

# AutoSVA: Democratizing Formal Verification of RTL Module Interactions

Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff and Margaret Martonosi  
Department of Computer Science and Electrical Engineering, Princeton University

Princeton, New Jersey, USA

Email: {movera, amanocha, wentzlaf, mrm}@princeton.edu

**Abstract**—Modern SoC design relies on the ability to separately verify IP blocks relative to their own specifications. Formal verification (FV) using SystemVerilog Assertions (SVA) is an effective method to exhaustively verify blocks at unit-level. Unfortunately, FV has a steep learning curve and requires engineering effort that discourages hardware designers from using it during RTL module development. We propose AutoSVA, a framework to automatically generate FV testbenches that verify liveness and safety of control logic involved in module interactions. We demonstrate AutoSVA’s effectiveness and efficiency on deadlock-critical modules of widely-used open-source hardware projects.

**Index Terms**—automatic, modular, formal, verification, SVA

## I. INTRODUCTION

Heterogeneous SoC design is a lengthy, expensive process that necessitates verification at early stages of development to avoid late bug fixes that thwart performance or area goals [14]. SoC modules may be developed in various contexts and exhibit complicated interactions [2]. With the numerous dependencies that occur between them, module interface verification is necessary to prevent opportunities for livelock and deadlock. Fig. 1 presents the Ariane core [18] and the cache hierarchy of the OpenPiton manycore [1], used as an example throughout the paper. Among the module interactions, the Load-Store Unit (LSU) is critical for the forward progress of the system.

SystemVerilog Assertions (SVA) [10] is often used for RTL verification because it is a powerful language for defining a design’s properties and specifying temporal dependencies. SVA properties can be checked through both test-driven simulation and Formal Verification (FV) in order to reveal bugs. However, only FV tools can exhaustively test a given Design-Under-Test (DUT) and consequently are most suitable for verifying forward progress [5], [17]. Unfortunately, these tools present a steep learning curve and require significant engineering effort to set up a *useful* FV testbench, i.e. writing appropriate properties and specification constraints. *This upfront knowledge and effort discourages hardware designers from using FV* [15]. Some tools generate SVA from a higher abstraction layer [9], [11], but creating a high-level model and mapping it to RTL signals is still cumbersome. These tools also do not cover important properties like forward progress.

With the goal of making FV *agile* and *widely used* among hardware designers, we make two **key observations**: (1) Although interactions between RTL modules may take place via different mechanisms, a common design pattern across

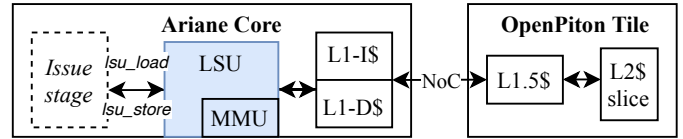


Fig. 1. Heterogeneous SoCs such as OpenPiton+Ariane, involve dependencies between modules. Verifying these interactions is critical to guarantee forward progress in the system. For example, a load request to the LSU (blue) must receive a response for a memory request to eventually complete.

many of them is *request and response*. We advocate for automatic support of FV for this pattern. (2) Capturing the request/response abstraction in a model allows for *automated reasoning* about RTL interfaces and their expected interactions. This work proposes a language centered around a *transaction* model. This model’s applicability is not limited to modules with explicit requests/responses; it can express other interface mechanisms, e.g. pipeline stages that receive requests from a previous stage and send them to the next stage.

**Approach:** Given our observations, this work proposes AutoSVA, a framework to automatically generate Formal verification Testbenches (FT) for a given DUT. The designer of the DUT only needs to identify relevant transactions and annotate them in the module interface using a simple language. The framework then generates properties that verify the transactions are *well-formed* and make *forward-progress*: they satisfy *liveness* (every request is eventually followed by a response) and *safety* (expectations for attributes of the response). Through its automated reasoning, AutoSVA creates the necessary scaffolding code to express these properties and tool-specific commands to drive the FV process, alleviating the hardware designer from significant engineering effort and democratizing the use of FV for verifying forward progress.

FTs generated by AutoSVA can then be supplied to a FV tool, e.g. JasperGold [5] or the open-source SymbiYosys tool [17]. AutoSVA thereby provides a frontend for automatic FV of an important subset of the correctness problem—ensuring RTL modules’ interface expectations.

### Our main contributions are:

- A language that creates a unified transaction abstraction to denote RTL interface interactions and dependencies. This enables automatic reasoning about RTL properties.
- An automated procedure to generate FTs that express liveness properties about transaction temporal dependencies and safety properties about control-logic attributes.
- Demonstration of AutoSVA’s effectiveness on 7 control-

arXiv:2104.04003v1 [cs.AR] 8 Apr 2021

```

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
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));
//Assert that every request has response, and every response 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);

```

Fig. 2. To verify the load interface of the LSU, a hardware designer would need to write many SVA properties and auxiliary code. AutoSVA automatically generates all modeling, removing the burden from the designer.

critical RTL modules of the widely-used open-source projects Ariane and OpenPiton [1], [18]. As one example, within 1 hour, AutoSVA generated a FT for Ariane’s MMU, discovered a bug, and verified the bug-fix.

## II. MOTIVATING EXAMPLE

SVA is SystemVerilog’s formal specification language [10], and offers a mature approach for verifying RTL. It can express Linear Temporal Logic (LTL) formulas over interface signals to build properties about module interactions. LTL specifies temporal relations, which fall into two major classes: safety and liveness properties. *Safety* properties specify that “nothing bad will happen”, e.g. a response must have had a request; while *liveness* specify that “something good will happen”, e.g. a request is eventually acknowledged.

Fig. 2 presents a subset of the modeling and properties that are necessary to verify forward progress for the load-store unit (LSU) in Ariane (depicted in Fig. 6). For example, *lsu\_load\_eventual\_response* is a liveness property to check that any load request eventually receives a response with the same transaction ID. Verifying expectations about module interfaces goes beyond writing properties; it requires code to sample transactions and symbolic variables to track them. AutoSVA automatically generates all of this necessary code.

Properties in SVA can use one of three directives: *assert*, *assume* and *cover*. Assumptions have different meanings based on how input stimuli are generated. In RTL simulation, inputs are driven either by manual or random tests, and thus *assume* has the same meaning as *assert*, i.e they check that the property holds. Conversely, FV tools treat inputs as Boolean variables, and *assumptions* constrain the state space exploration by preventing some behaviors, while *assertions* check that properties hold on the explored paths. FV tools then use a variety of solver engines [6] based on formal methods, such as model checking, which uses SAT (satisfiability) [3] or BDD (binary decision diagrams) [12], to exhaustively search for property violations. FV search may result in a counterexample

```

/*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
*/

```

Fig. 3. The LSU designer only needs to annotate the RTL interface using AutoSVA’s language to generate a FT (containing among other things the properties and modeling code shown in Fig. 2). The first line describes a relation between a request (italic blue) and a response (italic green) interface; the remaining lines map RTL interface signals to transaction attributes (bold).

(CEX) that highlights the violation of a property, or proof that properties hold, i.e. the solver converges and finds no CEXs.

The underlying dynamics of FV and SVA make it difficult to intuitively understand the consequences of various properties expressed, such as the behavior of symbolic variables, e.g. *symb\_lsu\_transid* in Fig. 2, which allow the tracking of indices with a single assertion. Furthermore, subtle mistakes in assumptions, e.g. using the  $|-\>$  implication symbol in the *lsu\_stability* assumption, can *over-constrain* the state space and end up proving vacuity. Manually inserting assertions can be cumbersome and error-prone for a hardware designer, and particularly frustrating when CEXs appear due to illegal inputs of not yet modeled interfaces [8]. Thus, AutoSVA automatically models and expresses the expected behavior for module transactions.

Hardware designers can employ SVA properties for Test-Driven-Development (TDD), where CEXs help to refine the design [16]. Moreover, this can be applied at early stages of RTL module development by using unit-level FV [4]. However, FV’s steep learning curve and necessary engineering effort preclude designers from using it in practice [15]. **AutoSVA democratizes FV for hardware designers and makes TDD practical by automating a key component of the FV problem: liveness and safety of module interfaces.** Instead of aiming to support functional FV, which is very implementation-dependent, AutoSVA focuses on verifying that modules interact through *well-formed transactions*. This verification entails checking certain attributes over the control logic involved in transactions and mapping them to properties that ensure that every module makes progress (does not hang).

Fig. 3 presents an example of the simple usage of AutoSVA’s language. These annotations unleash automated reasoning to generate the modeling and properties (shown in Fig. 2) surrounding liveness and safety for the Ariane core’s LSU. Section III explains the syntax of the AutoSVA language and semantics of each annotation. Section IV shows how these annotations can be applied to different interface styles.

## III. THE AUTOSVA FRAMEWORK

AutoSVA focuses on verifying liveness, and its well-formed transactions allow it to utilize a common abstraction from RTL interfaces that avoids the complexity of specific module implementations. By capturing a common design-pattern, AutoSVA can automatically generate *useful* Formal Testbenches (FT). We denote a FT as *useful* when it (1) has sufficient mod-

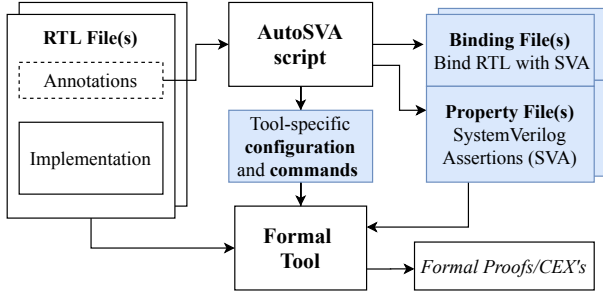


Fig. 4. AutoSVA is an agile framework for FV of RTL using SVA. The files that define the FT are denoted in blue. Dotted lines indicate designer input.

ule interface modeling to avoid spurious CEXs and capture relevant CEXs which lead to uncovering bugs, and (2) does not miss legal scenarios due to over-constraining assumptions. Moreover, AutoSVA reduces the state-explosion scalability problem because it deliberately focuses on control logic and FV tools can be instructed to automatically ignore datapaths.

Fig. 4 presents an overview of AutoSVA’s verification process. AutoSVA takes as input the interface declaration section of the RTL module acting as the DUT. The interfaces should be annotated using AutoSVA’s language for interface abstraction (defined at Section III-A). Once the abstraction is defined for a DUT, AutoSVA generates the FT that includes a property file describing the properties to verify, all necessary modeling about RTL blocks external to the DUT, and a binding file to connect the properties to signals in the DUT.

Based on the FV tool to target, AutoSVA generates configuration and command files. AutoSVA currently supports JasperGold [5] and SymbiYosys [17]. Once the properties, binding and tool-specific files are generated, AutoSVA invokes the FV tool to start the verification process. This returns either property proofs or CEXs that highlight possible bugs in the RTL. A hardware designer can then quickly set up a FT and locate bugs by using AutoSVA as a frontend for FV tools.

#### A. AutoSVA Language to Express Transactions

AutoSVA’s transaction abstraction involves two events connected with an implication relation. From the DUT’s perspective there are two types of transactions: (1) *incoming* transactions describe when a DUT receives a request and is responsible for eventually triggering a well-formed response or another request, and (2) *outgoing* transactions describe when a DUT triggers a request that eventually must receive a response.

The two events in a transaction are associated with RTL *interfaces*, which are the connection points of RTL modules. For example, incoming transactions can map a cache lookup interface to define a liveness condition that the cache lookup should eventually have a response, and to define a safety condition that this response must satisfy certain properties, e.g. maintain the same transaction ID the request had.

AutoSVA maps transaction events to interfaces through annotations expressed in its language. These language annotations are written as Verilog comments on the interface declaration section of an RTL file to identify module interfaces that participate in transactions. To distinguish these annotations

TABLE I  
THE AUTOSVA LANGUAGE. CONSTANTS ARE WRITTEN IN LOWERCASE AND SYNTAX IN UPPERCASE. STR AND ASSIGN ARE VERILOG’S SYNTAX FOR STRINGS AND ASSIGNMENTS.

$TRANSACTION ::= TNAME: RELATION ATTRIB$
$RELATION ::= P -in> Q \mid P -out> Q$
$ATTRIB ::= ATTRIB, ATTRIB \mid SIG = ASSIGN \mid input SIG \mid output SIG$
$SIG ::= [STR:0] FIELD \mid STR FIELD$
$FIELD ::= P\_SUFFIX \mid Q\_SUFFIX$
$SUFFIX ::= val \mid ack \mid transid \mid transid\_unique \mid active \mid stable \mid data$
$TNAME, P, Q ::= STR$

TABLE II  
PROPERTIES GENERATED FOR EACH TRANSACTION ATTRIBUTE.

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

from regular code comments, AutoSVA requires annotations to be preceded with an *AUTOSVA* macro, or be contained within a multi-line comment region that starts with it.

Table I presents the formalization of the AutoSVA language.  $P$  and  $Q$  represent two interfaces which have a temporal implication relation, which is either incoming “ $-in>$ ” or outgoing “ $-out>$ ” from the DUT’s perspective, and share a transaction named  $TNAME$ . Multiple transactions can be defined with unique names. *ATTRIB* definitions map interface signals to transaction attributes. Each definition must be placed on a separate line in the RTL, i.e. distinct line number, and must be prefixed with the interface name.

*Implicit definitions* are native interface signal declarations (preceded by input/output signals) that are already defined in the RTL design. If they follow the *FIELD* naming convention, AutoSVA can automatically identify these fields without annotations, which is especially useful for early-stage RTL verification. AutoSVA’s parser ignores signal declarations that do not match  $P$  or  $Q$  prefixes and the language’s legal suffixes.

*Explicit definitions* define new signals to extract transaction attributes that are not explicitly defined with interface signals. These are useful for renaming signals that do not match AutoSVA’s language, extracting fields within structs, and defining attributes based on multiple interface signals. Fig. 7 presents examples of these definitions for a few modules.

#### B. Property Generation Based on Transaction Attributes

AutoSVA generates properties based on how transactions are defined, as more attributes indicate more characteristics to verify. Table II presents the properties that result from the presence of each attribute. AutoSVA does not require all possible transaction attributes to be defined in order to generate meaningful properties. For example, an implication relation between  $P$  and  $Q$  with just the *val* attribute defined indicates

the two interfaces communicate and thus a liveness property is generated for the transaction. The absence of an *ack* signal indicates the request/response is always accepted.

Some of the properties expressed in Table II cannot be expressed in SVA directly, and thus AutoSVA generates all necessary auxiliary Verilog code. For example, verifying that every response followed a previous request requires counting the number of ongoing transactions (done with registers in Fig 2). The *transid* attribute allows tracking transactions to reason about other attributes, such as *data*, which is used to verify data integrity. This is important for interface fields which are immutable between  $P$  and  $Q$ , e.g. data in a queue or address in a memory request.

Attributes marked with \* at Table II generate properties that are asserted when the transaction is *incoming* and assumed when *outgoing*. E.g., for the *val* attribute, the word "eventually" indicates liveness when the DUT is expected to respond and fairness when it is waiting for a response. For attributes *stable* and *transid\_unique*, the opposite holds; properties are assumed on incoming and asserted on outgoing transactions. The attribute *active* is always asserted when defined.

**Submodule Properties:** When the DUT has a submodule whose inputs are driven by actual logic, it is worthwhile to ensure that assumptions about these inputs hold. AutoSVA assumptions can be converted into assertions by changing the value of the *ASSERT\_INPUTS* parameter. Submodule properties can be linked to the parent's through AutoSVA's parameters: "-AM" includes the properties when the submodule was the DUT (assumptions over outgoing requests) and "-AS" converts all assumptions into assertions.

**End-to-End Properties:** SVA allows writing properties that use internal RTL logic (not visible at the interface). While this is often necessary for full functional verification, it causes properties to depend on RTL implementation details. To overcome this, AutoSVA writes end-to-end properties which solely describe interface signals, but cover the whole path from input to output interface. End-to-end properties are implementation-agnostic, and thus can be automatically generated pre-RTL, making AutoSVA a great framework for Test-Driven-Development (TDD).

**Property Reuse:** In addition to FV, AutoSVA property files can be utilized in a simulation testbench to ensure that assumptions hold during system-level testing. Although many RTL simulation tools do not support liveness properties, all control-safety properties and X-propagation assertions can be checked during simulation. AutoSVA generates X-propagation assertions, which check that when the *val* signal of an interface is asserted, none of the other attributes have value X (concurrently 0 and 1). Because formal tools do not consider X's and instead assign arbitrary values of 0 or 1, these assertions are only checked during simulation (under a *XPROP* macro).

### C. AutoSVA Implementation and Process Steps

AutoSVA is implemented in Python using only standard libraries to provide portability and ease of use. **AutoSVA generates FTs in under a second.** Fig. 5 details its five steps.

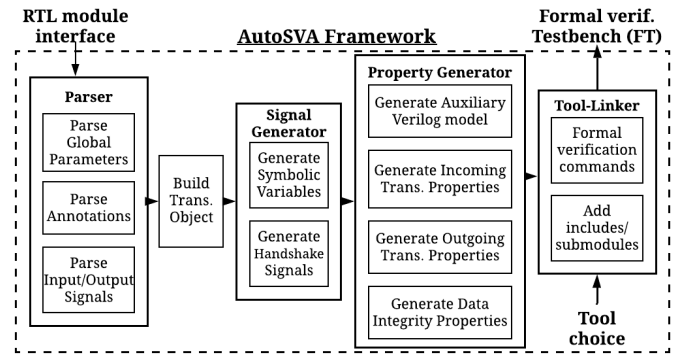


Fig. 5. Steps of the AutoSVA framework. It receives an annotated RTL file and the FV tool to target, and it outputs a FT that is ready to be run.

(1) **Parser:** AutoSVA parses the signal declaration section of the annotated RTL file to identify global parameters, e.g. cache associativity or queue size, annotations in the AutoSVA language, and interface input/output signals. Based on the annotations, the parser identifies which pairs of interfaces participate in transactions and creates a mapping from interface pairs ( $P$  and  $Q$ ) to a list of their attribute definitions.

(2) **Transaction Builder:** AutoSVA builds transaction objects based on interface fields and implication relations identified by the parser. During this process, AutoSVA can detect syntax errors in annotations, e.g. when *transid* or *data* fields are defined in only one of the interfaces of a transaction, or with mismatched data widths.

(3) **Signal Generator:** Before generating properties based on transactions, AutoSVA generates auxiliary signals, such as symbolics, which are unassigned variables used to build assertions. Symbolic variables are unconstrained, and allow FV tools to explore all their possible values in a single assertion. For example, a single assertion can be used to reason about all lines of a cache if a symbolic signal is used to index the cacheline. AutoSVA also generates handshake signals (as conjunctions of *val* and *ack*) to indicate that a request or response takes place.

(4) **Property Generator:** AutoSVA creates properties based on the transaction attributes and type (incoming or outgoing). These properties can verify liveness, uniqueness, data integrity, stability, or X-propagation (detailed in Section III-B). SVA properties are explicitly written in the property file. AutoSVA does not use SVA macros or checkers to provide better readability in case the user wants to explore the properties or a verification engineer wants to extend the FT for functional correctness. The properties are tool-agnostic, and written to be most efficient for FV tools to run, e.g. using symbolic indexes for *transid* tracking. The authors have created this tool based on lessons learnt from prior art [7], [8], [15] and years of industry and academic experience with FV of RTL.

(5) **FV Tool Setup:** Once the SVA properties are generated, AutoSVA links them to the FV tool that the hardware designer selects. Furthermore, AutoSVA supports linking the FTs of submodules of the DUT, that had already been generated, by using script parameters during AutoSVA's invocation.



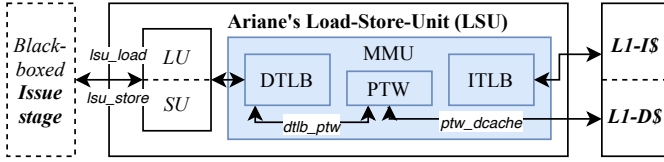


Fig. 6. AutoSVA verifies several modules in a hierarchy in Ariane. By testing at the MMU module level (blue box), AutoSVA revealed Bug1.

```

ptw_dcache: ptw_req -out> dcache_res
ptw_req_val = req_port_o.data_req
ptw_req_ack = req_port_i.data_gnt
dcache_res_val = req_port_i.data_rvalid
dtlb_ptw: dtlb -in> ptw_update
dtlb_active = ptw_active_o
dtlb_val = enable_translation & dtlb_access_i & dtlb_hit_i
dtlb_ack = !ptw_active_o
[riscv::VLEN-1:0] dtlb_stable = dtlb_vaddr_i
[riscv::VLEN-1:0] dtlb_data = dtlb_vaddr_i
ptw_update_val = ptw_update_o.valid | ptw_error_o
[riscv::VLEN-1:0] ptw_update_data = update_vaddr_o
mem-engine_noc: noc1buffer_req -in> noc1buffer_enc
[MSHR_ID:0] noc1buffer_req_transid = noc1buffer_req_mshrid
[MSHR ID:0] noc1buffer_enc_transid = noc1buffer_enc_mshrid

```

Fig. 7. AutoSVA annotations to define PTW’s outgoing transaction to the data cache (ptw\_dcachereq) and incoming transaction from the DTLB-miss interface (dtlb\_ptw), and OpenPiton buffer’s incoming transaction from Mem Engine towards NoC1 encoder (val and ack attributes match interface names).

#### IV. EVALUATING THE AUTOSVA FRAMEWORK

We utilize multiple metrics to evaluate AutoSVA: (1) its *ability to find bugs*, both known (open issues) and new bugs; (2) the *speed of bug discovery*, based on tool runtime and trace length; (3) *amount of engineering effort*, measured in time spent writing the transaction annotations; and (4) *bug-fix confidence*, whether the bug-fix leads to a proof or new CEX.

We study these metrics in mature, taped-out, open-source hardware projects: 64-bit RISC-V Ariane Core [18] and the OpenPiton manycore framework [1]. We have selected 7 RTL modules that are critical for forward-progress and thus require exhaustive testing. Table III lists these modules as well as the outcome of formally verifying them using testbenches generated by AutoSVA. These outcomes consist of proofs and bugs, demonstrating that AutoSVA is useful and effective at generating properties and models to verify forward progress. We also demonstrate AutoSVA for early-stage verification by applying it to a new unit, *Mem Engine*, which connects to OpenPiton’s NoC by reusing its encode/decoder buffers.

AutoSVA supports several FV tools, so we elect to perform evaluations using JasperGold 2015.12. Additionally, to check that AutoSVA properties are compatible with system-level simulation, we bind the property files to the in-place testbench using VCS-MX 2018.09.

**Applying the AutoSVA language to RTL modules:** A key component of AutoSVA is its transaction abstraction that is broad enough to apply to most RTL interface styles and specific enough to generate useful properties. Fig. 7 presents

TABLE III  
RTL MODULES TESTED WITH AUTOSVA. ARIANE MODULES ARE INDICATED WITH A, AND OPENPITON WITH O

RTL Module	Result
A1. Page Table Walker (PTW)	100% liveness/safety properties proof
A2. Trans. Look. Buffer (TLB)	100% liveness/safety properties proof
A3. Memory Mgmt. Unit (MMU)	Bug found and fixed → 100% proof
A4. Load Store Unit (LSU)	Hit known bug (issue #538)
A5. L1-I\$ (write-back)	Hit known bug (issue #474)
O1. NoC Buffer	Bug found and fixed → 100% proof
O2. L1.5\$ (private)	NoC Buffer proof, other CEXs

a few examples of how AutoSVA can be applied to a wide range of interfaces based on common possible scenarios.

*Single Ongoing Transaction:* When there is only one ongoing transaction in a module, it can be modeled simply by not defining the *transid* attribute, which is the case for the *ptw\_dcachereq* and *dtlb\_ptw* transactions in Fig. 7. This principle works for both incoming and outgoing transactions.

*Multiple Outstanding Transactions:* When transactions can be in-flight simultaneously, it can be modeled by annotating the tracking field with *transid*, e.g. *mshrid* for *mem-engine\_noc*. Tracking requests allows AutoSVA reasoning about integrity of *transid* and *data* fields. If requests are not tracked, AutoSVA still checks that there are no more responses than requests and that every transaction eventually finishes.

*No Ack signal:* When an interface does not have an *ack* signal but the module cannot always accept new requests, AutoSVA allows defining *ack* by reasoning about other signals. In the case of *dtlb\_ptw*, the *ack* field is defined based on the active signal, that indicates when the PTW is busy. Defining *stable* alongside *ack* means that AutoSVA will model the payload to remain stable until the request is ack’ed.

**Results:** Table III presents the 7 Ariane and OpenPiton modules that were tested using FTs. AutoSVA generated a total of 236 unique properties based on 110 LoC of annotations.

First, FTs of Ariane’s PTW and TLB resulted in 100% of the properties being proven at unit-level after 30 minutes of human effort to define the correct transactions. Next, the MMU FT was set up after 10 minutes of adding a new transaction and reusing the properties of its submodules’ FTs. These results demonstrate that AutoSVA is quick to use and effective at verifying forward progress in control-critical modules.

Fig. 6 shows the hierarchy of the Ariane modules we have tested. The MMU FT (blue) checks that every request from the LSU eventually receives a response, and that no response occurs without a prior request. Before it uncovered a real bug, AutoSVA found an interesting CEX: an ITLB miss was never filled because the PTW was always busy with DTLB misses, i.e. DTLB has static priority over ITLB. This fairness problem cannot happen in practice since one instruction cannot do many DTLB lookups. Since the trace was quick (<1s) and short (<4 cycles), it was straightforward to identify the cause of the CEX and add an assumption to remove it.

*Bug1. Ghost Response on MMU:* The next CEX uncovered a bug that was triggered when the MMU receives a misaligned request from the LSU. The MMU responds immediately with a bad alignment response, but the DTLB still misses and the PTW is activated (bad behavior). In the case of a page fault, the MMU generates a second "ghost" response to the LSU, raising an exception. This bug was found by the FV tool in less than a second, producing a 5-cycle trace that allowed us to quickly identify the problem and produce a bug-fix (masking the PTW request with the misaligned signal) with high confidence, as the formal tool found a proof in few seconds for the previously failing assertion. In 5 minutes, the MMU FT proof-rate was 100%. The Ariane maintainers confirmed the bug and the fix.

*Hitting Known Bugs:* The LSU FT hit (in 1 second) a bug that was recently discovered on a long FPGA run: an ongoing load hits an exception caused by a later load. The Ariane maintainers welcomed a FT where they could validate that the bug-fix solves the problem and does not break anything else. Similarly, the L1-1\$ FT was able to hit a reported bug.

*Bug2. Deadlock in NoC Buffer:* AutoSVA found a deadlock bug in an underdeveloped part of *Mem Engine* that connects to the OpenPiton NoC. Since the interfaces mostly matched the AutoSVA language, the FT was generated with just 3 lines of code (shown in *mem-engine\_noc* at Fig. 7). The first CEX to the liveness assertion revealed a bug that arises from the reuse of the NoC buffer from the L1.5\$ for *Mem Engine*. The buffer assumes that the input does not drive more requests than the number of buffer entries, which is violated in *Mem Engine*. After fixing the bug (adding a "not-full" condition to the *ack* signal), the formal tool resulted in a proof.

Lastly, the FT of OpenPiton's L1.5\$ showed that the condition added to the NoC buffer did not break the properties for its buffer instance. Other properties, e.g. that every cache miss is eventually filled, showed CEXs due to under-constraints in the message types. AutoSVA provides the FT foundation that the L1.5\$ designer can refine with assumptions to remove spurious CEXs. The testbench can also be extended with more assertions to achieve complete functional verification.

## V. RELATED WORK

Early works focused on developing methods for formal verification of RTL correctness [3], [12], [13]. These include model checking, which relies on SAT solvers or BDDs, and Assertion-Based Verification (ABV), which builds on top of model checking to verify control logic, design interfaces, and data integrity. The emergence of SVA [10] popularized ABV for verification engineers [15]. More recent work focuses on generating SVA from a higher level language [9], [11]: RTLCheck verifies RTL pipeline implementations against their memory consistency model (MCM) axiomatic specifications; ILA generates a Verilog model of the design from the ILAng functional specification, and compares it against the RTL implementation [9]. Although these methods automatically generate SVA, defining high-level models and matching them to RTL signals still require significant effort and knowledge, which may discourage hardware designers from using them.

## VI. CONCLUSION

This work presents AutoSVA, a tool to automatically generate testbenches for unit-level FV. Based on annotations made in the signal declaration section of an RTL module, AutoSVA generates liveness and safety properties about control logic to verify forward-progress. Thus, hardware designers can verify their designs at unit-level without requiring FV expertise and with the minimal effort of writing RTL module interface annotations. This pays off quickly, as performing FV early can save significant debugging time during system-level simulation and increase designer confidence that the system will not hang.

We have shown that AutoSVA is useful and effective with an evaluation on widely used open-source hardware projects. This discovered bugs and provided proofs of 7 control-critical RTL modules. AutoSVA generated a total of 236 unique properties (no loops) based on 110 LoC of annotations. The generated FTs are extensible and so they are included in the open-source repository of this work<sup>1</sup>. We envision AutoSVA to become a standard language to define RTL modules' interface expectations, eventually integrated into the SVA specification.

## REFERENCES

- [1] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini, "OpenPiton+Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores," in *Computer Architecture Research with RISC-V, CARRV*, vol. 19, 2019.
- [2] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba *et al.*, "BYOC: a 'bring your own core' framework for heterogeneous-ISA," in *ASPLOS'20*, 2020, pp. 699–714.
- [3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.
- [4] J. Buckingham, "Formal for designers," in *Agile Test Driven Development for ASIC*, 2016.
- [5] I. Cadence Design Systems, "Jaspergold apps user's guide," 2015.
- [6] —, "Jaspergold engine selection guide," 2016.
- [7] E. Cerny, S. Dudani, J. Havlicek, D. Korchemny *et al.*, *SVA: the power of assertions in SystemVerilog*. Springer, 2015.
- [8] C. Cumming, "SystemVerilog Assertions - best known practices for simple SVA usage," *SNUG Silicon Valley*, 2016.
- [9] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip verification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.
- [10] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2012 Std., 2013.
- [11] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *2017 50th Annual IEEE/ACM MICRO*, 2017, pp. 463–476.
- [12] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*. Springer, 1993, pp. 25–60.
- [13] Ping Yeung and K. Larsen, "Practical assertion-based formal verification for SoC," in *2005 Intl. Symposium on System-on-Chip*, 2005, pp. 58–61.
- [14] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-formal," in *CAV*, S. Chaudhuri and A. Farzan, Eds. Springer International Publishing, 2016, pp. 42–58.
- [15] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015.
- [16] S. Sutherland, "Who put assertions in my RTL code? And why? How RTL design engineers can benefit from the use of SystemVerilog Assertions," *SNUG Silicon Valley*, pp. 1–26, 2015.
- [17] C. Wolf, "SymbiYosys," <https://github.com/YosysHQ/SymbiYosys>.
- [18] F. Zaruba and L. Benini, "64-Bit RISC-V 6-stage Ariane core (CVA6)," <https://github.com/openhwgroup/cva6>.

<sup>1</sup><https://github.com/PrincetonUniversity/AutoSVA>