Journal of Software Engineering and Applications, 2023, 16, 170-192
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Review of Software Model-Checking Techniques for Dealing with Error Detection in Program Codes

Ednah Olubunmi Aliyu

Department of Computer Science, Adekunle Ajasin University, Akungba-Akoko, Nigeria
Email: Olubunmi.aliyu@aaua.edu.ng

How to cite this paper: Aliyu, E.O. (2023) Review of Software Model-Checking Techniques for Dealing with Error Detection in Program Codes. *Journal of Software Engineering and Applications*, **16**, 170-192.
https://doi.org/10.4236/jsea.2023.166010

Received: April 17, 2023
Accepted: June 25, 2023
Published: June 28, 2023

Abstract

Debugging software code has been a challenge for software developers since the early days of computer programming. A simple need, because the world is run by software. So perhaps the biggest engineering challenge is finding ways to make software more reliable. This review provides an overview of techniques developed over time in the field of software model checking to solve the problem of detecting errors in program code. In addition, the challenges posed by this technology are discussed and ways to mitigate them in future research and applications are proposed. A comprehensive examination of the various model verification methods used to detect program code errors is intended to lay the foundation for future research in this area.

Keywords

Software Model Checking, Symbolic Execution, State Explosion, Abstraction, Test Case Generations

1. Introduction

This article traced the history of program verification to the mid-sixties [1], when the founding fathers of Computer Science such as Robert Floyd, Tony Hoare and Edsger Dijkstra realized that one of the most important challenges computing faced was to tame complexity. So, they devised a methodology to study the property of a system. One way to study the properties of a program is to treat it as a black box and execute it. The merit of this approach is its ease of implementation while its drawback is that it can only reveal how the program behaves for a particular input value. To guarantee that some property is held on a different input, one would have to execute the program over every possible input. To solve

the problem, the research community decided to open up the black box. This led to the development of higher-level programming languages with precise semantics using mathematical objects to reason about program properties such as operational semantics, denotational semantics and axiomatic semantics as pointed out by [2]. Having laid down the foundation for describing program semantics, researchers began to follow two distinct but overlapping paths. The first was the design of a programming language of increasing sophistication, which promoted various styles of programming, while the second was the development of techniques for reasoning about the properties of programs.

Meanwhile, the Peer Review method used for software verification [3] is a method carried out by a team of software engineers who preferably have not been involved in the development of the software under review. They analyse code statically without executing it. The shortcoming of this method is that the errors discovered are errors in coding standards, style and readability, but does not display errors in program behaviour. Another verification method is testing. It is a dynamic technique that runs the software. Testing takes the piece of software under consideration and provides its compiled code with an input called tests. The main advantage of testing is that it can be applied to all sorts of software ranging from application software, to compiler and operating systems. However, a problem with testing according to [3] and [4] is that testing can never be complete, that is, it can only show the presence of errors, and it does not show the absence of errors. Also, determining when the testing operation is to stop is a challenge. Due to this problem, [5] stated that early discovery of errors makes development cheaper.

Static code analysis is another method of debugging by examining code before a program is run. It is implemented by analyzing a set of codes against a set of coding rules using automatic tools. [6] mentioned some static analysis tools that have emerged over the years such as Coverity, FindBugs, C++Test and JTest respectively, enabling common programming errors and style violations to be seen in the early stages of development. The two major challenges of static code analysis according to [7] are false positives and false negatives. False positive indicates the presence of a defect or vulnerability that is not actually present in the code while false negatives indicate that the code is free from defects or vulnerabilities when, in fact, it is not.

In the quest of making progress in developing tools for verifying requirements and design, the techniques organized themselves into distinct sub-areas under the general heading of formal methods such as: *Type checking* which define a set of values to be assigned to a variable, the operations that can be performed on a variable, the way a variable is stored in the memory and done automatically within the compiler [8]. *Program analysis*: A technology that is concerned with approximation about the program automatically, detecting invariants in a program and run-time errors such as division by zero, array bound overflow and arithmetic overflow.

Theorem prover: Technology for simplifying proof in response to proofs

by hand using a Floyd-Hoare style logic [9]. This discipline concentrates on the construction of models by successive refinements, symbolic techniques in the performance of constraint solver that underlie effective symbolic representation for propositional logic, as well as Binary Decision Diagram (BDD) and for the combination of first-order theories of equality with un-interpreted function [10] [11]. *Model Checking*: Checks the properties of models of program, usually, these models denote finite state machines and the properties to be checked are temporal properties [8]. Also, Ben-Ari [12] describes model checking as one of the most powerful formal methods that generate all possible states of a program and check whether the correctness specifications hold in each state.

This article focuses on model checking and the application of model checking involves the following phases according to [13].

1) First, it is needed to formally specify the system to check, written in a formal language such as temporal logic or process algebra.

2) The requirements of the system are also specified in the same language as the specification or in a different one.

3) A model of the system is generated from the formal specification usually, finite state automata or a labelled transition system.

4) Finally, the satisfaction or refutation of the properties is checked against the model.

However, each sub-area has its own community and philosophy. But what unifies this sub-area is that to prove the program never enters an undesirable state. To prove that, one has to compute from the program's description the set of possible states the program may ever enter. [1] documented that this set is not computable, what needs to be done is to demonstrate by construction the existence of a safe set that contains all the states the program may enter, but does not contain an undesirable state. From this point on, the sub-areas diverged in the kinds of error states and the programs being analyzed as shown in **Table 1**.

This review is based on software model checking which is the algorithmic analysis of programs to prove properties of their execution. More recently, software model checking has been influenced by three parallel but somewhat distinct developments. First, the development of program logic and associated decision procedures, this provided a framework and basic algorithmic tools to reason about infinite state spaces. Second, automatic model-checking techniques for temporal logic, this 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 [1]. While the second categories focus on symbolic model checking which has been an active area of research in the field of computer science for several decades, this study intends to review the state-of-the-art of symbolic model checking, especially the use of machine learning techniques to improve the scalability of symbolic model checking.

The remaining part of this review is organized as follows. Section 2 contains software model-checking techniques for dealing with error detection in program

Table 1. Features of formal methods.

Distinguishing Features of Sub-area	Type System	Program Analysis	Theorem Prover	Model Checking
Program Code	Applicable	Applicable	Not applicable	Not applicable
Model (Abstraction)	Not applicable	Control-Flow Graph (CFG)	Model construction	Finite-State Machine (FSM)
Type Checking	Applicable	Not applicable	Not applicable	Not applicable
Invariant Checking	Not applicable	Applicable	Applicable	Applicable
Functional Correctness	Not applicable	Applicable	Not applicable	Not applicable
Temporal Properties	Not applicable	Not applicable	Not applicable	Applicable
Symbolic Execution	Not applicable	Not applicable	Applicable	Applicable
Precision	Not applicable	Not applicable	Applicable	Applicable
Scalability	Not applicable	Applicable	Applicable	Not applicable
Exhaustive	Not applicable	Applicable	Applicable	Applicable
Efficiency	Not applicable	Applicable	Not applicable	Not applicable
Undesirable State	Type error	Spurious error	Invalid query	Spurious/coarse abstraction
Counter-example	Not applicable	Not applicable	Applicable	Applicable
Refinement	Not applicable	Not applicable	Applicable	Applicable
Proposed Properties	Not applicable	Applicable	Not applicable	Applicable
Inner Properties	Applicable	Not applicable	Applicable	Not applicable

codes. Section 3 analyse the situation of the symbolic model checking. Section 4 contains a discussion of challenges and future prospects. Section 5 presents the conclusion.

2. Software Model-Checking Techniques for Dealing with Error Detection in Program Codes

Several notions have emerged in the field of software model checking to deal with properties of a system. Typically, software model checking was based on explicit enumeration. That is, it manipulates individual states of the program that are stored in a large hash table (that is, states are stored as bit vectors). The state space graph is explored with depth-first search, looking for violation of safety property. An overview of general approach is presented in Figure 1.

In the literature, different approaches that brought enhancement in the field of software model checking are depicted in Table 2.

2.1. Explicit-State Model Checking

Explicit-state model checking is a formal verification technique that checks whether a system model satisfies a desired property by exploring all possible states of the system. One significant limitation of explicit-state model checking is the state explosion problem, which occurs when the number of possible states or transitions in the system grows exponentially with the size of the system. This

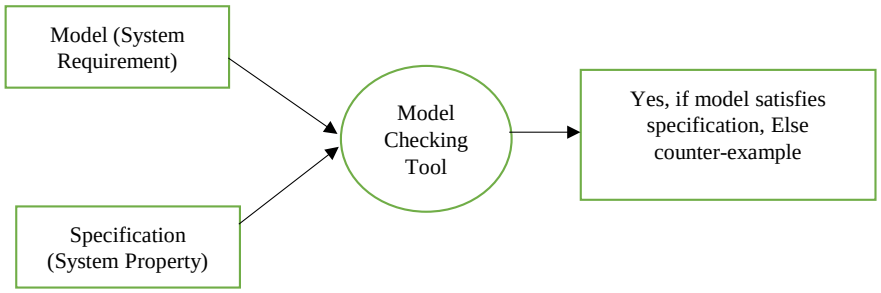

Figure 1. Model-checking approach.

Table 2. Different approaches in software model checking.

Approaches	Knowledge Source	Strength	Limitation
Explicit-state Model Checking	State graph or state transition system	Detect errors in highly concurrent design	State explosion problem.
Symbolic Model Checking (Boolean Encoding for State Machine and Set of States)	Ordered Binary Decision Diagram (OBDD) + Fix-point characterization of temporal operators	1) It can handle systems with large state spaces more efficiently. 2) It can verify properties that are specified in rich formal languages, such as temporal logic or mu-calculus spaces more efficiently. 3) It can verify systems that have unbounded data domains, such as integer and real variables. 4) It can verify parameterized systems, where the number of components or processes in the system is not fixed.	Many functions do not have a succinct BDD representation.
Model Checking with Abstraction	Predicate abstraction	Ability to balance the trade-off between accuracy and efficiency in formal verification.	1) Risk of losing important information or details that are necessary for detecting certain types of errors or properties. 2) Difficulty of ensuring the soundness and completeness of the verification results.
Bounded Model Checking	Formal language such as temporal logic or automata, and the design or implementation of the system being verified	1) Effectively handle systems with a large number of states and transitions. 2) Detect errors and violations quickly, without having to explore the entire state space of the system.	1) Cannot detect errors or violations that occur beyond the bound. 2) If the model does not accurately capture the behavior of the system, BMC may produce false negatives or false positives, which can lead to incorrect verification results. 3) State explosion problem.
Dynamic Model Checking	Program code and its associated documentation, such as requirements and design specifications	1) It can handle systems with dynamic or continuous behavior, such as control systems, analog circuits, and hybrid systems. 2) It detects errors and violations in real-time during system execution.	1) It only verifies the behavior of the system during execution. 2) It requires a set of inputs that exercise the system's behavior adequately. 3) It suffers from the state explosion problem.

can make explicit-state model checking infeasible or impractical for verifying large and complex systems.

In the sequel, various techniques to improve space consumption in explicit state model checkers were introduced such as: Reduction based techniques—consists of partial-order reduction, symmetry reduction or minimization based on behavioural equivalences such as simulation or bi-simulation, behavioural equivalences such as similarity or bi-similarity. Furthermore, compositional techniques were also introduced to reduce the safety verification problem on the original program to prove properties on subprograms, such that the results of model checking the subprograms can be combined to deduce the safety of the original program. Then, the idea of search heuristic in the field of artificial intelligence was introduced to find bugs quickly [1]. In addition, there was a special case of enumerative verification known as systematic execution exploration which checks the runtime system of a programming language implementation to implement enumerative state space exploration. [1] stated that the benefit of this approach was that it sidesteps the need to be able to formally represent the semantics of the programming language and machine instructions as a transition relation, but the number of states is still too large to be handled by the model checker.

In [9], a method of constructing concurrent program automatically using branching time temporal logic was proposed. They specified the behaviour of a finite number of fixed processes $p_1 \cdots p_n$ running in parallel using Computation Tree Logic (CTL) and applied Tableau-Based decision procedure for satisfiability of CTL formula. Model-checking algorithm (that is, Tableau-Based decision procedure) automatically factors out the synchronization skeletons of the individual processes from the global system flow-graph defined by the model.

Another study [14] described as a set of Communicating Sequential Processes (CSPs), where a translator transformed the description program into an Interpreted Petri-Nets (IPNs). To verify the conformity of the described system, the model was treated by an analyzer (CESAR) given the specification in branching time logic which expressed invariant, liveness and response to some action properties.

Also, [15] improved on the work carried out in [14] by developing automatic verification of finite-state concurrent systems using temporal logic specifications and moved fairness requirements into the semantic of Computation Tree Logic (CTL).

Model Checking Using Net Unfolding were proposed in [16] and made precise effort about the arbitrary interleaving of concurrent actions with certain properties such as reachability and mutual exclusion with the development of a new model-checking algorithm based on net unfolding to improve the model-checking algorithm developed by [17] which required exponential time.

In [18], presented the development of SPIN model checker to prove the correctness interaction between processes. The system model was described in Process

Metalanguage (PROMELA) for modelling process synchronization and coordination while specification properties are written in Linear Temporal Logic (LTL) formula.

2.2. Symbolic Model Checking

The need to overcome state-explosion-problem led to research on symbolic algorithms which manipulate representations of sets of states, rather than individual state, and perform state exploration through the symbolic transformation of these representations [1] [19]. One strength of symbolic model checking is that it can handle systems with large state spaces more efficiently than explicit-state model checking. The first to create a publicly available symbolic model checker called Symbolic Model Verifier (SMV) based on Binary Decision Diagrams (BDDs) was Ken McMillan, at Carnegie-Mellon University (CMU). The demerit of BDDs was its sensitivity to orderings of variables, that is, the order in which variables are encountered from the root to the leaves of the BDD can vary dramatically with different orderings.

The issue of scalability is addressed in [20] with explicit state model checker by extending Java Pathfinder model checker with symbolic execution and Satisfiability (SAT) solver.

In [21], looked into the issue of modeling complex data structures such as arrays with the development of a prototype implementation called SMT-CBMC approach.

Improving the scalability of the composite model-checking algorithms to verify source code level sequential programs is investigated in [22] against programming bug such as array bound violation, use of uninitialized variables, memory leaks, locking rule violations and division by zero with the objectives of combining multiple symbolic representation at Boolean, integer and Boolean combination of Linear constraints on reals to model programs defined using Composite Symbolic Formula (CSF).

Context-enhanced directed model-checking techniques introduced in [23] to tackle state explosion problem. The goal is to find error states in concurrent systems with the objectives of developing a heuristic search to explore those parts of the state space belonging to the same part of the system using the notion of interference context.

In [24], proposed Configurable Program Analysis Checker (CPACHECKER) with the development of Event Condition Action (ECA) systems using Binary Decision Diagrams (BDDs).

The issue of static verification techniques of Boolean formula satisfiability solvers were proposed in [25] such as, Satisfiability Theory (SAT) and Satisfiability Modulo Theory (SMT) solvers that operate on conjunctive normal form and first order logic formulae respectively to validate programs with the development of And-Inverter-Graph (AIG), encoding the problem definition in a program.

2.3. Model Checking with Abstraction

Model checking with abstraction is a technique that involves simplifying the formal model of a system being verified by abstracting away some details that are not relevant to the property being verified. Cousot [26] described abstract interpretation to be either partial or preorder. Being partial meaning that it is considered a subset of the concrete semantics. This is regarded as abstraction from below (Under-approximation). Preorder meaning that abstractions are "from above" that is, the abstract semantics describes a superset or logical consequence of the concrete semantics which is known as over-approximation.

One major improvement in this area was achieved through Satisfiability Modulo Theories (SMT); an algorithm that computes an abstraction of a program with respect to a given abstract interpretation by replacing loops and function calls in the Control Flow Graph (CFG) by their symbolic transformer [19] [27].

The work of [28] focused on model checking C programs, which are notoriously difficult to verify due to their complexity and the potential for errors in memory management. To address these challenges, the authors propose a method that uses both Boolean and Cartesian abstraction. Boolean abstraction involves replacing program variables with Boolean variables, which allows for efficient verification of properties that depend only on the truth values of the variables. Cartesian abstraction, on the other hand, involves partitioning the state space of the program into a finite set of Cartesian product abstractions, which can be used to efficiently verify complex temporal properties.

In [29], proposed an automatic method for generating predicate abstractions of C programs using a combination of abstract interpretation and decision tree learning. The authors first use abstract interpretation to generate a set of abstract states that over-approximate the behavior of the program. They then use decision tree learning to identify predicates that can be used to partition the abstract states into smaller sets that correspond to different program behaviors.

Predicate Abstraction for Software Verification by [30], proposed a method for automatically generating predicate abstractions using a combination of abstract interpretation and refinement The authors first use abstract interpretation to generate a set of abstract states that over-approximate the behavior of the program. They then use a refinement process to generate a more precise set of predicates that can be used to partition the abstract states into smaller sets that correspond to different program behaviors.

The paper "Model Checking Software at Compile Time" by [31] describes a novel approach to integrating model checking into the software development process at compile time, rather than after the software has been developed and compiled. It involves the automatic generation of a model of the software system at compile time, based on the source code of the program. This model is then used to perform model checking to verify the correctness of the software with respect to a given specification.

In [32], a methodology that combines automated verification techniques, such

as model checking, with automated repair techniques, such as program synthesis were presented. The proposed methodology consists of three main phases: verification, diagnosis, and repair. In the verification phase, the software is analyzed using model checking to identify any errors or bugs. In the diagnosis phase, the errors are further analyzed to determine the root cause of the problem. Finally, in the repair phase, a repair algorithm is used to automatically generate corrected code to fix the identified errors.

Symbolic abstraction algorithm called symba were introduced in [33] using the power of SMT solver (Z3), encoded all execution as a formula in Quantifier Free-Linear Real Arithmetic (QF-LRA), a commonly used theory for encoding program execution.

Model Checking with Abstraction by [34], proposed a method for verifying multi-agent systems using bounded model checking and abstraction techniques. The approach involves modeling the agents as finite-state machines, and using abstraction techniques to reduce the complexity of the model. The model is then checked for correctness using bounded model checking.

2.4. Bounded Model Checking

According to [19], satisfiability solver was observed to handle much larger formulas than BDDs. This led to the development of Bounded Model Checking (BMC) at Carnegie-Mellon University CMU in late 1990s. Bounded Model Checking (BMC) unrolls the control flow graph for a fixed number of steps, and checks if the error location can be reached within this number of steps and satisfiability of this constraint is checked by a constraint solver using backtracking.

However, the limitation of this technique according to [1] was the scalability performance of constraint solvers for non-linear arithmetic. Additionally, [35] stated that the disadvantage of bounded model checking is that the method lacks completeness and the types of properties that can currently be checked are very limited.

In [21], addressed the issue of modeling complex data structures such as arrays with the development of a prototype implementation called SMT-CBMC approach. Experimental results show that 1) encoding technique generated more compact formula than CBMC when arrays are involved in the input program; 2) On problems involving complex interactions of arithmetic and arrays manipulation, SMT-CBMC outperforms CBMC as the size of the arrays occurring in the input program increases.

Bounded Model Checking (BMC) for C and C++ programming languages were introduced in [36] using Low Level Virtual Machine (LLVM) complier front-end such as clang or llvm-gcc to translate C and C++ programs into LLVM's Intermediate Representation (LLVM-IR); usually in Static Single Assignment (SSA) form, LLVM-IR program is then converted into LLBMC's Internal Logical Representation (ILR). The result shown that LLBMC performed favourably compare to CBMC and ESBMC in terms of detecting an error such as arithmetic overflow,

invalid memory allocations, invalid memory access, invalid free and memory leak detection.

2.5. Dynamic Model Checking

Dynamic model checking is a technique for verifying the correctness of software systems by analyzing their behavior during execution. Unlike static analysis techniques, which analyze the source code of a program without actually running it, dynamic model checking involves running the program on sample inputs or test cases and observing its behavior to detect errors or violations of specified properties. As mentioned in "Model Checking" by [37], dynamic model checking can be particularly effective in detecting errors in complex systems where static analysis may be impractical or ineffective. Also, as discussed in "Model Checking Software: A Survey" by [11], dynamic model checking can handle systems with non-deterministic behaviour by generating and exploring all possible execution paths, and verifying that the properties of interest hold for each path.

Dynamic model checking by [38] proposed a new approach to model checking that involves dynamically executing the program under test and monitoring its behavior in real-time. This approach allows for more comprehensive testing of the software's behavioral correctness, as it can capture non-deterministic behavior and edge cases that may not be revealed by traditional model-checking techniques.

Dynamic model checking with evolving software and specifications by [39], proposed an approach to dynamic model checking that can handle evolving software and specifications. The authors' approach involves using a combination of dynamic model checking and dynamic symbolic execution. Dynamic model checking involves running the program under test and checking for violations of certain properties, while dynamic symbolic execution involves exploring the program's execution paths to generate test inputs that can achieve higher coverage. The authors demonstrate the effectiveness of their approach by applying it to several case studies, including a web application and a mobile phone application.

Dynamic model checking for concurrent software by [40] proposed a dynamic model-checking approach specifically for concurrent software. The authors' approach involves a combination of dynamic model checking and path-based symbolic execution. Dynamic model checking is used to explore the execution of the program and detect errors such as deadlocks and race conditions, while path-based symbolic execution is used to generate test cases that explore different paths through the program. The authors show the effectiveness of their approach by applying it to several case studies, including a file server and a memory allocator.

In [41], proposed a dynamic model-checking approach for software product lines. The authors' approach involves using a feature model to guide the testing process. The feature model captures the possible feature combinations of the soft-

ware product line, and the dynamic model-checking approach is used to explore the possible execution paths of the software products based on these feature combinations. The authors also use slicing techniques to reduce the number of paths explored during the testing process. The authors demonstrate the effectiveness of their approach by applying it to several case studies, including a file system and a configuration tool.

3. State-of-the-Art of Symbolic Model Checking

Symbolic model checking as mentioned in section two is a technique for verifying the correctness of software systems by analyzing a symbolic representation of the program's behavior. This technique analyzes a symbolic representation of the system's behavior and checks whether it satisfies a given set of properties [35]. The symbolic representation is constructed using a set of symbolic variables and their constraints, which allows for the exploration of all possible paths in the system's behavior without having to explicitly execute the system.

For example, suppose we want to verify a simple counter that counts from 0 to 3 and then resets to 0. We can represent the counter using a symbolic variable x, which takes values from 0 to 3.

We can define the constraints on x as follows:

If x is 0, then the counter increments to 1.

If x is 1, then the counter increments to 2.

If x is 2, then the counter increments to 3.

If x is 3, then the counter resets to 0.

We can encode these constraints as a Boolean formula using a temporal logic such as Linear Temporal Logic (LTL):

$$G\left(\left(x=0\rightarrow X\left(x=1\right)\right)\wedge\left(x=1\rightarrow X\left(x=2\right)\right)\wedge\left(x=2\rightarrow X\left(x=3\right)\right)\wedge\left(x=3\rightarrow X\left(x=0\right)\right)\right) \quad (1)$$

This formula states that for all future states G, if the current state is $x = 0$, then the next state must be $x = 1$, and so on. We can use a symbolic model checker to check whether this formula is satisfied by the symbolic representation of the counter. If the formula is satisfied, then we can conclude that the counter satisfies the given properties. If the formula is not satisfied, then we can use the counterexample generated by the symbolic model checker to diagnose and fix the bug in the counter.

Symbolic model checking is useful because it can handle complex systems with a large number of states and transitions. It can also find subtle bugs that may be difficult to detect through manual testing or other verification techniques. Additionally, symbolic model checking can be used to generate counterexamples, which are concrete executions of the system that violate a given property. These counterexamples can be used to diagnose and fix the bugs in the system.

However, it has been successfully applied to a variety of domains, including software, hardware, communication protocols, and distributed systems. Its purpose is to provide a rigorous and automated approach to verify the correctness of com-

plex systems.

3.1. Recent Advancement in Symbolic Model Checking

While Section 2 provides examples of recent research papers that have made significant contributions to the technique, this section discusses recent advancements in the technique, including new algorithms, tools, or applications as well as the technique's capabilities and limitations.

As mentioned above that symbolic model checking can be used in different domains, including software, hardware, communication protocols, and distributed systems. Here, a few examples of several new algorithms, tools, and applications of symbolic model checking that have been proposed for software verification and testing in recent years is as follows:

1) *The Symbolic Execution Intermediate Language* (*SEIL*): It is a programming language for specifying symbolic execution of software. It is a high-level language that can be used to describe symbolic execution of complex software systems. The SEIL tool was developed by [42] and can be used to generate test cases automatically.

2) *AFLSmart*: The AFLSmart tool is a mutation-based testing framework that uses symbolic execution to generate test cases. It is an extension of the AFL fuzzer and can handle complex software systems. The AFLSmart tool was proposed by [43] and has been shown to be effective in detecting subtle bugs in real-world software.

3) *JPF-SymSpark*: The Java PathFinder (JPF) tool is a symbolic model checker for Java programs. JPF-SymSpark is an extension of JPF that uses symbolic execution to generate test cases for Apache Spark applications. It was proposed by [44] and has been shown to be effective in detecting bugs in real-world Spark applications.

4) *Symbiotic*: Symbiotic is a tool for symbolic verification of C and C++ programs. It uses a combination of program slicing and symbolic execution to check for memory safety, concurrency bugs, and other types of bugs. Symbiotic was proposed by [45] and has been shown to be effective in finding bugs in large C and C++ codebases.

5) *SymDIVINE*: SymDIVINE is a tool for the verification of distributed software systems. It uses a combination of symbolic execution and model checking to check for safety and liveness properties. SymDIVINE was proposed by [46] and has been shown to be effective in detecting subtle bugs in distributed software systems.

Symbolic model checking is a powerful verification and testing method, but it also has advantages and disadvantages. Here are a few of them:

3.2. Capabilities of Symbolic Model Checking

1) *Automatic test case generation*: Symbolic model checking can automatically generate test cases that cover different execution paths and edge cases in the

software. This can help identify bugs and ensure that the software behaves correctly in all scenarios [47].

2) *Scalability*: Symbolic model checking can handle large and complex software systems with many possible execution paths and inputs. It can also handle programs with dynamic memory allocation and recursion, which are difficult to test with traditional methods [45].

3) *Precision*: Symbolic model checking can analyze the software behavior precisely and exhaustively. It can check for correctness properties such as absence of runtime errors, assertion violations, deadlocks, and data races [48].

3.3. Limitations of Symbolic Model Checking

1) *State space explosion*: Symbolic model checking suffers from the state space explosion problem, which arises when the number of possible program states and transitions is very large. This can cause the model checker to run out of memory or take a very long time to complete [17] [49].

2) *Incomplete specifications*: Symbolic model checking relies on formal specifications to check the correctness of the software. If the specification is incomplete or incorrect, the model checker may not be able to detect all the possible bugs in the software [50].

3) *Input generation*: Symbolic model checking can generate test cases automatically, but it may not be able to generate inputs that are representative of real-world usage scenarios. This can limit the effectiveness of the test cases in detecting bugs in the software [51].

4) *Limited support for non-determinism*: Symbolic model checking assumes that the program behavior is deterministic, but many software systems exhibit non-deterministic behavior. This can limit the applicability of symbolic model checking to certain types of software systems [24].

3.3.1. State Space Explosion

As mentioned above that the challenge of state space explosion in software model checking refers to the problem of dealing with a very large number of possible program states and transitions, which makes it difficult to verify the correctness of the software exhaustively. This problem arises because symbolic model-checking algorithms typically operate on a symbolic representation of the program that captures all possible executions of the program, rather than executing the program concretely.

To address the challenge of state space explosion, researchers have developed several techniques:

1) *Abstraction*: This involves reducing the size of the state space by ignoring some details of the program that are not relevant for verifying the desired property. For example, some variables can be abstracted away or combined into equivalence classes to reduce the number of possible states [52].

2) *Incremental checking*: This involves dividing the verification problem into smaller subproblems that can be checked separately. For example, the program

can be divided into modules or functions, and each module or function can be checked independently [53].

3) *Model checking with partial order reduction*: This technique reduces the size of the state space by exploiting the fact that many program executions are equivalent up to a certain point. By reducing the number of equivalent executions that are explored, this technique can reduce the overall size of the state space [54].

4) *Heuristics and optimizations*: Researchers have developed various heuristics and optimizations to improve the efficiency of symbolic model-checking algorithms. For example, techniques such as state caching and lazy evaluation can reduce the amount of redundant work that the model checker has to do [49].

5) *Symbolic model checking with machine learning techniques*: Machine learning techniques such as decision trees and regression models have been used to learn the structure of the system and guide the symbolic model-checking process. For example, in the work "Guiding Symbolic Execution towards Unexplored Code" by [55], a decision tree is used to guide symbolic execution towards unexplored code paths, reducing the number of explored paths and increasing the efficiency of the analysis.

Another example is the work "Speeding up Symbolic Model Checking with Machine Learning" by [56], where a regression model is used to predict the reachability of states in the system, reducing the number of states to be explored.

"DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems" by [57]. This work proposes a framework for testing deep learning systems using multi-granularity testing criteria. The approach uses machine learning to generate test cases and evaluate the testing criteria.

In [58], the authors propose a framework for learning-based testing that uses a variety of machine learning techniques to generate test cases, execute the test cases, and analyze the results of the test cases. The authors evaluate the framework on a case study of a real world IoT software and show that the framework can be used to find a significant number of defects in the system.

"Learning to Verify Safety Properties" by [59]. This work proposes a learning-based approach for verifying safety properties of software systems. The approach uses machine learning to learn the properties of the system under test and verify them against a set of safety properties.

Neural-Guided Deductive Search (NGDS) was proposed in [60]. The authors synthesize real-time programs from a small number of examples with significantly less time and space overhead than traditional deductive search techniques.

3.3.2. Incomplete Specifications

The challenge of incomplete specifications in software model checking refers to the problem of verifying software that does not have a complete specification, *i.e.* a complete and formal description of the expected behavior of the software. This problem arises because model checking algorithms typically rely on a formal specification of the desired behavior of the software in order to check whether

the actual behavior of the software satisfies the specification.

To address the challenge of incomplete specifications, researchers have developed several techniques:

1) *Abductive reasoning*: This involves inferring the missing parts of the specification based on the observed behavior of the software. For example, if the software always returns a certain value under certain conditions, we can infer that this behavior is part of the specification [48].

2) *Falsification*: This involves searching for inputs that cause the software to violate the specification. By finding counterexamples to the specification, we can identify parts of the specification that are missing or incorrect [61].

3) *Model-based testing*: This involves generating test cases based on a model of the software, rather than the software itself. By generating tests from the model, we can ensure that the tests cover all possible behaviors of the software that are relevant to the specification [62].

4) *Runtime verification*: This involves monitoring the behavior of the software during execution and checking whether the observed behavior satisfies the specification. This approach is useful when the software is too complex or too difficult to model accurately [63].

3.3.3. Input Generation

The challenge of input generation in software model checking refers to the problem of generating inputs that can trigger interesting behaviors in the software, such as error conditions or violations of the desired properties. This problem arises because model-checking algorithms typically rely on a set of inputs to explore the state space of the software and check whether it satisfies the desired properties.

To address the challenge of input generation, researchers have developed several techniques:

1) *Randomized testing*: This involves generating inputs randomly and testing the software with these inputs. By generating a large number of random inputs, we can increase the chances of finding interesting behaviors in the software [64].

2) *Search-based testing*: This involves using search algorithms to find inputs that are likely to trigger interesting behaviors in the software. By searching the space of possible inputs, we can find inputs that are more likely to uncover errors or violations of the desired properties [65].

3) *Symbolic execution*: This involves using a symbolic representation of the program to explore the space of possible inputs and generate inputs that satisfy certain conditions or constraints. By generating inputs symbolically, we can explore a large number of possible inputs without actually executing the software with each input [66].

4) *Model-based fuzzing*: This involves generating inputs based on a model of the software, rather than the software itself. By generating inputs from the model, we can ensure that the inputs cover all possible behaviors of the software that are relevant to the desired properties [67].

3.3.4. Limited Support for Non-Determinism

The challenge of limited support for non-determinism in software model checking refers to the problem of verifying software that exhibits non-deterministic behavior, where the behavior of the software depends on factors outside the control of the software, such as timing, external events, or user input. This problem arises because model-checking algorithms typically assume a deterministic model of the software, where the behavior of the software is fully determined by its inputs and the internal state.

To address the challenge of limited support for non-determinism, researchers have developed several techniques:

1) *Probabilistic model checking*: This involves extending model-checking algorithms to handle non-deterministic behavior by using probabilistic models, where the behavior of the software is described in terms of probabilities rather than deterministic transitions. By modeling non-determinism probabilistically, we can capture the inherent uncertainty in the behavior of the software [68].

2) *Runtime verification*: This involves monitoring the behavior of the software during execution and checking whether the observed behavior satisfies the desired properties. By monitoring the software at runtime, we can capture non-deterministic behavior that is difficult to model statically.

3) *Symbolic execution with concolic testing*: This involves using symbolic execution to explore the space of possible executions of the software, while also generating concrete inputs that satisfy certain conditions or constraints. By generating inputs that capture non-deterministic behavior, we can explore a larger space of possible behaviors than with purely symbolic execution [69].

4) *Model-based testing with adaptive input generation*: This involves generating inputs based on a model of the software, while also adapting the input generation process to capture non-deterministic behavior. By adapting the input generation process, we can explore a larger space of possible behaviors and capture non-deterministic behavior that is difficult to model statically [70].

4. Discussion: Challenges and Future Prospects

4.1. Summary of Challenges

From the current symbolic model-checking techniques and approaches identified, one interesting method that has been commonly applied to address the limitations is symbolic execution. Studies [71] [72] [73] show that many systems exhibit non-determinism behaviour. One such example is complex data structures that involve non-linear operations, such as matrices and polynomials.

Matrices, for example, can be represented as multi-dimensional arrays, and operations on matrices often involve non-linear operations such as matrix multiplication and matrix inversion. Symbolic execution can generate constraints on the values stored in matrix elements, but analyzing the constraints can be computationally expensive due to the non-linear nature of the operations.

Similarly, polynomials involve non-linear operations such as multiplication and

exponentiation, which can make symbolic execution of programs involving polynomials challenging. However, recent research Baldoni *et al.* (2020) has shown that techniques such as SMT-based constraint solving and approximation can be used to overcome some of the challenges of symbolic execution for non-linear operations.

It is stated in [74] that "Hybrid approaches can overcome the limitations of symbolic execution for non-linear operations by leveraging the strengths of other techniques" such as abstract interpretation or model checking.

However, one gap in using machine learning techniques with symbolic model checking is the potential loss of precision or soundness in the analysis. Machine learning models can be biased or incomplete, and may not capture all the relevant features or behaviors of the system under verification. This can lead to missed errors or false positives, which can be problematic for safety-critical systems.

For example, in the work "A Critical Survey of Machine Learning-Assisted Verification" by [75], the authors note that machine learning models can suffer from overfitting, where they memorize the training data rather than learning general patterns, leading to poor performance on new, unseen data. They also highlight the challenge of incorporating domain knowledge and constraints into machine learning models, which can affect their accuracy and usefulness for verification tasks.

Another gap is the potential increase in computational overhead and complexity when integrating machine learning with symbolic model checking. Machine learning models can be computationally expensive to train and evaluate, and may require large amounts of memory and storage. This can limit their scalability and practicality for real-world verification tasks.

4.2. Future Prospects

So far, software model-checking techniques for dealing with errors in software programs were introduced. Symbolic execution can be applied to a wide range of complex data structures but there are still some data structures involving non-linear operations that present challenges for symbolic execution. As mentioned by [74] that hybrid approach can be used to combat the problem, this study proposed dynamic symbolic execution which combines concrete execution with symbolic execution and concolic testing that combines symbolic execution with concrete testing to test cases that maximize code coverage.

Besides the above, recent studies [55] [56] [57] [76] [77] [78] demonstrated the use of machine learning for software model checking. However, further research and development in integrating machine learning with symbolic model checking, to ensure that the combined approach is accurate, scalable, and efficient needs more exploration.

In general, the challenges to model-checking software highlight the ongoing need for research and innovation in this field. As software systems continue to grow in complexity and importance, there will be a growing need for effective

and efficient verification techniques to ensure their correctness and reliability.

5. Conclusions

In this article, a review of recent techniques for error detection in program codes using model-checking technology was considered. First, an overview of the model-checking process and its importance in software verification is presented. Followed several techniques that have been developed to address some of the limitations of traditional model-checking techniques, including abstraction techniques, dynamic model checking, and model checking for software product lines.

Also, it highlighted some of the challenges and future prospects for model-checking software, including the state space explosion problem, incomplete and incorrect specifications, scalability, and the need to develop new algorithms and techniques to handle new types of software systems.

Based on this review of these techniques, model checking is a powerful and effective technique for detecting errors in program codes, but it also has some limitations that need to be addressed. However, with ongoing research and innovation, model checking will continue to be an important tool in the software verification process, helping to ensure the correctness and reliability of complex software systems.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Henzinger, T.A., Jhala, R., Majumdar, R. and Sutre G. (2002) Lazy Abstraction. *Proceedings of the* 29*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, 16-18 January 2002, 58-70. https://doi.org/10.1145/503272.503279

[2] Louden, K.C. (1993) Programming Languages Principles and Practice. URL.

[3] Baier, C. and Katoen, J. (2008) Principle of Model Checking. MIT Press, Cambridge.

[4] Aggarwal, K.K. and Yogesh, S. (2001) Software Engineering. 3rd Edition, New Age International Ltd, New Delhi.

[5] Van-Hung, D. (2005) Model-Checking and the SPIN Model Checker. International Institute for Software Technology, Macau.

[6] Mount, S. (2013) A Language-Independent Static Checking System for Coding Conventions. Ph.D. Thesis, University of Wolverhampton, Wolverhampton. https://www.semanticscholar.org/paper/A-language-independent-static-checking-system-for-Mount/a13e0e45b0c16ac8dda081666b9d2037c44e6b10

[7] Li, Y., Jiang, X., Zhang, Y., Xie, T. and Zhang, L. (2017) An Empirical Study on the limitations of Static Code Analysis for Vulnerability Detection. *IEEE Transactions on Software Engineering*, **43**, 462-477.

[8] Abrial, J. (2009) Modelling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge.

[9] Clarke, E.M. (2008) The Birth of Model Checking. In: Grumberg, O. and Veith, H.,

Eds., 25 *Years of Model Checking*, Springer, Berlin, 1-26.

[10] Strunk, E.A., Aiello, M.A. and Knight, J.C. (2006) A Survey of Tools for Model Checking and Model-Based Development. Technical Report, Department of Computer Science, University of Virginia, Charlottesville.

[11] Jhala, R. and Majumdar, R. (2009) Software Model Checking. *ACM Computing Surveys*, **41**, 1-54. https://doi.org/10.1145/1592434.1592438

[12] Ben-Ari, M. (2008) Principles of SPIN Model Checker. Springer-Verlag, London.

[13] Valero, M. (2005) Modal Abstraction and Replication of Processes with Data. Springer, Berlin.
https://pdfs.semanticscholar.org/804b/02fd88d199360b65fb6efe13e93e84428596.pdf

[14] Queille, J.P. and Sifakis, J. (1982) Specification and Verification of Concurrent System in CESAR. In: Dezani-Ciancaglini, M. and Montanari, U. Eds., *Programming* 1982: *International Symposium on Programming*, Springer, Berlin, 337-351.
https://link.springer.com/chapter/10.1007/3-540-11494-7_22
https://doi.org/10.1007/3-540-11494-7_22

[15] Clarke, E.M., Emerson, E.A. and Sistla, A.P. (1986) Automatic Verification of Finite-StateConcurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, **8**, 244-263.
https://doi.org/10.1145/5397.5399

[16] Esparza, J. (1994) Model Checking Using Net Unfolding. *Science of Computer Programming*, **23**, 151-195. https://doi.org/10.1016/0167-6423(94)00019-0

[17] McMillan, K.L. (1992) Symbolic Model Checking: An Approach to the State Explosion Problem. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, 1-212.

[18] Yin, X. (2006) An Introduction to the SPIN Model Checker. Tools for Model Checking and Model-Based Development Project Report: Tool Description and Analysis.

[19] Biere, A. (2008) Tutorial on Model Checking: Modelling and Verification in Computer Science. In: Horimoto, K., Regensburger, G., Rosenkranz, M. and Yoshida, H., Eds., *AB* 2008: *Algebraic Biology*, *Lecture Notes in Computer Science*, Springer, Berlin, 16-21. https://link.springer.com/chapter/10.1007/978-3-540-85101-1_2
https://doi.org/10.1007/978-3-540-85101-1_2

[20] Khurshid, S., Pasareanu, C.S. and Visser, W. (2003) Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H. and Hatcliff, J., Eds., *TACAS* 2003: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, 553-568. https://doi.org/10.1007/3-540-36577-X_40

[21] Armando, A., Mantovani, J. and Platania, L. (2006) Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. In: Valmari, A., Ed., *SPIN* 2006: *Model Checking Software*, Springer, Berlin, 146-162.
https://doi.org/10.1007/11691617_9

[22] Yang, Z., Wang, C., Gupta, A. and Ivančić, F. (2008) Model Checking Sequential Software Programs via Mixed Symbolic Analysis. *ACM Transaction on Design Automation of Electronic System*, **14**, 1-26. https://doi.org/10.1145/1455229.1455239

[23] Wehrle, M. and Kupferschmid, S. (2010) Context-Enhanced Directed Model Checking. In: Van De Pol, J. and Weber, M., Eds., *SPIN* 2010: *Model Checking Software*, Springer, Berlin, 88-105. https://doi.org/10.1007/978-3-642-16164-3_7
https://link.springer.com/chapter/10.1007/978-3-642-16164-3_7

[24] Eyer, D. and Stahlbauer, A. (2013) BDD-Based Software Model Checking with CPACHECKER. In: Kučera, A., Henzinger, T.A., Nešetřil, J., Vojnar, T. and Antoš, D., Eds., *MEMICS* 2012: *Mathematical and Engineering Methods in Computer*

Science, Springer, Berlin, 1-11. https://doi.org/10.1007/978-3-642-36046-6_1 https://link.springer.com/chapter/10.1007/978-3-642-36046-6_1

[25] Noureddine, M. and Zaraket, F.A. (2016) Model Checking Software with First Order Logic Specifications Using AIG Solvers. *IEEE Transactions on Software Engineering*, **42**, 741-763. https://scholar.google.com/citations?user=iLJ3TyQAAAAJ&hl=en https://doi.org/10.1109/TSE.2016.2520468

[26] Cousot, P. (2001) Abstract Interpretation Based Formal Methods and Future Challenge. https://www.di.ens.fr/~cousot/publications.www/Cousot-LNCS2000-sv-sb.pdf

[27] Abraham, E. and Kremer, G. (2017) Satisfiability Checking: Theory and Application. Springer International Publishing, New York.

[28] Ball, T., Podelski, A. and Rajamani, S.K. (2000) Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T. and Yi, W., Eds., *TACAS* 2001: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, 268-283. https://link.springer.com/chapter/10.1007/3-540-45319-9_19 https://doi.org/10.1007/3-540-45319-9_19

[29] Ball, T., Majumdar, R., Millstein, T. and Rajamani, S.K. (2001) Automatic Predicate Abstraction of C Programs. *Proceedings of the ACM SIGPLAN* 2001 *Conference on Programming Language Design and Implementation*, Snowbird, June 2001, 203-213. https://doi.org/10.1145/378795.378846

[30] Flanagan, C. and Qadeer, S. (2002) Predicate Abstraction for Software Verification. *Proceedings of the* 29*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming*, Portland, 16-18 January 2002, 191-202. https://doi.org/10.1145/503272.503291

[31] Fehnker, A., Brauer, J., Huuck, R. and Seefried, S. (2008) Goanna: Syntactic Software Model Checking. In: Cha, S., Choi, J.Y., Kim, M., Lee, I. and Viswanathan, M., Eds., *ATVA* 2008: *Automated Technology for Verification and Analysis*, Springer, Berlin, 216-221. https://link.springer.com/chapter/10.1007/978-3-540-88387-6_17

[32] Griesmayer, A. (2007) Debugging Software from Verification to Repair. Ph.D. Thesis, Graz University of Technology, Austria, 1-108

[33] Li, Y. (2014) Symbolic Abstraction with SMT Solvers. *Proceedings of the* 41*st ACM SIGPLAN-SIGACT Symposium on Principles of Programming*, San Diego, 22-24 January 2014, 607-618. https://doi.org/10.1145/2535838.2535857

[34] Lomuscio, A. and Michaliszyn, J. (2015) Verifying Multi-Agent Systems by Model Checking Three-valued Abstractions. *Proceedings of the* 2015 *International Conference on Autonomous Agents and Multiagent*, Istanbul, 4-8 May 2015, 189-198.

[35] Clarke, E., Biere, A., Rami, R. and Zhu, Y. (2001) Bounded Model Checking Using Satisfiability Solving. *Formal Method in System Design*, **19**, 7-34. https://doi.org/10.1023/A:1011276507260

[36] Merz, F., Falke, S. and Sinz, C. (2012) LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In: Joshi, R., Müller, P. and Podelski, A., Eds., *VSTTE* 2012: *Verified Software*: *Theories*, *Tools*, *Experiments*, Springer, Berlin, 146-161. https://doi.org/10.1007/978-3-642-27705-4_12

[37] Peled, D. (2001) Model Checking. Department of Computer Science, Bar llan University, Ramat Gan.

[38] Guo, H.Y., Wu, M., Zhou, L.D., Hu, G., Yang, J.F. and Zhang L. (2011) Practical Software Model Checking via Dynamic Interface Reduction. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, 23-26

October 2011, 265-278. https://doi.org/10.1145/2043556.2043582

[39] Nguyen, V.Y. and Ruys, T.C. (2013) Selected Dynamic Issues in Software Model Checking. *International Journal on Software Tools for Technology Transfer*, **15**, 337-362. https://doi.org/10.1007/s10009-012-0261-y

[40] Gupta, A., Kahlon, V., Qadeer, S. and Touili, T. (2018) Model Checking Concurrent Programs. In: Clarke, E., Henzinger, T., Veith, H. and Bloem, R., Eds., *Handbook of Model Checking*, Springer, Cham, 573-611. https://doi.org/10.1007/978-3-319-10575-8_18

[41] Santos, I.S., Rocha, L.S., Santos-Neto, P.A. and Andrade, R.M.C. (2016) Model Verification of Dynamic Software Product Lines. *Proceedings of the XXX Brazilian Symposium on Software Engineering*, Maringá, 19-23 September 2016, 113-122. https://doi.org/10.1145/2973839.2973852

[42] Avgerinos, T., Rebert, A., Cha, S.K. and Brumley, D. (2014) Enhancing Symbolic Execution with Veritesting. *Proceedings of the* 36*th International Conference on Software Engineering*, Hyderabad, 31 May-7 June 2014, 1083-1094. https://doi.org/10.1145/2568225.2568293

[43] Bohme, M., Pham, V.T. and Roychoudhury, A. (2017) Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, **45**, 489-506. https://doi.org/10.1109/TSE.2017.2785841

[44] Gulzar, M.A., Musuvathi, M. and Kim, M. (2020) BigTest: A Symbolic Execution Based Systematic Test Generation Tool for Apache Spark. *Proceedings of the ACM/IEEE* 42*nd International Conference on Software Engineering*: *Companion Proceedings*, Seoul South, 27 June-19 July 2020, 61-64. https://doi.org/10.1145/3377812.3382145

[45] Chalupa, M., Jasek, T., Movak, J., Rechtackova, A., Sokova, V. and Strejcek, J. (2021) Symbiotic 8: Beyond Symbolic Execution. In: Groote, J.F. and Larsen, K.G., Eds., *TACAS* 2021: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Cham, 453-457. https://doi.org/10.1007/978-3-030-72013-1_31

[46] Mrazek, J., Bauch, P., Lauko, H. and Barnat, J. (2016) SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration. In: Bošnački, D. and Wijs, A., Eds., *SPIN* 2016: *Model Checking Software*, Springer, Cham, 208-213. https://doi.org/10.1007/978-3-319-32582-8_14

[47] Visser, W., Pasareanu, C.S. and Khurshid, S. (2004) Test Input Generation with Java Pathfinder. *ACM SIGSOFT Software Engineering Notes*, **29**, 97-107. https://doi.org/10.1145/1013886.1007526

[48] Penczek, W., Szreter, M., Gerth, R. and Kuiper, R. (2000) Improving Partial Order Reductions for Universal Branching Time Properties. *Fundameta Informaticae*, **43**, 245-267. https://doi.org/10.3233/FI-2000-43123413

[49] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L. and Hwang, L.J. (1992) Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, **98**, 142-170. https://doi.org/10.1016/0890-5401(92)90017-A

[50] Havelund, K. and Pressburger, T. (1999) Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, **2**, 366-381. https://doi.org/10.1007/s100090050043

[51] Cadar, C., Dunbar, D. and Engler, D.R. (2008) KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Proceedings of the* 8*th USENIX Conference on Operating Systems Design and Implementation*, San Diego, 8-10 December 2008, 209-224.

[52] Clarke, E.M., Grumberg, O., Kroening D., Peled, D. and Veith H. (2018) Model Checking. 2nd Edition, MIT Press, Cambridge.

[53] McMillan, K.L. (1999) Symbolic Model Checking. In: Inan, M.K. and Kurshan, R.P., Eds., *Verification of Digital and Hybrid Systems*, Springer, Berlin, 117-137. https://doi.org/10.1007/978-3-642-59615-5_6

[54] Godefroid, P. (1996) Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Berlin. https://doi.org/10.1007/3-540-60761-7

[55] Sen, K., Marinov, D., Agha, G. and Sridharan, M. (2016) Guiding Symbolic Execution towards Unexplored Code. *Proceedings of the* 2016 *ACM SIGPLAN International Conference on Object-Oriented Programming*, *Systems*, *Languages and Applications*, Amsterdam, 2-4 November 2016, 753-771.

[56] Bao, X., Yin, H., Chen, X. and Zhang, L. (2018) Speeding Up Symbolic Model Checking with Machine Learning. *Proceedings of the* 33*rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, 3-7 September 2018, 570-580.

[57] Ma, L., Zhang, F.Y., *et al.* (2018) DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. ArXiv: 1803.07519.

[58] Durrelli, V.H.S., Durelli, R.S., Borges, S.S., Endo, A.T., Eler, M.M., Dias, D.R.C. and Guimaraes, M.P. (2019) Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Realiability*, **68**, 1189-1212. https://doi.org/10.1109/TR.2019.2892517

[59] Vardhan, A., Sen, K., Viswanathan, M. and Agha, G. (2004) Learning to Verify Safety Properties. In: Davies, J., Schulte, W. and Barnett, M., Eds., *ICFEM* 2004: *Formal Methods and Software Engineering*, Springer, Berlin, 274-289. https://doi.org/10.1007/978-3-540-30482-1_26

[60] Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P. and Gulwani, S. (2018) Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. ArXiv: 1804.01186. https://arxiv.org/abs/1804.01186

[61] Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A. and Schwartz, E.J. (2018) Executable Counterexamples in Software Model Checking. In: Piskac, R. and Rümmer, P., Eds., *VSTTE* 2018: *Verified Software*: *Theories*, *Tools and Experiments*, Vol. 11294, Springer, Cham, 17-37. https://doi.org/10.1007/978-3-030-03592-1_2

[62] Utting, M. and Legeard, B. (2006) Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco.

[63] Havelund, K. and Rosu, G. (2002) Monitoring Programs Using Rewriting. *Proceedings of the* 14*th International Conference on Computer Aided Verification*, Copenhagen, 27-31 July 2002, 450-462.

[64] Pacheco, C., Lahiri, S.K., Ernst, M.D. and Ball, T. (2007) Feedback-Directed Random Test Generation. 29*th International Conference on Software Engineering*, Minneapolis, MN, 20-26 May 2007, 75-84. https://doi.org/10.1109/ICSE.2007.37

[65] Harman, M. and Sthamer, H. (2002) Search-Based Software Testing. *Proceedings of the* 2002 *International Symposium on Software Testing and Analysis*, Roma, 22-24 July 2002, 249-258.

[66] King, J.C. (1976) Symbolic Execution and Program Testing. *Communications of the ACM*, **19**, 385-394. https://doi.org/10.1145/360248.360252

[67] Liang, G., Liao, L., Xu, X., Du, J., Li, G. and Zhao, H. (2013) Effective Fuzzing Based on Dynamic Taint Analysis. 2013 *Ninth International Conference on Computational Intelligence and Security*, Emeishan, 14-15 December 2013, 615-619. https://doi.org/10.1109/CIS.2013.135

[68] Kwiatkowska, M., Norman, G. and Parker, D. (2018) Probabilistic Model Checking: Advances and Applications. In: Drechsler, R., Ed., *Formal System Verification*, Springer, Cham, 73-121. https://doi.org/10.1007/978-3-319-57685-5_3

[69] Sen, K., Marinov, D. and Agha, G. (2005) CUTE: A Concolic Unit Testing Engine for C. *Proceedings of the* 10*th European Software Engineering Conference Held Jointly with* 13*th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC*/*FSE*-13), Chicago, 5-9 September 2005, 263-272.

[70] Satpathy, M. and Ramesh, S. (2007) Test Case Generation from Formal Models through Abstraction Refinement and Model Checking. *Proceedings of the* 3*rd International Workshop on Advances in Model-Based Testing*, London, 9-12 July 2007, 85-94. https://doi.org/10.1145/1291535.1291544

[71] Emerson, E.A. and Namjoshi, K.S. (1998) Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, **8**, 244-263.

[72] Lamport, L. (1986) On Interprocess Communication. *Distributed Computing*, **1**, 77-85. https://doi.org/10.1007/BF01786227

[73] Alur, R., Bodík, R., Juniwal, G. and Seshia, S.A. (2015) Synthesis of Cyber-Physical Systems. *Proceedings of the IEEE*, **103**, 1589-1609.

[74] Gurfinkel, A. and Chaki, S. (2015) Combining Static Analysis and Model Checking for Program Analysis. *Formal Methods in System Design*, **47**, 62-91.

[75] Zhang, L., Chen, Y., Zhang, Y. and Liu, Y. (2020) A Critical Survey of Machine Learning-Assisted Verification. *IEEE Transactions on Software Engineering*, **47**, 1064-1087.

[76] Pei, K., Cao, Y., Yang, J. and Jia, Z. (2017) Deep Learning-Based Software-Defined Networking for IoT Security. *IEEE Network*, **31**, 80-85.

[77] Gao, Y., Wei, Y., Sun, J. and Zhang, X. (2019) Learning-Based Abstraction Refinement for Software Model Checking. *Frontiers of Information Technology and Electronic Engineering*, **20**, 372-382.

[78] Zhang, X., Liu, Y., Huang, L. and Zhang, Y. (2021) Reinforcement Learning-Based Heuristic for Accelerating Software Model Checking. *Journal of Systems and Software*, **174**, Article ID: 110950.