AutoSVA: Democratizing Formal Verification of RTL Module Interactions

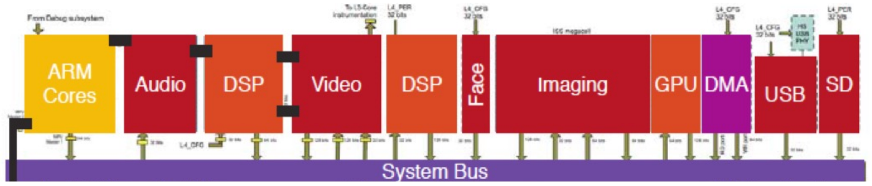
Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff and Margaret Martonosi

Presented by: Marcelo Orenes-Vera
movera@princeton.edu

PRINCETON UNIVERSITY

Verifying Module Interactions is Challenging

- Modern heterogeneous SoC design is complex and time-consuming
    1. Multiple modules developed in different contexts
    2. These modules interact with each other
    3. System can hang if one module never replies

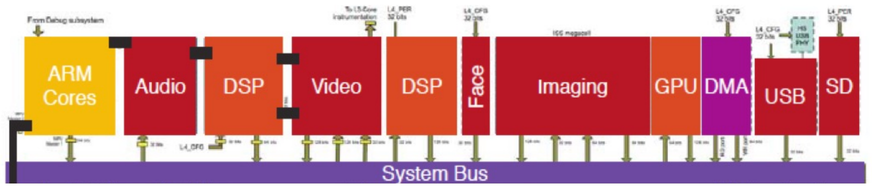


TI OMAP4 ***Heterogeneous SoC***

Image Credit: Texas Instruments

Verifying Module Interactions is Challenging

- Modern heterogeneous SoC design is complex and time-consuming
  1. Multiple modules developed in different contexts
  2. These modules interact with each other
  3. System can hang if one module never replies



TI OMAP4 ***Heterogeneous SoC***

Image Credit: Texas Instruments

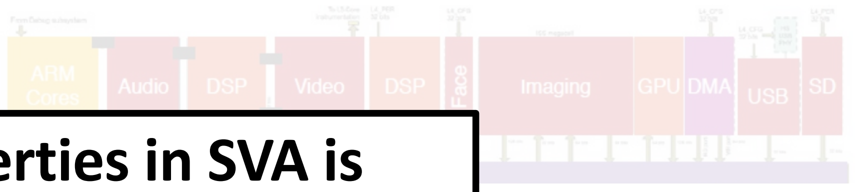
- **SystemVerilog Assertions (SVA)** is a language to describe properties about a hardware module. These properties can be:
  - *Safety properties:* nothing bad will happen, e.g. mem request with invalid addr.
  - *Liveness properties:* something good will eventually happen, e.g. get a response
  - These can be *asserted* (check always), *covered* (observed at least once) or assumed

E.g.:

```
co__lsu_request_happens: cover property (lsu_load_sampled > 0);
// Assert that if a valid transaction then eventually is ack'ed or dropped
as__lsu_load_hsk_or_drop: assert property (lsu_req_val |->
                                      s_eventually(!lsu_req_val || lsu_req_rdy));
```

Verifying Module Interactions is Challenging

- Modern heterogeneous SoC design is complex and time-consuming
  1. Multiple modules developed in different contexts
  2. These modules
  3. System can har



eterogeneous SoC

exas Instruments

- **SystemVerilog Asse**
  properties about a
  - *Safety properties:*
  addr.
  - *Liven*
  respo

Hand-writing RTL properties in SVA is tedious and error-prone. However, properties are very important to check, as the forward progress of the system depends on all modules interacting as expected!

Properties

Assert,
Assumes
and Covers

```
co__lsu_request_happens: cover property (lsu_load_sampled > 0);
// Assume that a transaction is stable until acknowledged
am__lsu_load_stability: assume property (lsu_req_val && !lsu_req_rdy |=>
                                         $stable({lsu_req_stable}) );
// Assert that if a valid transaction then eventually is ack'ed or dropped
as__lsu_load_hsk_or_drop: assert property (lsu_req_val |->
                                          s_eventually(!lsu_req_val || lsu_req_rdy));
```

Need exhaustive testing of properties

- While properties can be checked during simulation-based verification, i.e. running tests, there is no confidence that the SVA properties hold outside the tested scenarios.
  - Very long traces on properties failing on system-level simulation
  - Often only safety properties supported, and not liveness.

Need exhaustive testing of properties

- While properties can be checked during simulation-based verification, i.e. running tests, there is no confidence that the SVA properties hold outside the tested scenarios.
  - Very long traces on properties failing on system-level simulation
  - Often only safety properties supported, and not liveness.

- Properties can be checked thoroughly using Formal Property Verification (FPV) tools, since they check every possible combination in the space state. *FPV is more suitable for verifying liveness properties and forward progress*. But…

1. FPV has a **steep learning curve**
2. FPV requires both **significant knowledge and engineering effort**
   - Need to write many properties and additional modeling code in Verilog

We need an automated method!

Need exhaustive testing of properties

- While properties can be checked during simulation-based verification, i.e. running tests, there is no confidence that the SVA properties hold outside the tested scenarios.
  - Very long traces on pro
  - Often only safety prop

Such methodology would allow hardware designers to verify the modules they are developing

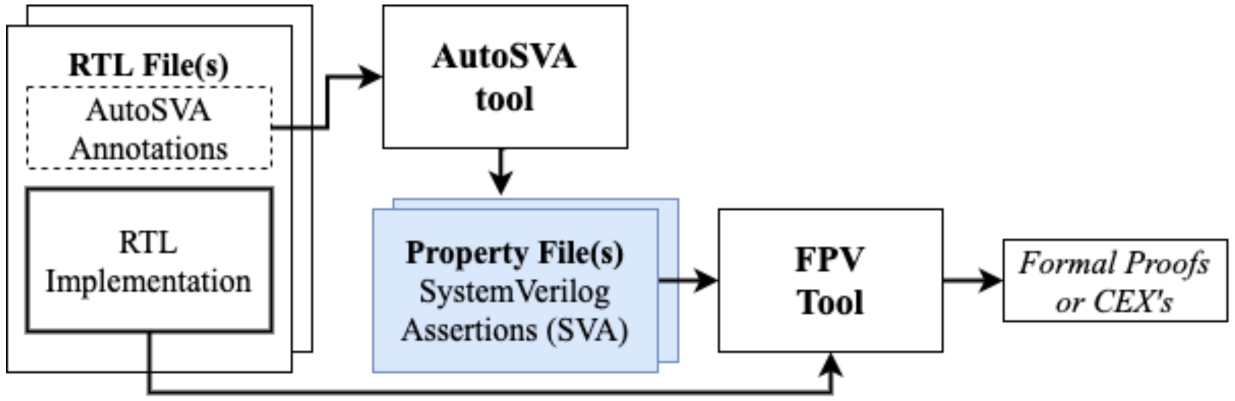
- Properties can be chec fication (FPV) tools, since they check every *PV is more suitable for verifying liveness properties and forward progress*. But…

1. FPV has a **steep learning curve**
2. FPV requires both **significant knowledge and engineering effort**
   - Need to write many properties and additional modeling code in Verilog

We need an automated method!

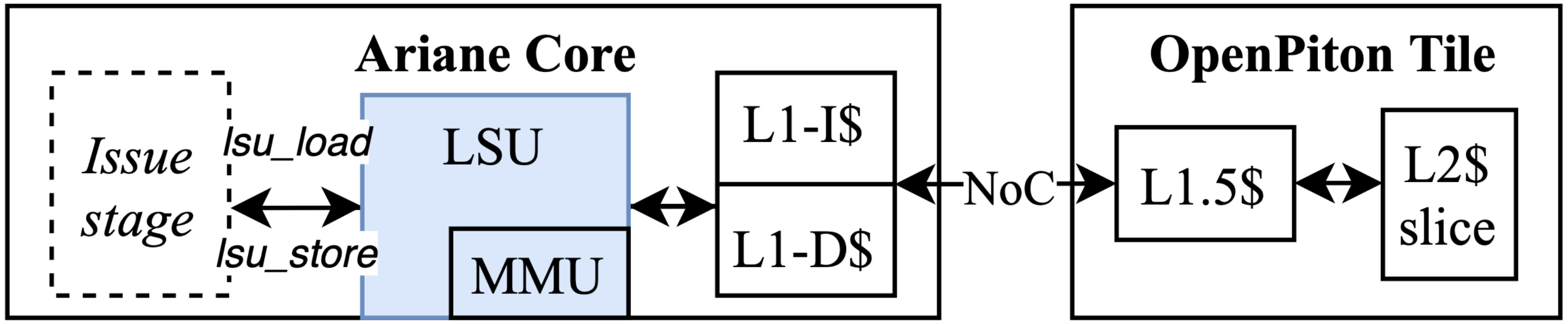
The AutoSVA Framework

- AutoSVA: a framework for automatic generation of FPV testbenches to verify *well-formed* transactions and *forward progress of RTL modules*
  - AutoSVA introduces a ***transaction* abstraction that enables automated reasoning about liveness and safety properties of module interactions thus allowing hardware designers to efficiently formally verify their RTL** by simply writing annotations at module interfaces



AutoSVA

AutoSVA to Verify RTL Module Interactions



Load-Store Unit (LSU) Load interface

```
input  logic                      lsu_valid_i,
output logic                      lsu_ready_o,
input  fu_data_t                  fu_data_i,

output logic                      load_valid_o,
output logic [TRANS_ID_BITS-1:0]  load_trans_id_o,
output riscv::xlen_t              load_result_o,
```

Need to verify that RTL modules interact as expected!

AutoSVA Offers a Simple but Rich Language

- Transaction involves **two events** with an **implication relation**
  - e.g., request->response, or any action->effect
- Transactions are named and can have various **attributes**
  - e.g., *valid*, *ready, trans_id, data,* etc.
- Attributes can be defined **explicitly** in the RTL (by writing annotations, as shown in the example), or **implicitly** (no annotation, when signals match our name convention)

AutoSVA Explicit Annotations to the LSU Load interface

```
/*AUTOSVA
lsu_load: lsu_req -in> lsu_res
lsu_req_val = lsu_valid_i && fu_data_i.fu == LOAD
lsu_req_rdy = lsu_ready_o
[TRANS_ID_BITS-1:0] lsu_req_transid = fu_data_i.trans_id
[CTRL_BITS-1:0] lsu_req_stable = {fu_data_i.trans_id,fu_data_i.fu}
lsu_res_val = load_valid_o
[TRANS_ID_BITS-1:0] lsu_res_transid = load_trans_id_o
*/
```

AutoSVA Offers a Simple but Rich Language

- Transactions can be incoming and outgoing
  - **Incoming (in):** An external module sends a request to the *Design-Under-Test (DUT),* so that AutoSVA properties will assert that there is an eventual effect or response, and that this follows certain conditions
  - **Outgoing (out)**: The DUT sends a request to an external module. Since the behavior of this is outside the scope of the DUT, we assume that this transaction behaves as expected (based on the annotations)

AutoSVA Explicit Annotations to the LSU Load interface (incoming)

```
/*AUTOSVA
lsu_load: lsu_req -in> lsu_res
lsu_req_val = lsu_valid_i && fu_data_i.fu == LOAD
lsu_req_rdy = lsu_ready_o
[TRANS_ID_BITS-1:0] lsu_req_transid = fu_data_i.trans_id
[CTRL_BITS-1:0] lsu_req_stable = {fu_data_i.trans_id,fu_data_i.fu}
lsu_res_val = load_valid_o
[TRANS_ID_BITS-1:0] lsu_res_transid = load_trans_id_o
*/
```

Mapping Transactions to Properties

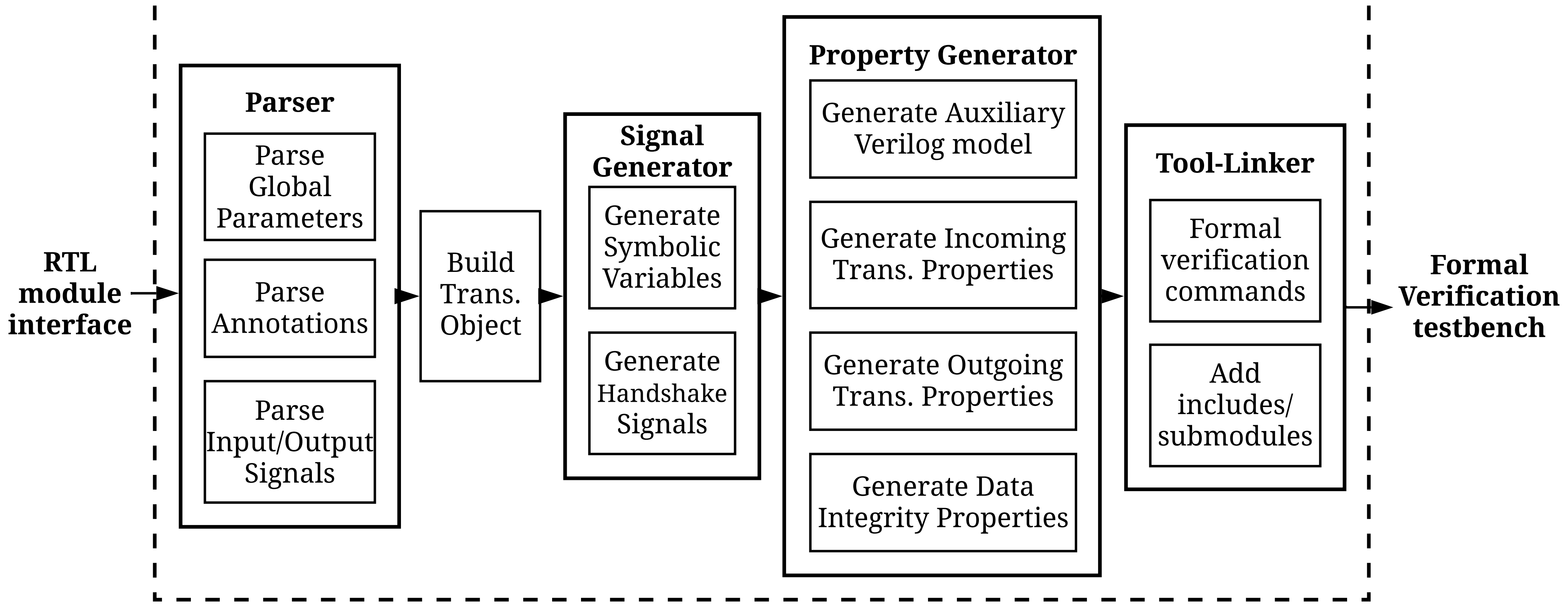
Attribute	Properties generated
*val**	If P is valid, then eventually Q will be valid and for each Q valid, there is a P valid
*ack**	If P is valid, eventually P is ack'ed or P is dropped (if its *stable* signal is not defined)
stable	If P is valid and not ack'ed, then it is *stable* next cycle
active	This signal is asserted while transaction is ongoing
*transid**	Each Q will have the same transaction ID as P
transid_unique	There can only be 1 ongoing transaction per ID
*data**	Each Q will have the same data as P

Fragment of the code and properties generated by AutoSVA

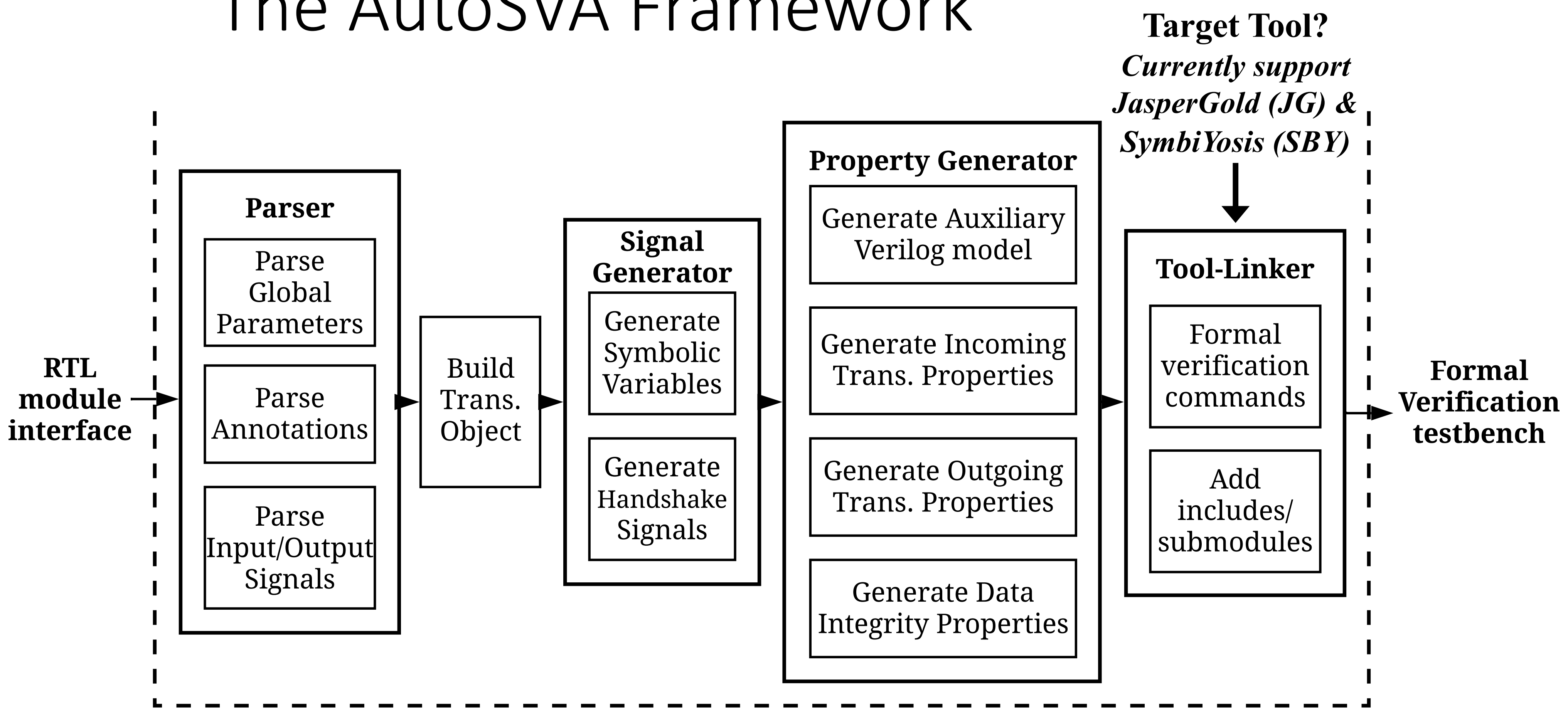
```
reg [TRANS_WIDTH-1:0] lsu_load_transid_sampled;
wire lsu_req_hsk = lsu_req_val && lsu_req_rdy;
wire lsu_load_set = lsu_req_hsk && lsu_req_transid == symb_lsu_transid;
wire lsu_load_response = lsu_res_val && lsu_res_transid ==symb_lsu_transid
always_ff @(posedge clk_i or negedge rst_ni) begin
 if(!rst_ni) //counting transaction
   lsu_load_sampled <= '0;
 end else if (lsu_load_set || lsu_load_response)
   lsu_load_sampled <= lsu_load_sampled + lsu_load_set - lsu_load_response
end
// Assert that every request has response, and every reponse had a request
as__lsu_load_eventual_response: assert property (lsu_load_set |->
                            s_eventually(lsu_load_response)));
as__lsu_load_had_a_request: assert property (lsu_load_response |->
                            lsu_load_set || lsu_load_sampled > 0);
```

AutoSVA generates both the necessary scaffolding code and design properties based on the attributes defined in the interface annotations

The AutoSVA Framework

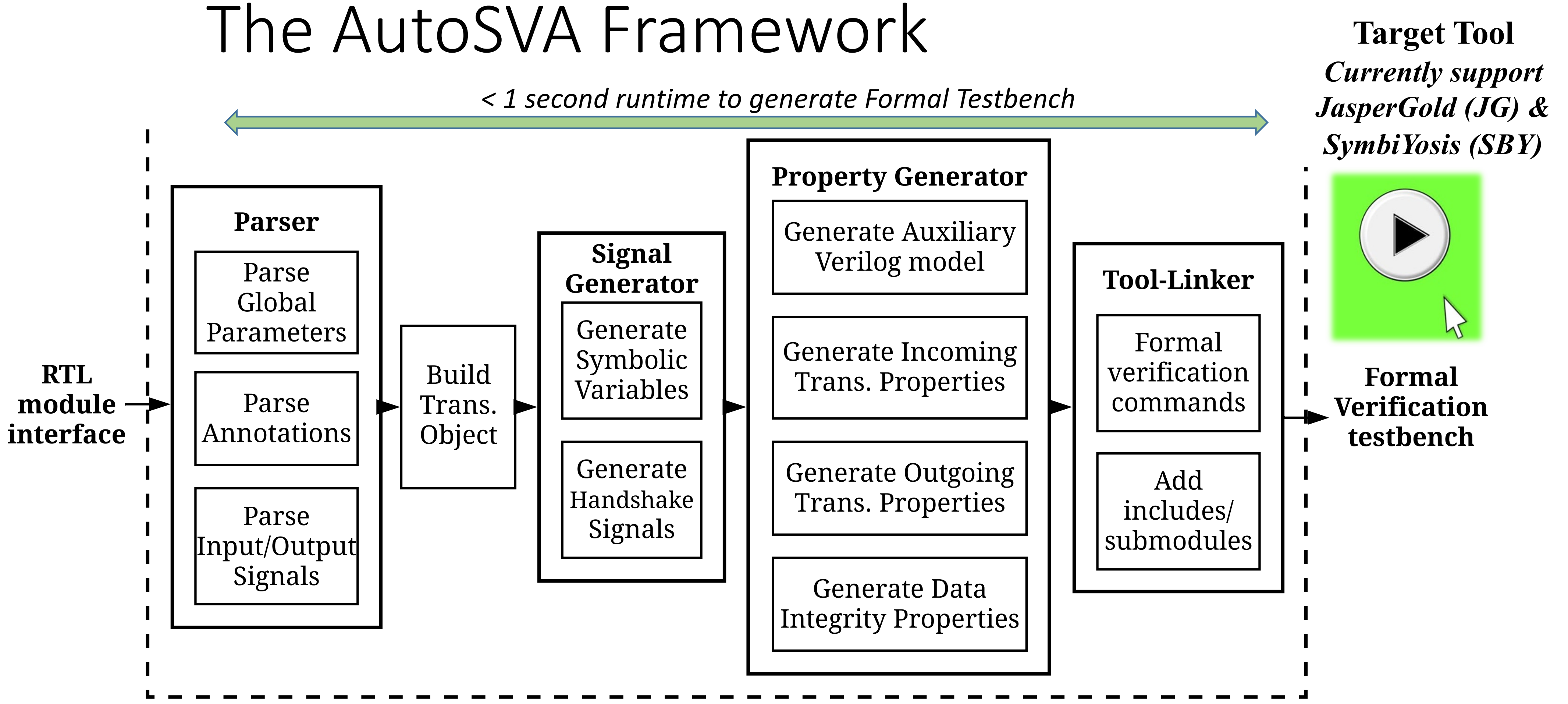

RTL module interface
Parser
Parse Global Parameters
Parse Annotations
Parse Input/Output Signals
Build Trans. Object
Signal Generator
Generate Symbolic Variables
Generate Handshake Signals
Property Generator
Generate Auxiliary Verilog model
Generate Incoming Trans. Properties
Generate Outgoing Trans. Properties
Generate Data Integrity Properties
Tool-Linker
Formal verification commands
Add includes/ submodules
Formal Verification testbench

The AutoSVA Framework



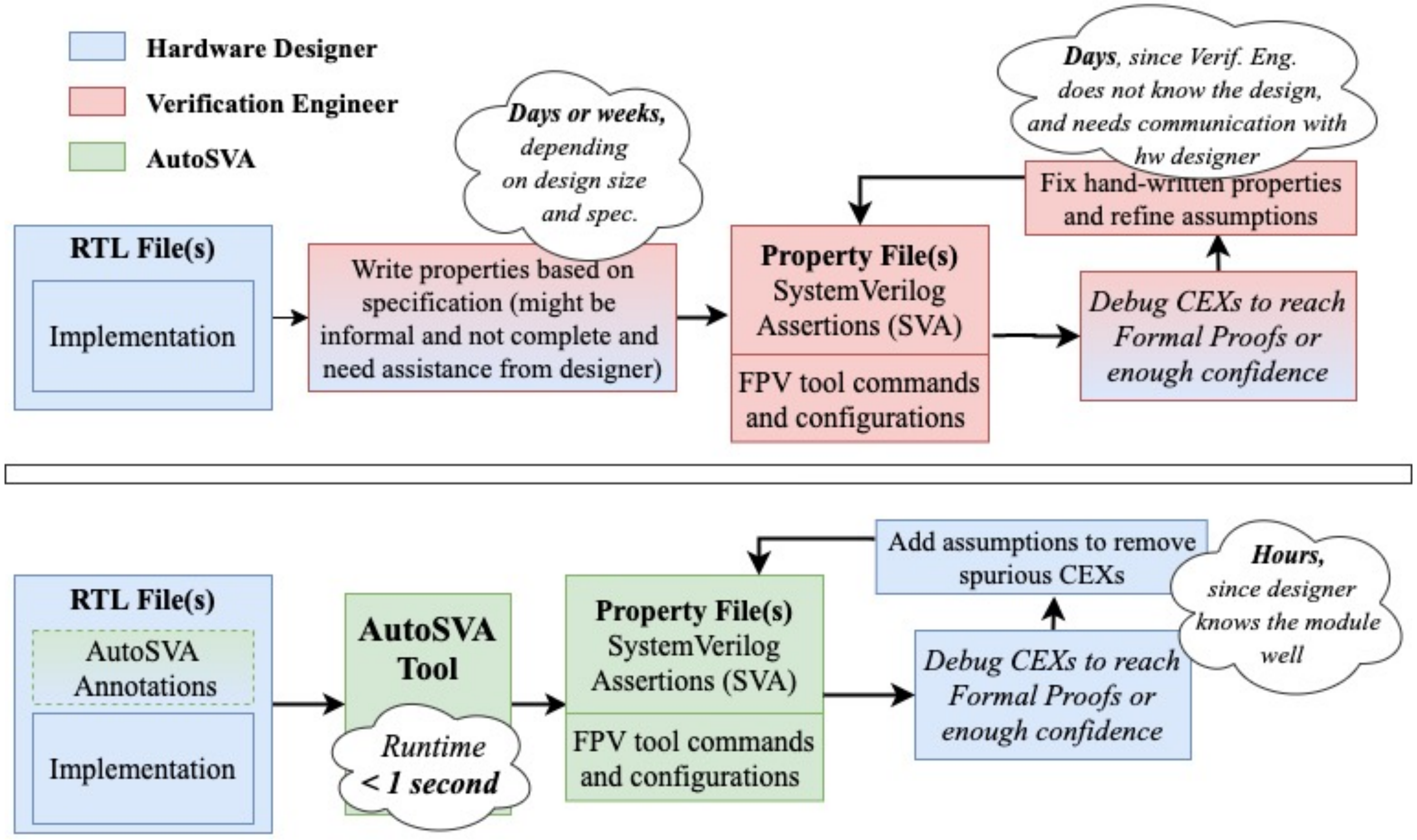
Target Tool?
Currently support JasperGold (JG) & SymbiYosis (SBY)
RTL module interface
Parser
Parse Global Parameters
Parse Annotations
Parse Input/Output Signals
Build Trans. Object
Signal Generator
Generate Symbolic Variables
Generate Handshake Signals
Property Generator
Generate Auxiliary Verilog model
Generate Incoming Trans. Properties
Generate Outgoing Trans. Properties
Generate Data Integrity Properties
Tool-Linker
Formal verification commands
Add includes/ submodules
Formal Verification testbench

The AutoSVA Framework



< 1 second runtime to generate Formal Testbench
Target Tool
Currently support JasperGold (JG) & SymbiYosis (SBY)
RTL module interface
Parser
Parse Global Parameters
Parse Annotations
Parse Input/Output Signals
Build Trans. Object
Signal Generator
Generate Symbolic Variables
Generate Handshake Signals
Property Generator
Generate Auxiliary Verilog model
Generate Incoming Trans. Properties
Generate Outgoing Trans. Properties
Generate Data Integrity Properties
Tool-Linker
Formal verification commands
Add includes/ submodules
Formal Verification testbench

AutoSVA vs Common FPV methodology[1]



Hardware Designer
Verification Engineer
AutoSVA
RTL File(s)
Implementation
Write properties based on specification (might be informal and not complete and need assistance from designer)
Days or weeks, depending on design size and spec.
Property File(s) SystemVerilog Assertions (SVA)
FPV tool commands and configurations
Debug CEXs to reach Formal Proofs or enough confidence
Fix hand-written properties and refine assumptions
Days, since Verif. Eng. does not know the design, and needs communication with hw designer
RTL File(s)
AutoSVA Annotations
Implementation
AutoSVA Tool
Runtime < 1 second
Property File(s) SystemVerilog Assertions (SVA)
FPV tool commands and configurations
Debug CEXs to reach Formal Proofs or enough confidence
Add assumptions to remove spurious CEXs
Hours, since designer knows the module well

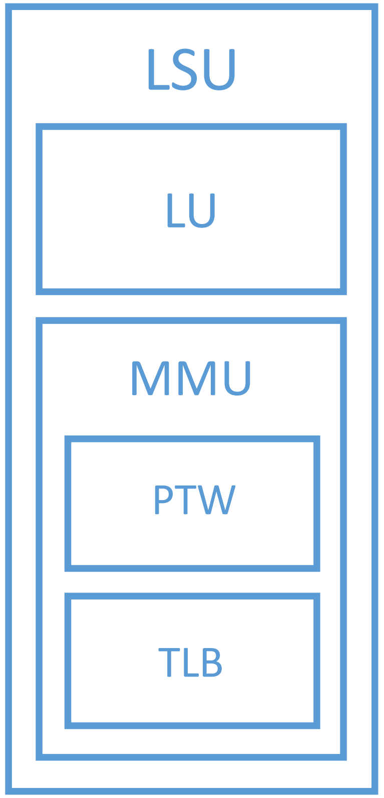
1. *Formal Verification: An Essential Toolkit for Modern VLSI Design. E. Seligman, T. Schubert, and A.K. Kumar*

Evaluation Target: Open-Source Hardware

- We focus on modules of renown, open-source hardware projects:
  - OpenPiton Manycore framework
    - *L1.5 and NoC buffers*
  - Ariane RISC-V Core
    - *Load-Store Unit (LSU) and its submodules: Page Table Walker (PTW), Translation Lookaside Buffer (TLB) and Memory Management Unit (MMU)*
    - *L1-Instruction Cache*

OpenPiton Ariane

AutoSVA Hierarchy strategy

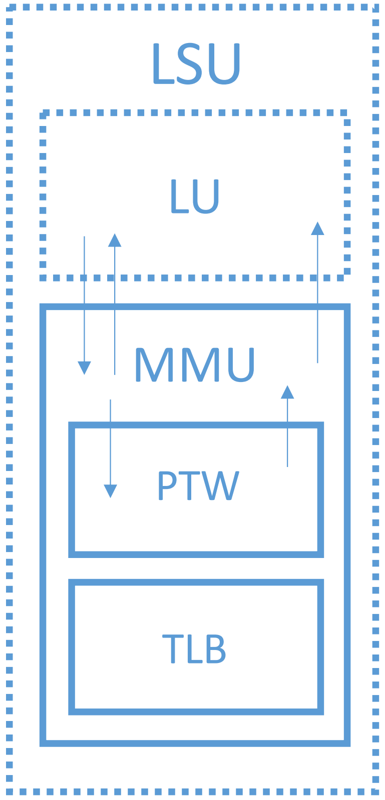


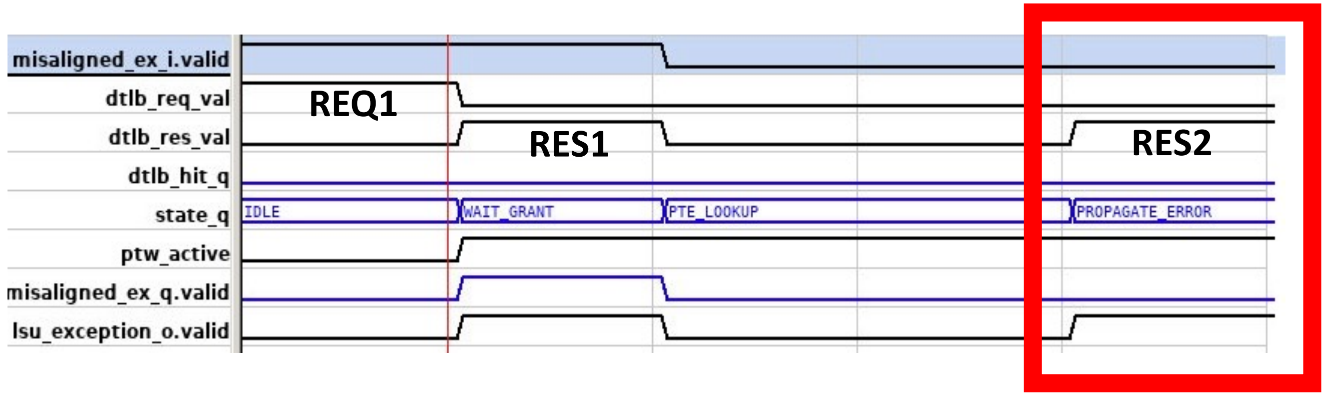
1. Smaller modules should be verified first, and we move on to parent modules once their submodules have been verified, e.g. MMU once TLB and PTW are verified

2. The submodule properties might have involved an outgoing transaction to a module which is now included within the parent, e.g. TLB triggering Page table walks.

The switch between assumptions to assertion is also controlled by the AutoSVA tool parameters

Finding a ghost-response bug in the MMU

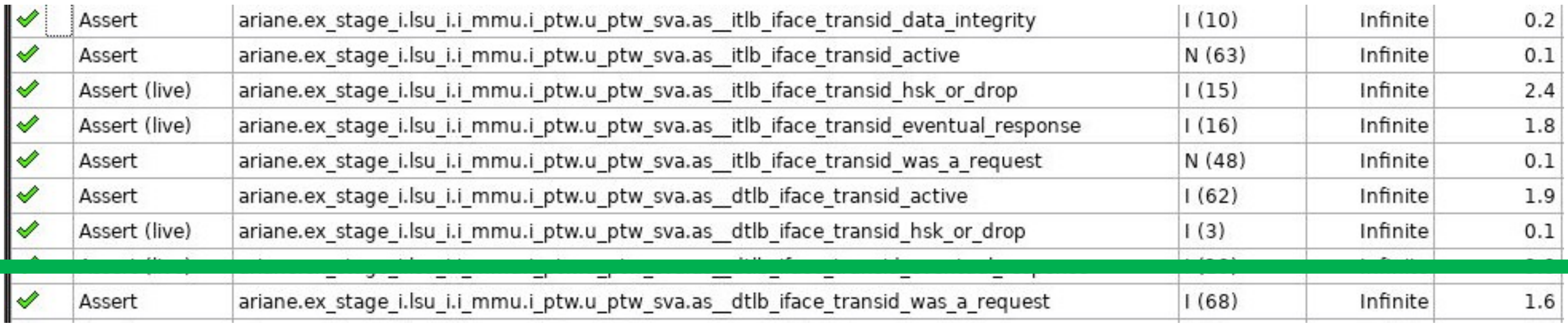




1. Writing AutoSVA annotations: **10 min**
2. AutoSVA properties generation: **0.7s**
3. Debugging time due to spurious CEXs: **30 min**
4. Trace length of Bug Hit: **5 cycles**
5. FPV tool runtime to generate trace: **0.2s**

AutoSVA

MMU ghost response bug-fix



✓	Assert	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__itlb_iface_transid_data_integrity	I (10)	Infinite	0.2
✓	Assert	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__itlb_iface_transid_active	N (63)	Infinite	0.1
✓	Assert (live)	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__itlb_iface_transid_hsk_or_drop	I (15)	Infinite	2.4
✓	Assert (live)	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__itlb_iface_transid_eventual_response	I (16)	Infinite	1.8
✓	Assert	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__itlb_iface_transid_was_a_request	N (48)	Infinite	0.1
✓	Assert	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__dtlb_iface_transid_active	I (62)	Infinite	1.9
✓	Assert (live)	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__dtlb_iface_transid_hsk_or_drop	I (3)	Infinite	0.1
✓	Assert (live)	[illegible]	[illegible]	[illegible]	[illegible]
✓	Assert	ariane.ex_stage_i.lsu_i.i_mmu.i_ptw.u_ptw_sva.as__dtlb_iface_transid_was_a_request	I (68)	Infinite	1.6

- We made a tentative bug-fix and got proof of no CEX!
- **The total invested time from writing AutoSVA annotations to finding the bug, fixing it, and getting bug-fix proof was around 1h**

Why should I use AutoSVA? To...

- **Assist hardware designers at every stage of RTL development**, by providing them with a formal testbench (FT) that they can run to get CEXs or proofs to work in progress

- **Provide a FT quickstart** that can be extended through manual addition of other properties, e.g. functional logic or if Full-Proof FPV is needed

- **Complement system-level simulation.** Properties generated by AutoSVA can be connected to the system-level testbench so that are also checked during simulation

Conclusions

- Verifying liveness and control-safety properties in an RTL design is complex and challenging**.** Formal property verification of modules' RTL can exhaustively search for bugs via assertions at a very early project stage, but **SVA and FPV tools are hard to use and reason about.**

- AutoSVA offers a framework to **automatically generate Formal Testbenches** that check module interface expectations, based on designer-written annotations.

- **This pays off quickly**, as it saves debugging time during simulation and increase designer confidence that the module will not hang within the system.

AutoSVA

Thanks for attending!

Questions?

Contact

- movera@princeton.edu
- https://cs.princeton.edu/~movera

Open-Source Repository

- https://github.com/PrincetonUniversity/AutoSVA
- Happy to assist on usage!

AutoSVA tutorial

- https://youtu.be/Gb5wT1D7dxU