Integrating Formal Verification in an Online Judge for e-Learning Logic Circuit Design

Javier de San Pedro Josep Carmona Jordi Cortadella Jordi Petit
Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya

ABSTRACT

This paper investigates the use of formal verification techniques to create online judges that can assist in teaching logic circuit design. Formal verification not only contributes to give an exact assessment about correctness, but also saves the instructor the tedious task of designing test cases. The paper explains how formal verification has been integrated in an online judge. It also describes the courseware created for a course on logic circuits and the successful experience of using it in a one-week summer course with students from secondary and high school.

Categories and Subject Descriptors

K.3.1 [**Computers and Education**]: Computer Uses in Education—*computer-assisted instruction*; B.6.3 [**Logic Design**]: Design Aids—*hardware description languages, verification*

General Terms

Algorithms, Design, Human factors

Keywords

Formal verification, logic circuit design, online judge, Verilog

1. INTRODUCTION

Most bachelor programs in Computer Science/Engineering include some course on designing logic circuits. At the time students take this course, they are usually familiar with basic programming concepts in some high-level language and have some knowledge about good programming practices: simplicity, cost, modularity, re-usability, etc. Most of their programming skills have been obtained by trying to solve many programming problems of increasing difficulty.

One of the most important decisions in creating a course syllabus on logic design is to choose the formalism to specify and describe circuits. In a similar way as students are exposed to programming by using high-level languages (e.g., C, C++, Java) and automatic tools (compilers) to translate a program into machine language, we strongly advocate for using a similar approach when designing circuits [17, 19]. This approach is not as disruptive as it looks when compared with the modern practice in industry today. Designers describe their circuits with *Hardware Description Languages* (HDL) and implement their circuits using CAD tools that automatically generate logic netlists [12]. The netlists are later transformed into layouts using physical synthesis tools. Indeed, as the time of assembly language passed long ago and common programming continuously evolves towards more abstract paradigms, the time of schematic capture has also passed and circuit design has evolved towards a more reliable and productive methodology based on HDLs.

Another important issue when creating a course on logic circuits is designing a set of problems that enables students to apply the theoretical concepts they learn to practical problems, and how to handle the assessment of these problems. Drawing a parallelism with our experience on computer programming, we advocate for the use of a collection of problems of increasing difficulty and the use of a online judge to automatically correct their solutions. Online judges have a variety of advantages that make them appealing for modern teaching methodologies: permanent and immediate assessment, adaptation to the progress of each student, more productivity, etc. See [5, 13] for an overview on online programming judges.

This paper introduces a novel online judge that uses formal verification to automatically correct problems on logic circuit design. Some features of this judge make it unique among the existing ones:

- Circuits are described using a HDL.
- Correctness is proved for all possible input stimuli by using formal verification techniques and tools.
- No test cases have to be designed to assess about correctness.
- When an incorrect circuit is submitted, the judge provides a counterexample trace for which the circuit behavior differs from the expected one.

These features have been implemented in Jutge.org, a open access educational online programming judge that already offers 800 programming problems [10] using several programming languages. Thanks to the new innovative correction engine, Jutge.org currently offers an introductory course of digital circuit design that includes a large variety of assignments on logic design, from simple combinational circuits to complex sequential controllers, all carefully organized and sorted by difficulty. About 60 new problems on circuits can now be solved in Verilog, other HDLs being under preparation.

To the best of our knowledge, it is the first time that formal verification is used to assess correctness with an online judge using an

HDL. An approach with similar goals was proposed in [8], where students can submit their design with LogicFlash [9], a graphic tool for schematic capture. Combinational circuits are checked by generating all possible input patterns. Sequential circuits are checked by generating a set of test patterns that cover all the state transitions of an automaton designed by the instructor. That approach suffers from various problems: (1) the cost of checking correctness becomes prohibitive for large circuits (e.g. a 32-bit adder), (2) many sequential circuits (e.g., counters, shift registers, sequential arithmetic circuits, register files, etc) cannot be practically described with automata and, (3) the coverage of all state transitions in an automaton does not guarantee correctness for all possible input traces. In contrast, the approach we present is able to handle circuits of medium complexity. For example, circuits that generate the Fibonacci series (8-bit numbers) or compute the greatest common divisor using Euclid's algorithm (6-bit numbers) have been proved (or disproved) to be correct. The verification of a simple 8-bit CPU with more than 30 internal flip-flops is also possible.

The paper is organized as follows. To start, as preliminaries, we overview HDLs. Afterwards, we present a guided tour that presents our courseware and its use. Then, we explain which formal verification techniques we apply to assess the correction of the candidate solutions and we show their implementation and integration in Jutge.org. Finally, we describe our experience using the system in a one-week summer course with talented high-school students, draw some conclusions, and advance ideas towards future work.

2. PRELIMINARIES: HDLS

The design, simulation, synthesis and verification of hardware devices can be accomplished by the use of a Hardware Description Language (HDL) [15].

The primitives of a HDL can describe the structure and behavior of logic circuits. In contrast to programming languages, HDLs incorporate three aspects that make them suitable for describing hardware:

1. *Notion of time*: the language provides statements to support time and delays of the components.
2. *Concurrency/Parallelism*: the semantics of the language is inherently concurrent, i.e., different components of the system are assumed to operate in parallel.
3. *Simulation capabilities*: circuit specifications can be validated by designing *testbenches*, i.e., modules that instantiate the design and monitor the reaction to a set of input stimuli. This capability contributes to gain confidence on the design's intended function, but also enables *architectural exploration*, i.e., evaluating and comparing different implementations of the same circuit.

Verilog [18] and VHDL [3] are the most popular HDLs used in logic circuit design. Verilog has been chosen for the on-line judge, however an extension to VHDL would also be feasible by using the appropriate synthesis tools.

Verilog. The primitives provided by Verilog allow different descriptions of the functionality of a circuit. *Structural* models are used to described hierarchical netlists of logic gates or instances of other modules. *Dataflow* models are used to describe combinational logic by means of different types of operators (logical, arithmetic, relational, bit-wise, etc). Finally, *behavioral* models are used to describe the functionality using primitives similar to those provided in programming languages (conditional, looping, concurrency, etc).

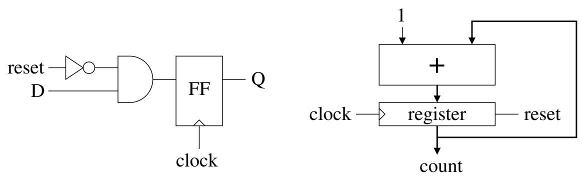
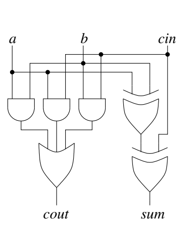


Figure 1: Circuits inferred by the description of a flip-flop and an *n*-bit counter with synchronous reset.

The syntax of Verilog is similar to the C programming language, also very popular both in education and industry. A Verilog design consists of a hierarchy of modules, where modules may communicate through input and output ports. A module can contain a combination of the following: net/variable declarations, concurrent and sequential statement blocks, and instances of other modules.

Below we can see a structural description of a full adder. The circuit has three inputs (*a*, *b* and *cin*) and two outputs (*sum* and *cout*) and contains a set of logic gates (*xor*, *and*, *or*) connected to the inputs, outputs and wires.



```
module full_adder (a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;
  wire x1, x2, x3;
  xor (sum, a, b, cin);
  and (x1, a, b);
  and (x2, a, cin);
  and (x3, b, cin);
  or (cout, x1, x2, x3);
endmodule
```

An implementation with the same behavior can be described with a dataflow model using logic equations, as shown below.

```
module full_adder (a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;
  assign sum = a ^ b ^ cin;
  assign cout = a & b | cin & (a | b);
endmodule
```

For sequential systems, Verilog provides the *always* statement, commonly used to model the behavior of memory elements triggered by signal events. This is the statement typically used to describe flip-flops and latches in sequential circuits. For example, the next two examples illustrate how the behavior of a flip-flop and an *n*-bit counter with synchronous reset can be modeled using the *always* statement.

```
// flip-flop behavior              // counter behavior
always @(posedge clock)            always @(posedge clock)
  if ( reset ) Q ⇐ 0;                if ( reset ) count ⇐ 0;
  else Q ⇐ D;                        else count ⇐ count + 1;
```

A Verilog compiler would automatically infer the logic gates and flip-flops to implement the behavior described above. Possible netlists derived by the compiler are shown in Fig. 1. The circuits could be later simplified by CAD tools that would automatically propagate the constants at the inputs and eliminate the redundant logic gates.

Synthesizable HDL. Both Verilog and VHDL have language subsets declared as *synthesizable* [2], meaning that CAD tools can automatically infer circuit structures that implement the specified functionality. Typically, the synthesizable subset includes the structural and dataflow models of the language. Whereas some complex operators (such as multiplication or division) may not always be included, addition/subtraction, relational and bit-wise operators are always included. The synthesizable subset is the one used by the students and the online judge to design and verify the circuits.

Besides various commercial tools, there are several open-source Verilog compilers/simulators. We have used Icarus Verilog [20].

3. GUIDED TOUR

This section offers a guided tour to the course on logic circuit design and the new judge engine that we have created. We encourage the reader to follow these steps in his/her web browser.

The course is available at Jutge.org, an open access educational online programming judge [10]. In order to access it, users can log in at https://www.jutge.org using the demo account or freely registering their own account. Afterwards, users should enroll the "*Introduction to Digital Circuit Design*" course in the Courses section. At this point, users can access its 60 problems (identified with a short hash code) and organized in the following lists:

Combinational circuits: Simple controllers, multiplexers, voting systems, priority encoders, properties on numbers, ...

Arithmetic circuits: Starts from 1-bit adders/comparators and progresses to *n*-bit adder/subtractors, comparators, incrementers and small ALUs.

Sequential circuits: Includes various up/down counters, sequence recognizers, simple control circuits (e.g., a traffic-light controller) and some sequential arithmetic (e.g., a generator of Fibonacci series or a calculator of the GCD using Euclid's algorithm).

A simple CPU: The design is split in four components (ALU, datapath, program counter and control unit) that are finally connected to create a complete 8-bit CPU with small instruction/data memories.

As an example, consider problem X12983 (the second problem in the Arithmetic Circuits list), which asks for the implementation of a full adder.

To solve this problem, a student could write an implementation either in structural form (using logic gates) or in dataflow form (using logic equations), such as the ones presented in the previous section for the module *full_adder*. In either case, these descriptions can be automatically synthesized into circuits that can be later verified for correctness.

To validate the implementation, our student would submit this piece of code to the judge, which would produce a verdict in a few seconds. Typical verdicts include "*Compilation error*", "*Accepted*", or "*Wrong answer*". In the case of the latest, the judge would also offer a set of values for the input ports that causes such wrong answer.

In order to test the correctness of a student's implementation, the problem setter has provided a *golden model* that will be never be disclosed to the students. This model can be at any level of abstraction, e.g., using arithmetic operators. For instance, this is the actual golden model provided for the full adder problem:

```
module full_adder (a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;
  assign {cout, sum} = a + b + cin;
endmodule
```

After solving the previous problem, our student could tackle problem X84292, which asks for an *N*-bit adder. A possible implementation could use an array of *N* instances of full adders:

```
module adder (A, B, cin, SUM, cout);
  parameter N = 16;
  input [N−1:0] A, B;
  input cin;
  output [N−1:0] SUM;
  output cout;
  wire [N−1:1] carry;
  full_adder FA[N−1:0] (A, B, {carry,cin}, SUM, {cout,carry});
endmodule
```

The flow is similar for sequential circuits. For example, the problem X05944 proposes the design of a (*mod 3*)-counter that generates the sequence $0, 1, 2, 0, 1, 2, \cdots$. The golden model implements the counter using a behavioral conditional statement:

```
module mod3_counter (count, clk, rst);
  input clk, rst;
  output reg [1:0] count;
  always @(posedge clk)
    if (rst | (count == 2)) count ⇐ 0;
    else count ⇐ count + 1;
endmodule
```

Let us assume the student misunderstands the specification of the problem and implements a conventional (*mod 4*)-counter with a 2-bit adder and a register as follows:

```
module mod3_counter(count, clk, rst);
  input clk, rst;
  output [1:0] count;
  wire [1:0] next_count;
  wire cout;
  adder #(2) _add_ (count, 1, 0, next_count, cout);
  register #(2) _reg_ (next_count, count, clk, rst);
endmodule
```

In this case, the judge will report a *"Wrong answer"*. More interestingly, the judge will also provide *the shortest trace* from the initial state to an erroneous state. This trace will be reported in textual and graphical form, as shown in Fig. 2, where the circle indicates the unexpected value according to the specification of the problem.

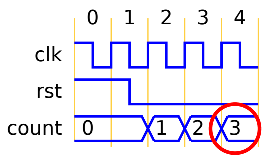


Figure 2: Waveform report for the mod3_counter circuit.

In the next two sections a detailed explanation on the underlying theory and techniques is provided for the interested reader.

4. FORMAL VERIFICATION

Testing shows the presence, not the absence of bugs.
E.W. Dijkstra.

Although simulation may provide insights on the correctness of a circuit, one cannot have an absolute guarantee of its correctness unless the circuit is exercised with *all possible input stimuli*. This requirement makes simulation inviable for the verification of circuits with medium/large complexity.

In the last three decades, there has been a significant progress in Boolean reasoning methods for equivalence checking. Two main techniques have been proposed to symbolically enumerate all possible states of a circuit: Binary Decision Diagrams (BDD) [4] and SAT solvers [11]. Still, the methods for the equivalence checking of sequential machines can only handle small circuits (e.g., few dozens of state variables) when the internal sequential elements of

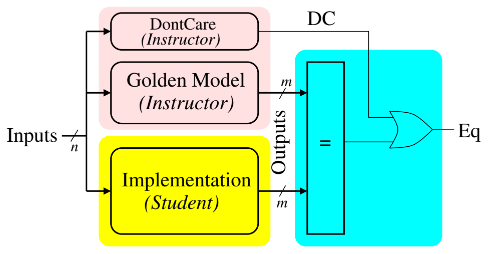


Figure 3: Miter created to verify sequential equivalence.

the two circuits (golden model and implementation) are different. When the sequential structure is the same, the problem can be reduced to combinational equivalence checking [14].

Unfortunately, the problem we are facing needs to prove, in general, the equivalence of sequential machines that are internally different, since the states variables of the golden model may differ from the ones used by the student. They may have different number of flip-flops, state encoding and initial state.

Golden model vs. implementation. The equivalence of the two circuits is checked by creating a *miter*, as shown in Fig. 3. A miter is a circuit that is derived from the two original circuits, inserting a comparator to compare the outputs of the two circuits. The circuits are equivalent when the output of the miter, *Eq*, is proved to be a tautology (always at 1).

In case of combinational circuits, equivalence is reduced to a satisfiability problem, i.e., finding an assignment that contradicts *Eq*. If it does not exist, the circuits are equivalent.

In case of sequential circuits, all the reachable states of the miter must be visited using all the possible input stimuli. This is a much costly task and symbolic techniques (e.g., using BDDs) are often used to mitigate the state explosion problem.

In our case, we have chosen an open-source tool that has become state-of-the-art in the verification community: *NuSMV* [6]. NuSMV is a symbolic model checker [7] that can verify circuit properties specified in temporal logic [16]. For the verification of sequential equivalence, NuSMV receives the miter circuit and the invariant property that "*Eq must always be true*".

Don't cares. In different situations, equivalence does not need to be strict. For example, before the two circuits are reset, the observed behavior may differ. Two circuits transferring data do not need to show the same output while the "*valid*" bit is not asserted.

For such cases, the miter can be enhanced with an auxiliary circuit (DontCare in Fig. 3) that masks the equivalence of the outputs when they do not need to be equivalent. The DontCare circuit must be provided by the instructor together with the golden model. More specifically, the *don't care* conditions (DC) can be specified for each individual output of the circuit (not shown in the figure).

Error traces. Under the unsatisfactory case that the implementation is not equivalent to the golden model, formal verification tools can provide a valuable feedback to the student: a counterexample that disproves the equivalence of the circuits. In case of a combinational circuit, the counterexample is simply an assignment to the inputs for which the outputs differ. In case of sequential circuits, the tool can report a trace of input stimuli that drives the erroneous circuit from the initial state to the state that produces the unexpected output. Typically, sequential verifiers are capable of generating the shortest trace to an error condition.

MODULE *example*(*a*)
VAR *r*: *boolean*;
ASSIGN **next**(*r*) := !(*a* | *r*);

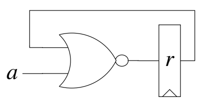


Figure 4: NuSMV model for a simple circuit.

5. VERIFICATION PROCESS

5.1 Overall flow

Verification starts by first synthesizing the Verilog models. For this, we use Icarus Verilog [20], which transforms the model into an internal hierarchical representation. Icarus Verilog can also be customized with plugins that can access the internal representation and generate other models. This is the strategy we have used to generate NuSMV models from Verilog.

Once the model provided by the student has been transformed into a NuSMV module, it is combined with the golden model provided by the instructor to create the miter circuit shown in Fig. 3. We next provide more details on the verification flow.

5.2 Conversion to NuSMV model

NuSMV has a modeling language for finite systems that can describe the sequential behavior of Boolean signals. Figure 4 depicts a simple example of a circuit with one input signal (*a*) and one output signal (*r*), where the output signal is also a state variable (declaration of `VAR`). The `ASSIGN` statement describes the behavior of each signal assigning a Boolean expression. When the behavior is sequential, `next(`*r*`)` is used to denote the value of the signal at the *next* cycle. More details about the NuSMV language can be found in [6, 1].

The circuit internal representation of Icarus Verilog is a hypergraph, where the nodes represent the synthesized components and the hyperedges represent the wires connecting the components.

The nodes can represent logic gates or special modules from a *Library of Parametrized Modules* (LPMs) performing higher level functions (e.g., integer addition, register, ...). The LPMs are assumed to be implemented with pre-designed logic blocks for each target technology.

Let us illustrate the conversion to NuSMV with a the example below, that models a counter with a control signal (*inc*) that indicates when the value of the counter must be incremented.

```
module counter(clk, rst, inc, count);
  input clk, rst, inc;
  output reg [1:0] count;
  always @(posedge clk)
    if (rst) count <= 0;
    else if (inc) count <= count + 1;
endmodule
```

Icarus Verilog will produce an internal representation as the one shown in Fig. 5. The ports from the original module are represented by ellipsis. Additionally, two LPMs have been synthesized: a register and an adder. The hyperedges are represented by the circles between nodes.

This representation corresponds to the schematic shown in Fig. 6 with an *N*-bit counter and one additional connection of the *inc* port to the Chip Enable (CE) port of the register.

By traversing the graph backward from the output ports and state variables, it is possible to reach the driver of each signal in the circuit and derive the expressions that determine their behavior.

The NuSMV model is generated by traversing the graph and using the following conversion rules:

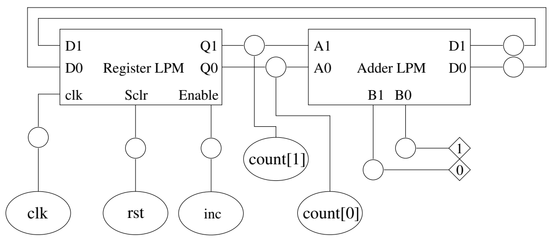


Figure 5: Icarus internal graph for the counter circuit.

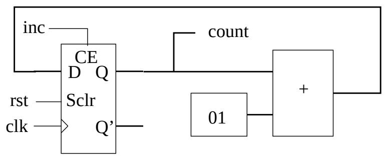


Figure 6: Schematic for the counter circuit.

1. Verilog modules are converted into NuSMV modules and input ports are converted into module parameters.
2. Instances inside a module are converted into NuSMV instances, and the real argument expressions that are passed to those instances are generated by graph traversal.
3. Output ports are converted into `define` statements, and their expressions are calculated by graph traversal. Other generated expressions can then refer to these definitions as if they were NuSMV output parameters.

By using the previous transformation rules, the following NuSMV model is obtained for the counter:

```
MODULE counter(inc, rst)
VAR
    ff: lpm_register (add.Q, inc , rst );
    add: lpm_adder(ff.Q, 0b2_01);
DEFINE
    count := ff.Q;
```

5.3 Generation of the miter

Once both NuSMV models have been generated, the miter module is generated by adding a comparator of all the output ports. The miter is the top NuSMV module that will be subject to verification.

At this stage, the property for verification is also generated. The property is an invariant that guarantees that *Eq* is always asserted after the circuit has been reset.

LPM definitions are also included as NuSMV modules. These definitions are instanced from a manually created set of NuSMV modules for each of the possible LPMs. For example, the definition of the *lpm_register* with support for synchronous clear and chip enable signals (like the one in Fig. 5) is the following:

```
MODULE lpm_register(Data, Sclr, Enable)
VAR     Q : word[< register_size >];
ASSIGN  next(Q) := case
    Sclr  :  0;      -- If clear is raised
    Enable : Data;   -- If chip enabled
    TRUE : Q;        -- Otherwise, keep current value
  esac;
```

An important observation is that the NuSMV models ignore the clock signal. The reason for that is because the circuits to be verified are assumed to have a single clock that triggers all sequential elements on the rising edge. In this way, the NuSMV model can safely assume that the implicit clock of the system corresponds to the common clock used in the Verilog models.

5.4 Verification and error diagnosis

After executing the top model with the miter, NuSMV gives an answer stating whether the requested property is asserted. In case it is not asserted, NuSMV also emits an error diagnosis by providing a counterexample trace of the input stimuli that drive the circuit from the initial state to another state in which the outputs differ.

Since the structure of the NuSMV model is isomorphic to that of the original circuit, the system can easily generate an equivalent trace in Verilog-like syntax and a graphical representation as a waveform, as shown in Fig. 2.

5.5 Integration in Jutge.org

The verification process previously presented has been integrated in Jutge.org. Internally, Jutge.org is designed as a distributed master-slave system, with a master computer that offers the web server, stores the database and keeps the submission queue, and several slave servers that host the different virtual machines that perform the actual correction tasks under a secure environment [10]. In order to correct a submission, a slave machine receives the problem description (including the golden model), the candidate solution and a driver module, which guides the compilation, execution and checking process to produce a verdict.

Thanks to the modular architecture of Jutge.org, extending the system to accommodate problems on circuits has been an easy task. On the back-end (subsystem that corrects submissions), it was only needed to create a new driver to guide the correction of problems on circuits. This driver implements all of the validation steps as detailed in this section. On the front-end (web interface), it was just necessary to add some features to offer a nice display of the information computed together with the verdict as, for instance, the error traces and waveforms.

6. A SUMMER COURSE ON CIRCUITS

The courseware we have described was first used in an intensive, one-week summer course (7 hours/day) on logic circuit design during July 2011. There were 26 students from secondary and high-school (mostly 16/17-year old) that had no previous knowledge about circuit design. The course was sponsored by the Joves i Ciència program of CatalunyaCaixa, whose mission is to enhance the scientific activity of young students with interest, motivation and talent for science.

The main goal of the course was to expose the students to logic design and to exercise their capabilities of logic reasoning by posing challenging problems. Given the origin of the students, there was no special intention of teaching a particular methodology for circuit design, although some common strategies were introduced during the lectures to help them in solving exercises. Except for short periods of time in which some expository lectures were taught (not longer than one hour per day), the students were spending most of the time solving problems on their computers and testing them with the online judge.

Few of the students managed to complete the CPU, whereas most of the students could complete several sequential circuits of different complexity. Interestingly, a few students also worked at night from home with the online judge, even after such an exhausting working day.

Students were free to select their favorite exercises depending on their skills and confidence of success. At the end of the course, 3 students could solve more than 50 exercises, 3 students between 40 and 49 exercises, 13 students between 30 and 39, and 7 students between 20 and 29.

The online judge was essential to achieve a high design productivity that would not have been possible with only the assistance of the 2 or 3 instructors that were permanently in the lecture room. The bandwidth provided by the judge and the assessment delivered by the formal verification tool (counterexamples in erroneous circuits) contributed to have an environment that could continuously help the students without the need of an instructor.

Schematics were rarely used during the course, mostly on the first day to provide the intuitive sense that the components of a circuit are connected with wires. After that, some block diagrams were shown when necessary. The logic gates were introduced on the first day. After a couple of simple exercises, combinational logic was always specified with Boolean equations.

At the end of the course, the students filled up a survey and showed a very high satisfaction with the use of Verilog and the online judge for learning logic circuit design.

On the negative side, we observed that the feedback provided by the judge was so informative that the students were too eager to submit the solution to the judge even before simulating and validating the correctness. While this aspect was not important for the main goals of the course (logic reasoning), it should be taken into account for Bachelor's courses, in which a design methodology including simulation should be stimulated.

7. CONCLUSIONS

While using formal verification to fully prove software correctness is still a utopia, it is becoming a common successful practice in hardware design. In this paper we have shown that it is possible to build an introductory course on digital circuit design around a collection of problems that are automatically corrected by an online judge without test cases. The introduction of formal verification methods in online judges provides a novel paradigm that contributes to broaden the use of such tools in particular and of e-learning techniques in general.

For introductory courses, in which the complexity of the circuits is still small, formal verification can be effectively used to handle a large variety of design examples. This paradigm starts hitting the state explosion wall when the number of state variables of the circuit increases. In these cases, divide-and-conquer schemes to design and verify smaller components of the system can still be used. However, we will have to resort to more conventional approaches based on the simulation of test cases when formal verification cannot go that far, which is the classical approach used today in online judges for programming courses and contests.

The time required by an instructor to write a problem on circuits with formal verification is much less than the time to write programming problems. In both cases the correct solution must be written, but in the former case, there is no need to write an extensive test set as in the latter.

The experience in using this courseware in a one-week intensive summer course with talented high-school students has shown a remarkable productivity in designing different types of circuits with growing complexity, from simple combinational circuits to complex state machines, including a small CPU.

As future work, we are planning to enhance the online judge with the evaluation of different metrics that can assess on the quality and/or suitability of the solution. The number of logic gates or the logic depth of the circuit can be calculated to give a quality metric on complexity and delay. This metric can be compared with the one of the golden model and provide feedback to the student in case a low-quality design has been submitted. For example, if a student designs a slow ripple carry adder when a fast carry-lookahead adder was requested, the judge would report that a circuit with excessive logic depth has been submitted.

8. REFERENCES

[1] NuSMV v2.5 User Manual. http://nusmv.fbk.eu.

[2] J. Bhasker. *Verilog HDL Synthesis: A Practical Primer*. Star Galaxy Publishing, 1998.

[3] J. Bhasker. *A VHDL Primer*. Prentice Hall, third edition, 1999.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[5] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[6] A. Cimatti et al. NuSMV2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[8] M. Damm, F. Bauer, and G. Zucker. Solving Digital Logic Assignments with Automatic Verification in SCORM Modules. In *International Conference on Interactive Computer-Aided Learning (ICL)*, pages 359–363, 2009.

[9] M. Damm, B. Klauer, and K. Waldschmidt. LogiFlash - A Flash-based Logic-Simulator for Educational Purposes. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications*, pages 748–750, 2003.

[10] O. Giménez, J. Petit, and S. Roura. Jutge.org. Technical report. http://www.jutge.org/documentation/jutge.pdf,, 2011.

[11] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 114–121, 2001.

[12] D. Jansen. *The Electronic Design Automation Handbook*. Kluwer Academic Publishers, 2003.

[13] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3), 2005.

[14] A. Kuehlmann and C. A. J. van Eijk. *Combinational and sequential equivalence checking*, pages 343–372. Kluwer Academic Publishers, 2002.

[15] J. P. Mermet. *Fundamentals and Standards in Hardware Description Languages*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[16] A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.

[17] A. Sagahyroon and M. Massoumi. On the use of hardware description languages in teaching VLSI design courses. In *Proceedings of the 26th Annual Frontiers in Education (FIE)*, volume 2, pages 713–716, 1996.

[18] D. E. Thomas and P. Moorby. *The Verilog hardware description language*. Kluwer, third edition, 1996.

[19] Z. Vranesic and S. Brown. Use of HDLs in teaching of computer hardware courses. In *Workshop on Computer Architecture Education (WCAE)*, 2003.

[20] S. Williams. Icarus Verilog. http://iverilog.icarus.com.