct behavior, and then check DUT outputs for correct behavior
- Generators, monitors, and maybe transactors are portable across chips that use same protocol
- Monitors are also usable regardless of what drives the DUT inputs
  - Unit level monitors can be reused at chip level and again at system level

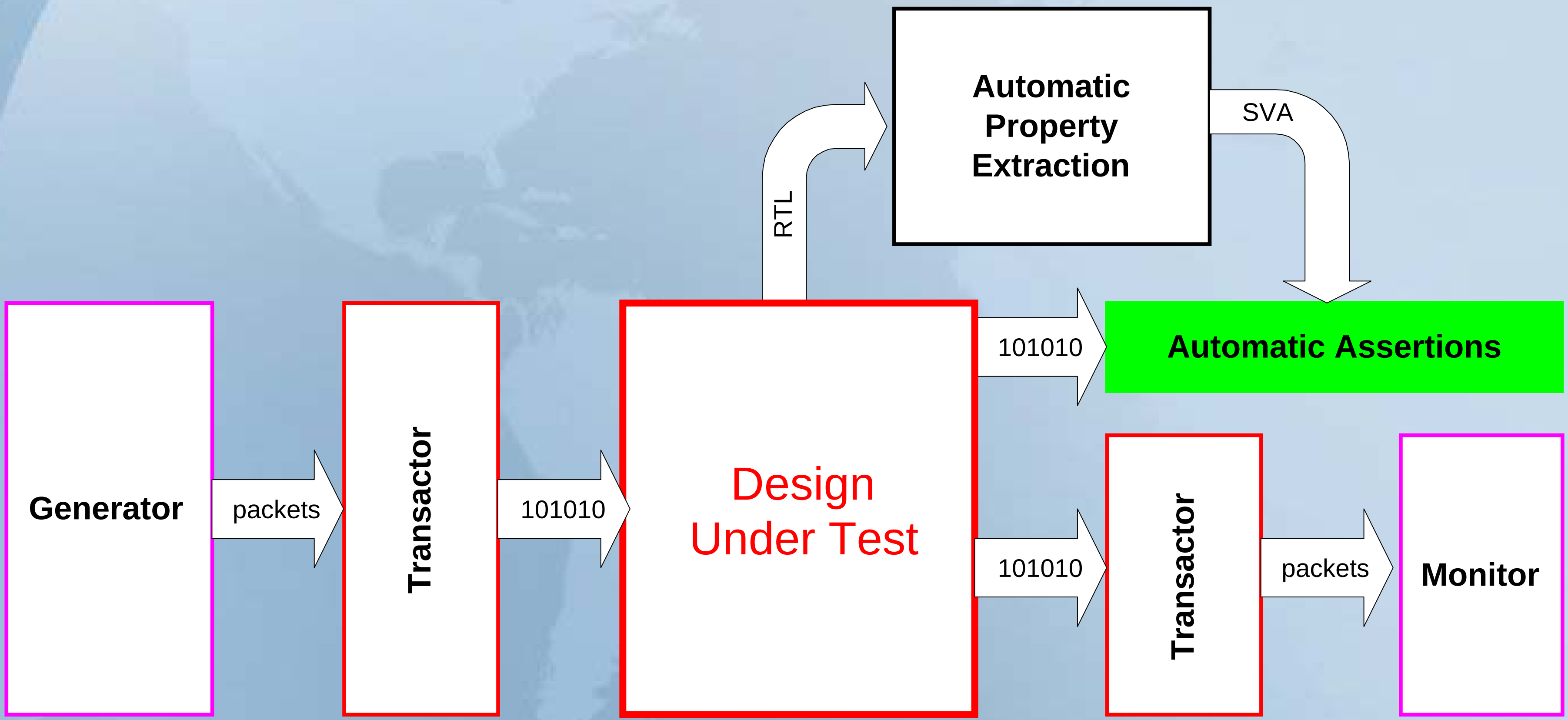
Two Components of Assertion Languages

- Assertions have two components
  - The assert, cover, assume, etc. keywords that say what to do with a sequence or expression
  - The temporal language usually associated with assertion languages that describes what is happening
- This talk uses the assertion language to specify temporal behaviors that are reacted to by the procedural part of the testbench
  - Implementation detail: The action block of assert and cover constructs enables reactivity

Assertion Strengths

- Invariants
  - Ex: one-hot state vector is always one hot
- Temporal properties
  - Ex: For every request, a grant comes back 3-10 cycles later
- Convenient for expressing coverage points
- Can be proven formally
- Usually better suited to control logic than to data path

First Step – Automatically Extract Properties



Automatically Extractable Properties

- Inferred from the design structure
  - Parallel case, full case
    - For legacy Verilog - priority/unique case in SystemVerilog provides this functionality
  - X propagation
  - Array out of bounds
  - One-hot or one-cold violations
  - Dead (unreachable) code
  - Synchronization errors
  - Etc

Example Automatic Property

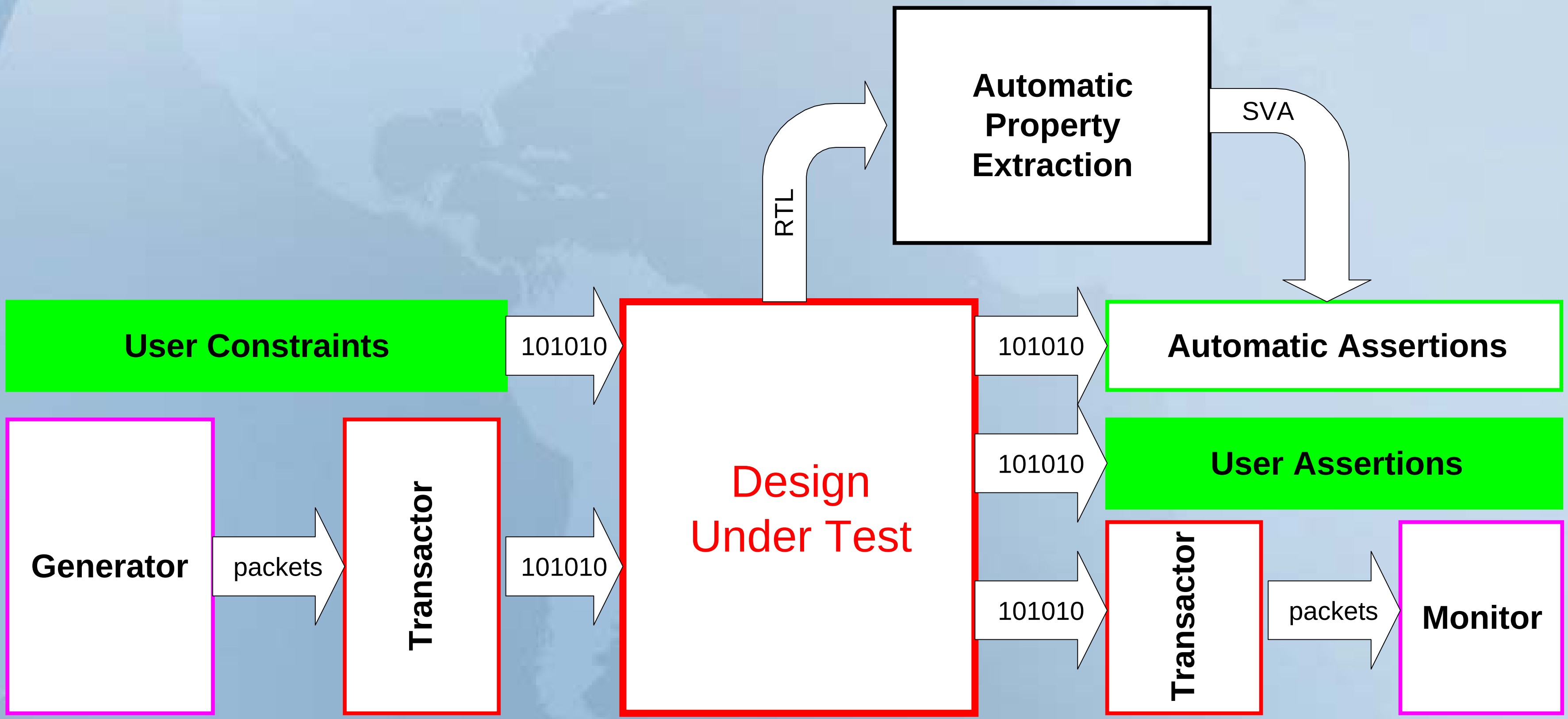
```
case (state)
  3'b001: …
  3'b010: …
  3'b100: …
endcase

automatic_property: assert ($onehot(state));
```

Why Use Automatic Properties?

- Why write properties yourself when a tool will generate them automatically in seconds?
- They find some classes of bugs sooner and more easily than otherwise possible
  - They complement your testbench nicely
  - Typically handle implementation checks
    - Sort of "extended lint" checks
- They give you examples of assertions for your design and therefore can jumpstart the process

Next Step – Add User Assertions and Constraints

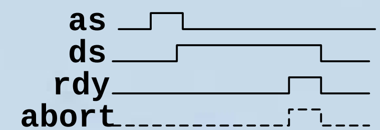


Adding User Constraints

- Formal proof of assertions may require user constraints to reflect the operating environment
- Temporal user constraints can simplify stimulus generator
  - Ex: A new bus cycle may not start for 2 clock cycles after an abort cycle has completed.
- Temporal user constraints can provide functional coverage feedback to generator
  - Ex: Skew packets toward output port 0 until the FIFO becomes full while transferring a max_length packet, then distribute packets evenly

Simplification Example

A new bus cycle may not start for 2 clock cycles after an abort cycle has completed




```
sequence abort_cycle;
 !rdy throughout (as ##1 ds[*1:$] ##1 abort);
endsequence
```

```
cover property (@(posedge clk) abort_cycle)
   wait_cnt = 2;
```

```
property wait_after_abort;
  @(posedge clk) abort_cycle |=> !as[*2];
endproperty
assert property (wait_after_abort);
```

Simulation Monitors and Constraints for Formal Analysis

```
program manual_stimulus_generator;
repeat(1000) begin
  generate_transaction(addr,data);
  while(wait_cnt > 0)
    @(posedge clk) wait_cnt--;
  end
endprogram
```

Feedback Example

Skew packets toward output port 0 until the FIFO becomes full while transferring a max_length packet, then distribute packets evenly

```
sequence full_in_max;
 fifo_full within (start_pkt ##1 !end_pkt[*MAX_LEN-1] ##1 end_pkt);
endsequence
```

```
cover property (@(posedge clk) full_in_max) skew_factor = 0;
```

```
class pktClass;
...
rand logic[2:0] destPort;
constraint pktSkew {destPort dist {0 := 5 + skew_factor,
                                   [1:3] := 5};}
endclass
```

```
program test2;
int skew_factor = 10; pktClass pktObj;
repeat(1000) begin
  pktObj.randomize();
  sendPkt(pktObj);
  end
endprogram
```

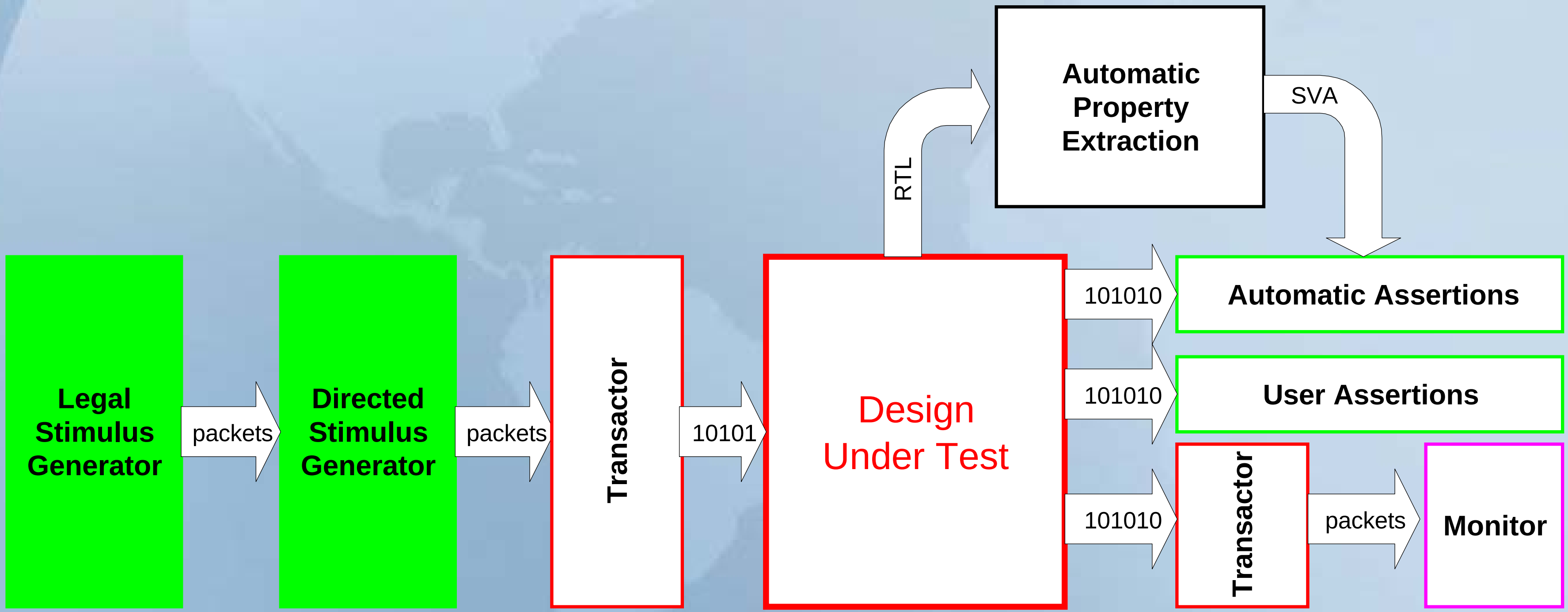
Adding User Assertions

- First, assertions enhance existing testbench
- Later, assertions replace parts of transactors and monitors
  - Used when temporal language more concisely expresses desired behavior than do existing testbench technologies
- Used with formal analysis and functional simulation
- Ex: grant comes 3-5 cycles after request

Benefits of Combined Methodology

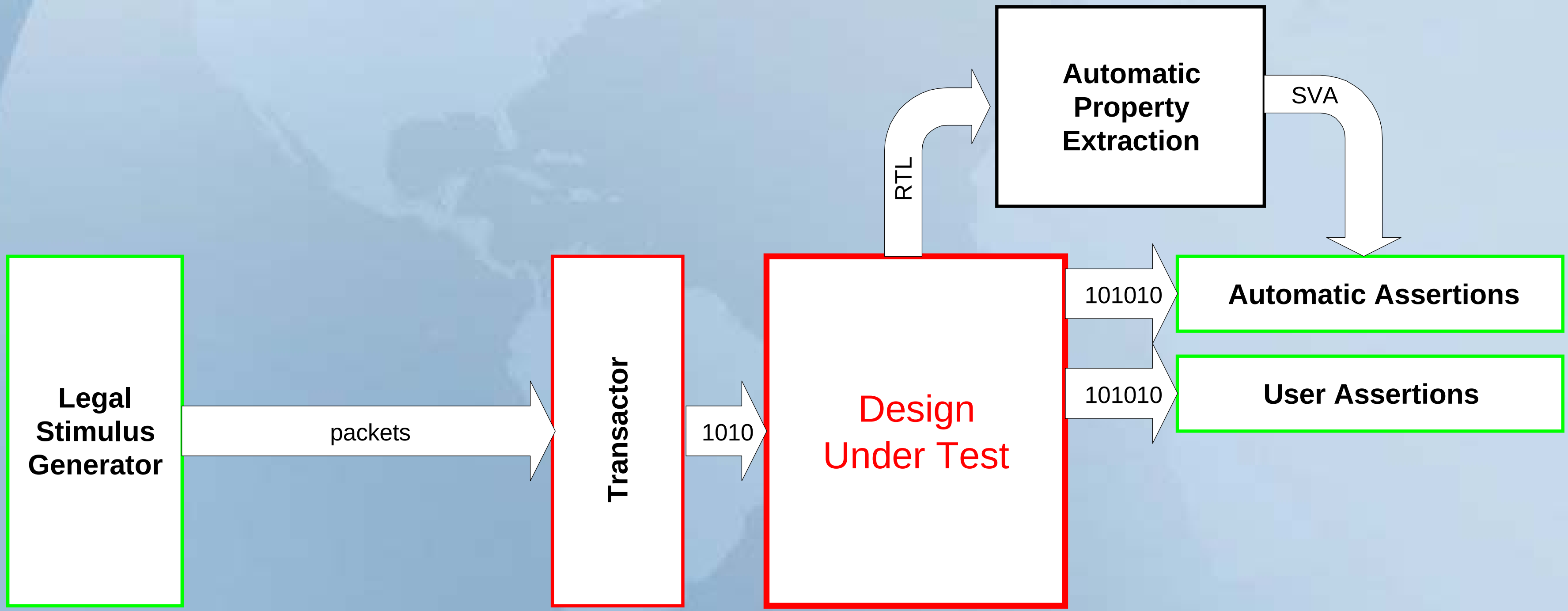
- Assertion language speeds development and debug of simulation based environment
  - Finds bugs faster
- Formal methodologies find
  - Some bugs faster
  - Some bugs which would never be found in simulation
- No change to power of simulation environment, just a more efficient reorganization
- SystemVerilog 3.1, a standard, integrated language makes these fully combined methodologies possible

Future Possibilities – Modularize Stimulus Generation



This is the complete simulation testbench

Formal Flow



Simply remove pieces of the simulation testbench

Modularizing Stimulus Generation

- Generation is split into two components
  - Legal stimulus
    - Completely random but legal stimulus
    - Used by both formal analysis and functional simulation
    - Ex: legal Ethernet packet sizes
  - Directed stimulus generation
    - Creates directed randoms from legal stimulus
    - Used by functional simulation
    - Ex: 40% min Ethernet size, 40% max, 20% other
- Generator components can be implemented as base classes being constrained in derived classes
  - Assertions used wherever appropriate

Modularizing Stimulus Generation Example

```
sequence interface_busy; @(posedge clk)
   interface_valid or ~interface_ready or ~min_delay_passed;
   endsequence
sequence min_delay_passed; @(posedge clk)
   $negedge(interface_valid) |=> ~interface_valid [*min_delay];
   endsequence
assert property (interface_busy) iBusy = 0 else begin
   iBusy = 1; $info();
   end
```

```
class inject_legal;
  constraint valid_dist { valid dist {0 := 1,  1 := iBusy}};
endclass


class inject_directed extends inject_legal;
  constraint valid_dist {
    valid dist {0 := ~iBusy + 1-valid_prob,
                1 := iBusy * valid_prob}; };
endclass
```

Hurdles for Modularizing Stimulus Generation

- Formal constraint generation from SystemVerilog testbench constructs (with embedded assertions)
  - Ex: Constrained, extended classes to formal constraints
- Capacity of formal tools

Conclusion

- SystemVerilog 3.1 enables a single DV environment that harnesses the benefits of both traditional testbench automation technologies and assertions
- Integration of testbench and assertions improves productivity of simulation based methodology
- Integration allows same environment to be used for formal analysis as well
- All together, more bugs are found faster