VCEGAR: Verilog CounterExample Guided Abstraction Refinement*

Himanshu Jain[1], Daniel Kroening[2], Natasha Sharygina[1,3], and Edmund Clarke[1]

[1] Carnegie Mellon University, School of Computer Science
[2] ETH Zuerich, Switzerland
[3] Informatics Department, University of Lugano

Abstract. As first step, most model checkers used in the hardware industry convert a high-level register transfer language (RTL) design into a netlist. However, algorithms that operate at the netlist level are unable to exploit the structure of the higher abstraction levels, and thus, are less scalable. The RTL level of a hardware description language such as Verilog is similar to a software program with special features for hardware design such as bit-vector arithmetic and concurrency. We describe a hardware model checking tool, VCEGAR, which performs verification at the RTL level using software verification techniques. It implements predicate abstraction and a refinement loop as used in software verification. The novel aspects are the generation of new word-level predicates, an efficient predicate image computation in presence of a large number of predicates, and precise modeling of the bit-vector semantics of hardware designs.

1 Introduction

Most new hardware designs are implemented at a high level of abstraction, e.g., using *register transfer language (RTL)*, or even at the system-level. The RTL level of a hardware description language such as Verilog is very similar to a software program in ANSI-C, and offers special features for hardware designers such as bit-vector arithmetic and concurrency. However, most model checkers used in hardware industry still operate on a low-level design representation called a *netlist*. This is due to lack of automated verification techniques at the RTL level. Converting a high-level RTL design into a netlist results in a significant loss of structure present at the RTL level. This makes verification at the netlist level inherently more difficult and less scalable.

VCEGAR, the tool presented in this paper, is a hardware model checker that performs verification at the RTL level directly. In order to reduce the state space explosion problem during model checking, VCEGAR performs abstraction. Abstraction techniques reduce the state space by mapping the set of states

* This research was sponsored by the Gigascale Systems Research Center (GSRC), Semiconductor Research Corporation (SRC), the National Science Foundation (NSF), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), the Defense Advanced Research Projects Agency, the Army Research Office (ARO), and the General Motors Collaborative Research Lab at CMU.

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 583–586, 2007.

of the actual, concrete system to an abstract, and smaller, set of states in a way that preserves the relevant behaviors of the system. Since high-level hardware designs are similar to concurrent software, it implements abstraction algorithms that have been devised for software verification. VCEGAR employs predicate abstraction [1], a key technique used in the SLAM software verification project [2]. Predicate abstraction removes data by only keeping track of certain predicates on the data. Each predicate is represented by a Boolean variable in the abstract model, while the original data paths are eliminated.

The abstract model is computed as a *conservative* over-approximation of the original circuit. This implies that if the abstraction satisfies the property, the property also holds on the original circuit. The drawback of the conservative abstraction is that when model checking of the abstraction fails, it may produce a counterexample that does not correspond to any concrete counterexample. This is usually called a *spurious counterexample.* The basic idea of abstraction refinement techniques [3,4,2] is to create a new abstract model that contains more detail in order to prevent the spurious counterexample. This process is iterated until the property is either proved or disproved. It is known as the *Counterexample Guided Abstraction Refinement* framework [4].

VCEGAR is geared towards application by hardware designers. It accepts Verilog, a popular hardware description language, as input. VCEGAR checks safety properties of the hardware designs.

Related Work. In the hardware domain, the most commonly used abstraction technique is *localization reduction* [3]. The abstract model is created from a given netlist level circuit by removing a large number of latches together with the logic required to compute their next state. During refinement, some of the removed latches may be added back to make the abstract model more precise. While localization reduction is a special case of predicate abstraction, predicate abstraction can result in a much smaller abstract model. As an example, assume a circuit contains two sets of latches, each encoding a number. Predicate abstraction can keep track of a numerical relation between the two numbers using a single predicate, and thus, using a single state bit in the abstract model. Localization reduction typically turns all bits of the two words into visible latches, and thus, the abstraction is identical to the original model.

Clarke et al. introduce a SAT-based technique for predicate abstraction of circuits given in Verilog [5]. The first step is to obtain predicates from the control flow guards in the Verilog file. The circuit is then synthesized into netlist level. Any refinement steps are carried out at the netlist level, new word-level predicates are never introduced. VCEGAR operates at the RTL level also during refinement and uses weakest pre-conditions to derive new word-level predicates.

2 Word-Level Circuit Verification with VCEGAR

This section provides a short overview of ideas implemented in VCEGAR. For more information, we refer the reader to [6,7]. The abstraction step in VCEGAR is performed by computing a predicate image. Two problems arise when applying predicate abstraction to RTL level circuits: 1) The computation of the abstract model is hard in presence of large number of predicates, and 2) discovery of suitable word-level predicates for abstraction refinement.

In order to address the first problem, the tool divides the set of predicates into *clusters* of related predicates. The abstraction is computed separately with respect to the predicates in each cluster. Since each cluster contains only a small number of predicates, the computation of the abstraction becomes more efficient. We refer to this technique as *predicate clustering.* We do not require the clusters to be disjoint, that is, they can have common predicates.

Example: Let x, y denote the current state and x', y' denote the next state of a hardware design. Let the transition relation $R(x, y, x', y')$ be $x' = y \wedge y' = x$. Let the set of predicates be $\{x = 1, y = 1, x' = 1, y' = 1\}$. The value of the predicate $y' = 1$ is affected by the value of x (as y' equals x). Note that the value of $y' = 1$ is not affected by the value of y. Thus, we keep $x = 1$ and $y' = 1$ together in a cluster C_1. Similarly, the other cluster $C_2 := \{y = 1, x' = 1\}$ is obtained.

The tool provides various options for predicate clustering. These options control the precision of the abstraction and the time required to compute the abstraction. The tool uses a SAT solver to compute the abstract model [8].

Due to predicate clustering, additional spurious counterexamples are introduced, which have to be removed during the refinement phase. When a spurious counterexample is encountered, we first check whether each transition in the counterexample can be simulated on the original program. This is done by creating a SAT instance for the simulation of each abstract transition. If the SAT instance for an abstract transition is unsatisfiable, then the abstract transition is spurious. In this case, we refine the abstraction by adding constraints on the abstract transition relation, which eliminate the spurious transition. We make use of the *unsatisfiable core* of the SAT instance to identify a small subset of the existing predicates that are causing the transition to be spurious. The fewer predicates are found, the more spurious transitions are eliminated in one step.

When all SAT instances for the simulation of abstract transitions are satisfiable, it means that none of the abstract transitions is spurious due to the predicate clustering. The immediate conclusion is that the spurious counterexample is caused by a lack of appropriate predicates. For this case, VCEGAR uses a refinement technique employed in software verification tools. It first determines a set of predicates that causes the simulation to fail. Subsequently, it computes the weakest precondition of these predicates with respect to the transition function given by the circuit in order to obtain new word-level predicates. *Example:* Let the property be $x < 3$, and the next state function for the register x be $((x < 5)?(x + 2) : x)$, where ? denotes a conditional operator. Suppose we obtain a spurious counterexample of length equal to 1. The weakest precondition *wp* of $x < 3$ is given as $(((x\ <\ 5)\ ?\ (x{+}2)\ :\ x\ )<\ 3)$. Refinement corresponds to adding the Boolean expressions occurring in *wp* to the existing set of predicates.

In case of a long spurious counterexample, the weakest precondition computation may become expensive due to a blowup in the size of weakest pre-conditions. We address this problem by applying a syntactic *simplification* to the weakest preconditions at each step. The simplification uses data from the abstract error trace. We exploit the fact that many of the control flow guards in the Verilog file are also present in the current set of predicates. The abstract trace assigns truth values to these predicates in each abstract state. In order to simplify the weakest preconditions, we substitute the guards in the weakest preconditions with their truth values. Furthermore, we only add the atomic predicates in the simplified weakest precondition as the new predicates (more details in [6]).

Example: Suppose the guard $x < 5$ is present in the current set of predicates. Let the value of $x < 5$ in an abstract state $\bar{s}$ be `true`. The weakest precondition given as $(((x \ < \ 5) \ ? \ (x+2) \ : \ x \ ) < \ 3)$ can be simplified in $\bar{s}$ by substituting the value of $x < 5$. This results in a new atomic predicate $x + 2 < 3$ (or $x < 1$).

VCEGAR was used to check safety properties of Instruction Cache Unit and Instruction Cache RAM (ICRAM) of Sun PicoJava II microprocessor in [7]. It has also been applied to examples from the opencores (`www.opencores.org`), and the Texas97 and VIS benchmark suites.

3 Conclusion

This paper describes a hardware model checker, VCEGAR, that implements counterexample guided abstraction and a refinement loop for RTL Verilog designs. It uses the idea of predicate abstraction from software verification tools. VCEGAR provides various options for balancing the precision of abstraction and the time required for abstraction computation. For abstraction-refinement new word-level predicates are discovered by computing syntactic weakest preconditions of predicates with respect to Verilog statements. This technique has not been applied to RTL circuits before. A user of the tool needs to provide the input program, property to check, and a few options. Given these inputs, the tool performs all the steps of the CEGAR loop automatically.

References

1. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV. Volume 1254 of LNCS., Springer (1997) 72–83
2. Ball, T., Rajamani, S.: Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research (2000)
3. Kurshan, R.: Computer-Aided Verification of Coordinating Processes. Princeton University Press, Princeton (1995)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
5. Clarke, E., Talupur, M., Wang, D.: SAT based predicate abstraction for hardware verification. In: SAT. (2003)
6. Clarke, E., Jain, H., Kroening, D.: Predicate Abstraction and Refinement Techniques for Verifying Verilog. Technical Report CMU-CS-04-139 (2004)
7. Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word level predicate abstraction and refinement for verifying RTL In: DAC. (2005) 445–450
8. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. Formal Methods in System Design:25 (2004) 105–127