Deductive Formal Verification of Synthesizable, Transaction-level Hardware Designs Using Coq

Tobias Strauch
R&D
EDAptix e.K.
Munich, Germany
Email: tobias@edaptix.com

***Abstract*—We present the compilation process of synthesizable, transaction-level hardware designs into Gallina code. This allows the Coq theorem prover to execute formal verification scripts on top of the automatically generated code to prove individual theorems. The novelty of this work is the use of PDVL (Programming Design and Verification Language), which adds specific language constructs to SystemVerilog to enable an aspect-oriented, transaction-level design style.**

In this paper, we show that PDVL code can also be compiled into Gallina code, with the advantage that it is highly usable for theorem proving and perfect for using the Coq proof assistant. We outline individual proof strategies for theorem proving, sequential equivalence checking, property checking, and show applications of symbolic simulation techniques. This is done by using a complex timer peripheral, AES and UART modules as well as RISC-V based designs.

***Index Terms*—Transaction level design, deductive formal verification, aspect-oriented programming, Coq proof assistent**

I. Introduction

In this paper, we build a bridge between the SystemVerilog (SV) world for digital hardware design and the Coq theorem prover domain for deductive formal verification (FV). The novelty of our approach is an additional layer of supporting language constructs on top of SV. This layer is defined by the aspect-oriented and transaction-level Programming, Design, and Verification Language PDVL [1]. PDVL code can be compiled into synthesizable SV code as well as into Gallina code, which can be used by the Coq theorem prover. We demonstrate proof strategies for functional equivalence and inductive proposition checking (permutation, sorted, etc.) as well as for symbolic simulation by using examples.

In Section II we give a very quick overview of the relevant key features of PDVL. We then introduce the PDVL to Gallina code compiler. Section IV gives a comprehensive list of examples of the different formal verification disciplines for which the proposed flow can be used. The paper finishes with a result and a conclusion section.

II. PDVL

PDVL was introduced in 2017 [1]. We therefore only give a very brief overview. PDVL encapsulates conditions and assignments (called datapaths). Transactions define under which conditions individual datapaths are valid. PDVL is not limited to CPUs, but a quick example of a RISC-V instruction is given in Alg. 1 and Alg. 2.

Algorithm 1 PDVL: RISC-V instruction example (ADDI)

```
1: cl_instr_addi {                                          (* cluster *)
2: c_instr_i_addi { if (opcode_i == 7'h13                   (* condition *)
3:   & funct3i == 3'h0) this; }
4: d_addi { dp_out = rs1_dato+instr[31:20]; }(* datapath *)
5: tr_rv32i_addi {                                          (* transaction *)
6:   unique @c_instr_i_addi { d_rs1i_addr; d_addi;
7:     d_rd_dp_out; d_rd_addr; c_rf_write; d_pc4; } } }
```

Algorithm 2 PDVL: Generating a module hierarchy, joining a cluster into a submodule and defining clock input and edge sensitivity for registers.

```
1: build TB {
2:   build i_duv DUV;
3:   join cl_rv32i cl_rv32imc;                            (* joining *)
4:   join { tr_reg { @e_clk { tr_rv32i_addi; }}} cl_rv32imc;
5:   join cl_rv32imc i_duv; }
```

The aspect-oriented paradigm of PDVL becomes obvious, when looking at the hardware generation process (Alg. 2). Aforementioned elements are stored in clusters. A hardware module hierarchy is built, and clusters are then joined into the individual modules. The merging and signal routing must be handled by the compiler. More information can be found in the PDVL specification [2].

III. PDVL to Gallina compiler

In this section we introduce the tool MRPHS, which compiles PDVL code into Gallina [3] code. Gallina is a specification language that makes it possible to develop mathematical theories and prove specifications of programs. It is used by the Coq theorem prover [4].

1) Preparation of the design under verification (DUV): Before PDVL code is compiled into a Gallina representation, the design under verification (DUV) must be generated by using the relevant PDVL build commands. Once the logic is joined, all sequential and combinatorial signals of the DUV are identified.

2) Inductively defined dynamic, finite state list: We mention the standard Boolean type ”bool” for signal bits in this paper, but any user-defined type can be referenced. Alg. 3 shows that all signals are added to an inductively defined type t_item and

979-8-3503-4859-0/DATE24/© 2024 EDAA

that a dynamic list t_state can be generated based on t_item members.

Only signals with a defined value are added to the t_state list. A signal is removed from the list once its value becomes undefined. This applies to both sequential and combinatorial signals.

Alg. 3, line 12 shows that design-specific functions are defined, which are used in the following Gallina design description. MRPHS also generates a set of more general functions which, for instance, extract the value of a specific item in the t_state list (e.g. f_get) or assign a given value to an item in the t_state list (e.g. f_set). The generated Gallina code mentioned so far is stored as a DUV specific library.

3) Conditions, datapaths, and transactions: For each condition in the PDVL source code, a Boolean signal is generated and added to the item list as well. The condition body is converted into a Gallina definition.

All condition, datapath, and transaction definitions in PDVL are compiled into Gallina code and stored as a second DUV-specific library, which is used by the final proof scripts. Alg. 4 gives an example related to the aforementioned RISC-V ADDI instructions. We will see in the following section, that theorems can be proven based on the automatically generated Gallina code mentioned so far.

4) Automatic type checking by the Coq theorem prover: When compiling the generated Gallina code, the design is automatically type-checked by the Coq theorem prover. Since the signal names and the logic structures are maintained when compiling PDVL into Gallina code, it is very easy to locate the problematic section in the PDVL source code.

5) Signal update and cycle execution: The following extensions are mainly relevant for proofs using symbolic simulation techniques. SV and PDVL are both concurrent programming languages. This program paradigm must be reflected by the Gallina code for theorems utilizing symbolic simulation techniques. To achieve this, MRPHS performs the following steps once the logic is joined:

1) Clock tree and clock domain dependencies analysis.
2) Register and signal ordering based on their dependencies.
3) Writing the definitions for the simulator support.

For sequential elements, the clock trees of the individual clock domains must be identified. Clock tree and clock domain dependencies analysis of digital designs as well as register and signal ordering techniques for digital design model compilers are already discussed in the literature [5].

MRPHS generates Gallina definitions to support symbolic simulation techniques. The definition sim_update (Alg. 5, line 1) walks through the complete design and updates all possible non-sequential signals. Assuming the DUV has only one single clock, then the definition sim_cycle (Alg. 5, line 1) simulates a complete cycle, which includes an update of all sequential elements, followed by an update of all non-sequential signals. For more complex clock domain structures, the sim_cycle definition is adapted accordingly. Alg. 5, line 3 demonstrates, how two cycles of the design can be simulated.

Algorithm 3 Gallina: Signal definition examples, the design state list and a boolean function example

```
1:  Inductive t_item : Type :=
2:    | i_nil
3:    | instr ( l : t_bus32 )
4:    | ...
5:    | pc ( l : t_bus20 )
6:    | reg_file ( l : t_arr32x32 ) .
7:
8:  Inductive t_state : Type :=
9:    | st_nil
10:   | st_cons ( s : t_item ) ( l : t_state ) .
11:
12: Definition f_equal32 ( a b : t_bus32 ) : bool := ...
```

Algorithm 4 Gallina: Compiled condition, datapath and transaction definitions (ADDI example)

```
1: Definition c_instr_i_addi ( st : t_state ) : t_state := ...
2: Definition d_addi ( st : t_state ) : t_state := ...
3: Definition tr_rv32i_addi ( st : t_state ) : t_state := ...
```

IV. Examples

Space does not permit a detailed presentation of all examples, but this list gives a good idea of how PDVL's aspect-oriented and translation-level programming paradigms work together to make FV feasible in these cases.

A. Theorem proving based on inductively defined propositions (Calendar)

We define a calendar as a complex timer that can be used in a System-on-Chip design. It stores a given list of events as a time-ordered linked list in ascending order. As soon as an internal counter matches the first list entry, it generates an interrupt, see Fig. 1. A read command must then remove this first entry from the list. Alg. 6 gives a top level overview of the calendar algorithm written in PDVL.

We want to prove by induction that the output list is a sorted permutation of the programmed list over an unlimited period of time.

The fundamental problem is that synthesizable hardware does not support the concept of an unbound number. The calendar memory is limited in size and the time counter inevitably runs into an overflow. Another problem is that it takes a variable number of cycles to insert the event marker into the memory. Also, an event is only removed when the SoC bus issues a read command. Therefore, there are several reasons for subsequent write commands to be stalled.

Algorithm 5 Gallina: Helper definitions for symbolic simulation

```
1: Definition sim_update ( st : t_state ) : t_state := ...
2: Definition sim_cycle ( st : t_state ) : t_state := ...
3: Definition two_cycles ( st : t_state ) : t_state :=
4:   sim_cycle (sim_cycle st).
```

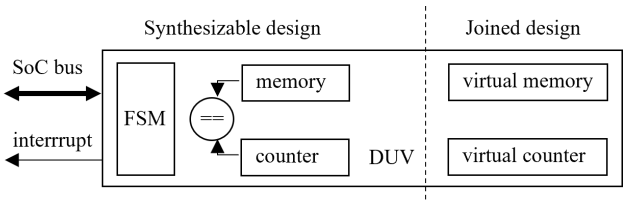

Fig. 1. Calendar overview, syntheziable design and joined design sections

Algorithm 6 PDVL: Calendar top level algorithm

```
1:  tr_top { finite top {
2:    INIT: { tr_init; }
3:    WAIT: {
4:      @c_rd { tr_read; }
5:      else @c_wr {
6:          @c_fifo_empty { tr_write; }
7:          else @c_lesser { tr_head; }
8:          else LOOP; }
9:    LOOP: {
10:     @next_empty { tr_append; }
11:     else @c_lesser { tr_insert; }
12:     else { tr_get_next; } } } }
```

In defining the deductive FV environment, we discovered some specification gaps, which also helped to establish a thorough deductive FV process. One of the extensions to the specification we added was that we defined a minimum processing time (MPT) as the time it takes to complete a write command. An event is only valid if it should happen MPT cycles after a write command has been executed.

PDVL supports aspect-oriented programming, which means that logic can be joined together to form a design. This allows us to join testbench logic like virtual memories and virtual time counters of unlimited size with the design under verification (DUV) for verification purposes only (Fig. 1).

By joining a virtual memory of unlimited size, we solved the problem of the missing concept of unlimited numbers in digital designs. A virtual time counter is also joined. We added additional information like virtual time, variable access times for write commands, etc. to each event. We refer to this bundle of information as a virtual token in the following. The event is stored in the original memory, but also as a token combined with additional information in the virtual memory. The output of a read command is accompanied by the information stored in the token.

We adapted our proof goal to a token-based concept. The goal now is to prove by induction that the tokens issuing an interrupt are a sorted permutation of the write commands and their associated relevant event time.

Alg. 6 lists the top-level transaction. It defines the key states of a finite state machine (FSM) as INIT, WAIT, and LOOP. FSM transitions can also be expressed by transactions that may define additional FSM states. In this case, the transactions at a lower transaction level hierarchy are tr_init, tr_read, tr_write, tr_head, tr_append, tr_insert, and tr_get_next. We have extended the original calendar design written in PDVL with the testbench logic mentioned above, such as the virtual memory and the virtual counter.

The design including all synthesizable transactions as well as transactions defining virtual behavior is compiled into Gallina code. We were able to independently prove relevant statements based on these transactions for later use.

Alg. 7 shows a manually defined helper definition (Fixpoint insert_new), which handles the insertion of natural number into an inductively defined list of natural numbers, in particular a virtual memory. The goal is to bridge the gap between a loop in a FSM (Alg. 6) and the recursive function defined in Alg. 7, which is needed for a proof based on an inductive proposition.

The transaction definitions called by the helper definition insert_new then directly or indirectly reference transactions, which are the results of a PDVL to Gallina compilation process, which also represent signals and signal assignments. By doing that, the virtual memory and the virtual counter definitions are successively replaced by their counterparts defined by the synthesizable PDVL code.

We were able to prove by induction that the calendar tokens which issue an interrupt are an ordered permutation of the write commands and their relevant event time. The proof theorem is presented in Alg. 8, which is exactly the same proof defined in the verification of a simple sorting algorithm based entirely on natural numbers only.

B. Deadlock property checking (Calendar)

In the following, we demonstrate the feasibility of our proposed methodology in terms of property checking, specifically for deadlock verification, which can be viewed as the process of detecting whether a hardware design can enter a deadlock state or not. A deadlock state is a state that once reached, no combination of inputs, or combination of internal state interactions will cause the state to be exited.

To do this, we have used the calendar design we just outlined. A top-level view of the central FSM can be seen in Alg. 6. It has already been mentioned that the transactions which are called by the displayed tr_top transaction expand the FSM by several other states and their related transitions. However, it is obvious that the WAIT state is the key state of the FSM and any transition sequence starting from this state should inevitably lead back to the WAIT state. It is also important to note that the FSM will only begin transitioning from the WAIT statement to other states based on a given read or write condition.

We prove by deduction that starting from the WAIT statement, each transition from it based on a write- or read-condition results in a return to the WAIT state. Our prove strategy relies on the fact that an undefined input value forces the FSM into an undefined state. Therefore, only correctly defined signal values and input sequences are applied when verifying that the FSM reaches the WAIT state again. Having defined all possible input sequences by an inductive definition, all possible FSM traversals are evaluated based on deductive verification mechanisms provided by the Coq theorem prover.

Algorithm 7 Gallina: Manually defined helper definitions

```
1: Fixpoint insert_new (i:nat) : (l: list nat) :=
2:   match l with
3:   | nil =>insert_rd (insert_wr i reset)
4:   | h::t =>
5:     if leb i h then
6:     insert_rd (insert_hd (st_cons (arroo (h::t)) st_nil))
7:     else match nxt (st_cons (arroo (h::t)) st_nil) with
8:     | nil =>nil
9:     | h'::t' =>h' :: insert_new i t'
10:  end
11: end.
```

Algorithm 8 Gallina: Final proof theorem

```
1: Definition calendar_sorting_algorithm (f: list nat ->list nat)
2:              := forall al, Permutation al (f al) /\sorted (f al).
3: Theorem calendar_sort_correct:
4:   calendar_sorting_algorithm sort_new.
5: Proof.
6:   split. apply sort_perm. apply sort_sorted.
7: Qed.
```

C. Sequential equivalence checking (SEC) by using equivalence theorems (AES)

We demonstrate the usage of sequential equivalence checking (SEC) based on a cryptographic design (Advanced Encryption Standard, AES, [6]). The key idea in this example is that we prove the sequential equivalence of transactions defined in both the design and the golden reference model.

Our verification work flow is outlined in Fig. 2. We manually defined an AES specification in Gallina (here model A) while considering the future PDVL implementation at the same time (1). We then manually converted the Gallina A specification into a PDVL synthesizable code (2), which we optimized for a decent area and timing trade-off by inserting registers and a control FSM. The MRPHS tool was used to automatically generate a synthesizable SV version (3) and a Gallina B version from the PDVL source (4).

In this case it is not surprising to find matching pairs of definitions in the Gallina A and B representations. This is true for all functions that characterize an AES algorithm (sBox, SubBytes, MixColumn, . . . , Round). We were then able to prove theorems which state that the individual pairs are functional equivalent (Fig. 2 (5)). This can be done by reusing previously defined theorems. For example, the proof that the sBox transactions are functional equivalent can be used by the theorem, which proves that the SubBytes transactions are functional equivalent as well. An example of an AES Round algorithm in Gallina is given in Alg. 9. Based on that, the individual proofs are straightforward and simple, and are predominantly based on simple Coq tactics such as unfold, rewrite and f_equal. An exception to this direct approach is related to transactions which are sequential in nature. Here the correct number of cycles must be added to match the asynchronous specification of the Gallina A reference version.

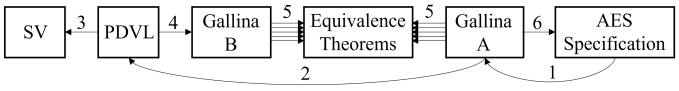

Fig. 2. PDVL and Gallina AES design formal verification flow.

Algorithm 9 Gallina: Definition of an AES round

```
1: Definition AES_Round (st : t_state) : t_state :=
2:   AES_AddRoundKey (AES_MixColumns
3:     (AES_ShiftRows (AES_SubBytes st))).
```

In this paper we concentrate on the verification flow and show that an AES implementation in PDVL is formally provable to be functional equivalent to a Gallina representation. It cannot be concluded from this that the reference model implementation is correct (Fig. 2 (6)). Literature shows how to formally verify an AES implementation or how to obtain an end-to-end proof, linking two high-level specifications [7], [8].

Based on the method shown above, we also checked a constant time property, which states that the runtime of the AES implementation has an exact and therefore constant cycle time. We have also proven an entropy property which states that the ciphertext output of the AES engine always remains constant, except for cycles when the output is valid.

D. Proof by symbolic simulation (RISC-V SoC)

Another class of theorems in FV is based on symbolic simulation, as demonstrated in [9]. Theorems can be proven that for a given initial state and a given stimulation sequence, where state and sequence values can have a symbolic type, all possible resulting state sequences match the expected behavior. We sketch an example based on a RISC-V SoC (here CPU, AXI and UART) with emphasis on the possibility to specify functional behavior based on transaction hierarchies provided by both PDVL and Gallina (Alg. 10).

Algorithm 10 Gallina: RISC-V SoC UART driver sequence

```
1: Definition tr_transfer_uart                    (* compiled *)
2:     (data : t_bus8) (st : t_state) t_state :=
3:     tr_set_NOP (
4:     tr_store_uart ( 0 (                        (* sw a5,0(s0) *)
5:     tr_set_data ( 1 (                          (* li a5,0x1 *)
6:     tr_store_uart ( 4 (                        (* sw a5,4(s0) *)
7:     tr_set_data ( data (                       (* li a5,data *)
8:     tr_set_base_uart st ))))))))).             (* lui s0,0x80030 *)
9: Definition tr_driver_uart                      (* user-defined *)
10:    (data : t_bus8) (st : t_state) t_state :=
11:    tr_read_monitor_uart (
12:    tr_transfer_uart ( data (
13:    tr_conf_uart (
14:    tr_turn_on_uart st)))).
15: Lemma le_txrx_packet :                        (* lemma *)
16:    forall (data : t_bus8), data = tr_txrx_packet data conf.
17: Theorem th_driver_uart :                      (* final theorem *)
18:    forall (data : t_bus8), data = tr_driver_uart data reset.
```

1) Sequence generating transactions:

Alg. 10, lines 9-14 show a user-defined, verification case specific transaction tr_driver_uart written in Gallina, which is built from a nested list of transactions. Given the design state st, the tr_turn_on_uart transaction generates RISC-V code to turn on the UART peripherals, followed by a UART configuration instruction sequence tr_conf_uart. The data is passed to tr_transfer_uart, which takes care of writing the data to the UART via CPU instructions and finally starts the transfer. The transaction tr_read_monitor_uart forces the sequence to wait until the transfer has finished, reads the receive data register in the UART testbench monitor and returns its value.

An example for such a transaction is given in Alg. 10, lines 1-8. The transaction tr_set_base_uart sets the UART peripheral base address and the relevant data in the RISC-V's register file and issues a store instruction to store the data in the UART peripheral transmit register with an address offset of 4. A fixed data value of 1 is then written into the UART's configuration register (no offset), starting the transfer sequence. Then the instruction needs to be set to NOP for the upcoming cycles.

The theorem shown in Alg. 10, lines 17-18 states, that any data value given to the driver sequence tr_driver_uart after the reset state, will be programmed via the CPU into the UART and will be transferred to the testbench UART monitor.

2) Proving lemmas based on combined transactions:

The ability to define more abstract transactions to build a transaction hierarchy allows for the formalization and proof of lemmas. For the given RISC-V SoC example, it can be argued that a UART transmit (TX) control finite state machine (FSM) has its counterpart in the UART receive (RX) FSM in the testbench monitor. For the sake of simplicity, let's assume that the transmission is based on a synchronous single-bit protocol.

The transaction in the TX-FSM that defines the transmission of a single bit is combined with the transaction in the RX-FSM to form a new transaction that defines the entire transmission process of a single bit value. This abstract transaction defined in PDVL can be reused for verification and compiled into Gallina code. A lemma is proven stating that the transmitted data bit value is correct.

The same mechanism is applied for the transfer of a byte and a full packet, while using the previously formally verified lemmas to prove the expected behavior of abstract transactions. Alg. 10, lines 15-16 state the lemma, that any data written to the UART TX data register will be correctly transferred to the RX data register in the testbench monitor. The lemmas are finally used for the overall strategy to prove the final theorem tr_driver_uart by symbolic simulation based on combined transactions.

E. SEC by symbolic simulation (RISC-V ISA)

We demonstrate the feasibility of our proposed methodology for SEC based on symbolic simulation and a RISC-V (RV32IMC) example. In Alg. 1 the PDVL code of an individual instruction is shown (here ADDI). It can be argued that PDVL is a human- and machine-readable executable specification which can be compiled into synthesizable SV code and Gallina code alike. Alg. 4 lists the automatically compiled Gallina code of the relevant sections of the example instruction.

Fig. 3 shows the verification flow for this RISC-V example. Based on an executable specification in PDVL, we automatically generated a Gallina A reference for the RISC-V instruction set (1). The PDVL source code was then manually enhanced to support a 3-stage pipeline architecture (2). From this new custom PDVL model, we automatically generated an SV implementation (3) and a custom Gallina B model (4) using the proposed MRPHS tool. In the last step (5), we used the Coq theorem prover to perform sequential equivalence checking proofs for instructions used in a custom 3-stage RISC-V and the reference 1-stage RISC-V implementation.

Alg. 11 shows that definitions must be provided, which set the design into a relevant state (set_addi) and which sequentially compare relevant sections of the design state (sec_addi).

V. Related work

Coquet: The framework Coquet [11] is based on a library to model and reason about hardware circuits in the Coq theorem prover. It supports the proving of high-level specifications through the use of type-isomorphism. If it turns out to be beneficial to use more complex types as they are defined by Coquet, the basic PDVL to Gallina compilation process as proposed in this paper can be adapted and is likely to remain intact due to Gallina's functional programming paradigm.

VeriCoq: The tool VeriCoq [12] supports most of the synthesizable Verilog constructs and converts parameters, arrays, module hierarchies and module instantiations effectively into their Gallina representation. The process is hierarchy preserving and the output is module based. In this paper we propose MRPHS, which compiles aspect-oriented and transaction-level PDVL designs into a Gallina representation. It exposes the complete design and the associated conditions and datapaths definitions to be accessed by the Coq theorem prover on transaction level.

BSV: Bluespec SystemVerilog (BSV) [13] is a high-level hardware description language of guarded atomic actions. The BSV concept is based on BSV rules and a term rewriting system, whereas each rule can be viewed as a declarative assertion expressing a potential atomic state transition. In [14] it is discussed, how the modular concept of BSV generates new challenges for predicting the compiler output. The sequential behavior of the hardware defined by PDVL is exact and the cycle behavior of the compiler's SV and Gallina code is therefore predictable since no scheduling is involved.

KAMI: According to [15], KAMI is a framework to support implementing, specifying, formally verifying, and compiling hardware designs based on BSV and the Coq theorem prover. It emphasizes modular verification of digital hardware. In contrast, using PDVL provides a transaction-level design paradigm and a transaction-level formal verification paradigm. Transactions can span multiple modules, creating a transaction-level hierarchy that can range from low-level cycle-accurate transactions to approximately-timed or untimed transactions. The concept of a module becomes only relevant when the final SV code is generated.

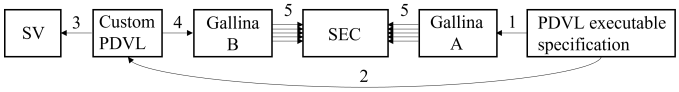

Fig. 3. PDVL and Gallina RISC-V ISA formal verification flow.

Algorithm 11 Gallina: Using symbolic simulation for SEC of 3-stage and 1-stage pipelined instructions.

```
1: Definition sec_addi ( st_a st_b : t_state ) : Prop := ...
2: Definition set_addi ( st : t_state ) : t_state := ...
3: Theorem th_addi : forall (data : t_bus8), sec_addi (
4:     ( sim_update ( set_addi data ) )
5:     ( sim_cycle ( sim_cycle ( set_addi data ) ) ) ).
```

Sail: In recent years, the language Sail [10] for describing the semantics of the instruction set architecture (ISA) of processors has gained increasing attention. Noteworthy is the RISC-V Sail specification and the capability to automatically compile the specification to a Coq compatible Gallina code. To the best of the authors' knowledge, Sail models cannot be converted into predictable, synthesizable RTL code, as PDVL supports it. PDVL is not limited to an ISA. In our work we have made good progress in defining a RISC-V specification similar to the RISC-V Sail model. We demonstrated a compiled Gallina code which is highly reusable. The actual Gallina snapshot resulting from the RISC-V Sail model is greater than 50k lines of code and its usability remains to be seen.

VI. Results

In this section we give an overview (Table I) of the goals we have achieved with the presented formal verification flow. The list of examples we gave in Section III covers a control-oriented design (calendar), a datapath-oriented design (AES), a design based on an instruction set architecture (RISC-V) as well as a system spanning multiple modules (SoC).

The applied formal verification techniques are quite diverse. Table I gives an overview of the individual use cases of formal verification, although it does not always make sense to apply every technique to a specific design example. They range from theorem proving based on inductively defined propositions, property checking theorems (deadlock, constant time, entropy) and sequential equivalence checking (SEC). We also demonstrated symbolic simulation capabilities by using sequence generating transactions and guided transaction abstraction. We also prove SEC with the help of symbolic simulation.

VII. Conclusion and Future Work

In this paper we link digital design and deductive formal verification. The individual deductive formal verification techniques are very diverse. It is not useful to apply all techniques to any given design. We provided a comprehensive list of design examples and showed how individual theorems can be proven. The proposed flow supports proof strategies for functional equivalence and inductive proposition checking (permutation, sorted, etc.) as well as for symbolic simulation.

TABLE I
Individual FV techniques applied on various example designs.

FV technique	**Calendar**	**AES**	**RISC-V**	**SoC**
Inductively defined propositions	x			
Property checking (deadlock)	x			
Property checking (const. time)		x		
Property checking (entropy)		x		
SEC equivalence theorems		x		
Symbolic simulation (sequence)			x	x
Symbolic simulation (abstrac.)				x
SEC symbolic simulation			x	

Future work will be based on coverage-driven formal verification, specifically on how functional coverage, intended to serve as a guide for dynamic verification, as well as assertion-based properties, can be reused and converted into proof theorems for deductive formal verification and how to automatically solve them. We will also open source the PDVL to Gallina compiler.

References

[1] T. Strauch, "An Aspect and Transaction Oriented Programming, Design and Verification Language", IEEE Euromicro DSD 2017, 30 Aug. - 1 Sep., Vienna, Austria, pp. 30 - 39 .

[2] T. Strauch. PDVL Specification v0.1. [Online]. Available: https://github.com/cloudxcc/PDVL

[3] Gallina Development Team. The Gallina specification language. [Online]. Available: https://coq.github.io/doc/v8.9/refman/language/gallina-specification-language.html

[4] Coq Development Team. The coq proof assistant. [Online]. Available: http://coq.inria.fr/

[5] T. Strauch, "Deriving AOC C-Models from D&V Languages for Single- or Multi-Threaded Execution using C or C++", 18. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2015, 3-4 March, Chemnitz, Germany, pp. 1-12, Available Online: https://doi.org/10.48550/arXiv.1807.05442

[6] National Institute of Standards and Technology. Advanced Encryption Standard. NIST FIPS PUB 197, 2001.

[7] A. Bitat, and S. Merniz, "Formal Verification of Pipelined Cryptographic Circuits: A Functional Approach", NISS19: Proceedings of the 2nd International Conference on Networking, Information Systems & Security, March 2019, article no. 35, pp 1–6.

[8] P. Haselwarter, B. Hvass, L. Hansen, T. Winterhalter, C. Hritcu, and B. Spitters, "The Last Yard: Foundational End-to-End Verification of High-Speed Cryptography", Cryptology ePrint archive, paper 2023/185. [Online]. Available: https://eprint.iacr.org/2023/185

[9] Y. Morihiro, and T. Toneda, "Formal verification of Data-path Circuits Based on Symbolic Simulation", Proc. of the Ninth Asian Test Symposium, 2000, 6th Dec, Taipei, Taiwan, pp. 1-8.

[10] The Sail Development Team. Sail. [Online]. Available: https://www.cl.cam.ac.uk/ pes20/sail/

[11] T. Braibant, "Coquet: A Coq Library for Verifying Hardware", Proc. of the First Intern. Conf. on Certified Programs and Proofs, 7-9 Dec. 2011, Kenting, Taiwan, pp. 330–345.

[12] MM. Bidmeshki, and Y. Makris, "VeriCoq: A Verilog-to-Coq Converter for Proof-Carrying Hardware Automation", IEEE Intern. Symp. on Circuits and Systems, ISCAS, 2015, 24-27 May, Lisbon, Portugal, pp. 29-32.

[13] R. Nikhil, "Bluespec System Verilog: Efficient, correct RTL from high level specifications," Proc. 2nd ACM and IEEE Int. Conf. Formal Methods and Models for Co-Design , MEMOCODE'04, 23-25 June 2004, San Diego, CA, USA, pp. 69–70.

[14] M. Vijayaraghavan, N. Dave, and Arvind, "Modular Compilation of Guarded Atomic Actions", ACM/IEEE Intern. Conf. on Formal Methods and Models for Codesign, MEMOCODE 2013, 18-20 Oct. 2013, Portland, OR, USA, pp. 177-188.

[15] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind. "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification", Proceedings of the ACM on Programming Languages, Volume 1, Issue ICFP, Article No.: 24, pp 1–30.