An Automated Framework Towards Widespread Formal Verification of Complex Hardware Designs

Jonathan Certes, Benoît Morgan

An Automated Framework Towards Widespread Formal Verification of Complex Hardware Designs

Jonathan CERTES, Benoît MORGAN
IRIT-ENSEEIHT, University of Toulouse
email : *firstname.lastname*@irit.fr

***Abstract*—Verification is an essential step of critical systems design flow with regard to safety and security. It supports respectively fault and vulnerability removal. *Model-checking* ensures that a design meets its specifications by using an exhaustive state exploration approach. It has been adopted to design critical and/or secure by design embedded hardware systems. On the one hand, *model-checking* is fully automated; on the other hand, it does not scale and faces state-space explosion when working with large industrial circuits and complex specifications. Compositional *model-checking* and theorem proving enable to verify large designs at the cost of finding abstractions and proving an implication of the specifications. In this paper, we present a method along with a framework to reduce this cost and improve hardware verification performances.**

We formally verified a hardware security monitor involved in a remote attestation scheme for microprocessors. This verification could not be achieved using classical approaches as it faces state-space explosion: the monitor is complex enough to be unfit for *model-checking*. So, we applied our method to the verification and successfully proved the security of remote attestation using symbolic *model-checking* and automatic theorem proving. Our verified security monitor is described with an hardware description language and its specifications are written in property specification language. Our method makes extensive use of compositional *model-checking* techniques to leverage the modular and partitioned aspects of most automata involved in hardware modelling. This method is fully automated in a framework build on top of free model checkers and automatic theorem provers. Automation relies on synthesis and translation tools which exploit the modular structure of sequential circuits and avoid having to re-design.

***Index Terms*—Automation, formal verification, security, FPGA**

I. Introduction

Verification of hardware designs is traditionally performed through simulation and testing. Selected input vectors are fed into the system and outputs are checked for correctness. The drawback of these traditional verification techniques is that they aim at tracing the most potential defects and suffer from incompleteness.

Another approach, adopted to design critical and/or secure hardware systems [1], [2], relies on formal methods to ensure that the design meets its specifications. Formal verification is generally conducted in three steps. First, the system (hardware or software) is modeled, for example as an automaton. Then, properties to be satisfied by the system are formally expressed. Eventually, *model-checking* and/or proof is used against the model to demonstrate that it satisfies the required properties.

The major obstacle for widespread application of *model-checking* to real-world designs is called the state-space explosion problem [3]: when the number of states needed to model the system accurately turns out to exceed the physical limits of the computer memory. One of the most successful ways to cope with this problem is to use abstract models and check the property on them instead of the original model. However, over-approximation, as a consequence of abstraction, turns out to generate false negatives [4]. Finding satisfactory abstractions, which reduce the model enough and do not lead to false negatives, is non-trivial. Compositional *model-checking* techniques, such as *localization reduction* and *partitioned transition relations*, guarantee the absence of false negatives [5]. These techniques are particularly adapted to abstract hardware designs as partitioned structure of sequential circuits ensures no interdependence between processes.

In this paper we propose a mostly fully automated framework to improve hardware verification performances and efficiently remove the presence of faults (consequently vulnerabilities) in realistic hardware designs. Our method is based on a specific case of compositional *model-checking* and exploits the modular and partitioned structure of sequential circuits. Then, we extensively present how we applied this approach to formally verify a hardware security monitor for remote attestation of microprocessor software [6]. One of the objectives of this monitor is to preserve the confidentiality of a secret key used to compute a HMAC. It relies on a complex trace decompresser for ARM *CoreSight* debug interface [7] whose verification is the cornerstone of overall system security. Our main objective is to prove the security for remote attestation of microprocessor software from locally verified properties.

A lot of work has been dedicated to efficient and/or automatic abstraction for Register Transfer Level (RTL) *Verilog*, either with a smart predicate partitioning through slicing of bit vectors [8], [9], using uninterpreted functions abstraction [10] or by applying algebraic rewriting to extract arithmetic functions [11]. The authors automatically provide sound partitions for the system to increase compositional *model-checking* efficiency. Unfortunately, writing consistent properties to be verified regarding the content of the generated abstract model is left to the designer's expertise. This process is inconvenient since the main goal is to verify overall specifications (i.e. formal definition of security), potentially expressed with signals or memories that have been removed or split/renamed in the generated abstract models. Also, local properties are

then to be expressed with generated names while making sure that their conjunction still implies the overall specifications. A more convenient approach, that we extensively use in our method, is to write local properties with a proof deduction in mind and then find abstract models that are adapted to their verification.

For a wide adoption of formal methods to hardware verification, expressing specifications and dividing them into local properties must be convenient and theorem proving must be automated. As a consequence, automatic theorem proving must be conducted on highly expressive temporal logics. Moreover, abstraction that is adapted to the verification of these local properties must be automated too. To answer the aforementioned bottleneck and challenges, we introduce in this paper two main contributions:

1) an automated translation of Property Specification Language (PSL) [12], dedicated to automatic theorem proving. The tool also supports uninterpreted functions abstraction, if needed, in order to relieve deductive proof systems in solving the satisfiability problem;
2) an automated framework that computes, for complex hardware designs, adapted abstract models from temporal properties, verifies their soundness and feeds *model-checking* software.

Source code for algorithms that rewrite PSL and verifies the soundness of abstract models is publicly available at [13].

This article is organized as follows: state of the art is given in Section II. Section III details our automated verification framework. The case study is described in Section IV and Section V shows the application of our framework. Eventually, limitations are listed in Section VI.

II. State of the Art

In this section we provide background on modeling complex hardware designs and formal verification.

A. Modeling of complex hardware designs

Automatic translation tool *Verilog2SMV* [14] converts *Verilog* to *SMV* language, which supports similar high-level types of state variables as Hardware Description Languages (HDL) for wires and registers. In particular, *Verilog2SMV* aims at handling designs with memories efficiently [14]. *Verilog2SMV* is built on top of *Yosys* [15], a free *Verilog* synthesis suite. It first flattens the *Verilog* high-level design, synthesizes the RTL circuit while providing optimizations and then translates the output into a corresponding *SMV* model.

Tuerk et al. have formally validated the correctness of a translation from PSL to Linear Temporal Logic (LTL) [16], they produced a *model-checking* infrastructure that works by translating *model-checking* problems to equivalent checks for the existence of fair paths through a Kripke structure [17].

B. Formal verification

Deterministic Büchi automata are specific ω-automata that can accept infinite words. More especially, a word is accepted if and only if the automaton goes infinitely often through accepting states, called acceptance set. A deterministic Büchi automaton can be defined as a tuple $A = \langle \mathcal{Q}, \Sigma, \Delta, I, \mathcal{F} \rangle$ where:

- $\mathcal{Q}$ is a finite set of states
- Σ is an alphabet
- $\Delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function
- $I \subseteq \mathcal{Q}$ is a set of initial states
- $\mathcal{F} \subseteq \mathcal{Q}$ is an acceptance set

Deterministic Büchi automata are used to model finite state machines and formulae in temporal logics [18]. *Model-checking* algorithms manipulate them according to the automata theory approach.

NuSMV is an open-source model checker which implements *symbolic model-checking*, using a fixed point algorithm with *Binary Decision Diagrams* (BDD), where a set of states is represented by a BDD instead of an exhaustive list of individual states [19]. Models are described in SMV language and *NuSMV* supports the analysis of specifications expressed as invariants or in temporal logics, including LTL and PSL.

Duret-Lutz et al. presented *Spot*, a C++ library with Python bindings designed to manipulate LTL and ω-automata [20]. *Spot* contains algorithms to perform the usual tasks in *model-checking*, including filtering, conversion and transformation for LTL formulae and Büchi automata.

As a response to the state-space explosion problem in *model-checking*, R. Kurshan, E. Clarke and H. Veieth formalized the most commonly used abstraction techniques [4]: *localization reduction* abstracts models by hiding variables that are not referenced in the verified property; *predicate abstraction* is an over-approximation technique which may produce spurious counterexamples; *counterexample-guided abstraction refinement* is an iterative process to create new abstract models and checks spurious counterexamples on the concrete model until the checked property is either proved or disproved. Berezin et al. describe the rules to follow to ensure the soundness of compositional *model-checking* techniques, including with *partitioned transition relations* [5], where the global transition relation of a system is written as the conjunction or disjunction of transition relations for the individual components of this system. Also, R. Milner introduced the concept of simulation between automata [21]. Bensalem et al. demonstrated the preservation of properties in the case of simulations parameterized by a Galois connection [22]: establishing a simulation relation between systems, which is straightforward in the case of compositions, allows to share a proof that a property is verified.

III. Automated verification framework

In our framework, overall specifications are formally expressed with PSL and local properties for compositional *model-checking* are elaborated manually to be sufficient to imply the specifications. All PSL properties are translated into LTL by model checker *NuSMV*, which is considered correct. Nevertheless, it is possible to also rely on proven translation works [16], which preserve PSL semantics. Then, *Spot* is

leveraged to prove the implication of the overall specifications from the conjunction of the local properties.

The hardware design is described in *Verilog*, it is synthesized and translated using *Verilog2SMV* into a Büchi automaton, which model is described in *SMV* language. To avoid state-space explosion, we propose a method to automatically generate abstract models, with *localization reduction*, for each PSL property to verify. Property preserving simulation relations are established between the abstract models and the concrete model: this provides a certificate that the abstractions are sound. Then, PSL properties are verified locally with model checker *NuSMV*.

Our approach differs from previous works as abstract models are deduced from the properties to verify. We believe that writing local properties with a proof deduction in mind is key to proving the implication of complex specifications, even if it implies a non-optimal reduction for the abstractions. Furthermore, our framework is automated and relies on synthesis tools so that the generated models are close to the final implementation of the system.

Several Electronic Design Automation (EDA) suite are available in the industry, most of which providing solutions for synthesis and assertion based formal verification. The approach we propose is agnostic to the choice of an EDA toolchain: it can be applied to any of these suite as soon as the dedicated tools can be instrumented with a high level of granularity. For this reason, we instantiated it in a framework which relies on open-source software.

Now, we present our automated framework, depicted in Figure 1, which follows those three major steps:

- Step 1: as opposed to [8], [9], [10], [11], we start by rewriting the specifications in local properties. We take great care in ensuring that the conjunction those local properties is sufficient to imply the specifications.
- Step 2: once we have the local properties, we prove that their conjunction imply the specifications, i.e. when this implication is a tautology. To do so, we rely on *Spot* which provides filtering algorithms for LTL properties [20].
- Step 3: finally, we can move on to the automatic abstraction of the system using the local properties. Hopefully, those local properties are shrunk enough to generate *localization reductions* of the system that can be model checked in a reasonable amount of time. *Model-checking* with *NuSMV* ensures the verification of the local properties.

A. Step 1: Formal expression of specifications and local properties with PSL

In our framework, we take advantage of extended next operators, a subset of PSL that can be seen as syntactic sugar for LTL. This subset is supported by *NuSMV* for *model-checking* [23]. PSL can be described as follows:

In addition to propositional operators, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), PSL features temporal operators along with replicators, which are

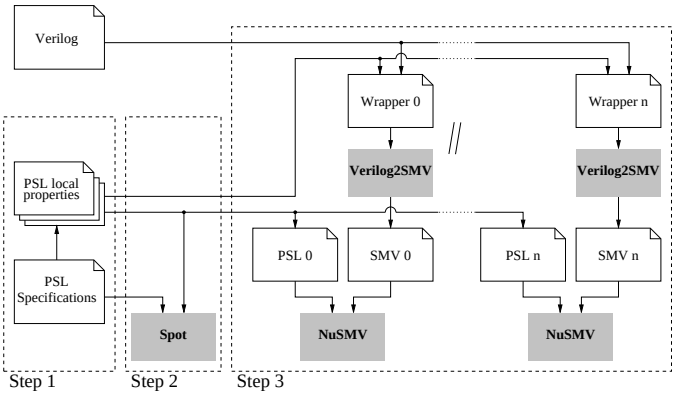

Fig. 1. Automated verification framework

quantification operators. The following operators find their equivalent in LTL, they are simple but of interest to express our specifications:

- $\mathbf{always}(\phi)$: holds if property ϕ is true for all future states;
- $\mathbf{next}(\phi)$: holds if ϕ is true at the next system state;
- (ψ) $\mathbf{until}$ (ϕ): holds if there is a future state where ϕ is true and ψ is true for all states before that.

In addition to these basic temporal operators, PSL also offers extended next operators, including:

- $\mathbf{next\_event\_a}(\psi)[range](\phi)$: holds if ϕ is true at all the next states where ψ is true in the range defined by *range* (where a range is a set of consecutive integer numbers);
- $\mathbf{next\_event}(\psi)(\phi)$: this is a shorthand for $\mathbf{next\_event\_a}(\psi)[1:1](\phi)$.

The last interesting operator is the $\mathbf{forall}$ replicator:

- $\mathbf{forall}\ i\ \mathbf{in}\ \{range\} : \phi(i)$: holds if the conjunction of parameterized sub-properties $\phi(i)$ is true for all possible values of identifier i in the range defined by *range*.

Specifications are formally expressed with PSL and local properties are elaborated while trying to keep their conjunction sufficient for the implication. These tasks rely on the designer's expertise in logics. This is the only part of our framework which is manual.

B. Step 2: Proof strategy

Implication of the specifications is obtained through automatic theorem proving using *Spot*. To achieve this, we have instrumented parsing and translation functions from *NuSMV* in order to convert PSL, which is not entirely supported by *Spot*, to LTL. As a consequence, extended next operators are only expressed with basic temporal operators; for instance, operator $\mathbf{next\_event}$ is expressed as:

$$\mathbf{next\_event}(a)(b) \equiv (\neg(a)\ \ \mathbf{until}\ \ (a \wedge b)) \vee \mathbf{always}\,(\neg(a))$$

After the translation, we generate the following formula from a conjunction of the local properties and an implication of the specifications:

$$(\mathit{property}_0) \wedge (\mathit{property}_1) \wedge \ldots \rightarrow (\mathit{specifications})$$

We rely on filtering functions from *Spot* [20] to automatically process this formula:

- if the filtered formula is a tautology, then specifications are implied;
- if the filtered formula is a temporal expression, then the conjunction of local properties is not sufficient to imply the specifications;
- if the physical limits of the computer memory are reached, we cannot conclude.

Despite being automatic, proving the implication consists in solving a satisfiability problem which suffers from state-space explosion just like *model-checking*. To tackle this problem, we provide uninterpreted functions abstractions to reduce the proof effort:

- logic operations between bit vectors are abstracted into booleans. For instance, equality between two vectors a and b is only considered as either true or false to prove the implication, regardless of the vectors size;
- **forall** replicators are not expanded into a conjunction of sub-properties: only replicated sub-properties (i.e. expressions using the identifier) are considered. This greatly reduces the complexity of the proof but comes at the cost of manually making sure that all possible values are verified through *model-checking*.

For example, the following expression gives a property that has been verified through *model-checking*:

$$\textbf{forall } i \textbf{ in } \{0:255\}: \ \textbf{always}\{(a=i) \rightarrow \textbf{next}(b=i)\}$$

Our abstractions reduce it to boolean "$(a = i)$" always implying boolean "$(b = i)$" at the next state. Since sizes for vectors a and b do not appear in the expression, it is the responsibility of the designer to ensure that *model-checking* with range $\{0:255\}$ for identifier i covers all possible values for both a and b. In a case where the provided abstractions are still not sufficient to run the proof, rewriting local properties from *Step 1* or separating the proof into several steps is required.

Our translation and abstraction algorithm is available at [13]. It has been implemented using *pyNUSMV* [24], a Python framework for prototyping and experimenting with BDD-based *model-checking* algorithms from *NuSMV*.

C. Step 3: Automatic localization reduction and model-checking

A model of the hardware design is described in *Verilog* at RTL. To automatically generate abstract models dedicated to the verification of the properties, we rely on hypothesis **H0**, defined as follows:

H0: the concrete model is composed of multiple finite systems running in parallel. When outputs are unused, optimizations step of synthesis separates them, removes unused registers and leaves the useful parts of the system untouched.

We take advantage of synthesis optimizations step to create *localization reductions* of the model: outputs are left unconnected and all state variables from the model that are irrelevant in verifying the property are abstracted. With **H0**, we assume that the useful parts of the system are left untouched and that there is a simulation relation parameterized by a Galois connection between the concrete model and the generated abstract models.

Since **H0** is a hypothesis, we conduct an *a posteriori* verification and provide a certificate that the abstract model is simulated by the concrete model. Verified properties are then preserved. This task is embedded in our framework and is automatically applied to all generated abstract models. The strength of this approach is that it makes the verification process possible for any synthesis optimizations algorithm and configuration as soon as our certificate guarantees its soundness. The only restriction is to use the same algorithm and configuration for converting both the concrete model and its abstract model.

Regarding our framework, we proceed as follows:

- a *Verilog* wrapper is automatically created where all the module outputs which do not appear in the property are left unconnected. This is achieved using the Verilog Procedural Interface of *Icarus Verilog* [25].
- *Verilog2SMV* synthesizes and translates the *Verilog* model, extended with the previously generated wrapper, and converts it to SMV. Optimizations step from *Yosys* proceeds to a *localization reduction* of the model which is dedicated to verify the property.
- Verification of replicated properties is split into the verification of several non-replicated sub-properties, one for each value of the replicator. This reduces the cost of *model-checking* as size of the BDD grows exponentially with the complexity of the property.
- *NuSMV* is used to verify that the property holds on its dedicated abstract model. If the property does not hold, *NuSMV* generates a counterexample which is converted into Value Change Dump (VCD) format.

This operation is repeated for each property as depicted in sub-graph "Step 3" from Figure 1.

Certificate of soundness

We establish a simulation relation between the concrete model and the generated abstract model, this is performed by comparing transition functions of Büchi automata. To understand the verification of soundness, we first need to introduce some concepts regarding SMV models.

A SMV model is defined by input variables, state variables, initial values and transition functions [14], [19]:

- SMV input variables represent the inputs of the hardware circuit; all possible tuples for SMV input variables values represent the alphabet (Σ) of the automaton.
- SMV state variables represent *Verilog* registers as either single bits, bit vectors or multidimensional arrays; all possible tuples for SMV state variables values represent the finite set of states ($\mathcal{Q}$) of the automaton.
- Their possible initial values are set as a result of the translation from *Verilog initial* directive; initial states (I)

of the automaton are the product of initial values for all SMV state variables.
- Finally, SMV transition functions determine, for each SMV state variable, which SMV state the variable should be set to from its current state and SMV input variables; the global transition function (Δ) of the Büchi automaton is also the result of a product between SMV transition functions for all SMV state variables.

There is no acceptance set ($\mathcal{F}$) in the SMV model since it is a consequence of having a property to be verified: final states are, in *model-checking*, states that satisfy the negation of the property on the product of both the model and the automaton generated from the property.

SMV models are then represented by a conjunctive partitioned transition relation [5]: each partition (δ) of the transition function (Δ) for the model is then a transition function for one SMV state variable. To establish a simulation relation, we verify that all partitions of the transition function and initial states from the abstract model are identical in the concrete model.

Algorithm 1 Comparison of transition functions

Input: C: concrete model, A: abstract model
Output: $boolean$: soundness of the abstraction

```
1: [Δ_C, I_C] = parse(C)
2: [Δ_A, I_A] = parse(A)
3: normalize(Δ_C); normalize(I_C)
4: normalize(Δ_A); normalize(I_A)
5: for all δ_A in partition(Δ_A) do
6:    if not (∃δ_C ∈ partition(Δ_C)|δ_C ≡ δ_A) then
7:       return False
8:    end if
9: end for
10: for all q_A in partition(I_A) do
11:    if not (∃q_C ∈ partition(I_C)|q_C ≡ q_A) then
12:       return False
13:    end if
14: end for
15: return True
```

Algorithm 1 presents how the comparison of transition functions is performed. It requires two models generated from the RTL description of the circuit: the concrete model (C) and the abstract model (A), where both are generated using *Verilog2SMV* with the same synthesis optimization algorithm and configurations. It returns a boolean giving the soundness of the abstraction. This returned value is true if the expression of all transition functions and initial states from the abstract model exist with the same expression in the concrete model.

- First, it parses both the concrete model and the abstract model to extract their global transition function (Δ_C, Δ_A) and sets of initial states (I_C, I_A). As explained earlier, global transition functions of both automata are the results of a product between several transition functions (δ_C, δ_A) from all state variables. Respectively, global initial states of both automata are the results of a product between several initial states (q_C, q_A) from the same state variables.
- Then, a normalization is performed so that descriptions of initial states and transition functions are only expressed with members of the alphabet (Σ) and other states variables ($\mathcal{Q}$). This step is important since different circuits have been processed by the optimization algorithm and descriptions may not result from the same wiring (*Verilog2SMV* preserves the hierarchy of the circuits by expressing wires through the definition of symbols [14]; initial states and transition functions are expressed with these symbols). Normalization allows to abstract the wiring of the circuit as these are not memories and do not alter the BDD when *model-checking*.

An implementation of Algorithm 1 is available at [13]. Equivalence of partitioned transition functions is verified through a comparison at syntactic level. This is justified by the fact that many abstract models have to be checked for soundness (one per property) for a single circuit and it makes the algorithm more efficient. Also, this makes the algorithm easily adaptable to fit a particular purpose if enhanced with the use of semantic comparison instead of checking for an equivalence. It has been implemented using *pyNUSMV*. Thus, parsing and normalization functions follow an instrumentation of *NuSMV*: the same model checker is used to verify the soundness of the abstraction and the satisfiability of the properties.

IV. Case study

We successfully applied our framework to the verification of a hardware security monitor involved in a remote attestation scheme for ARM microprocessors [6]. This could not be achieved with classical automated approaches.

Modern Systems on Chip (SoC), such as Xilinx Zynq-7000, integrate ARM microprocessors along with programmable logic in a single device. This combines the flexibility and the parallelism of a Field-Programmable Gate Array (FPGA) with the performances of an Application-Specific Integrated Circuit. Spatial partitioning for sensitive memories and implementation of our hardware security monitor takes place in the FPGA.

ARM microprocessors come with a debug interface called *CoreSight* which enables real-time instruction flow tracing without slowing down execution. Traces contain information to reconstruct the execution of a program which, in our case study, is composed of cryptographic primitives.

During the computation of an integrity check, the activation of program flow tracing, combined with the addition of specific instructions, provides data that can be used for monitoring. These traces are accessed by the hardware security monitor in the FPGA and processed to achieve remote attestation security.

The proof strategy to achieve remote attestation security is described in [6] along with the architecture of our security monitor. Proving the security is an iterative process involving *model-checking* and proving lemmas for each of four hardware sub-modules as described in figure 2.

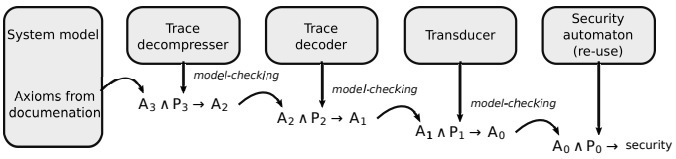

Fig. 2. Proof strategy

Overall security is based on axioms that are formally expressed from the documentation of the SoC (A_3). In particular, the format of ARM *CoreSight* traces and the events causing their output are described in PSL. The security monitor is a composition of four sub-modules dedicated to trace decompression, trace decoding, transduction and security enforcing. *Model-checking* ensures that each sub-module verifies local properties P_i (with $i \in [0:3]$) and proofs of lemmas provide new axioms A_i for the next sub-module.

Understanding the overall verification of the security monitor is not a prerequisite to appreciate the application of our method as it is an iterative process. To illustrate our approach, we focus on one iteration: the verification of temporal properties and proof of a lemma on the *CoreSight* trace decompresser. This is relevant because the trace decompresser is the sub-module of the security monitor which comes with most memories, hence the bottleneck to *model-checking* regarding the state-space explosion problem.

A. The decompresser

Traces are transmitted packet-wise by *CoreSight*. These packets are compressed so that unmodified data between two transmissions is not repeated. A packet header determines the type of packet being transmitted and compression bits inform about the presence of following bytes of data [26]. Figure 3 gives an overview of the decompresser's input/output.

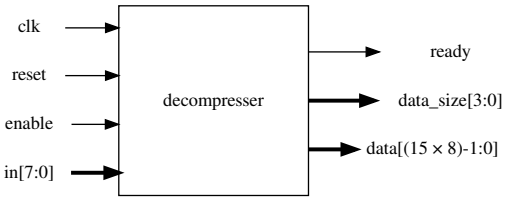

Fig. 3. Overview of the decompresser's inputs/outputs

The decompresser retrieves data from a 8-bit vector (in) and features a synchronization clock (clk) and an $enable$ bit: data is read from input in only when *CoreSight* asserts this $enable$ bit, which can be de-asserted in the middle of a reception for an undefined amount of time. Once all data is received, it is output to a longer vector which contains the content of the whole packet. Minimal size for the decompresser's memory is fixed by the length of the biggest packet, hence a 15 bytes memory to store the payload of *isync* packets [26].

The type of packet being decompressed is identified from the received header. Expected size for a packet depends on this identification and the content of the packet; it can vary between 1 and 15 bytes. Output $ready$ is set at the reception of the last byte, output $data\_size$ gives the number of bytes in the decompressed packet and output $data$ gives its content.

B. Specifications

To guaranty the overall security for the monitor, decompression for several types of packets must be correct. In particular, *CoreSight branch* packets, that provide destination address for an indirect branch and exception informations, are the longest packets which decompression must be verified [26], [6]. In this section, we describe the specifications for correct decompression of a *branch* packet.

- data is memorized from input in, at rising edge of clk, when $enable$ bit is asserted;
- packet type is deduced according to the received header, i.e. the first received byte;
- if input $reset$ is asserted at rising edge of clk, memorization and packet type deduction are discarded;
- output $ready$ rises when the decompresser receives the last byte of a *branch* packet and is set only during one clock cycle;
- output $data\_size$ gives the number of bytes in the decompressed packet when $ready$ rises;
- output $data$ takes the value of the decompressed packet when $ready$ rises.

Figure 4 shows an example of the reception of a 4 bytes packet, where w, x, y and z are received bytes (w contains the packet header) and signal $memory$ refers to an internal memory. The last received byte is output at its reception.

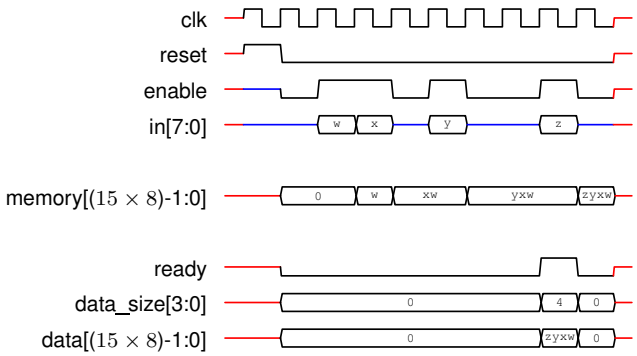

Fig. 4. Description of the data flow stream

V. Application to the Case Study

In this Section, we show how we applied our approach to verification of the decompression for ARM *CoreSight branch* packets.

The concrete model for the decompresser uses a total of 132 bits for SMV state variables and 11 bits for SMV input variables. Its description in *Verilog* HDL has approximately five hundred lines of code. *Verilog2SMV* is used to re-create the SMV model from the HDL, which has approximately five thousand lines of code. Both *Verilog* and SMV models are available at [13].

Verification is conducted using our framework on a computer cluster of 256 nodes for parallelization. Each node has $4GB$ of RAM and a single-core CPU running at $3GHz$. Computation times and memory usage for this application may differ if our approach is instantiated in an other toolchain.

A. Step 1: Formal expression of specifications and local properties

The first step consists in manually expressing the specifications and local properties in PSL. Formal expression of specifications is a difficult process and might give different results depending on the designer's writing choices: several PSL expressions may translate the same specifications in natural language.

Here, we provide an example for the formal expression of specifications from section IV-B. To ease understanding, we purposefully omit certain aspects of the specifications, such as the input filtering depending on the format of *CoreSight branch* packets. We also omit the duplication of **forall** operators for all replicators: this is not syntactically correct as a PSL replicator only accepts one identifier but it greatly reduces the representation of the expression. Complete PSL expressions with correct syntax and dependencies to *CoreSight* format are available at [13].

To guaranty the overall security for the monitor, decompression for the first 7 bytes of *branch* packets must be correct as they contain destination addresses and exception informations [26]. This is formally expressed by specification 1.

$$
\begin{aligned}
&\textbf{forall } i_0,i_1,i_2,i_3,i_4,i_5,i_6 \textbf{ in } \{0:255\}: \\
&\textbf{always}(\\
&\quad ((clk \wedge reset) \vee (clk \wedge ready)) \wedge \textbf{next}(\\
&\qquad \neg(clk \wedge reset) \textbf{ until } (clk \wedge ready) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data\_size \geq 7) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[1:1](in = i_0) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[2:2](in = i_1) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[3:3](in = i_2) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[4:4](in = i_3) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[5:5](in = i_4) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[6:6](in = i_5) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[7:7](in = i_6) \\
&\quad) \\
&\rightarrow \\
&\quad \textbf{next}(\\
&\qquad \textbf{next\_event}(clk \wedge ready)(data[7:0] = i_0) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[15:8] = i_1) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[23:16] = i_2) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[31:24] = i_3) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[39:32] = i_4) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[47:40] = i_5) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data[55:48] = i_6) \\
&\quad) \\
&) \qquad (1)
\end{aligned}
$$

This PSL specification deals with traces starting when the decompresser is reset or ready. From the next state, a restriction for the traces is that input $reset$ cannot be asserted at a rising edge of clock until output $ready$ is set. Also, only traces where $data\_size$ is greater or equal to 7 are considered.

Left side of the implication expresses that data is memorized from input in, at rising edge of clk, when $enable$ bit is asserted. This is true for all possible values between 0 and 255 for each of the 7 bytes of data. Right side of the implication expresses that output $data$ takes the value of the decompressed packet when $ready$ rises.

Expression of local properties is a manual process. Conjunction for these local properties must imply PSL specification 1. The designer can keep in mind that this implication is proven by an automatic theorem prover and that abstracting a module's output in a property reduces both the complexity of the property and the size of the automaton for *model-checking*.

One possible solution for expressing local properties is to split the content of output $data$ into seven bytes, where each local property has one replicator of 256 possible values. An other solution is to split the content of output $data$ into 7×8 bits, where each local property has one replicator of 2 possible values. The first solution is straightforward as expressing a local property only consists in removing **next_event_a** operators from specification 1. The second solution requires more rewritings but reduces the complexity of the property and the size of the automaton even more, enabling *model-checking* for more complex specifications.

For our case study, following the first solution is sufficient to enable formal verification. An example for one local property is given by expression 2. A total of seven local properties, following the same approach, is needed to imply the specification.

It is possible to give more expressive power to a local property — in case the same property helps proving more than one specification. For this reason, the value of output $data\_size$ in property 2 is now greater or equal to one —

$$
\begin{aligned}
&\textbf{forall } i_0 \textbf{ in } \{0:255\}: \\
&\textbf{always}(\\
&\quad ((clk \wedge reset) \vee (clk \wedge ready)) \wedge \textbf{next}(\\
&\qquad \neg(clk \wedge reset) \textbf{ until } (clk \wedge ready) \\
&\quad \wedge \textbf{next\_event}(clk \wedge ready)(data\_size \geq 1) \\
&\quad \wedge \textbf{next\_event\_a}(clk \wedge enable)[1:1](in = i_0) \\
&\quad) \\
&\rightarrow \\
&\quad \textbf{next}(\\
&\qquad \textbf{next\_event}(clk \wedge ready)(data[7:0] = i_0) \\
&\quad) \\
&) \qquad (2)
\end{aligned}
$$

since we only verify the first byte of data. But, care must be taken, when modifying local properties, that the specification remains provable.

B. Step 2: Proof strategy

Once all seven local properties are available, a proof that the specifications are implied must be conducted. Since we wrote the local properties with proof deduction in mind, the proof strategy is straightforward:

- expressions that appear in both the specification and the local properties can be left uninterpreted and abstracted; for instance, we can replace the following PSL expressions with single booleans:
  - **next_event_a**$(clk \wedge enable)[1:1](in = i_0)$
  - **next_event**$(clk \wedge ready)(data[7:0] = i_0)$
- axioms are added if we gave more expressive power to a local property. For instance, one axiom can be as follows: if $data\_size$ is greater than 7, then it is greater than 1.
- we rely on our framework to translate PSL into LTL, provide abstractions to reduce the proof effort and generate the formula. *Spot* is leveraged to prove the implication.

Figure 5 shows how we prove for the decompression for ARM *CoreSight branch* packets. This can be separated in three steps:

1) uninterpreted PSL expressions are replaced with single booleans ;
2) a proof is conducted that the specification is implied by the conjunction of local properties. This step is where *Spot* is leveraged, it is represented in red in Figure 5.
3) uninterpreted PSL expressions are refined to re-create the specification.

Optimizations can be achieved by separating the proof into several steps. In this case, we create an intermediate property, which is larger than the local property but smaller than the specifications to prove, then we follow the same approach. For example, an intermediate property can describe the decompression of the first four bytes; then an other intermediate property describes the decompression of the last three bytes. Steps $2'$ and $2''$ in Figure 5 represent how we optimize the proof. Sources to reproduce the experiment are available at [13]. Table I summarizes memory usage and computation times.

Optimizations	%MEM	Computation time
No (step 2)	9.6	2 minutes
Yes (steps $2'$ and $2''$)	< 1	< 2 seconds

TABLE I
PROOF: MEMORY USAGE AND COMPUTATION TIMES

Note: separating the proof into several steps is also an elegant solution in case we opt for the second solution when expressing the local properties, i.e. a split of output $data$ into 7×8 bits. In this case, property from expression 2 serves as an intermediate property. It must be proven from a conjunction of the local properties and the rest of the proof remains the same.

Regarding the abstractions of uninterpreted expressions, i.e. steps 1 and 3 in Figure 5, we provide a *coq* proof at [13] that this is correct at semantic level for any temporal property. This proof relies on a LTL library written in *coq* [27]. We assume that LTL semantic is identical for both *Spot* and this *coq* library. So, for this particular proof, verifying the correctness in *Spot* for each abstraction is unnecessary. Nevertheless, to advocate in favor of abstraction of uninterpreted functions, we leveraged *Spot* for a verification in some cases. A verification consists in proving an implication between a local property and its abstracted form (step 1 in Figure 5). In the abstracted form, for each value of integer j, the following PSL expressions are replaced with single booleans:

- **next_event_a**$(clk \wedge enable)[j+1:j+1](in = i_j)$
- **next_event**$(clk \wedge ready)(data[8 \times (j+1) - 1 : 8 \times j)] = i_j)$

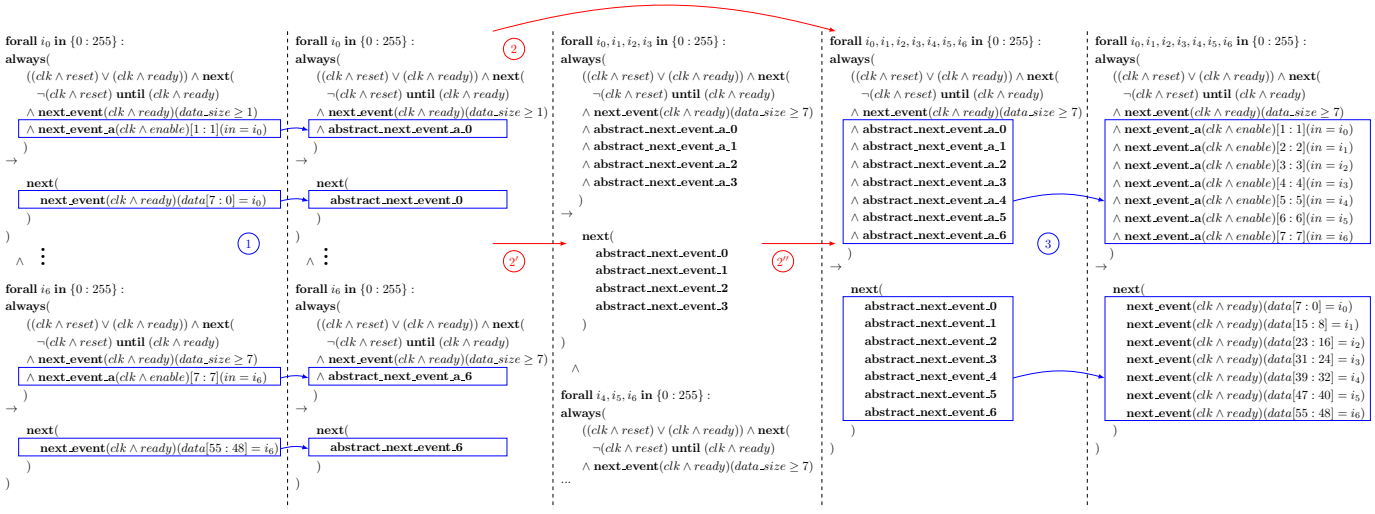

Fig. 5. Proving the decompression for ARM *CoreSight branch* packets

This verification is resource-intensive as complexity of the properties grows exponentially with the value of integer j. Sources to reproduce the experiment are available at [13]. Table II summarizes memory usage and computation times.

j	%MEM	Computation time
0	< 1	10 seconds
1	< 1	35 seconds
2	1.8	4 minutes
3	100	(runs out of memory)

TABLE II
UNNECESSARY VERIFICATION OF ABSTRACTIONS

This shows the limitations of automatic theorem proving when dealing with complex properties. Expression of extended next operators with basic temporal operators, which is not required if they appear in both the local properties and the specification, prevents the proof to complete. This is because the number of basic temporal operators grows exponentially with the value of integer j for our PSL expressions.

C. Step 3: Automatic localization reduction and model-checking

The conjunction of our local properties is sufficient to imply the specifications. Now, *model-checking* must guaranty that the decompresser verifies these local properties.

Since we opted for local properties where the content of output $data$ is split into bytes, our framework automatically removes 14 bytes of memory from the model, out of 15, at each abstraction. Also, since our local properties have one replicator of 256 possible values, our framework automatically splits the verification into 256 steps where properties are not replicated. As a consequence, a verification of a local property on our concrete model results in 256 verifications of sub-properties that are 2^8 times smaller, on an abstract model that is $2^{(14\times 8)}$ times smaller, hence an exponentially reduced BDD.

Note: even if the specification is not split into local properties, our framework would also automatically remove 8 bytes of memory out of 15 in the abstraction. This is because these 8 bytes of memory do not appear in the specification either. The verification process would also be split into 7×256 steps since the specification has 7 replicators with a range of 256 values each.

To verify the soundness, our framework automatically creates one concrete SMV model — in addition to an abstract SMV model for each sub-property. For each abstract SMV model, it tries to establish a simulation relation with the concrete SMV model:

- in case of success, *NuSMV* is used to verify that the property holds;
- in case of failure, it shows in a log file the differences between transition functions and initial states for the state variables that differ.

NuSMV computes *symbolic model-checking* of the sub-properties on their dedicated abstract model. It shows in a log file the sub-properties and the results of the computation:

- in case a sub-property holds, *NuSMV* provides the mention "is true";
- in case a sub-property does not hold, *NuSMV* provides a counterexample. In addition to the log file, our framework converts the counterexample into a VCD file which can be visualized with a waveform viewer.

Memory usage and computation times depend on both the size of the abstract model and the complexity of the property. Size of the abstract model is identical for each of our seven local properties. Complexity of the property mainly results of having a high integer number in the range of PSL operator **next_event_a**. Table III and figure 6 summarize memory usage and computation times on one node of our computer cluster, i.e. for the verification of one sub-property. Row indexes represent the integer number in the range of PSL operator **next_event_a**. Indexes 1 to 7 are of interest as they represent our seven local properties.

Note: for the sake of the experiment, we verified two unnecessary properties to evaluate our strategy of decomposition for more complex specifications. This shows that we could rely on the same solution if we specify the decompression of packets with 8 bytes of data, but we nearly reach the limits of our computer's memory at the 9th byte.

#	%MEM	Computation time
1	21.9	7 seconds
2	28.4	8 seconds
3	29.9	10 seconds
4	39.5	15 seconds
5	60.2	35 seconds
6	28.7	7 minutes
7	41.6	25 minutes
8	47.9	39 minutes
9	95.0	6 hours 55 minutes

TABLE III
MODEL-CHECKING: MEMORY USAGE AND COMPUTATION TIMES

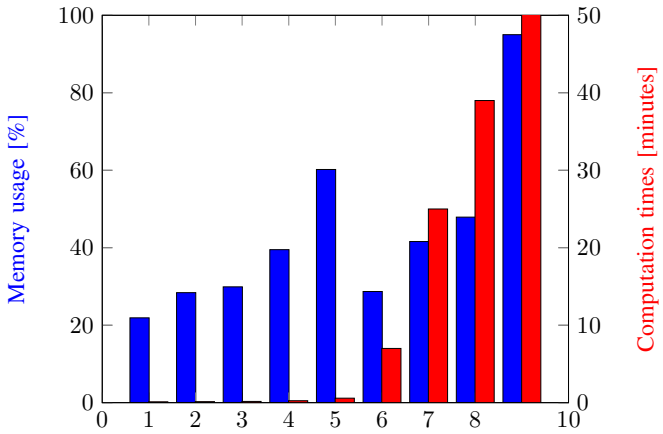

Fig. 6. Model-checking: memory usage and computation times

Sources to reproduce the experiment are available at [13]. We recommend to proceed to *model-checking* on a computer cluster of 256 nodes. Otherwise, computation times to verify one local property would be approximately 256 times higher.

D. Summary

To sum up, our framework automatically provided sound abstractions for our ARM *CoreSight* trace decompresser and increased compositional *model-checking* efficiency. Synthesis optimizations algorithms helped deducing these abstractions from the local properties to verify.

We had several solutions to express our local properties: we opted for a decomposition of an output vector into bytes as it eases proof deduction and allows *model-checking* in a reasonable amount of time. Fortunately, expression is conducted with proof deduction in mind. So, finding a strategy to prove the specification is straightforward: we opted to abstract functions depending on the decomposed bytes as they appear in both the specifications and the local properties.

In the case where our specification would have been more complex, we could also have opted for a decomposition of the output vector into bits. On the one hand, this solution would have forced us to add one step in the proof strategy since local properties would be smaller. On the other hand, it would have greatly reduced *model-checking* computation times as the abstract model would automatically be reduced as well.

We also applied our automated framework to the verification of several hardware modules: we verified the correctness for trace decompression, trace decoding, transduction and security enforcing. In the end, it allowed to verify the design of our whole security monitor and to prove the security for remote attestation of microprocessor software [6].

VI. Limitations

This approach can be generalized to the verification of other sequential circuits with complex specifications. Although, automatic theorem proving may show some limitations. Despite many abstractions, implication of the specifications may be too large to be processed in an acceptable amount of time. A solution is then to create an intermediate property, which is larger than the ones that are verified through *model-checking* but smaller than the specifications to prove. Then the proof must be separated into several steps following the same approach. An example of a proof in more than one step is publicly available at [13].

Other limitations may occur when it comes to verifying systems depending on a high number of inputs or when the memory of an atomic automaton already exceeds the tolerated size. For such systems, a workaround consists in altering their architecture so that they can be turned into partitioned systems. Such a strategy must be anticipated as freezing the architecture of a system is part of the early stages of the design flow for integrated circuits.

In a case where the specification contains PSL operators $\mathbf{next\_event\_a}$ with a too high integer number in the range, a different proof strategy must be considered. This might imply using *model-checking* to verify simpler properties — for instance, about the internal memory of the hardware — and rely on a more complex proof to imply the specifications. A drawback is that it increases the level of expertise needed to conduct the proof. In such case, relying on a proof assistant might be a more elegant solution than decomposing the proof in a high number of steps.

VII. Conclusion

The major obstacles for widespread application of formal verification to real-world designs are the complexity of their specifications and the state-space explosion problem. A lot of work has been dedicated to automatically provide smart abstractions for RTL *Verilog* and increase compositional *model-checking* efficiency [8], [9], [10], [11]. However, dealing with complex specifications requires to prove their implication from local properties, and writing properties function of automatically generated abstract models is inconvenient for automatic theorem proving.

In this paper we propose a framework to overcome this challenge and help the community going towards the adoption of formal methods. Our framework leverages the modular and partitioned aspects of sequential circuits and relies on synthesis and translation tools [14] to avoid a profound redesign. It computes convenient and sound abstract models from RTL *Verilog* and the properties to verify, allowing to prove complex specifications for hardware designs. Translation of PSL properties, to be fed to automatic theorem provers, is also automated and supports uninterpreted functions abstraction to relieve deductive proof systems in solving the satisfiability problem.

Further work can be conducted such as using a proven translation from PSL to LTL [16] to ensure the correctness of the LTL properties to manipulate. Nevertheless, our framework has been successfully applied as is to verify the design of a hardware security monitor for remote attestation of microprocessor software [6]. Such hardware design contains many memories and has similar complexity to the ones we find in industrial designs. This brings hope to see our approach adopted in the industry, especially since it is agnostic to the choice of an EDA toolchain.

References

[1] W. Khan, M. Kamran, S. R. Naqvi, F. A. Khan, A. S. Alghamdi, and E. Alsolami, "Formal verification of hardware components in critical systems," *Wireless Communications and Mobile Computing*, 2020.

[2] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *28th USENIX Security Symposium*. Santa Clara, CA: USENIX Association, Aug. 2019.

[3] C. Wang, G. D. Hachtel, and F. Somenzi, *Abstraction Refinement for Large Scale Model Checking*. Springer, 2006.

[4] E. M. Clarke, R. P. Kurshan, and H. Veith, "The localization reduction and counterexample-guided abstraction refinement," in *Time for Verification, Essays in Memory of Amir Pnueli*, Z. Manna and D. A. Peled, Eds. Springer, 2010.

[5] S. Berezin, S. V. A. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany. Revised Lectures*, W. P. de Roever, H. Langmaack, and A. Pnueli, Eds. Springer, 1997.

[6] J. Certes and B. Morgan, "Remote attestation of bare-metal microprocessor software: a formally verified security monitor," *The 5th International Workshop on Cyber-Security and Functional Safety in Cyber-Physical Systems (IWCFS)*, 2021.

[7] *Coresight Technology System Desgin Guide*, no. ARM DGI 0012D ID062610, 2006-2013.

[8] Z. S. Andraus and K. A. Sakallah, "Automatic abstraction and verification of verilog models," in *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA*, S. Malik, L. Fix, and A. B. Kahng, Eds. ACM, 2004.

[9] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word-level predicate-abstraction and refinement techniques for verifying RTL verilog," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2008.

[10] Y. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. K. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *2016 Formal Methods in Computer-Aided Design, FMCAD, Mountain View, CA, USA*, R. Piskac and M. Talupur, Eds. IEEE, 2016.

[11] C. Yu and M. J. Ciesielski, "Automatic word-level abstraction of datapath," in *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada*. IEEE, 2016.

[12] *1850-2010 IEEE Standard for Property Specification Language (PSL)*. IEEE, 2010.

[13] J. Certes and B. Morgan, "Verification materials: source code and examples," 2021. [Online]. Available: https://gitlab.irit.fr/these-jonathan-certes-public/erts-2022

[14] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani, "Verilog2smv: A tool for word-level verification," in *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany*, 2016.

[15] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[16] T. Tuerk and K. Schneider, "From PSL to LTL: A formal validation in HOL," in *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, Proceedings*, J. Hurd and T. F. Melham, Eds. Springer, 2005.

[17] T. Tuerk, K. Schneider, and M. Gordon, "Model checking PSL using HOL and SMV," in *Hardware and Software, Verification and Testing, Second International Haifa Verification Conference, HVC 2006, Haifa, Israel, Revised Selected Papers*, E. Bin, A. Ziv, and S. Ur, Eds., 2005.

[18] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, Proceedings)*, 1995.

[19] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv version 2: An opensource tool for symbolic model checking," in *Proc. International Conference on Computer-Aided Verification (CAV)*. Copenhagen, Denmark: Springer, July 2002.

[20] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — a framework for LTL and ω-automata manipulation," in *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, Oct. 2016.

[21] R. Milner, "An algebraic definition of simulation between programs," in *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK*, 1971.

[22] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis, "Property preserving simulations," in *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, Proceedings*, G. von Bochmann and D. K. Probst, Eds. Springer, 1992.

[23] *NuSMV 2.6 User Manual*. FBK-irst - Via Sommarive 18, 38055 Povo (Trento) – Italy, 2015.

[24] S. Busard and C. Pecheur, "Pynusmv: Nusmv as a python library," ser. LNCS, G. Brat, N. Rungta, , and A. Venet, Eds., vol. 7871. Springer-Verlag, 2013, pp. 453–458.

[25] S. Williams, "Icarus verilog," http://iverilog.icarus.com/.

[26] *CoreSight Program Flow Trace Architecture Specification*, no. ARM IHI 0035B ID060811, 1999-2011.

[27] C. M. V. Reyes, "Ltl (linear temporal logic) in coq," https://github.com/spidermoy/LTL_Coq.