Volume 3, Issue 4, April 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Ontology of Formal Verification

Madhu Nashipudi[a], Rupali Kale [b]
[a] *Pillai Institute of Information Technology, Information Technology Department,*
[b] *Pillai Institute of Information Technology, Computer Department*

Abstract: Formal verification requires you to think differently. For example, simulation is empirical i.e. you use trial and error to try to uncover bugs that can take an intractable amount of time to try all possible combinations. Hence, it is never complete. Furthermore, since engineers have to define and generate a significant number of input scenarios, they are focusing their effort on how to break the design not on what the design is supposed to do. Formal verification, on the other hand, is mathematical, exhaustive and allows the engineer to focus solely on intent or "What is the design's correct behaviour?" Formal verification is the process of proving or disproving properties using formal methods (i.e., mathematically precise, algorithmic methods). A formal proof of a property provides a guarantee that no simulation of the system considered will violate the property. This eliminates the need for writing additional test cases to check the property. Ontologies are widely used in many areas. Different automatic or semiautomatic extraction techniques have been proposed for building domain ontology in recent years. The correctness of the extracted ontology, however, has often been ignored or not verified formally. With increasingly complex and sophisticated real world domains, the issue of correctness and verification of ontology is becoming more important.

Keywords:

1.0 Introduction

Computers are ubiquitous these days and are used to control various safety critical systems like aircrafts, satellites, medical instruments, etc. Software that is developed to operate these systems is complex, consisting of many modules, each with thousands of lines of code. Any error in the functioning of these programs results in serious loss of life and money. An example of such a failure is that of the Ariane 5 rocket, which exploded in June 1996, less than forty seconds after it was launched. The reason for this explosion was a software error in the computer that was responsible for calculating the rocket's horizontal velocity. An exception was caused during the conversion of a 64-bit floating point number into a 16-bit signed integer. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This caused the computer to fail and the backup computer too failed due to the same reason. Consequently, no instruction regarding altitude was transmitted to the onboard computer and this resulted in the rocket failing. This example (and many more) clearly indicates the need for developing reliable hardware and software systems [1]. Formal verification is the process of checking whether a design satisfies some requirements (properties)[3]. We are concerned with the formal verification of designs that may be specified hierarchically, this is also consistent with how a human designer operates. In order to formally verify a design, it must first be converted into a simpler "verifiable" format. The design is specified as a set of interacting systems; each has a finite number of configurations, called states. States and transition between states constitute FSMs. The entire system is an FSM, which can be obtained by composing the FSMs associated with each component. Hence the first step in verification consists of obtaining a complete FSM description of the system. Given a present state (or current configuration), the next state (or successive configuration) of an FSM can be written as a function of its present state and inputs. The two most popular methods for automatic formal verification are language containment and model checking[2] .

2.0 Formal Verification Methods

This section describes all methods of formal methods in detail. These are widely used in various industries since it has automation techniques.

2.1 Model Checking

Model checking aka property checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification. Typically, one has hardware or software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking [5] is a technique for automatically verifying correctness properties of finite-state systems.

Software model checking is the algorithmic analysis of programs to prove properties of their executions. While focus here is on analysing the behaviour of a program relative to given correctness specifications, the development of specification mechanisms happened in parallel, and merits a different survey. More recently, software model checking has been influenced by three parallel but somewhat distinct developments. First, development of program logics and

associated decision procedures provided a framework and basic algorithmic tools to reason about infinite state spaces. Second, automatic model checking techniques for temporal logics provided basic algorithmic tools for state-space exploration. Third, compiler analysis formalized by abstract interpretation provided connections between the logical world of infinite state spaces and the algorithmic world of finite representations. [5]

Software model checking is recognized as a breakthrough in software verification and resulted, for example, in tools shipped with the Windows Driver Development Kit. The original idea of Model Checking (whose authors obtained the 2007 Turing Award) is to build a model of the system to be verified, and to automatically analyse it in order to check whether all behaviours respect a property typically expressed in temporal logic. As a result, the approach states whether the model satisfies the property and, maybe more importantly, exhibits a counter example in case it does not. This counter example can be of tremendous help to the human designer or programmer in order to track the error and to fix it. Applying directly the idea to modern computer systems is intractable due to the huge number, practically infinite, of possible states and behaviours the underlying system might exhibit. This is known as the state explosion problem, and several approaches or techniques like partial order reduction, symbolic representations, bounded model checking, abstraction, interpolation, and SAT and SMT solvers have been developed to combat it. As a result, software model checking is the combination of the original automatic model checking framework with decision procedures, symbolic representations, and static analysis as formalized by abstract interpretation.

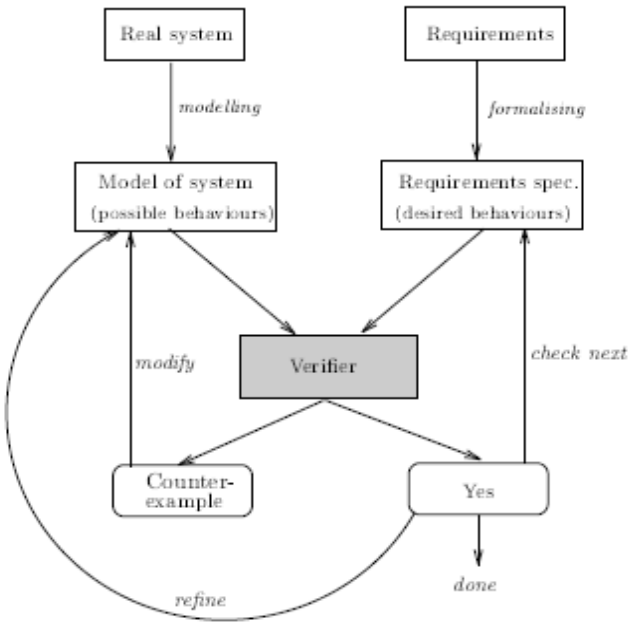
The basic concept refers to mathematical logic. Model checking determines whether a given structure is a model of a given logical formula. When checking hardware and software systems, the terms model and property are often used instead of mathematically rigorous terms structure and formula.

Models accurately describe the behaviour of systems. Both finite and infinite systems are supported by different model checking tools. In either case, the state of the system during checking holds the current values of variables, program counter, etc. Transitions define how the system moves from one state to another. Models are written in special-purpose modelling languages (e.g. Promela [7]) or derived from general programming languages (e.g., old versions of JPF).

Properties are derived from system's specifications and tell what the system should do and what not (e.g., a property requires that a deadlock, uncaught exception, or null-pointer dereference does not occur in the system). Properties are usually expressed as formulas in temporal logic (e.g., LTL in Spin [7]) or by general programming languages (e.g., Java in JPF [6]). Some properties are generic and they can be hard coded in model checkers (e.g., a deadlock property in Spin).

The main objective of model checking is to efficiently find subtle errors in models or prove they are error free. In other words, it is used to check whether a set of properties hold in a model or not. The answer provided by model checkers is either 'yes' or 'no + counterexample'. A counterexample (or error trace) describes the complete execution path leading to the property violation.

The user inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The counter-example consists of a scenario in which the model behaves in an undesired way. Thus the counter-example provides evidence that the model is faulty and needs to be revised. This allows the user to locate the error and to repair the model specification before continuing. If no errors are found, the user can refine its model description (e.g. by taking more design decisions into account, so that the model becomes more concrete / realistic) and can restart the verification process.



Verification Methodology of Model Checking

2.1.1 Explicit Representation Model Checking

The first successful model checking approach is explicit model checking. There are different approaches based on LTL [4] and CTL [7] properties. In all approaches, the state space is represented explicitly, and searched by forward exploration until a violation of a property is found. For example, in LTL model checking the negation of a property is represented as an automaton that accepts infinite words (B¨uchi automaton). If the synchronous product of model and B¨uchi automaton contains any accepting path, then this path proves property violation (the path shows that the negation of the property is accepted by the model automaton, and therefore the property itself is violated). The counterexample is simply the path back to the initial state. The search algorithm can either be depth- or breadth-first search; recently heuristic search has also been considered. Breadth-first search always finds the shortest possible counterexamples, but the memory demands are significantly higher than for depth-first search. In CTL model checking, all states satisfying a given property are determined by recursively calculating the satisfied sub-formulas for each state. If all states are visited and no violation is detected, then the property is consistent with the model.

Directed model checking extends explicit model checking with heuristic search to increase the speed with which errors are found and counterexamples are generated. Such a technique is applicable if the aim of model checking is not a proof of correctness, but the generation of counterexample. As such, this idea is well suited for test case generation.

2.1.2 Symbolic Model Checking

a) Binary Decision Diagrams (BDDs)
b) Propositional Satisfiability Checkers (SAT solvers)

In symbolic model checking [3], sets of states are represented implicitly using Boolean functions. For example, assume that the behavior of some system is determined by the two variables v1 and v2, and that (11, 01, 10) are the three combinations of values that can be assigned to these variables in any execution of this system. Rather than keeping and manipulating this explicit list of states (as was done in explicit model checking), it is more efficient to handle a Boolean function that represents this set, e.g. v1 _v2. Manipulating Boolean formulas can be done efficiently with Reduced Ordered Binary Decision Diagrams (ROBDD, or BDD for short), a compact, canonical graph representation of Boolean functions. The process works roughly as follows: The set of initial states is represented as a BDD. [3] The procedure then starts an iterative process, where at each step i, the set of states that can first be reached in i steps from an initial state are added to the BDD. At each such step, the set of new states is intersected with the set of states that satisfy the negation of the property. If the resulting set is non-empty, it means that an error has been detected. This process terminates when the set of newly added states is empty or an error is found. The first case indicates that the property holds, because no reachable state contradicts it. In the latter case, the model checker prints a counterexample. Note that termination is guaranteed, since there are only finitely many states. The combination of symbolic model checking with BDDs pushed the barrier to systems with 1020 states and more. Combining certain, mostly manual, abstraction techniques into this process pushed the bound even further. For the first time a significant number of realistic systems could be verified, which resulted in a gradual adoption of these procedures to the industry. Companies like Intel and IBM started developing their own in-house model checkers, first as experimental projects, and later as one more component in their overall quality verification process of their chip designs. Intel has invested significantly in this technology especially after the famous Pentium bug a few years ago.

2.1.3 Dynamic Symbolic Execution

Concolic (CONCrete + symbOLIC) testing (also known as dynamic symbolic execution or white-box fuzzing) combines concrete dynamic analysis and static symbolic analysis to automatically generate test cases to explore execution paths of a program. To apply concolic testing, we must specify symbolic variables in a target program, based on which symbolic path formulas are generated at runtime.

- **Key Idea**

Automatically generate test inputs that cover unexplored branches in the program.

Essentially, a concolic testing algorithm operates as follows:

a) Classify a particular set of variables as input variables. These variables will be treated as symbolic variables during symbolic execution. All other variables will be treated as concrete values.
b) Instrument the program so that each operation which may affect a symbolic variable value or a path condition is logged to a trace file, as well as any error that occurs.
c) Choose an arbitrary input to begin with.
d) Execute the program.
e) Symbolically re-execute the program on the trace, generating a set of symbolic constraints (including path conditions).
f) Negate the last path condition not already negated in order to visit a new execution path. If there is no such path condition, the algorithm terminates.
g) Invoke an automated theorem prover to generate a new input. If there is no input satisfying the constraints, return to step 6 to try the next execution path.
h) Return to step 4.

There are a few complications to the above procedure:

The algorithm performs a depth-first search over an implicit tree of possible execution paths. In practice programs may have very large or infinite path trees — a common example is testing data structures that have an unbounded size or length. To prevent spending too much time on one small area of the program, the search may be depth-limited (bounded). Symbolic execution and automated theorem provers have limitations on the classes of constraints they can represent and solve. For example, a theorem prover based on linear arithmetic will be unable to cope with the nonlinear path condition xy = 6. Any time that such constraints arise, the symbolic execution may substitute the current concrete value of one of the variables to simplify the problem. An important part of the design of a concolic testing system is selecting a symbolic representation precise enough to represent the constraints of interest.

- **Concolic Execution - Properties**
  a) Quick path exploration
     - Concrete input values
     - We only need to observe the execution
  b) Solver called only to get new inputs
     - Much less calls in the initial stages when compared to symbolic execution
  c) Incomplete
     - New path != new state
     - Often diverges in presence of loops and/or concurrency
     - Heuristics for branch choice are important
     - Branches feasibility depends on the trace prefix
  d) Copes well with uninterruptable stuff
     - If parts of code ignored, we still get some input for the next test

- **Concolic Execution - Tools**
  a) Pex
     - MS Research
     - .NET platform, Visual studio integration
  b) Sage
     - MS Research
     - Machine code, Visual studio integration
  c) AgitarOne
     - Agitar technologies
     - Java, Eclipse integration
  d) ExpliSAT
     - IBM, C

3.0 Language Containment

There are properties of practical interest that cannot be described in CTL. An example is the ``almost always" property: a condition, q, always holds after a finite number of transitions (note that formulas $FG\,q$ and $AF\,G\,q$ would express this, but these are not legal CTL formulas). This property looks a lot like $AF\,AG\,q$, but it is not the same. One can exhibit a transition system where $AF\,G\,q$ is true, while $AF\,AG\,q$ is false.

A solution would be to use a more expressive type of temporal logic (for instance, the previous property could be expressed in PLTL or CTL*). But there would be drawbacks, such as the higher complexity of algorithms for model checking. An alternative is to use another verification paradigm, called language containment, based on the theory of ω-automata. For example, it is easy to express the previous ``almost always" property using an automaton. [5]

Currently VIS supports a restricted form of language containment. We review briefly the idea of language containment: for a system to satisfy a property it must be that $L(S) \subseteq L(T)$, where S is an ω-automaton representing the system, T is an ω-automaton representing the property and L is the language accepted by the automaton. It is a fact that $L(S) \subseteq L(T)$ is equivalent to $L(S) \cap L(T) = \emptyset$

To achieve language containment checking we represent the composition of the given system with a model representing the negation of the property and check it for language emptiness. The language of the composed system is empty if and only if the system satisfies the property T.

Language emptiness is used not only to verify properties that cannot be expressed in Fair CTL, but also to check whether the abstraction of a system still contains the original system. In both cases one must complement an ω-automaton (T), and this is hard to do if the automaton is nondeterministic (as is usually the case for an abstraction). The fact that complementation of a deterministic property is easy, while complementation of a nondeterministic property may be hard, is a key problem with language containment. This has prompted a lot of research on different classes of ω-automata with different expressiveness and difficulty of complementation. Currently VIS supports language emptiness of nondeterministic Büchi automata; only it is the responsibility of the user to derive the complement of a given nondeterministic property. Büchi automata acceptance conditions are states that must be reached infinitely often and they are specified by means of fairness constraints. Thus to use language containment, the user must insert in the Verilog

hierarchy a monitor, which represents the complement automaton structure, and impose a set of fairness conditions to specify the complement automaton acceptance conditions, i.e., the acceptance conditions are specified in terms of fair paths.

As a final note, inside VIS, language emptiness (language containment) is reduced to CTL, by checking the CTL formula $EG\,true$ on the system (system composed with complemented property), i.e., whether there is an infinite path (notice that $true$ is always satisfied), satisfying appropriate fairness constraints.

4.0 Advantages of Formal Verification

- Reduce vector set for either dynamic or random simulations.
- Formal verification is vector less and checks assumptions made by designers.
- Smaller vector set for corner cases are easier to detect at block level and bugs can be fixed prior to SoC integration.
- Automatic generation of testbench.
- Bugs found at Post layout due to incorrect timing constraints are detected early. Example: Multi cycle paths and False Paths.
- Validates constraints from designers.
- Synthesis constraints are input to Formal Verification.
- Helps in any invalid timing constraints for IP by the designers.
- Helps in reducing post layout debug time when IP is integrated in SoC.

5.0 Challenges of Using Formal Techniques on Large Designs

- Apply constraints for intermodular signals (Not tried it yet.) and see if violations show up.
- Check for tied ports in the design and correlate with toggle coverage tools.
- Complex SoC vector initialization is required.
- Analog and Memory behavioural model checks are not fully available in EDA tools currently.
- Cannot use for multiple functional modes. Separate initialization sequences are required.

6.0 Conclusion

- Formal Verification is a must for quality verification.
- Efficient strategy is required to plan the flow for both block and full chip.
- Tools must be updated to improve the engines internally to handle larger designs and have the capability of changing the user defined rules dynamically.
- Model checking is a general approach and is applied in areas like hardware verification and software engineering. Due to its success in several projects and the high degree of support from tools, there is an increasing interest in industry in model checking various companies have started their own research groups and sometimes have developed their own in house model checkers.

References

[1] Formal Verification, B Meenakshi. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Formal Methods used in Software Verification, Robert T. Bauer IBM/SWG/Rational/ eavertonrobert.bauer@acm.org.

[3] Bounded Model Checking, Armin Biere1 Alessandro Cimatti2 Edmund M. Clarke3 Ofer Strichman3 Yunshan Zhu4? 1 Institute of Computer Systems, ETH Zurich, 8092 Zurich, Switzerland.

[4] Formal verification techniques, Rekha Bangalore,March 2007.

[5] Model Checking, mark reith,CS 7123Research seminar 22Oct 2007.

[6] Language Containment Wikipedia

[7] Java PathFinder (JPF), http://javapathfinder.sourceforge.net

