CS569

A Survey Report on Formal Software Verification Methods

By knowing following terms, one can understand the title of the paper and what the paper is about.

Software Verification: Software verification is to assure that software fully satisfies all the expected requirements. It has two types: Static Analysis and Dynamic Analysis. We mainly concentrate about Static Analysis.

Static Analysis: Static verification is the process of checking its correctness by inspecting the code before it runs. Whereas Dynamic verification is by running the code and testing (Usually testing procedure). One of the methods of Static Analysis is Formal Verification.

Formal Verification: Formal verification is the act of proving or disproving the correctness of program with respect to a certain formal specification or property. In simple words, proving correctness of a program. One of the approaches for Formal Verification is Model Checking.

Model Checking: Given a program, exploring all states and transitions in the program, by using smart and domain-specific abstraction techniques to consider whole groups of states in a single operation and reduce computing time.

The four techniques I surveyed are:

1. Abstract Static Analysis
2. Model Checking
3. Bounded Model Checking
4. Explicit State Model Checking

Abstract static analysis is used in software development tools for pointer analysis in modern compilers. A formal basis for such techniques is Abstract Interpretation. Model Checking is to determine if a correctness property holds by exhaustively exploring the reachable states of a program. If the property does not hold, the model checking algorithm generates a counterexample, an execution trace leading to a state in which the property is violated. As the state space of software programs is typically too large to be analyzed explicitly, model checking is often combined with abstraction techniques like predicate abstraction. Bounded Model Checking explores program behavior exhaustively, but only up to a given depth. Bugs that require longer paths are missed.

1. Abstract Static Analysis:

Static analysis techniques typically propagate a set of values through a program until the set does not change with further propagation. The program flow is determined by a CFG with each step or track of variable in form of states and interpreting the flow of program and changes of variable. The set of possible values of a variable is known as concrete domain and the procedure

is known concrete interpretation which is impossible because we end up in getting rapidly increasing states which are not scalable. So, they have different analysis methods based on your program: Flow Sensitive, Path sensitive and context sensitive. The names explain what they are.

An abstract domain is a set of concrete values. An abstraction function is used to map concrete values to abstract ones. An abstract interpretation involves evaluating the behavior of a program on an abstract domain to obtain an approximate solution. With abstract interpretation, an application can be tested into a domain of abstract values. With this, we obtain an abstract version of the application with abstract data. However, the abstract analysis may still require an infeasible number of iterations to reach a fixed point.

Along with this, they discussed a lot of stuff about Value-set Analysis, Shape Analysis, and Difference Bound Matrices, Which I didn't understand at all. In Simple words, Abstract Interpolation constructs the new state within range of specified domain and then we estimate the state of an intermediate value of independent variable.

The first Static analysis tool was LINT and later several modern tools like FINDBUGS, CODE-SONAR, K7, PREVENT, CGS were developed by extending LINT.

Most analyzers can be used to analyze large software systems with minimal user interaction. So, they are robust, can handle large inputs and are efficient. One advantage of this technique is that it is possible to know if the results of the program could be obtained in a finite time. But the properties that can be proved are often simple and are usually hard coded into the tools, for example, ensuring that array bounds are not exceeded or that arithmetic overflows do not occur and for assertion violations. The information obtained in the analysis is always an approximation of what really occurs in a program execution. Early static analyzers produced just warnings and hence no one used. Recent tools include options for the user to control the quality or format of the output and to specify which properties must be analyzed. Simple abstract domains and analyzes, which disregard program structure may be useful for compiler optimization but rarely sufficient for verification.

2. Model Checking:

This might be an interesting topic, because I have some practical experience on how it works. Using model checking we check if model of a system satisfies a correctness specification or not. A model of a program means a set of states, transitions between states and some specifications or a property. A state contains of the values of all program variables, and the configurations of the stack. Transitions show how a program goes from one state to another. In Model checking, we exhaustively examine the reachable states of a program. This procedure will terminate if the state space is finite. If a state violating a property is found, It returns a counter example which is an execution trace of program showing where the error is occurred or property is failed.

Model checking tools verify safety properties like unreachability of bad states (assertion violation, null pointer dereference or buffer overflow) and Liveness properties like something good eventually happens, like the condition that a program must eventually terminate.

Partial Order Reductions:
Allowing all possible orderings is a potential cause of the state explosion problem. The partial order reduction is aimed at reducing the size of the state space that needs to be searched. It exploits the commutativity of concurrently executed transitions, which result in the same state. Thus, this reduction technique is best suited for asynchronous systems. (In synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved.). We construct a reduced state graph to reduce the number of states while preserving the correctness of the property. We use Kripke Structures to represent them.

We have mainly two types of Algorithms for State exploration in Model checking: Symbolic Model checking and Explicit Model Checking. Explicit-state model checking algorithms directly index states, and use graph algorithms to explore the state space, starting from the initial states. Symbolic model checking algorithms use implicit representations of sets of states and may start from the initial states, error states, or the property. Let's see both in Detail.

2.1 Symbolic Model Checking:

Many of Model checking tools talk a lot about propositional Logic, Kripke Models, Buchi Automata, CFL CNF, e.t.c. For Better Understanding, You can know more about them here: http://www.springer.com/cda/content/document/cda_downloaddocument/9780387341552-c2.pdf?SGWID=0-0-45-321359-p173660384

Symbolic model checking methods represent set of states, rather than enumerating individual states using BDDs (Binary Decision Diagrams) and propositional logic for finite sets and finite automata for infinite sets. A BDD is obtained from a Boolean decision tree by maximally sharing nodes and eliminating redundant nodes. However, BDDs grow very large. The issues in using finite automata for infinite sets are analogous. Symbolic representations such as propositional logic formulas are more memory efficient, at the cost of computation time. Symbolic techniques work well for proving correctness and handling state-space explosion due to program variables and data types.

Few people have proposed Symbolic Model checking without using BDDs by using SAT and ACTL & ECTL which are some combinations of Temporal Logic. Some Symbolic Model checking tools available are: SMV, NuSMV 2 and PRISM. PRISM is a hybrid version which uses both Probabilistic and Symbolic Model checking.

I tried of looking into Some Disadvantages of Symbolic Model checker, but I think Symbolic and Explicit are two approaches towards solving state explosion problem right? But, not sure if these approaches really have any pros and cons to overlook.

2.2 Explicit State Model Checking:

Explicit state methods construct a state transition graph by recursively generating successors of initial states. The graph may be constructed in a depth-first, breadth-first, or heuristic manner. New states are checked for a property violation on the-fly, so that errors can be detected without building the entire graph. Explored states are compressed and stored in a hash table to avoid re-

computing their successors. If the available memory is insufficient, lossy compression methods can be used. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure, and property validation amounts to a partial or complete exploration of the state space.

The mainly used tools for Model checking are SPIN, which is an Explicit state Model Checker for C and JPF (Java Path Finder which is an Explicit State Model checker for Java Byte code.
SPIN supports a high level language to specify systems descriptions called PROMELA. Concurrency in SPIN is interpreted using an interleaving approach. Properties can be specified in various ways. To express safety properties, the Promela code can be augmented with assertions or deadlock state characterizations. In order to express liveness properties, Promela models can be extended by never claims. Spin can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Buchi automata.

Symbolic model checking methods represent sets of states, rather than enumerating individual states. In contrast, explicit state methods at the time scaled to a few thousand states. Symbolic techniques work well for proving correctness and handling state-space explosion due to program variables and data types. Explicit-state techniques are well suited to error detection and handling concurrency.

2.3 Predicate Abstraction:

An orthogonal approach to counter state-space explosion is abstraction. It is sufficient to analyze a sound abstraction of the program, with a smaller state space. Abstractions were manually constructed, but are constructed automatically in recent tools.
Predicate abstraction (used by SLAM) is currently the predominant abstraction technique in software model checking in which we use logical predicates to construct an abstract domain by partitioning a program's state space. This process differs from standard abstract interpretation because the abstraction is parametrized by, and specific to a program. The challenge in predicate abstraction is identifying predicates, since they determine the accuracy of the abstraction. In counterexample-guided abstraction refinement (CEGAR), if model checking the abstraction yields a counterexample, which does not exist in the concrete program, the abstract counterexample is used to identify new predicates, and obtain a more precise abstraction.

The success of predicate abstraction for software model checking was initiated by the SLAM. SLAM checks a set of about 30 predefined, system specific properties and comprises the predicate abstraction tool C2BP which is a BDD-based model checker BEBOP for Boolean programs. The tool BLAST uses a lazy abstraction approach: The refinement step triggers the re-abstraction of only the relevant parts of the original program. Unlike SLAM, BLAST uses Craig interpolation to derive refinement predicates from counterexamples. SATABS uses a SAT-solver instead of general purpose theorem provers to generate the abstraction and to simulate the counterexamples. The bit-level accurate representation of C programs makes it possible to model

arithmetic overflow, arrays and strings. SATABS automatically generates and checks proof conditions for array bound violations, invalid pointer dereferencing, division by zero, and assertions provided by the user. SATABS uses a SAT-based model checker BOPPO to compute the reachable states of the abstract program. To deal with concurrency, BOPPO combines symbolic model checking with partial order reduction. Unfortunately, this approach still has severe scalability issues

In practice, the counterexamples provided by model checkers are often of more value than a proof of correctness. Predicate abstraction in combination with CEGAR is suitable for checking control-flow related safety properties. Though no false positives are reported, the abstraction-refinement cycle may not terminate. Furthermore, the success of the approach depends crucially on the refinement step: Many existing refinement heuristics may yield a diverging set of predicates. The main field of application is currently verification of safety properties of device drivers and systems code. Predicate abstraction does not work well in the presence of complex heap-based data structures or arrays.

SATABS is a predicate abstraction tool using SAT which transforms a C/C++ program into a Boolean program, which is an abstraction of the original program in order to handle large amounts of code. The Boolean program is then passed to a Model Checker. Predicate abstraction is best suited for lightweight properties such as array bounds (buffer overflows), pointer safety, exceptions and control-flow oriented user-specified assertions.

3. Bounded Model Checking:

Bounded model checking (BMC) is one of the most commonly applied formal verification techniques in the embedded software. In BMC, the design under verification is unwound ‘k’ times together with a property to form a propositional formula, which is then passed to the SAT solver. The formula is satisfiable if and only if there is a trace of length ‘k’ that refuses the property. The technique is inconclusive if the formula is unsatisfiable, as there may be counterexamples longer than ‘k’ steps.

In each transition, the next program location is computed by following the control flow graph of the program. The basic block is converted into a formula by transforming it into Static Single Assignment (SSA) form. The arithmetic operators in the basic block are converted into simplistic circuit equivalents. Arrays and pointers are treated as in the case of memories in hardware verification: a large case split over the possible values of the address is encoded. Such an unwinding, when done with k steps, permits exploring all program paths of length ‘k’ or less. The size of this basic unwinding is ‘k’ times the size of the program. For large programs, this is prohibitive, and thus, several optimizations have been proposed. The first step is to perform an analysis of the control flow that is possible.

This motivated the idea of loop unwinding. Instead of unwinding the entire transition relation, each loop is unwound separately. Syntactically, this corresponds to a replication of the loop body together with an appropriate guard. This kind of unwinding may result in more compact formulas. It also requires fewer case-splits in the formula, as there are fewer successors for each time-frame. As a disadvantage, the loop based unwinding may require more time-frames in order

to reach a given depth. The CBMC tool first guesses a bound, and then checks an assertion that is violated by paths that exceed this bound. This assertion is called an unwinding assertion.

There are several advantages of bounded model checking. They do not require exponential space and large designs can be checked very fast, since the state space is searched in an arbitrary order. BDD based model checking usually operates in breadth first search consuming much more memory. Further, the procedure is able to find paths of minimal length, which helps the user understand the examples that are generated. Lastly, the SAT tools generally need far less by hand manipulation than BDDs.

There are few disadvantages to bounded model checking. It has so far only been used for specifications where fix point operations are easy to avoid. Additionally the method as applied is generally not complete, meaning one cannot be guaranteed a true or false determination for every specification. However, even with these disadvantages, the advantages of the method make it a valuable complement to existing verification techniques. It is able to find bugs and sometimes to determine correctness, in situations where other techniques fail completely.

BMC is the best technique to find shallow bugs, and it provides a full counterexample trace in case a bug is found. It supports the widest range of program constructions. This includes dynamically allocated data structures; for this, BMC does not require built-in knowledge about the data structures the program maintains. On the other hand, completeness is only obtainable on very ’shallow’ programs, i.e., programs without deep loops.

There are lot of tools that implement BMC: CBMC (Implements loop unrolling), F-SOFT (Implements unwinding of entire transition system), EBMC (For Hardware designs), LLBMC (For C++), TCBMC (for threaded C programs), JCBMC (for concurrent Java programs).

I am not talking much about CBMC because we already discussed about it in Assignment 1.

Summary:

I surveyed three main techniques for automatic formal verification of software. Static analysis techniques based on abstract interpretation scale well at the cost of limited precision, which manifests itself in a possibly large number of false warnings. The tool support is mature.
Model Checking tools that use abstraction can track invariants with complex Boolean structure, and are able to generate counterexamples. Bounded Model Checkers are very strong at detecting shallow bugs, but are unable to prove even simple properties if the program contains deep loops. The model checking-based tools for software are less robust than static analysis tools and the market for these tools is high.

The properties that can be proved by Abstract Static Analysis are often simple and are usually hard coded into the tools. The information obtained in the analysis is always an approximation of what really occurs in a program execution. Recent static analysis tools include options for the user to control the quality or format of the output and to specify which properties must be analyzed. Simple abstract domains and analyzes, which disregard program structure may be

useful for compiler optimization but rarely sufficient for verification. Symbolic techniques work well for proving correctness and handling state-space explosion due to program variables and data types. Explicit-state techniques are well suited to error detection and handling concurrency. Predicate abstraction is best suited for lightweight properties such as array bounds (buffer over-flows), pointer safety, exceptions and control-flow oriented user-specified assertions. But, It has limited problem size, long run times, and large memory consumption. BMC is generally not complete, meaning one cannot be guaranteed a true or false determination for every specification. And Finding exact bug location using BMC is also a bit difficult. So, We add some fault localization techniques to BMC. Concurrency has been subject of research for decades, few tools like TCBMMC were developed to verify concurrent programs but the few tools that analyze programs with shared-variable concurrency still scale poorly.

I didn't talk much about Propositional languages like CTL, LTL and automata and other stuff because, they don't make any sense unless you know stuff about Theory of computation and compiler language. The main outcome from this report is Why verification, Why this many methods, which one to use when? Which one performs better over other and what they can't solve?

Submitted By:
Lakshman Madhav Kollipara
For CS569 – Static Analysis & Model Checking (Survey Project)