Word-level Formal Verification Using Abstract Satisfaction

Rajdeep Mukherjee

Department of Computer Science, University of Oxford, UK
rajdeep.mukherjee@cs.ox.ac.uk

Abstract. With the ever-increasing complexity of hardware (HW) and SoC-based designs for mobile platforms, demand for scalable formal verification tools in the semi-conductor industry is always growing. The scalability of hardware model checking tools depends on three key factors: the *design representation*, the *verification engine*, and the *proof engine*. Conventional SAT-based bit-level formal property checking tools for hardware, as shown in the top flow of Figure 1, converts the design into a netlist, typically represented using And-Inverter graphs (AIGs). These tools can not exploit the word-level structure of designs given at the register transfer level (RTL). Over the last decade, formal hardware verification tools have therefore implemented a word-level representation of the transition relation, typically represented using BLIF format or tool-specific word-level formats, thus enabling the use of modern solvers for Satisfiability Modulo Theories (SMT). However, the performance of word-level symbolic execution engine is determined by the level of abstraction of the symbolic expressions and the power of the rewrite engines used by the SMT solvers. State-of-the-art bit-level and word-level formal verification tools scale up to block level or small IP level circuits. These tools generally do not scale to large IPs, subsystems, or full SoC designs. Figure 1 presents a conventional bit-level hardware/software co-

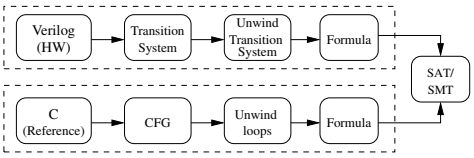

Fig. 1. Hardware property checking and HW/SW co-verification tool flow

verification flow in the presence of a firmware. The main limitation of this flow is that the hardware (typically represented in Verilog RTL) and software (in C) synthesis flow meet only at the level of the solver. The high-level design structure of C and RTL is already lost at this level and thus there is very limited opportunity for scalable reasoning.

In this thesis, we propose a novel approach for verifying hardware circuits, at the heart of which is a verifier for software. To this end, we translate hardware design, typically given in Verilog at register-transfer level into an equivalent word-level ANSI-C program, which we call *software netlist*, following synthesis semantics.

The property of interest is instrumented into the C program as an assertion. This change in the design representation at the front-end allows us to leverage the precision and scalability of state-of-the-art software analyzers in the back-end. In particular, we apply well known software verification techniques such as path-based symbolic execution [1], abstract interpretation [2] and abstract conflict driven learning [3] to hardware designs by synthesizing them to a software netlist model. These techniques have never been applied to formal hardware verification based on netlists.

Contributions We make the following contributions in this research.

1. We present a high-level software representation for hardware circuits expressed at register-transfer level. To this end, we develop a tool *v2c* to automatically synthesize a software netlist model in C from hardware circuits expressed in Verilog RTL following synthesis semantics. The work is published in TACAS 2016 [4].
2. The high-level representation of hardware circuits enable higher level reasoning using various software verification techniques. In particular, we propose a novel hardware verification flow for bounded and unbounded safety verification of hardware circuits using state-of-the-art software analyzers such as *CBMC*, *Astrée*, *Klee*, *CPAChecker*, *Ultimate Automizer*. The work is published in ISVLSI 2015 [5] and DATE 2016 [6].
3. We develop a HW/SW co-verification framework embodied in our tool, *VerifOx*, for bounded HW/SW co-verification of IP level and SoC level designs. VerifOx supports IEEE 1364-2005 System Verilog standards and the C89, C99 standards. VerifOx also supports SAT and SMT backends for constraint solving. It first builds an unified HW-SW model in C for co-verification task. A precise path-based forward symbolic execution is performed on this unified model using eager path pruning strategy and incremental SAT solving. Our experiments show that *VerifOx* is order of magnitude faster than conventional HW/SW co-verification tools based on netlist due to its path-based symbolic execution approach.
4. We develop a tool for C versus RTL equivalence checking using automatic trace partitioning to scale up sequential and combinational equivalence checking of complex arithmetic circuits and other data and control-intensive circuits. The work is published in ISVLSI 2015 [7].
5. We develop a new program analysis technique, *ACDCL*, based on Abstract Conflict Driven Clause Learning [3] for verifying software netlist models generated from hardware circuits in Verilog. ACDCL performs program and property driven trace partitioning to improve the precision of the analysis using abstract domains like – intervals, octagon, polyhedra and equality domains.

Research Methodology

The primary motivation for the transition from bit level to word level is to gain scalability [5,6]. The use of high-level structures as a design description for model checking is a holy grail of hardware verification. To this end, we take a step further and move beyond word-level verification by translating the hardware design at RTL to a software representation in C. Note that the software model is not an abstraction of the hardware circuit, but a bit-precise and cycle-accurate model.

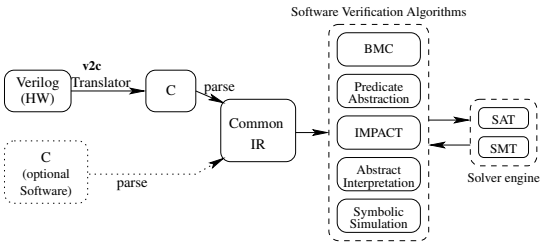

Fig. 2. Proposed hardware verification and co-verification tool flow

This preserves the word-level structure and the control-flow information of the input RTL design. The translation details are presented in [4].

Figure 2 gives an overview of the tool architecture we propose. The input Verilog design is synthesized to a software netlist model in C using *v2c*. This high-level representation allow us to verify hardware IP's using different state-of-the-art software verifiers. Note that the hardware design may be augmented with software, such as firmware or high-level models of surrounding IPs given in C. The proposed tool architecture also provides an unified framework to uncover hardware/software design bugs much early in the verification process. To the best of our knowledge, this is the first attempt to perform hardware verification and hardware/software co-verification using software analyzers.

Our evaluation using different program analysis techniques show that these techniques are competitive to the contemporary hardware verifiers based on netlist. For example, state-of-the-art abstract interpretation tools like Astrée was never optimised for precise hardware analysis, and we thus believe that there is scope for new tools that implement abstract interpretation using abstract domains developed specifically for this task, e.g., by applying abstract conflict driven clause learning (ACDCL) [3] – a new program analysis that embeds an abstract domain inside the Conflict Driven Clause Learning (CDCL) algorithm. From the abstract interpretation point of view, ACDCL is an abstract interpreter that uses decision and learning to increase transformer precision. From the decision procedure perspective, ACDCL is a SAT solver for program analysis constraints and is thus a strict generalisation of propositional CDCL solvers. Constraint propagation in ACDCL uses fixed point iteration, decisions restrict the range of intervals and learning generates program analysis constraints (not assumptions) that preserve the error reachability. Moreover, ACDCL implicitly perform program and property driven trace partitioning to increase the precision of the analysis. Our experiments show that the abstract domains necessary for hardware property checking are – booleans, bitfield, constants, interval, equality and octagons domains.

Let us consider a simple safe program P as shown in Figure 3A. Astrée fails to verify the safety using interval and trace-partition domain due to control-flow join at location $n5$. Astrée requires external hints, provided by manually annotating the code with partition directives at $n1$, to prove safety. In general, the imprecision is either intended by the tool because such high precision analysis is normally not required for runtime error analysis or the imprecision is unavoidable due to the complexity of the application under analysis.

On the other hand, ACDCL uses decision and learning to generate the proof using simple interval domain. The analysis associates an interval with each location

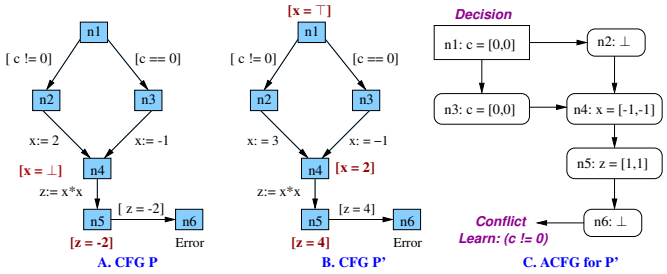

Fig. 3. A control flow graph for P, P' and abstract conflict graph (ACFG)

and variable. Assuming $bool : c = [0,1]$, ACDCL performs backward propagation starting from $n6$ by computing the weakest precondition for every statement in P and immediately infers that $(n4 : x = \bot)$, thus proving safety as shown in red color in figure 3A. Now, let us consider program P' in Figure 3B. Assuming $char : c$, ACDCL performs both forward and backward propagations (shown in red color in Figure 3B). The analysis then perform sequence of decisions starting with $c = [0,255]$ and reaches a conflict when the decision is $(n1 : c == [0,0])$ which is connected to $(n6 : \bot)$ and suffices to prove safety. The deductions made during fixed point iteration are represented by abstract conflict graph shown in Figure 3C. Similar to conflict analysis phase in SAT solvers, ACDCL learn $(n1 : c = [1,255])$, that is all error traces must satisfy $(c \neq 0)$ at n1. The analysis backtracks discarding all assumptions. Interval analysis is run with the learnt constraint and can prove safety. Thus, decisions and clause learning are used to avoid case based reasoning. The advantage of ACDCL over propositional SAT solver is that the decision heuristics in ACDCL can exploit the program structure by making decisions on interesting program variables (for example variables in conditional branches or loop variables). Experimental evidence shows that this leads to significantly less number of decisions and less number of conflicts compared to the propositional solver. Compared to classical abstract interpretation based tools, ACDCL automatically performs program and property driven trace partitioning using decision and clause learning to generate proofs using simpler domain – thus, it is more precise than classical abstract interpretation.

Abstract Conflict Driven Clause Learning

Static program analysis based on abstract interpretation [2] has been widely used to verify certain classes of properties for safety-critical systems. In abstract interpretation, a given program is analysed with respect to a set of given abstract domains. However, ACDCL is a novel program analysis technique that embeds an abstract domain inside the CDCL algorithm. From the abstract interpretation point of view, ACDCL is an abstract interpreter that uses decision and learning to increase transformer precision. From the decision procedure perspective, ACDCL is a SAT solver for program analysis constraints and is thus a strict generalisation of propositional CDCL solvers. Constraint propagation in ACDCL uses fixed point iteration, decisions restrict the range of intervals and learning generates program analysis constraints (not assumptions) that preserve the error reachability.

Algorithm 1: Abstract Conflict Driven Clause Learning

```
   input  : Program 𝒫 with properties specified with assert(c)
   output: The status (safe or unsafe) and a counterexample if unsafe
 1 assign result=deduce (P)
 2 if result ==⊥ then
 3 |  return safe
 4 else
 5 |  if is_gamma_complete() then
 6 |  |  return unsafe
 7 while true do
 8 |  assign decision_variable = decision_heuristics(P)
 9 |  if decision_variable == NULL then
10 |  |  return unknown
11 |  else
12 |  |  assign result=deduce (P)
13 |  |  if result == UNKNOWN then
14 |  |  |  if is_gamma_complete() then
15 |  |  |  |  return unsafe
16 |  |  |  else
17 |  |  |  |  continue
18 |  |  else
19 |  |  |  do
20 |  |  |  |  if ¬ conflict_analysis() then
21 |  |  |  |  |  return safe
22 |  |  |  |  assign result=deduce (P)
23 |  |  |  while result ==⊥
24 end
```

Moreover, ACDCL implicitly perform program and property driven trace partitioning to increase the precision of the analysis. We observe that the abstract domains necessary for hardware property checking are – booleans, constants, interval, equality and octagons domains.

Algorithm 1 presents an overview of the ACDCL algorithm. Given a program $\mathcal{P}$ with properties specified as $assert(c)$, ACDCL terminate with *safe* if there exist no counterexample trace violating the assertion or *unsafe* otherwise. The procedure $deduce(P)$ is similar to BCP step in SAT solvers where it computes least fix-point with strongest post-condition using forward analysis. If the result of $deduce(P)$ is BOTTOM ($\bot$), the algorithm terminates with *safe*. Else, the *gamma completeness* check [8] is performed to determine if it is a real counterexample. If the *gamma completeness* check is successful, then the counterexample is real. Else, the algorithm enters into the while($true$) loop and heuristically picks a meet irreducible for decision. For example, assuming interval domain, decisions restrict the range of intervals for variables, so the analysis jumps under a greatest fixed-point. Note that the widening operation in abstract interpretation jumps above a least fixed-point, so decisions can be viewed as dual widening. The procedure $deduce(P)$ is called next to deduce new facts for current decision. The algorithm terminates with *unsafe* if the result of $deduce(P)$ is *gamma complete*.

Else, the algorithm enters in to $conflict\_analysis()$ phase to learn the reason for *conflict*. There can be multiple incomparable reasons for conflict – based on the choice of Unique Implication Point (UIP), ACDCL heuristically choose one. A learnt clause must include asserting cuts which guarantees derivation of new information after backtracking. The clause learning and backtracking continues as long as the result of deduction is BOTTOM ($\perp$) or the analysis backtracks to decision level 0. If no further backtrack is possible, then the algorithm terminates with *safe*. Else, the algorithm makes a new decision and the above process is repeated until a real counterexample is obtained or the algorithm backtracks to decision level 0 after a conflict in which case it returns *safe*. Currently, ACDCL handles loops in the program by unrolling the bounded loops. However, we are developing a new technique for automatic invariant generation using ACDCL for unbounded proofs.

References

1. C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX, 2008, pp. 209–224.
2. P. Cousot, "Proving the absence of run-time errors in safety-critical avionics code," in *EMSOFT*, 2007, pp. 7–9.
3. V. D'Silva, L. Haller, and D. Kroening, "Abstract conflict driven learning," in *Principles of Programming Languages (POPL)*. ACM, 2013, pp. 143–154.
4. R. Mukherjee, M. Tautschnig, and D. Kroening, "v2c – a Verilog to C translator," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS. Springer, 2016.
5. R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12.
6. R. Mukherjee, P. Schrammel, D. Kroening, and T. Melham, "Unbounded safety verification for hardware using software analyzers," in *DATE*, 2016.
7. R. Mukherjee, D. Kroening, T. Melham, and M. Srivas, "Equivalence checking using trace partitioning," in *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 13–18.
8. R. Giacobazzi and E. Quintarelli, "Incompleteness, counterexamples, and refinements in abstract model-checking," in *SAS*, 2001, pp. 356–373.