Encyclopedia of Information Science and Technology, Third Edition

Mehdi Khosrow-Pour
Information Resources Management Association, USA

A volume in the

Information Science REFERENCE
An Imprint of IGI Global

Formal Verification Methods

S

Osman Hasan
National University of Sciences and Technology, Pakistan

Sofiène Tahar
Concordia University, Canada

BACKGROUND

On June 1, 2009, Air France Flight 447 from Rio de Janeiro to Paris crashed into the Atlantic Ocean, killing all 216 passengers and 12 crew members. Prior to the disappearance of the A330 aircraft, the automatic reporting system sent messages indicating disagreement in the airspeed readings, which led investigators to believe that the pilot probe sensors did not "accurately" measure airspeed and the autopilot may have automatically disengaged. Like the above probes, many systems are increasingly being used in safety-critical domains, such as medicine, transportation systems, chemical plants, etc. There is hence a dire need to ensure the accuracy of such systems as a system bug, or error, may endanger human life or lead to a significant financial loss.

In order to ensure error-free systems, the system design process is usually accompanied by a rigorous system analysis to check if the designed system would exhibit the desired behavior. The core of system analysis is based on mathematics and the fundamental idea is to create a mathematical model of the system and then use logical or mathematical reasoning to verify that the desired properties hold for this model. Traditionally, system analysis is done by paper-and-pencil proof methods. However, considering the complexity of present age systems, such kind of analysis is notoriously difficult, if not impossible, and is quite error prone due to the human error factor. Moreover, it is quite often the case that mathematicians forget to pen down all the assumptions that are required for the validity of their analysis. This fact may also result in designing erroneous systems. With the advent of computers, many computer based software tools based on the principle of testing or simulation have been introduced for system verification. Due to the reliable and efficient bookkeeping characteristic of computers, large systems can be analyzed and thus one of the limitations of paper-and-pencil proof methods can be overcome. However, the main problem with such kind of analysis is its completeness as the system is checked only for a subset of possible inputs since exhaustive testing is not possible for any system with a significant number of inputs due to the exponential growth of the test patterns. In Dijkstra's words "*Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*" Moreover, testing or simulation cannot be used to precisely verify properties about continuous systems due to the usage of computer arithmetic, like floating-point or fixed-point representations of real numbers. The above mentioned inaccuracy limitations of commonly used system analysis techniques can be held responsible for many unfortunate incidents that happened due to an erroneous system deployed in a safety-critical domain For example, the Therac-25 software bug and the Intel Pentium's floating-point division unit error).

INTRODUCTION

In order to raise the reliability of system analysis, a system analysis technique is required that can have the precision of paper-and-pencil based mathematical proofs, and thus does not rely upon computer-arithmetic, and utilizes the computers for bookkeeping, to be able to handle complex systems without having to worry about human-errors. Formal verification methods, which are primarily based on theoretical computer science fundamentals like logic calculi, automata theory and strongly type systems, fulfill these requirements. The main principle behind formal analysis of a system is

DOI: 10.4018/978-1-4666-5888-2.ch706

to construct a computer based mathematical model of the given system and formally verify, within a computer, that this model meets rigorous specifications of intended behavior. Due to the mathematical nature of the analysis, 100% accuracy can be guaranteed.

The history of formal methods dates back to Knuth and Dijkstra as both of them advocated the topic. Formal verification methods started to be investigated as computer-aided design (CAD) tools in the 1970s for software verification. However, the interest was marred by the fact that the software bugs can be easily fixed by releasing a software patch and thus the added reliability of software is not worth the rigorous exercise of formal verification. There was some research activity related to the formal verification of security systems funded by the US National Security Agency in the 1980s but the real catalyst for the active research interest in formal verification was their usage in verifying digital hardware systems in late 1980s. This is mainly because hardware descriptions are often more regular and hierarchical than software ones, hardware primitives are less obscure than the ones used in software and the cost of an uncaught design bug in hardware is much more profound than software since the hardware silicon chip once fabricated cannot be fixed by releasing a patch but instead has to be re-designed and re-fabricated, which costs considerable amount of time and money. The Intel floating-point division bug in 1994 further enhanced the interest in formal hardware verification and the industry started to adopt formal hardware verification tools in their design flows in late 1990s (Kropf, 1999). With the success of formal verification in hardware and due to some interesting developments in the underlying technologies, it started to be used again for software, transportation and security system analysis domains. Moreover, researchers started to explore the formal verification of physical systems, such as control systems, robotics and analog circuits, and biological systems by using powerful abstraction techniques to reduce the complexity of observable phenomena to what is relevant for a particular purpose. Recently, formal verification methods have also been used to verify complete system models, along with their continuous and unpredictable physical realities. The future of formal methods seems to be quite promising and besides academia, industry giants, like Intel and Microsoft, are also actively involved in formal methods related research.

The added benefits of formal verification methods come mainly at the cost of extreme rigor. Generally speaking, the expressiveness of a formal verification method is in direct proportion with the amount of required user intervention. Thus, formal verification of complex systems is more challenging and time consuming. Therefore, the general trend is to use a lightweight approach, i.e., use traditional verification methods, like simulation or testing, where accuracy of the analysis is not a big concern while using formal verification methods for the critical sections of the systems. On similar lines, hybrid formal verification methods are also being developed which allow us to partition the overall system model based on its complexity levels and thus facilitate using automatic formal verification methods for the rather simpler sections of the system while using the interactive methods with the complex sections.

Generally, formal verification methods are classified based on their underlying logic, expressiveness and decidability. The most commonly used formal verification methods include *theorem proving*, *symbolic simulation* and *model checking*. All of these have their own strengths and weaknesses. They have been used successfully to verify a variety of real-world systems. In the rest of this chapter, we provide a brief introduction to these widely used formal verification methods along with some of their practical applications. Finally, the chapter ends with some discussions and conclusions.

THEOREM PROVING

Theorem proving or automated reasoning is one of the most generic and widely used formal verification method. The system that needs to be analyzed is mathematically modeled in an appropriate logic and the properties of interest are verified using computer based software tools usually called theorem provers. The use of formal logics as a modeling medium makes theorem proving a very flexible verification method as it is possible to formally verify any system that can be described mathematically. The core of theorem provers usually consists of some well-known axioms and primitive inference rules. Soundness is assured as every new theorem must be created from these basic axioms and primitive inference rules or any other already proven theorems or inference rules. A question that may arise

here is that why do we need logic to model the system and why natural languages like English or other commonly used programming languages like C++ or Java may not suffice for carrying out theorem proving. The foremost answer to this question is that the meanings of these languages can be ambiguous and can lead to multiple interpretations depending on the context and implicit assumptions. Thus, statements specified in such languages cannot be used for theorem proving where the main goal is to verify formulas based on precise rigorous reasoning and we need a logical language with a syntax that can be described using a few basic rules and a semantics that can be unambiguously defined.

The human interaction or the manual proof effort required for proving logical formulas in a theorem prover varies from trivial to complex depending on the underlying logic. For instance, propositional logic is the logic of propositions or declarative sentences which can be *true* or *false*. The propositions can be combined using Boolean operators: *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implication* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$). Theoretically speaking, propositional logic is decidable, i.e., the logical correctness of a formula specified in propositional logic can be automatically verified using an algorithm. The main limitation of propositional logic is its limited expressiveness as it cannot be used to represent verification problems for all sorts of systems. First-order logic extends propositional calculus with quantifiers, i.e., *for all* ($\forall$) and *there exists* ($\exists$), and predicates, which are functions that return a Boolean value. One can declare constants, function names and free variables in first-order logic, which gives a considerable amount of flexibility in terms of expression. However, first-order logic is not completely decidable and is usually referred to as semi-decidable since all statements expressed in first-order logic cannot be automatically verified by a computer algorithm. Thus, the user of first-order-logic theorem provers may have to interactively verify some formulas by providing inputs to assist the tools. Finally, higher-order logic is the most expressive form of logic that allows quantification over functions and sets. These features make it so expressive that any system, along with its continuous and unpredictable elements, can be described using higher-order logic given that its behavior can be expressed in a closed mathematical form. This expressiveness comes at the cost of manual verification where user input is required to verify all formulas expressed in higher-order-logic, due to its un-decidable nature. Based on the required user intervention in the proof process, theorem proving can be broadly classified into two sub branches, i.e., automated theorem proving and interactive theorem proving.

Automated Theorem Proving

Automated theorem provers are primarily based upon propositional or first-order logics. The propositional logic is decidable theoretically, but in practice, exponential-time algorithms are required for automatic proofs. Thus, automatic proofs are mainly done by first reducing the formula to be verified to a propositional tautology or Boolean satisfiability checking problem. This way, efficient algorithms like Binary decision diagrams (BDD), Davis–Putnam–Logemann–Loveland (DPLL) based SAT (satisfiability) solvers or Stakmarck's procedure may be used to automatically check the validity of the formula. Recently, Satisfiability Modulo Theories (SMT) solvers (Nieuwenhuis, 2006) extend the capabilities of SAT solvers by handling arithmetic and some other decidable theories and have revolutionized the area of automated theorem proving.

From the automated theorem proving user's perspective, formal verification can be done by developing a formal model of the system under verification using the available logic, i.e., propositional or first-order logic. The next step is to formally specify the property that needs to be verified for the given system. This property can then be verified using the automatic verification utilities like SMT solvers.

Various standalone automated theorem proving tools have been developed and some of the prominent ones include ACL2, E, Key, MetiTarski, Prover 9 and Vampire. Some of the successful SMT solvers are mathsat 4, yices and Z3. In terms of real-world applications of automated theorem proving, digital logic circuits are primarily based on propositional logic so automated theorem provers find a direct application in their verification. The ability of the first-order logic to formalize axiomatic systems makes it very useful for formal verification and this fact along with the powerful automatic verifiers, like SAT and SMT solvers, have facilitated the usage of automated theorem proving to verify a wide variety of applications including both software (Denney, 2006; Beckert, 2007) and hardware (Flatau, 2002; Ray, 2010) designs. Despite these successful examples, automated theorem proving cannot

be used with systems that involve data types with an infinite domain, such as the real line, due to the limitations of the underlying logic.

Interactive Theorem Proving

In system analysis, we often come across systems, such as analog circuits or optical systems, whose behavior can only be described in terms of more general mathematics involving infinite sets, real numbers, etc. As described earlier, first-order logic cannot be used to model these kinds of systems and thus we have to use higher-order logic and interactive theorem proving, where the user is involved in the formal verification process along with the machine. Edinburgh LCF (Logic for Computable Functions) is one of the most commonly used methods for developing interactive theorem provers. LCF style theorem provers are implemented using the strongly-typed functional programming language ML (Meta Language) or its variants. An ML abstract data type is used to represent higher-order-logic theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. The Interactive theorem provers usually include many automatic proof assistants and automatic proof procedures to assist their user in the verification process. The user interacts with a proof editor and provides it with the necessary tactics to prove goals while using automatic proof procedures whenever the problem is reduced to a decidable subset. This process could be very tedious and usually takes thousands of lines of proof script and hundreds of man-hours for verifying the mathematical analysis presented in a couple of pages. However, the ability to build upon already verified results is a big strength of this technique, which allows us to broaden the scope of interactive theorem proving.

Some widely used theorem provers include HOL, Isabelle, PVS and CoQ. Many interesting formalizations, including real analysis theory (Harrison, 1998), C programming language (Norrish, 1998), Euclidian geometry (Harrison, 2005) and probability theory (Mhamdi, 2011), have been developed using interactive theorem proving. Moreover, these foundations have been utilized to verify real-world systems, like programming language compilers (Strecker, 2002), floating-point algorithms (Harrison, 2006), DSP systems (Akbarpour, 2006), optical systems (Hasan, 2009) and wireless sensor networks (Elleuch, 2011).

For illustration purposes, consider the formal verification of an algorithm that returns the minimum value of an array of real numbers. Due to the involvement of real numbers and an arbitrary number of elements, higher-order logic theorem proving is used for the verification. The first step in analyzing this algorithm is to formalize it in higher-order logic. This can be mainly done by the following recursive definition:

$$\vdash \forall x.\ \text{min\_list}\ ()\ (x{:}\text{real}) = x\ \hat{}$$
$$\forall h\ t\ x.\ \text{min\_list}\ (h::t)\ x = \text{minimum}\ (\text{min\_list}\ t\ h)\ x$$

where the function *minimum* takes two real numbers and returns the lesser one out of them and the symbol:: denotes the *cons* operation between the head and tail of an array. The function recursively finds the minimum real number of a list of real numbers and another real number *x*.

The next step after the formalization of the algorithm is to formalize the property of interest as a proof goal in the theorem prover. The following property serves this purpose:

$$\vdash \forall\ (L{:}\ \text{real list})\ x.\ \text{MEM}\ x\ L \Rightarrow \text{min\_list}\ (\text{TL}\ (L))\ (\text{HD}\ L) \leq x$$

The predicate *MEM x L* ensures that x is a member of list *L* and the functions *HD* and *TL* return the head and tail of their list argument, respectively. Thus, the goal guarantees that the value returned by the function *min_list* is less than or equal to all values of list L. This theorem can now be verified in a theorem prover using induction on the variable *L*. The rest of the reasoning is based on the above mentioned definitions and properties of real numbers and lists. The main strength of the analysis presented above is its generic nature, which is evident from the usage of the "*for all*" quantifier for variables *L* and *x*, and guaranteed accuracy, based on the inherent soundness of a theorem prover. However, on the downside, the formalization involves the understanding of higher-order logic and the theories of real numbers and lists. Moreover, the verification required human guidance and was done interactively.

SYMBOLIC SIMULATION

Symbolic simulation (Bryant, 1990) bridges the gap between the traditional simulation or testing approach and formal verification. The main idea is to use symbols (or variables) instead of the actual values in simulation and thus consider multiple executions of the system simultaneously. This way the number of test cases reduces and thus exhaustive simulation becomes realistic. For example, consider the case of verifying a 4-input *and* gate. In traditional simulation, this verification would require 2^4=16 test vectors but using symbolic simulation, we can assign Boolean variables to each input, say x_1, x_2, x_3 and x_4, and check if the output is equal to $x_1 \wedge x_2 \wedge x_3 \wedge x_4$ in one test run.

The Boolean expressions with the symbolic variables are generally expressed using canonical representations like the Reduced Order Binary Decision Diagrams (ROBDDs). This way, checking the equivalence between two circuits becomes very straightforward as we just have to make sure that the two circuits have the same directed acyclic graph. The main computationally expensive step is the composition of operations to obtain the final ROBDD. Symbolic simulation is often used in conjunction with ternary simulation where besides *true* and *false* a *don't care* (X) value is also considered. Symbolic simulation methods are quite frequently used with model checking. The main concept in this regard is Symbolic Trajectory Evaluation (STE), which is an extension for symbolic simulation and allows users to specify time dependent properties using temporal logic over bounded trajectories (Case, 2011).

One of the most common uses of symbolic simulation is for functional equivalence checking of digital designs. The field is quite mature and industrial tools are available for conducting symbolic simulations based equivalence checking. Examples include ESP-CV from Synopsys, Insight from Avery Design Systems and the Blue Pearl software, which uses symbolic simulation to speed-up timing closure of RTL designs. One of the main reasons why symbolic simulation has paved its way to the industry is its user friendliness since the tools usually accept hardware descriptive languages, like Verilog or VHDL, and work in a push button fashion. More recently, the usage of symbolic simulation in software verification has also been investigated and has brought promising results (Cadar, 2011).

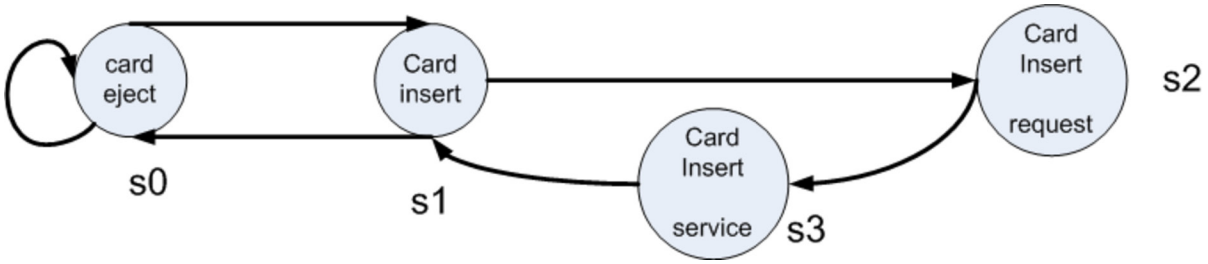
MODEL CHECKING

Model checking is primary used as the verification technique for reactive systems, i.e., the systems that exhibit a behavior that is dependent on time and their environment, like controller units of digital circuits and communication protocols. The inputs to a model checker include the finite-state model of the system that needs to be analyzed along with the intended system properties, which are expressed in temporal logic. The model checker automatically verifies if the properties hold for the given system while providing an error trace in case of a failing property. The main verification principle behind model-checking is to construct a precise state-based model of the given system and exhaustively verify the given property for each state of this model. The analysis is automatic which is why model checking is one of the most widely used formal verification technique. On the other hand, model-checking is limited to systems that can only be expressed as finite state machines. Another major limitation of the probabilistic model checking approach is state space explosion. The state space of a system can be very large, or sometimes even infinite. Thus, it becomes computationally impossible to explore the entire state space with limited resources of time and memory. This problem is usually resolved by working with abstract, less complex, models of the system by somewhat compromising the accuracy of the analysis.

Many techniques have been proposed to minimize the memory and computation requirements of model checking. Symbolic model checking (McMillan, 1993) is based on the idea of grouping multiple states together and assigning them a unique symbol and then running model checking algorithms on this symbolic state-space. The most commonly used data structure in symbolic model checking is BDDs. Bounded model checking (BMC) (Biere, 1999) is an extension of symbolic model checking and the main idea is to encode states as propositional logic formulas and then use SAT solvers for the analysis. A subset of executions with an upper bound on length, say *k*, is chosen and the counter-example is searched in this subset in BMC. If a contradiction is not found then the algorithm is run again with a higher *k*.

The above mentioned methods along with appropriate usage of abstraction have enabled a wide usage of model checking. Some of the commonly used model checking tools include SPIN (used for distributed sys-

Figure 1. State Transition System for the Simple ATM



tems; mostly software), NuSMV (used for concurrent systems including digital hardware), Uppaal (Real-Time Systems), Hytech (Hybrid Systems) and PRISM (probabilistic model checking). Candence Design Systems had released a commercial model checking tool FormalCheck in late 1990s that was capable of reading synthesizable Verilog code and thus the user of the tool had to only specify the properties to be verified and the rest of the analysis was automatic. However, most of the models checking tools do not offer such kind of a luxury and the users have to encode the model in the language of the tool. For example, SPIN accepts Promela models only. Model checking has been used to formally verify a wide variety of systems including software (Berard, 2010), digital hardware (Raffelsieper, 2009), security protocols (Armando, 2008), analog and mixed signal circuits (Zaki, 2008), etc.

For illustration purpose, consider a simple automatic bank teller machine (ATM): The first step to initiate a transaction is to insert the ATM card. Next, the user may make a request, like requesting and depositing money, or do nothing. If a request is made then it is serviced and the card is ejected else the card is ejected without providing any service to the user. The first step in analyzing this system using model checking is to construct a state-based model, depicted in Figure 1, for this system in the language supported by the model checker. It is interesting to note that more than one transition is possible from the state s1. To model such cases, all model checking languages support non-deterministic assignments.

Now, we may check that eventually the card is ejected for all executions. This property can be specified in linear temporal logic (LTL) as *GF (card eject),* where the temporal operator *G* represents the global validity of a property and *F* represents eventual verification of a property in the future. The model checker can be invoked to check the correctness of this property and it automatically returns *False* with the counter-example path: *s0,s1,s2,s3,s1*….This is the case when the user always initiates another request just when her previous request was serviced. Similarly, another property could be that every request is eventually serviced. This can be modeled in LTL as $G(request \rightarrow F\ response)$ and can be automatically verified by the model checker since it is valid for all possible state-paths.

The above example illustrates that model checking is a rigorous method and unlike testing it verifies a property only if it is true for all possible executions. This is a very useful feature as the verification engineer does not need to think about creating smart test cases to identify system problems. However, due to the same rigorous nature of verification, the technique may not be used for larger models.

CONCLUSION

This chapter provides a brief overview of formal verification methods, their strengths, weaknesses and applications. Formal verification provides precise system analysis, which is a dire need in safety-critical system design. However, this precision comes at the cost of extensive engineering time and effort. Theorem proving is one of the most generic formal verification methods as it can automatically handle the analysis of systems that can be expressed using propositional or first-order logic and can also handle complex systems, involving continuous and unpredictable components, using higher-order logic at the cost of significant manual effort. Symbolic simulation is an automatic and user friendly formal verification method but has a very limited scope in terms of the systems that can be analyzed. Finally, model checking requires manual efforts in formal specification of the system and proper-

ties but the verification is done automatically. However, it is limited by the state-space explosion problem and thus cannot handle very large systems. One of the recent trends in formal verification methods is to use a hybrid approach, i.e., leverage upon the strengths of each method to develop hybrid tools, e.g., KeYmaera combines model checking, theorem proving and symbolic methods to verify hybrid systems.

FUTURE RESEARCH DIRECTIONS

The chapter also provides some examples of using formal methods for ensuring the correctness of some real-world systems. It is interesting to note that besides the traditional computing problems, formal methods are also being widely explored these days in verifying some exotic problems involving the domains of physics, biology, economics, law, etc. However, most of these existing works have been conducted by academic research groups and the usage of formal methods in the industry has been somewhat scarce. Some of the main reasons include the time-to-market pressures, the non-friendly nature of available formal method tools and the challenges associated with the identification of the most appropriate formal method for a particular application. The situation can be improved by providing appropriate training to industrial engineers as they are usually not very savvy with formal methods and their notations. Similarly, regulations and standards imposing the usage of formal methods are also expected to improve the situation. In the past decade, some industries, including SAP, Siemens, Intel and SSF, which are involved in developing safety-critical systems, have started to utilize formal methods and a comprehensive survey of industrial usage of formal methods and their impact is presented in (FM4industry, 2013). Similarly, a worth reading motivating story about the adoption of formal methods by the railway signaling division of General Electric Transportation Systems (GETS) is presented in (Bacherini, 2006).

REFERENCES

Akbarpour, B., & Tahar, S. (2006). An Approach for the Formal Verification of DSP Designs using Theorem Proving. *IEEE Transactions on CAD of Integrated Circuits & Systems*, *25*(8), 1141–1457. doi:10.1109/TCAD.2005.857314

Arm&o, A., & Compagna, L. (2008). SAT-based Model-Checking for Security Protocols Analysis. *International Journal of Information Security, 7*(1), 3-32.

Bacherini, S., Fantechi, A., Tempestini, M., & Zingori, N. (2006). A Story about Formal Methods Adoption by a Railway Signaling Manufacturer. In Formal Methods (pp 179–189).

Beckert, B., Hähnle, R., & Schmitt, P. H. (2007). *Verification of Object-Oriented Software: The KeY Approach*. Springer.

Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., & Schnoebelen, P. (2010). *Systems & Software Verification: Model-Checking Techniques & Tools*. Springer.

Biere, C. Fujita, & Zhu. (1999). Symbolic Model Checking using SAT Procedures instead of BDDs. *Design Automation Conference* (pp. 317-320).

Bryant, R. E. (1990). Symbolic Simulation—Techniques and Applications. In *Design Automation Conference* (pp. 517-521).

Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011). Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In Software Engineering (1066-1071).

Case, M., Baumgartner, J., Mony, H., & Kanzelman, R. (2011). *Approximate Reachability with Combined Symbolic and Ternary Simulation* (pp. 109–115). Formal Methods in Computer-Aided Design.

Denney, E., Fischer, B., & Schumann, J. (2006). An Empirical Evaluation of Automated Theorem Provers in Software Certification. *International Journal of Artificial Intelligence Tools*, *15*(1), 81–107. doi:10.1142/S0218213006002576

Elleuch, M., Hasan, O., Tahar, S., & Abid, M. (2011). *Formal Analysis of a Scheduling Algorithm for Wireless Sensor Networks* (pp. 388–403). Formal Engineering Methods. doi:10.1007/978-3-642-24559-6_27

FM4Industry. (2013) Retrieved from http://www.fm4industry.org

Flatau, A., Kaufmann, M., Reed, D. F., Russinoff, D., Smith, E., & Sumners, R. (2002). Formal Verification of Microprocessors at AMD. In Designing Correct Circuits.

Harrison, J. (1998). *Theorem Proving with the Real Numbers*. Springer. doi:10.1007/978-1-4471-1591-5

Harrison, J. (2005). A HOL Theory of Euclidean Space. In Theorem Proving in Higher Order Logic (pp. 114-129).

Harrison, J. (2006). Floating-Point Verification using Theorem Proving. In *Formal Methods for the Design of Computer* (pp. 211–242). Communication, and Software Systems.

Hasan, O., Afshar, S. K., & Tahar, S. (2009). *Formal Analysis of Optical Waveguides in HOL* (pp. 228–243). Theorem Proving in Higher-Order Logics. doi:10.1007/978-3-642-03359-9_17

Hasan, O., & Tahar, S. (2010). Formally Analyzing Expected Time Complexity of Algorithms using Theorem Proving. *Journal of Computer Science & Technology*, *25*(6), 1305–1320. doi:10.1007/s11390-010-9407-0

Kropf, T. (1999). *Introduction to Formal Hardware Verification*. Springer. doi:10.1007/978-3-662-03809-3

McMillan. K. L., (1993). Symbolic Model Checking. Kluwer Academic Publishers.

Mhamdi, T. Hasan, O., & Tahar, S. (2011). Formalization of Entropy Measures in HOL. Interactive Theorem Proving (pp. 233-248).

Nieuwenhuis, R., Oliveras, A., & Tinelli, C. (2006). Solving SAT & SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Lovel & Procedure to DPLL. *Journal of the ACM*, *53*(6), 937–977. doi:10.1145/1217856.1217859

Norrish, M. (1998). *C Formalised in HOL*. PhD Thesis, University of Cambridge, UK.

Raffelsieper, M., Roorda, J.-W., & Mousavi, M.-R. (2009). Model Checking Verilog Descriptions of Cell Libraries. Application of Concurrency to System Design (pp. 128-137).

Ray, S., Bhadra, J., Portlock, T., & Syzdek, R. (2010). *Modeling & Verification of Industrial Flash Memories* (pp. 705–712). Quality Electronic Design.

Strecker, M. (2002). Formal Verification of a Java Compiler in Isabelle. In Automated Deduction (pp. 63–77).

Zaki, M., Tahar, S., & Bois, G. (2008). Formal Verification of Analog & Mixed Signal Designs: A Survey. *Microelectronics Journal*, *39*(12), 1395–1404. doi:10.1016/j.mejo.2008.05.013

ADDITIONAL READING

Abrial, J. R. (2009). Faultless Systems: Yes We Can! *Computer*, *42*(9), 30–36. doi:10.1109/MC.2009.283

Baier, C., & Katoen, J. P. (2008). *Principles of Model Checking*. MIT Press.

Boca, P. P., Bowen, J. P., & Siddiqi, J. I. (2009). *Formal Methods: State of the Art & New Directions*. Springer.

Hall, A. (2007). Realizing the Benefits of Formal Methods. *Journal of Universal Computer Science*, *13*(5), 669–678.

Harrison, J. (2009). *H&book of Practical Logic & Automated Reasoning*. Cambridge U. Press. doi:10.1017/CBO9780511576430

KEY TERMS AND DEFINITIONS

Binary Decision Diagram (BDD): A representation of a Boolean expression using a rooted directed acyclic graph (DAG) that consists of terminal (with constant values 0 or 1) or non-terminal nodes (variables). A Reduced ordered BDD (ROBDD), which is a widely used data structure in formal verification, is a BDD with a particular variable order where identical nodes are shared and redundant tests are eliminated.

Formal Verification Methods: Mathematical techniques, often supported by computer-based tools,

for the specification and verification of software and hardware systems. The main principle behind formal analysis of a system is to construct a computer based mathematical model of the given system and formally verify, within a computer, that this model meets rigorous specifications of intended behavior.

Higher-Order Logic: A system of deduction with a precise semantics. It differs from the more commonly-known predicate and first-order logics by allowing quantification over function variables. This extension substantially increases the expressiveness of the logic and thus higher-order logic can be used for the formal specification of most mathematical concepts and theories.

Satisfiability: A logical formula is termed to be satisfiable if and only if it is true for at least one combination of its variables.

Tautology: A logical formula is termed to be a tautology (valid) if and only if it is true for all the possible values of its variables. In other words, a formula is a tautology if its negation (¬F) is unsatisfiable. This relationship between satisfiablity and tautology is one of the foundational principles of using SAT solving for equivalence checking.

Temporal Logic: Temporal logic allows us to formally represent time-dependent propositions. For example, propositions like an event would happen in the next time step or sometime in the future or would never happen in the future, can be expressed using temporal logic operators. Temporal logic is used in model-checking to express the properties of interest about the reactive systems.