Faculty of Engineering and Computer Science
Expectations of Originality

This form sets out the requirements for originality for work submitted by students in the Faculty of Engineering and Computer Science. Submissions such as assignments, lab reports, project reports, computer programs and take-home exams must conform to the requirements stated on this form and to the Academic Code of Conduct. The course outline may stipulate additional requirements for the course.

1. Your submissions must be your own original work. Group submissions must be the original work of the students in the group.
2. Direct quotations must not exceed 5% of the content of a report, must be enclosed in quotation marks, and must be attributed to the source by a numerical reference citation[1]. Note that engineering reports rarely contain direct quotations.
3. Material paraphrased or taken from a source must be attributed to the source by a numerical reference citation.
4. Text that is inserted from a web site must be enclosed in quotation marks and attributed to the web site by numerical reference citation.
5. Drawings, diagrams, photos, maps or other visual material taken from a source must be attributed to that source by a numerical reference citation.
6. No part of any assignment, lab report or project report submitted for this course can be submitted for any other course.
7. In preparing your submissions, the work of other past or present students cannot be consulted, used, copied, paraphrased or relied upon in any manner whatsoever.
8. Your submissions must consist entirely of your own or your group's ideas, observations, calculations, information and conclusions, except for statements attributed to sources by numerical citation.
9. Your submissions cannot be edited or revised by any other student.
10. For lab reports, the data must be obtained from your own or your lab group's experimental work.
11. For software, the code must be composed by you or by the group submitting the work, except for code that is attributed to its sources by numerical reference.

You must write one of the following statements on each piece of work that you submit:
For individual work: **"I certify that this submission is my original work and meets the Faculty's Expectations of Originality",** with your signature, I.D. #, and the date.
For group work: **"We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality",** with the signatures and I.D. #s of all the team members and the date.

A signed copy of this form must be submitted to the instructor at the beginning of the semester in each course.

I certify that I have read the requirements set out on this form, and that I am aware of these requirements. I certify that all the work I will submit for this course will comply with these requirements and with additional requirements stated in the course outline.

Course Number:	COEN 6551 (Formal Hardware Verification)	Instructor:	Dr.S.Tahar
Name:	Thenmozhie Rajan	I.D. #	40192527
Signature:	[illegible]	Date:	09-05-2022

[1] Rules for reference citation can be found in "Form and Style" by Patrich MacDonagh and Jack Bordan, fourth edition, May, 2000, available at http://www.encs.concordia.ca/scs/Forms/Form&Style.pdf.
Approved by the ENCS Faculty Council February 10, 2012

FORMAL HARDWARE VERIFICATION TOOL AND TECHNIQUES : A SURVEY

COEN 6551

Thenmozhie Rajan
ID: 40192527
t_rajan@encs.concordia.ca

Abstract:
Technical systems are progressively influencing the world. Several crucial aspects of our lives even rely on them. Digital circuits, such as microprocessors or integrated circuits, plays a critical part in majority of systems. To avoid a system failure, we must either prove that the system is fault-free or fault tolerant or fail safe. System design process is meticulously analyzed to see if the proposed system will behave as expected. Formal Hardware Verification is a new technique for evaluating the functional correctness of designs at various stages of development. This report delves deeper into this subject and provides the details about various supporting tools and techniques.

I. Introduction

Simulation-based verification and formal based verification are two commonly employed logic verification methodologies in the verification area. Simulation-based verification is a conventional verification approach that involves manually examining the correctness of the outputs matching to the inputs after propagating a set of input test vectors through the design. Simulated outcomes are used to verify design accuracy.

Formal verification, on the other hand, does not require any input vectors and, when applied appropriately, ensures that the design is accurate [4]. Formal verification is a mathematical proof approach for confirming that the implementation of a design follows its specifications [2]. The essential principle behind system analysis is to develop a mathematical model of the system and then use logical or mathematical reasoning to verify that the required attributes hold true for this model [3]. If the time and effort spent on formal verification is less than the time and effort saved by identifying challenging flaws lot earlier, formal verification is a gain, regardless of whether we can claim that the system is accurate. This cost-benefit analysis supports automated and simple-to-use formal verification procedures, even if they lack theoretical expressiveness. This report focuses various effective formal hardware verification methodologies.

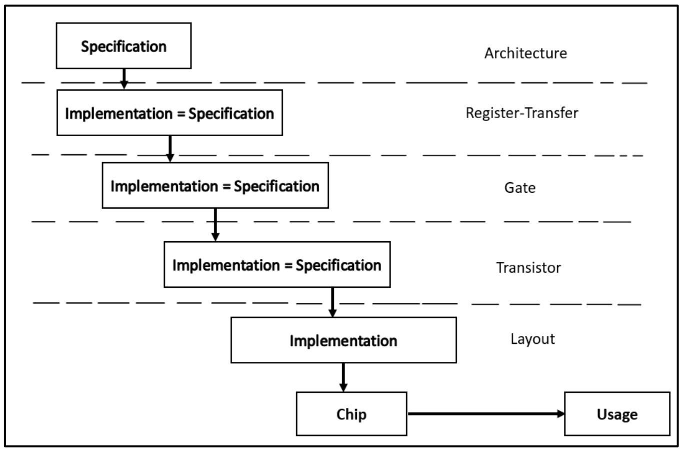
II. History of formal verification

As traditional simulation-based testing has failed to cope with surging design complexity, solutions based on formal verification approaches have silently grown in the background. A mathematical proof is a valid alternative to simulation for stating propositions about the whole behavior of a circuit. "Program testing can be used to show the presence of bugs, but never to show their absence," quoted Edsger Wybe Dijkstra. He urged the design community to think differently in April 1970. Even though Dijkstra's comment was delivered in the context of software verification, his call to action has had a considerably broader impact [5]. Martin Davis created the first computer-generated mathematical proof for a theorem in Presburger arithmetic, a decidable portion of first-order logic. The theorem stated that the product of two even integers is also an even number. Both the system model and the specification must be written in a precise mathematical language to solve a problem algorithmically. The concept was devised autonomously by Clarke and Emerson and by Queille and Sifakis in the early 1980s. Propositional temporal logic is commonly used to write specifications.

III. Design Cycle

The design process includes several parts, the first of which is formal specification, which involves transforming an informal design statement into a formal specification (language with mathematically defined syntax and semantics to describe the intended behavior of the system). Second, an implementation is developed from a specification by gradually adding information (i.e., an implementation expressed as a formula or a semantic model).

A real-world design flow is naturally more complicated and incorporates cycles, as design restrictions on a specific level are usually not satisfied, requiring the prior phase to be repeated with several other parameters. Different sorts of errors arise based on the stage of the design cycle.


Specification
Architecture
Implementation = Specification
Register-Transfer
Implementation = Specification
Gate
Implementation = Specification
Transistor
Implementation
Layout
Chip
Usage

Design Process

Debugging for formal verification grew in conjunction with new formal verification techniques that allow for significantly more automation than previously allowed. The BDD-based algorithms are the one under consideration. The binary decision diagram (BDD) is a data structure for describing Boolean functions. The BDD for a Boolean function may be constructed in the following way. First, construct a decision tree for the desired function, keeping in mind that no variable may exist more than once through any path from root to leaf, and that the variables must always appear in the same order [6]. Some automatic proof techniques developed for the bit-level are like BDDS or SAT (Satisfiability) solvers, and for the word-level are WLDDs or SMT (Satisfiability modulo theories) solver. The following are some formal hardware verification methodologies which makes use of some above-mentioned algorithms.

IV. Classification Of Formal Verification Methods

1. Equivalence Checking

Equivalence checking is important for ensuring that a design's function has not altered after an operation such as synthesis or the application of a functional Engineering Change Order. This technology uses mathematical modelling tools to demonstrate that two design representations behave similarly.

1.1. Combinational Equivalence Checking (CEC)

The most apparent use of BDDs in digital integrated circuit design is to assure design accuracy after logic synthesis and technology-dependent optimization. In today's electronic design automation (EDA) systems, fast and scalable solutions for combinational equivalence checking (CEC) are significant. CEC is used to prove structural equivalence of logic designs. Scalable CEC approaches are very beneficial in various important logic synthetic methods that rely on efficient computation of internal circuit node equivalence classes. ABC, an industrial-strength academic tool for logic synthesis and formal verification [7], required more than 24 hours to check equivalence, showing that proving equivalence gets more difficult as the design size grows.

Technology: Parallelism

There is a requirement for massively parallel EDA tools operating on the cloud to allow the next phase of technological innovation. Algorithms in a typical EDA environment operate on a sparse graph that represents the circuit. Due to the irregular structure of graphs represented using pointer-based data structures, exploiting parallelism of graph-based algorithms is generally hard. By using graph (mitre) partitioning to balance data sharing and data independence during SAT solving, parallelism for CEC could be accomplished. It is based on ABC &cec engine's state-of-the-art CEC engine, which makes use of the synergy between logic simulation and Boolean satisfiability (SAT) [7]. By using graph partitioning during mitering and SAT sweeping, these models trade off data sharing and data independence. This can increase the runtime and scalability of other applications [7].

Technology: Exploiting inherent characteristics of reversible circuits for faster CEC

Similarly, Checking the combinational equivalence of two reversible circuits is a significant but difficult (coNP-complete) problem. By using the intrinsic qualities of reversible circuits, such as bi-directional execution and XOR-richness, it is possible to achieve average speeds of more than one order of magnitude [8]. This significant advancement in the verification of reversible computation circuit descriptions also has the potential to be used in the verification of conventional designs.

1.2. Sequential Equivalence Checking (SEC)

Sequential equivalency checking is used to evaluate if two circuits under test are functionally equivalent. two sequential circuits are said to be similar [9] I If and only if no input sequences differentiate their respective outputs from their original states. The need that two designs have corresponding flip-flops is a key constraint of CEC. Synthesis tools frequently perform sequential optimizations on complicated designs, destroying the flip-flop correspondence in both the reference and implementation designs. Retiming, sequential redundancy elimination, and other improvements are examples [9]. Sequential equivalence checking should be used to check the equivalence of such designs (SEC).

Technology: Boolean Satisfiability

Based on invariant extraction, a SAT-based SEC methodology aimed at tackling difficult-to-verify tasks [9]. The SEC typically wastes time validating inappropriate assignments. The search space can be considerably reduced by restricting produced invariants via an upgraded invariant checking approach that uses timeframe-expansion and dynamic candidate selection procedures to extract some invariants in the miter circuit in advance. By finding dominant invariant candidates, the technique incorporates invariant checking and sequential SAT alternatively, and it may solve several difficult-to-detect problems with large solution lengths that are often unsuccessful when a limited model checking tool is employed [9]. This method has the ability to verify large-scale circuits sequentially.

Technology: Sequential Quantum Circuits-Hilbert space quantum mechanics

The quantum state machine, which is a natural quantum version of Mealy machines, was used to establish a formal framework for equivalence verification of sequential quantum circuits. The fact that quantum circuits' state spaces are continuums making verifying them extremely challenging. This problem was addressed by the main theorem, which showed that equivalence checking of two quantum Mealy machines can be done using input sequences chosen from a finite basis and having a length quadratic in the dimensions of the state Hilbert spaces of the machines. Based on this theoretical finding, an algorithm for testing the equivalence of sequential quantum circuits was created with a running time of $O\ (2^{3m}+5l\ (2^{3m} + 2^{3l}))$, where m and l represent the input and internal qubit counts, respectively. In the sense that both are exponential in the number of (qu)bits, the complexity of this method is comparable with the existing algorithms for checking classical sequential circuits [10].

A. Tools for Equivalence Checking

Supplier	**Tool Name**	**Design Level**	**Web Pointer**
Siemens	FormalPro	RTL/Gate	**Link**
Siemens	Questa SLEC	RTL	**Link**
Cadence	Conformal Equivalence Checker	RTL	**Link**
Cadence	Jasper SEC App	RTL	**Link**
Synopsys	Formality	RTL/Gate	**Link**
Synopsys	VC Formal	RTL	**Link**
Onespin Solution	360 EC- RTL	RTL/Gate	**Link**

2. Model Checking

Model checking allows us to ensure that a state machine adheres to a property specified using temporal logic [6]. Temporal logic is a model to describe features that vary over time. There are many kinds of temporal logic; Linear temporal logic (LTL), CTL (Computation Tree Logic), Computational Tree Logic Star (CTL*), Binary Decision Diagram are some of them. Model checking has the advantage of giving counterexamples when properties are not met. Formal modelling provides and evaluates the model's requirements. These methods are becoming increasingly common in industrial automation.

2.1. Symbolic Model Checking

Symbolic model checking uses BDDs in the model checking algorithm. For behavioral modelling of a system, symbolic model checking is utilized. States, events, and actions are used to represent the system. The framework for modelling systems, specification language, and verification technique are the three essential components of model checking [11]. The automated, model-based, and property-verification of a system are the essential features of model checking.

To construct a symbolic model checking algorithm [12], there are two steps: one is to design an algorithm based on operations on kinds of conditions, and the other is to represent sets of situations with propositional formulas and to perform all operations on sets with operations on formulas. One of the challenges with this verification method is that a model may contain an indefinite time-tree. We can have a finite representation for infinite time-trees by assuming that they have regular properties. Then we can convert a conceivable system structure to a symbolic model, allowing us to manipulate scenarios using Boolean formulae that express set of similar situations.

2.2. Runtime Probabilistic Model Checking

Many computer systems will have dynamic changes throughout their lifetime [13], which generally occur in the structure and components of the computer system and resulting in the addition, deletion, or alteration of its components, demanding runtime verification. The purpose of runtime probabilistic model testing [13] is to see if a system with random behavior matches the characteristics of the system at runtime.

The system models vary at various periods while monitoring the system. The model checker needs to constantly check freshly created models to see if the current model fits the system's requirements. This continuous verification at runtime comes at a high price. The notion of incremental verification was applied to the process of probabilistic model checking to increase the efficiency of runtime probabilistic model checking. Heuristics-based incremental probabilistic model checking at runtime was devised and enhanced the efficiency of probabilistic model checking by combining a heuristic technique [13]. The approach was implemented in the model testing tool PRISM, which employed two different types of benchmark case models to verify the model's reachability features.

2.3. CTL Model Checking

The major bottleneck in Computational Tree Logic (CTL) model checking is the state space explosion. This problem is challenging in the framework of typical model checking techniques. For this purpose, a technique is suggested that uses numerous Machine Learning (ML) methods to forecast the results of CTL model checking. Using the current CTL model checking tool, the data set with a few Kripke structures, CTL formulae, and their model checking outcomes is first collected. Second, the data set is trained using Random Forest (RF), Boosted Trees (BT), Decision Trees (DT), and Logistic Regression (LR) methods. The four ML models are derived from it to forecast the results of CTL model verification. Analysis revealed that the proposed method's accuracy is the same as current CTL model checking methods, and its average efficiency is 378 times greater than the old method if each CTL formula is 500 characters long [14]. It should be noted that larger-scale operating systems are a new application field for CTL model checking, and the low efficiency of this sector limits its development.

B. Tools for Model Checking

Supplier	Tool Name	Design Level	Web Pointer
Cadence	Incisive Formal Verifier	RTL	**Link**
PRISM	PRISM	RTL	**Link**
Carnegie Mellon University	NuSMV2	RTL	**Link**

3. Theorem Proving

Theorem proving, often known as automated reasoning, is a frequently used formal verification approach. The correctness result was then achieved by verifying that the specification and implementation were appropriately connected in logic [1]. The system which needs to be examined is mathematically modelled in the appropriate logic, and the features of interest are verified using computer-based software tools known as theorem provers. The use of formal logics as a modelling medium makes theorem proving a highly versatile verification approach, as any system that can be defined mathematically can be formally verified [3]. Some well-known axioms and fundamental inference procedures are the base of theorem provers. Every new theorem must be derived from these basic axioms and primitive inference rules, as well as any other previously established theorems or inference rules.

3.1. Automated Theorem Proving

Propositional or first-order logics are commonly used in automated theorem provers. Although propositional logic is decidable theoretically, automated proofs need exponential-time algorithms [3]. As a result, automatic proofs are primarily accomplished by first reducing the formula to a propositional tautology or a Boolean satisfiability checking condition.

3.1.1. Larch theorem prover [15]

Larch Prover (LP) is a theorem-proving tool that works with equations and is based on many-sorted first-order predicate logic. As Larch Prover is built on the idea of term rewriting systems, its primary function is to reduce a formula to a simpler formula. As a result, while using Larch Prover to prove a mathematical issue, the user could focus on the most important aspects of the problem. Larch Prover may be used to solve a wide range of issues; for example, it is used as a formal verification tool for distributed algorithms.

Axioms are equations that are converted to rewriting rules. The user then challenges LP to prove a hypothesis. Two or more proof techniques may be used when proving a hypothesis; for example, if a conjecture bears the form of implication P=>Q, we can assume P and prove a subgoal Q, but it may be conceivable to explore scenarios where alternative equations are true. We can see that LP is more of a tool to aid a user than a tool to automatically prove conjectures. These theorem provers appear to be based on higher-order logic, whereas LP's language is based on first-order predicate logic. In this regard, LP's language differs from that of Coq or Isabelle. However, by constraining the language, we can observe that LP can automatically prove numerous equations. A theorem-proving tool based on a lighter framework can be used to prove a complex mathematical issue.

3.2. Interactive Theorem Proving

We often see systems, such as analogue circuits or optical systems, whose behavior can only be characterized using more generic mathematics such as infinite sets, real numbers, and so on. First-order logic cannot be utilized to represent complex systems, demanding the use of higher-order logic and interactive theorem proving, in which the user participates in the formal verification process with the machine. One of the most often utilized approaches for constructing interactive theorem provers is Edinburgh LCF (Logic for Computable Functions) [3]. The highly typed functional programming language ML (Meta Language) or its derivatives are used to create LCF-style theorem provers. Higher-order-logic theorems are represented by an ML abstract data type, and the only method to interact with the theorem prover is to run ML procedures on values of these data types. Many automatic proof assistants and automatic proof methods are included in interactive theorem provers to aid the user in the verification process.

3.2.1. HOL Theorem Prover [16]

The hardware development technique is intended to make hardware verification in HOL simple while maintaining a strong connection to a subset of Verilog's formal semantics. The approach is focused on an automatic proof-producing tool that, given a functional version of a Verilog program in HOL, generates Verilog code and proves a correspondence theorem stating that the Verilog code and the functional code behave identically according to a formal semantics of Verilog we developed. The correspondence theorem allows system correctness results for functional code to be transferred to Verilog code.

A proof-producing translation tool from HOL circuits to Verilog circuits was developed, and a formal semantics for a subset of Verilog was devised as a prerequisite for this. The semantics have been carefully constructed to reflect the Verilog standard as closely as possible while being simple enough to utilize for reasoning. The latter is critical for our proof-producing translator, which must use automated reasoning to create arguments that ensure the accuracy of its translations. The translator offers a hardware development flow in which users construct theorems and theories based on shallowly embedded HOL circuits that can be readily transferred to deep embedded Verilog circuits.

C. Tools for Theorem Proving

Supplier	**Tool Name**	**Design Level**	**Web Pointer**
University of Texas at Austin	**ACL2**	Universal	**Link**
Cambridge University	**HOL4**	Universal	**Link**
SRI	**PVS**	Universal	**Link**

V. Summary

Formal hardware verification aims to overcome the limitations of non-exhaustive simulation by demonstrating the relationship between an abstract specification and the actual design. These formal techniques to hardware validation are suitable to scale with the complexity of VLSI designs in the long term. The fundamental reason for this is because they use advanced mathematical tools instead of brute force. Each verification approach has its own set of benefits and drawbacks. For example, model checking can proved counterexample when the result is false but it has state explosion problem, while theorem-proving is quite versatile but reasoning about detailed level circuit behavior is often challenging. Similarly, equivalency checking allows testers to focus on smaller data sets, increasing the chances of finding more faults, however equivalence class identification relies significantly on tester skill. Using the proper verification technique gradually increases the designer's confidence in the tool, the methodology, and his or her own ability to use formal verification tools.

VI. Reference

1. T. Kropf: Introduction to Formal Hardware Verification, Springer Verlag, 1999. (ISBN: 3540654453, 304 pages)
2. M. Girish, G. Gopakumar and D. S. Divya, "Formal and Simulation Verification: Comparing and Contrasting the two Verification Approaches," *2021 2nd International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS)*, 2021, pp. 41-44, doi: 10.1109/ACCESS51619.2021.9563305.
3. O. Hasan and S. Tahar, Formal Verification Methods, Encyclopedia of Information Science and Technology, pp. 7162-7170, IGI Global Pub., Germany, 2015.
4. R. Kumar, "Formal verification of hardware: misconception and reality," *Wescon/98. Conference Proceedings (Cat.*

No.98CH36265), 1998, pp. 135-138, doi: 10.1109/WESCON.1998.716435.

5. John Buxton and Brian Randell," Software Engineering Techniques,"1970 NATO Science Committee, England,1970. (130 pages)
6. A. J. Hu, "Formal hardware verification with BDDs: an introduction," *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM. 10 Years Networking the Pacific Rim, 1987-1997*, 1997, pp. 677-682 vol.2, doi: 10.1109/PACRIM.1997.620351.
7. V. N. Possani, A. Mishchenko, R. P. Ribas and A. I. Reis, "Parallel Combinational Equivalence Checking," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 3081-3092, Oct. 2020, doi: 10.1109/TCAD.2019.2946254.
8. L. Amarú, P. Gaillardon, R. Wille and G. De Micheli, "Exploiting inherent characteristics of reversible circuits for faster combinational equivalence checking," *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 175-180.
9. Feijun Zheng, Yanling Weng and Xiaolang Yan, "An efficient sequential equivalence checking framework using Boolean Satisfiability," *2007 7th International Conference on ASIC*, 2007, pp. 1174-1177, doi: 10.1109/ICASIC.2007.4415843.
10. Q. Wang, R. Li and M. Ying, "Equivalence Checking of Sequential Quantum Circuits," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, doi: 10.1109/TCAD.2021.3117506.
11. Karmakar, R. (2022). Symbolic Model Checking: A Comprehensive Review for Critical System Design. In: Tiwari, S., Trivedi, M.C., Kolhe, M.L., Mishra, K., Singh, B.K. (eds) Advances in Data and Information Sciences. Lecture Notes in Networks and Systems, vol 318. Springer, Singapore. https://doi-org.lib-ezproxy.concordia.ca/10.1007/978-981-16-5689-7_62
12. R. Chen and W. Zhang, "Verification of CTL_BDI Properties by Symbolic Model Checking," *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, 2019, pp. 102-109, doi: 10.1109/APSEC48747.2019.00023.
13. Y. Liu and C. He, "A Heuristics-Based Incremental Probabilistic Model Checking at Runtime," *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 2020, pp. 355-358, doi: 10.1109/ICSESS49938.2020.9237680.
14. W. Zhu, P. Feng and M. Deng, "An Approximate CTL Model Checking Approach," *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 2019, pp. 646-648, doi: 10.1109/ICSESS47205.2019.9040772.
15. S. Jaishy, N. Ito and Y. Kawabe, "Problem Solving with Interactive-Theorem Proving - A Case Study," *2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD)*, 2016, pp. 301-306, doi: 10.1109/ACIT-CSII-BCD.2016.064.
16. A. Lööw and M. O. Myreen, "A Proof-Producing Translator for Verilog Development in HOL," *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2019, pp. 99-108, doi: 10.1109/FormaliSE.2019.00020.