A Pyramid Of (Formal) Software Verification

Martin Brain[1(✉)] and Elizabeth Polgreen[2]

[1] City, University of London, London, England
martin.brain@city.ac.uk
[2] University of Edinburgh, Edinburgh, Scotland
elizabeth.polgreen@ed.ac.uk

Abstract. Over the past few years there has been significant progress in the various fields of software verification resulting in many useful tools and successful deployments, both academic and commercial. However much of the work describing these tools and ideas is written by and for the research community. The scale, diversity and focus of the literature can act as a barrier, separating industrial users and the wider academic community from the tools that could make their work more efficient, more certain and more productive. This tutorial gives a simple classification of verification techniques in terms of a pyramid and uses it to describe the six main schools of verification technologies. We have found this approach valuable for building collaborations with industry as it allows us to explain the intrinsic strengths and weaknesses of techniques and pick the right tool for any given industrial application. The model also highlights some of the cultural differences and unspoken assumptions of different areas of verification and illuminates future directions.

1 Introduction

Software verification is a large and diverse area of computer science research. Topics covered range from low-level, practical issues such as understanding the exact behaviour of various hardware and software constructs through to high-level, theoretical issues of expressibility and the limits of what is computable. The diversity of and connections between these areas can make it hard to understand and appreciate the full power and applicability of the ideas. This is compounded by the existence of several different academic traditions or schools each of which have their own terminology and foundations.

The field also has a strong culture of tool development, leading to a range of powerful academic and commercial tools. However potential users (both academic and commercial) are often faced with the problem of understanding how these various tools relate to each other and their various strengths and weaknesses. Their problem is not which tool to pick but *on what basis to make their decision*. For experienced academics and researchers, the answers are often 'obvious' but, again, this requires a broad and comprehensive knowledge of the different approaches and traditions of verification.

A. Platzer et al. (Eds.): FM 2024, LNCS 14934, pp. 393–419, 2025.
https://doi.org/10.1007/978-3-031-71177-0_24

This paper describes our approach to bridging this gap and communicating the 'big-picture' of software verification without requiring people to read many papers, attend numerous conferences or develop multiple tools. The approach has been developed and used in numerous industrial partnerships as well as in undergraduate and post-graduate teaching. It has provided a simple way of structuring the explanation of what we do and how this fits with particular organisations' needs. We regard this as a successful and efficient way of bridging the (common) divide between research and practice.

As with all overviews, there are exceptions and caveats to all of the classifications we give. There are also systems which combine multiple techniques, whose classification is debatable or ambiguous. If such exceptions and combined techniques do not already exist, they will likely do so soon as they represent novel research directions. In this regard, this paper should be thought of as a guidebook or a phrase book rather than an atlas or dictionary. Our goal is to describe the common 90% of papers in a field rather than the exceptional 10%.

We assume a base understanding of computing, but this paper is intended to be readable by commercial developers and does not assume prior knowledge of verification or theoretical computer science. We hope it will communicate:

- Our pyramid model, which gives a fundamental trade-off for software verification tools (Sect. 2).
- The kinds of tools available, their intrinsic strengths and weaknesses, and enough of the culture and terminology to communicate with and evaluate tools from the relevant research community (Sect. 3).
- The different ways of giving specifications (Sect. 4).
- How to select the right kind of tool(s) for a particular practical problem (Sect. 5).

2 A Pyramid of Verification

In the most general sense, verification is the process of checking the properties of a thing against a set of criteria. In our context, we will refer to the thing being checked as the *program* and the criteria as the *specification*. Figure 1 illustrates the verification process. The primary inputs are the specification and the program plus the verification process uses an amount of compute resource and human effort. The ideal outcome is that the system is verified but it is also possible that one or more defects are detected or that the verification is inconclusive and the final result is unknown. From a practical point of view, unknown is probably the worst outcome as the effort is expended without producing useful evidence. However, as we will see, reliably avoiding the unknown outcome turns out to be challenging.

Representing the program. Our primary focus is when the program (thing we are verifying) is a piece of software written in an imperative language. However many of these techniques have been successfully applied to hardware, parallel

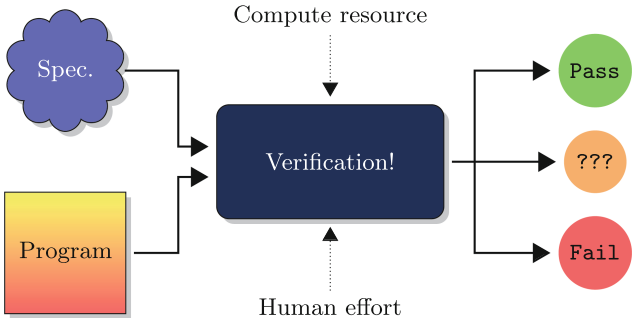

Fig. 1. The verification process in principle

and distributed software as well as more abstract models of computation such as protocols, process calculi, automata, cyber-physical systems, transition systems, etc. One common view in software verification, and that we will use in the following sections of the paper, is that a program is a *description* or *representation* of a set of traces. A *trace* is a single execution of the program; a sequence of states showing the step-by-step execution of the program. Running a program is computing a single trace. For a program of any size it is infeasible to compute all of the traces, so the set of all traces is a mathematical idea rather than something that is ever directly computed. This viewpoint is useful as it allows the program and the specification to be seen as the same kind of mathematical objects; sets of traces. It also gives an idea of *over-approximation* and *under-approximation* of a program. These are sets of traces that contain or are contained in the set of traces of the program.

Representing the specification. The specification is the set of criteria we check the program against. Software verification tools have traditionally focused on *universal* specifications; those of the form "every execution of the program must" or conversely "there must be no way of" rather than *existential* ones, "there must be at least one trace that". Other, more complicated, properties can be specified that reason about the interaction of traces (hyperproperties) or the likelihood of certain traces (probabilistic properties), but we do not discuss those here. If the program is understood as a representation of a set of traces then the specification can be understood as a representation of the set of all traces that have a required property. Formal verification of a universal specification can then be conceptually reduced to checking that the set of all program traces is *included* in the set of specified traces. We give further discussion on the ways of representing specifications in Sect. 4.

Given a program and a specification to verify it against, the ideal verification tool would be:

Automatic (run with no human interaction),
Never miss bugs (only say verified if the system meets the specification), and

Never give false alarms (always say verified if the system meets the specification).

Unfortunately, if the specification includes any notion of reachability ("if the program ever ...", "when the program ...") then Rice's Theorem [73], a consequence of Turing's famous result on the Halting Problem [78] means that it is not possible to create a verification tool that has all the desired properties[1] in all cases. As almost all significant specifications include some notion of reachability or location, this gives us the fundamental trade-off at the heart of software verification:

> It is not possible to create a verification tool which can take *any* program and *any* specification and *automatically* give an answer in a finite amount of time guaranteeing *no missed bugs* and *no false alarms*.

As with all applications of theoretical results to the real world, we should be mindful of the caveats. Turing's result applies to a theoretical model of computing which has an infinite state-space. We will assume that the state-space of programs is so large that is effectively infinite[2] and that this result applies. The subtleties of quantification are also important. This result applies to one, or a finite number of verification tools working on *all* programs and *all* specifications. For a particular program and specification, there is a verification tool that can automatically give a full answer (although writing it might require performing the verification by hand and then writing a program that will simply print the answer). Likewise, there are large sets of programs and specifications for which this is possible. For example, if the specification is "the program terminates", then any acyclic program (no loops or recursion) can easily be automatically verified in the time it takes to read the program and check it is acyclic.

Although theoretical computer science shows that it is not possible to build a universal verification tool, it is easy to get surprisingly close:

- 'Automatic' and 'no missed bugs' can be achieved by over-approximating the set of invalid traces (traces of the program that do not meet the specification). A tool that prints `verification failed` for all inputs would be the simplest example.

[1] We avoid the terms "sound" and "complete" as there is a cultural bias for authors to claim that their technique is "sound but not complete". This leads to two, opposite definitions of these terms. Some authors use "sound/complete (for proof)" while others use "sound/complete (for refutation)". No missed bugs is "sound for proof" and "complete for refutation" while no false alarms is "complete for proof" and "sound for refutation".

[2] Clearly, any individual computer has physical limits on the number of bits that can be stored and thus has a limit on the state-space of a program running on it. However, the state-space of a program running on a processor with 640KB of memory (up to $2^{5242880}$ states) may be regarded as *practically* infinite as it contains more states than particles in the universe.

- 'Automatic' and 'no false alarms' can be achieved by under-approximating the set of invalid traces. A tool that prints `verification succeeded` for all inputs would be the simplest example.
- Finally 'no missed bugs' and 'no false alarms' can be achieved by getting a human to create a formal proof of the verification and checking it.

As it is easy to achieve two of the three criteria, we can view verification tools as starting with up to two of these and then trading computational effort to achieve the third for an ever-increasing set of programs and specifications. For example, *over-approximate* tools (automatic and no missed bugs) use computation to reduce the number of false alarms. *Under-approximate* tools (automatic and no false alarms) use computation to reduce the number of missed bugs. *Human-assisted* tools (no missed bugs and no false alarms) use computation to reduce the amount of human effort required.

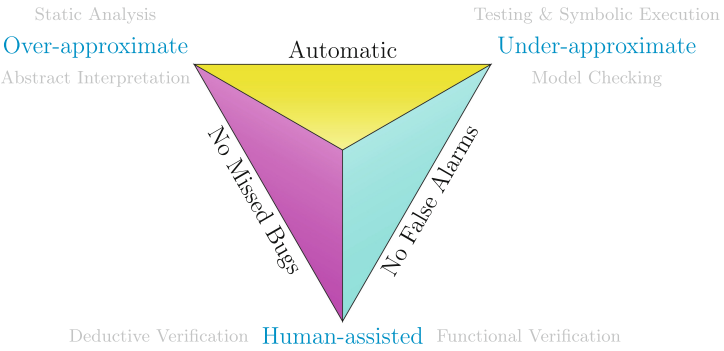

Fig. 2. The software verification pyramid with the six schools.

Figure 2 illustrates this trade-off with a three-sided pyramid (viewed from above). Each base edge of the pyramid represents one of the three attributes; automatic, no missed bugs and no false alarms. Corners on the base of the pyramid represent each of the three approaches; over-approximate, under-approximate and human-assisted. The top of the pyramid, in the centre of the diagram, represents the ideal system with all three attributes. Computation effort is then used to 'climb' the pyramid towards the top, with different techniques giving different routes up their chosen edge (or face).

There are many other dimensions on which software verification techniques can be classified, and many other trade-offs that are necessarily made by different tools. For instance, one could also consider how properties are specified, the categories of systems that are analysed, and the usability of the system and specification language. Our pyramid model is not intended to be exhaustive, but it is intended as a useful starting point when making the initial choices about how to solve a verification problem.

Table 1. Cultural attributes of the six schools.

	Over-Approximate		Under-Approximate		Human-Assisted	
	Static Analysis	Abstract Interpretation	Testing & Symbolic Execution	Model Checking	Deductive Verification	Functional Verification
Program	Procedural or O.O.	Procedural	Procedural or O.O.	Procedural or O.O.	Subsets of procedural	Functional
Commmon Means of Specification	Builtin	Annotation linked to the abstraction	Generally annotation	Annotation or external	Annotation	Type as annotation
Common Type of Specification	Data flow, aliasing, type, shape, taint	Value, shape, resource, data flow	Value, WCET, resource	Value, temporal, modal, liveness	Value, shape, termination, resource	Type, termination
Mathematical Foundations	Ad-hoc / operational semantics	Order theory	Ad-hoc / transition systems	Transition systems	Logic	Type theory
User Skill Required	Minimal	Low/Medium	Low	Medium	High	Very high
Compute Required	Minimal	Low/Medium upwards	Medium upwards	Medium/High upwards	Low/Medium	Low
Typical Output	Algorithm dependent	Alarms or abstract domains	Error traces	Error traces	Proof or local counter-examples	Type-checking errors
Major Systems	Lint[55], Coverity[1], Fortify[5], FindBugs[4], CPPCheck[2]	Astrée[33], Polyspace[7], Infer[57], Code Contracts[66]	CREST[3], JPF[50], Pex[77], KLEE[18]	CBMC[60], Blast[51], *SMV[22], CPAchecker[13]	SPARK[8], Dafny[64], Frama-C[34], Malpas[6], Esc/Java[45]	Coq[12], PVS [70], Agda[69], Isabelle/Hol[68]

3 Six Schools of Verification

Our pyramid model allows us to compare and contrast the different academic traditions in and aligned with software verification. Each of these represents a separate lineage of thought, approach and community, although there is interaction, overlap and cross-fertilisation between them. This section surveys them and attempts to give a qualitative assessment. Of the six schools, not all would regard themselves as software verification but all have an important impact on the field. As with all qualitative assessments, there is an element of subjectivity and no doubt researchers and practitioners from each of the fields have different views; the aim of this document is to provide an overview rather than a definitive assessment. There are also exceptions to most of the distinctions and classifications and our statements should be taken as normative or cultural observations rather than hard limits.

Table 1 gives the six schools with various cultural attributes. The kinds of programs verified, and the kind of specification (see Sect. 4) normally considered are given along with the mathematical formalisation used to express theoretical results and algorithms. It also gives quantitative valuation for the level of user skill required to apply the tools effectively and the computational requirements of the tools. Finally, it gives a few of the prominent or significant tools from each

of the areas. Many tool suites include tools from several of the different schools. These are listed under the school for their main tool or general approach.

As well as the traditional approaches of the six schools, there are also a significant number of combined techniques and tools. The simplest of these are pipe-lines that run different tools in sequence (or sometimes in parallel), feeding the results of one into the next. More sophisticated, synergistic combinations also exist and will likely form a major part of the future of software verification.

```
1 (int, int)
2 count (Array a, int target)
3 {
4   int found=0, last=-1, i=0;
5   while (i < a.length()) {
6     assert(0≤i<a.length());
7     if (a[i] == target) {
8       found = found + 1;
9       last = i;
10    }
11    i = i + 1;
12  }
13  assert(0≤last<a.length());
14  assert(found≠0 ⟹ a[last]=target);
15  return (found, last);
16 }
```

Fig. 3. The running example program. The specification we wish to check is in the form of 3 assertions.

Figure 3 gives an example program in a C-like language. The specification is given as three assertions (an example of what Sect. 4 describes as an annotation specification). They express:

1. All array accesses are in bounds, at line 6
2. `last` is in `a` at end at line 13
3. If `found` is non zero at end then `a[last] == target` at line 14.

Try working out which of these can be verified. If they can be verified, what knowledge about the program is required to show this? If not, how would you provide a counter-example? Answers can be found here[3].

3.1 Static Analysis

One of the oldest fields of program analysis and verification is static analysis. Unfortunately, this term is used in at least two ways. In the general sense, it refers to program analysis performed without running the program (hence static). In this sense, the majority of the techniques of software verification can be regarded as static analysis. In the narrow sense, it refers to a specific set of algorithms and techniques that are explicitly referred to as static analysis (as opposed to any other name), and are often developed with and to serve the needs of compilers, both in improving their warnings (verification against a specification) and optimising the code they generate.

[3] Assertion 1 always holds (i.e., is always true) as the initial assignment to `i` is `0`, it is only ever incremented and the assertion appears immediately after the loop condition checks that `i < a.length()`. Assertion 2 can fail, as if the array does not contain the `target`, `last` will still have its initial `-1` value. Assertion 3 always holds as the only place `last` is assigned guarantees this property, although some non-trivial reasoning about reachability is required.

The traditional focus of this area has been on over-approximate techniques. Spurious warnings are fine if not too numerous, likewise missing an optimisation is much more acceptable than miscompiled code based on a flawed assumption. The need to keep compilation fast and robust has meant the field has focused on fast and robust techniques for common problems such as finding uninitialised variables, eliminating possible aliasing between pointers and data flow for scheduling and optimisation rather than more complex specifications.

The first static analysis tools began to emerge in the 1970 s, beginning with Lint [55], developed at Bell Labs in 1978, which flagged suspicious constructs in code that could be suggestive of a bug in the code. There have been many versions of lint developed for many C and C++ compilers, and "linter" is sometimes used as a generic term for static analysis tools based on this paradigm of flagging patterns indicative of programming errors, bugs, stylistic errors and suspicious constructs. For example, Lint would flag this if statement as a suspicious construct since it always evaluates to true, which is probably not what the programmer intended:

```
1 unsigned x;
2 if(x < 0) ...
```

These static analysis tools can be broadly described as lexical scanners that look for patterns in code that are likely to be defects or vulnerabilities. Static analysis tools do not treat the process of finding bugs in software as a logical problem; none of the analyses performed by these static-analysis tools involves constructing proof objects, and, as a result, these static analysis tools are not able to discover many of the complex bugs that can be discovered by other verification tools. That said, there is a broad range of types of bugs that can be discovered through this kind of code analysis, many of which represent serious vulnerabilities, for example, potential buffer overflows.

To mitigate the large numbers of false alarms typically produced by most static analysis algorithms, modern static analysis tools may categorise the bugs into ranks by seriousness [4] (i.e., the likelihood of the bug being a serious vulnerability), or by applying filtering algorithms to the results [24,76].

Industry: Static analysis tools like the commercial Coverity [1] and open-source CPPCheck [2] are commonly used in industry and have been used to find bugs in projects like Mozilla [35].

Running Example: Static Analysis cannot check the specifications given in the running example. It could, however, flag simpler properties. For instance, in a typed program, it could flag if the comparison `i < a.length()` compares an integer with an unsigned integer.

3.2 Abstract Interpretation

The field of abstract interpretation started with a series of papers [30–32] on the theory underpinning a range of static analyses. These proposed using the tools of

order theory (partially ordered sets, lattices and Galois connections) to separate the analysis into two components: domains which track the information required for the specific analysis and abstract algorithms describing how the analysis is performed.

The *domain* describes a data-structure that is used to represent an over-approximate summary of the state of the program at a given point. Traditionally there will be an instance of the domain for every program location. For example, the *constant domain* contains a map from variables to constant values (plus flags for "not a constant" and "no value assigned"). If the map has `found` $\rightarrow$ `0` then we know that every time that location is reached, `found` will be `0`. The *interval domain* stores a map from variables to intervals (plus a "no value assigned" flag). If `i` $\rightarrow [0, 10]$, then we have a bound for the possible values for `i`. This can represent all the cases that a constant domain can and also represent things it can't so we say it is a more precise representation (i.e. less of an over-approximation). Domains have been created for a wide range of different analyses; data flow analysis, constant propagation, pointer analysis, etc. [21,74,81]. Creating a new kind of analysis can be as simple as specifying an appropriate domain.

The choice of domain depends on several factors. Using a more detailed domain (i.e. less of an overapproximation) can reduce the number of false alarms but requires additional computation. Often it is necessary to choose a domain that can precisely express the specification and the reasons why it is true. For example, if the specification includes proving bounds on variables then an interval domain is a good choice. However, if the specification includes proving the equality of variables, then intervals are unlikely to be very useful as they cannot represent the relationship between variables. The mathematics of abstract interpretation allows domains to be combined in various ways. Selecting the right combination of domains for a particular program and specification is one of the more advanced skills that can boost the effectiveness of abstract interpretation.

The second part of abstract interpretation is the analysis algorithms. These are stated in terms of mathematical operations on the domain, typically:

Join Combines two instances of the domain to create a new instance that over-approximates both. For example, if one instance has `x` $\rightarrow [1, 4]$ and the other has `x` $\rightarrow [6, 8]$ then in the join `x` $\rightarrow [1, 8]$ – the smallest interval that contains both inputs.

Transform Takes one instance and an instruction and creates a new instance which over-approximates the effects of the instruction. For example if an instance contains `x` $\rightarrow [1, 8]$ and `y` $\rightarrow [-4, 4]$ then the transformer for `z = x + y` would create a new domain containing `z` $\rightarrow [-3, 12]$ and all other variables mapped to the same as in the original instance.

Widen Creates an instance of the domain that over-approximates the fix-point of a series of instances. For example given a loop `for (i = 0; i < n; ++i)` the widen operator might create an instance with `i` $\rightarrow [0, MAX]$.

Using abstract operations such as these allows the analysis algorithms to be specified and implemented independently of the domain, giving another 'orthogonal'

space of possibilities. Analysis algorithms are often characterised by *sensitivity* to various program constructs:

Flow Sensitive:the order of instructions in the program is followed.
Path Sensitive:the branch conditions are applied to improve precision.
Context Sensitive:the calling context of functions is considered.

By using a more sensitive algorithm, the amount of over-approximation can be reduced and the set of specifications that can be automatically verified, with no false alarms, is increased. Of course, this can only be achieved by increasing the amount of computation required, so the more sensitive the algorithm the more expensive it will be to use. One of the practical strengths of abstract interpretation is modularity; assuming that variables or parts of a program are independent of each other is an over-approximation. This fits naturally within abstract interpretation. It is also the basis of many techniques for improving scalability.

Industry: Abstract interpretation has been used in industry to prove the absence of bugs in flight control software [56], and to analyse worst-case execution time for microprocessors [42]

Table 2. Table showing the result of abstract interpretation on the running example, using an interval domain.

	Line	found	last	i	Assertion result
L4	found = 0				
L4	last = -1	[0,0]			
L4	i = 0	[0,0]	[-1,-1]		
L5	i < a.length()	[0,x]	[-1, x-1]	[0,x]	
L6	$0 \leq$ i < a.length()	[0,x]	[-1, x-1]	[0,x]	pass
L7	a[i]=target	[0,x]	[-1, x-1]	[0,x]	
L8	found=found+1	[0,x]	[-1, x-1]	[0,x]	
L9	last=i	[1, x+1]	[-1, x-1]	[0,x]	
L11	i=i+1	[0, x+1]	[-1, x]	[0,x]	
L13	$0 \leq$ last $\leq$ a.length()	[0,x]	[-1, x-1]	[0,x]	unknown
L14	found$\neq$0 $\implies$ a[last]=target	[0,x]	[-1, x-1]	[0,x]	unknown

Running Example: Table 2 shows the result of abstract interpretation being applied to the running example, using the interval domain. Abstract interpretation can prove assertion 1 is always true, but cannot prove that assertion 2 fails or that assertion 3 is always true.

3.3 Testing and Symbolic Execution

Testing compiling and executing code on some concrete input(s), has a dual role in development. It is both a verification tool (does the test give a result allowed by the specification) and a validation tool (does the test do what I expected). In its verification role, testing is most suitable for existential specifications (things of the form "The software must be able to ...") because it is an underapproximate technique, only exploring a subset of possible traces of the program. Dijkstra famously described this situation [37] by saying: "Program testing can be used to show the presence of bugs, but never to show their absence!" . Showing the absence of bugs is the same as saying that every trace of the program does not trigger any bugs; a "universal" specification (Sect. 2).

One pragmatic option is to try to explore a 'sufficient' set of traces so that if there are bugs there is a high probability they will be found. This is the motivation behind coverage metrics which have a notion of a set of traces that is 'sufficient' and likely include at least one example of each kind of program behaviour. Coverage metrics are widely used and give a base level of certainty, even for universal specifications. Testing and coverage metrics are a large topic and [49,71] give a summary of the current-state-of-the-art. Test inputs can be defined manually by developers or automatically generated using *fuzz testing* to try to increase these coverage metrics.

Symbolic Execution [58] is a verification technique that was developed in the context of testing. It aims to build a logical expression that describes all of the traces that have taken the same path through the control flow graph. Rather than test a single trace it covers a subset of traces whose behaviour is similar. The subset of traces is then tested against the specification by testing the satisfiability of the logical expression representing the subset of traces and the negation of the specification, using a satisfiability solver.

A symbolic execution tool keeps a set of symbolic states. Each of these contains its current location in the program, a map from variables to expressions (describing the space of value it could take) and a *path condition* which is a set of expressions giving the conditions that must hold for that path to be taken. The set initially contains a single symbolic state, at the start of the program, with every program variable mapped to set to a fresh logical variable and an empty path condition. The analysis proceeds as follows:

Assign: Assign symbolic variables to each variable in the program state. Evaluate program statements using the symbolic variables until you reach a branching condition

Branch: On reaching a branching condition, e.g., an if statement, choose a fork to explore. This places a constraint over the symbolic variables, e.g., in the case of an if statement, if we explore the path when the if condition evaluates to true.

Check: When the path reaches an assertion, pass the constraints over the symbolic variables to the solver along with a constraint representing the violation of the assertion.

Treating the memory as fully symbolic does not scale in practice, so symbolic execution engines typically implement a partial memory model in which writes are concretized, but reads are modelled as reads from symbolic memory, up to a certain finite size of memory, and beyond that are concretized [29].

A limitation of symbolic execution is the path explosion problem: the number of paths in a program typically grows exponentially with the program size (and in the case of unbounded loops may be infinite). This means that the larger the program, the less likely it is symbolic execution will manage to find a subset of paths that exercises a particular bug. There are several heuristics the community has developed to try to mitigate this problem, including merging paths [61], exploring paths in parallel [79], and use of different heuristics to control the order in which paths are explored. In addition, combined approaches exist: concolic testing is a successful combination of symbolic execution combined with testing (or *concrete* execution), which treats program variables as symbolic, but inputs as concrete. It is used in conjunction with constraint solvers to generate new concrete inputs with the aim of maximising code coverage. A good survey on symbolic execution techniques is [20]

Industry: Testing is ubiquitous in industry, and needs no specific citation. Symbolic execution is the underlying technique in many popular tools used in industry, for example, KLEE [18] has been used for a variety of applications including wireless sensor networks and exploit generation [19], JPF has been used at NASA on the Orion control software [72], and Microsoft's SAGE, using a combination of fuzzing and symbolic execution, is used to find bugs in Windows applications [48].

Table 3. Table showing the result of symbolic execution for one arbitrarily choosen path on the running example. α and t are symbolic variables, and the path constraints are constraints inferred from choosing a branch at each branching condition.

	path constraints	symbolic environment
L0 (assign)	true	$a \mapsto \alpha, target \mapsto t$
L4 (assign)	true	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L5 (branch)	$0 < \alpha.length()$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L7 (branch)	$0 < \alpha.length() \wedge \alpha[0] \neq t$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 0$
L11 (assign)	$0 < \alpha.length() \wedge \alpha[0] \neq t$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 1$
L13 (check)	$0 < \alpha.length() \wedge \alpha[0] \neq t \wedge 0 \leq -1 < \alpha.length()$	$\ldots, found \mapsto 0, last \mapsto -1, i \mapsto 1$

Running Example: Table 3 shows a single path of symbolic execution being applied to the running example. We are able to prove that assertion 2 fails if the target is not in the array. No path in the graph is able to show assertion 1 or assertion 3 always hold.

Algorithm 1: Fixpoint

```
Result: Reachable states R
R = I;
while 1 do
    if R == R ∧ T then
        return R;
    else
        R = R∧T ;
    end
end
```

Algorithm 2: BMC

```
Result: Reachable states R
R = I;
i=0 ;
while i < k do
    R = R∧T ;
    i++ ;
end
return R ;
```

3.4 Software Model Checking

Model checking involves constructing transition systems, and checking that these systems are *models* of given logical specifications. Originally, the field focused on specifications written in temporal logic [9], and systems that were manually specified using a process calculus giving a *labelled transition system* (LTS), an automata-like structure with states corresponding to states of programs and transitions to the possible developments of the system. Verification could be reduced to showing that the system's LTS was a model of the logic, giving rise to the name of the field.

Explicit State Model Checking explores the states of the LTS one at a time, using graph algorithms such as depth-first search, until either a counter-example for the property has been found or all reachable states have been explored. This is limited by the number of states so tends to be used on protocols, high-level designs and abstraction of software systems. SPIN [53], FDR [47] and the TLA [83] tools are example of this style.

Symbolic Model Checking uses boolean formulae to represent sets of states in the system, the transition relation and the properties we wish to check. For instance, a formula representing the initial states of the running example is $found = 0 \wedge last = -1 \wedge i = 0$. The most basic symbolic model checking algorithm for systems with finite-states computes a formula that represents the total set of reachable states (R) by starting with the initial state (I) and "unrolling" the transition relation (T) repeatedly until a fixed-point is reached, as shown in Algorithm 1. The formula can then be checked to see whether it satisfies the specification. Critical to the performance of these systems is the use of compact and efficient data structure to manipulate Boolean formulae. A form of decision trees known as Binary Decision Diagrams (BDDs) are a popular choice [17,23].

Algorithms which compute the fix-point are able to find all bugs provided computing the fix-point is possible (i.e., the system does not contain any unbounded loops) and the representation of the system as a transition relation is precise enough to capture any bugs (for instance, memory models are

often approximate). If the system contains unbounded loops, the algorithm will never find a fix-point, and so approximations must be introduced in order to deal with these scenarios.

Bounded Model Checking (BMC) [14] is an under-approximate technique which, instead of computing the fix-point, unwinds a transition system to a finite bound k, and then checks for violations of the property within the states reachable in k steps. BMC thus only guarantees the absence of bugs that can be reached in k steps. For some systems, a completeness-threshold [25] can be computed such that it is guaranteed that, if no bugs exist within k steps, no bugs exist at all.

However, computing completeness thresholds for unbounded loops amounts to solving the Halting problem and so the technique remains, in general, on the under-approximate corner of the pyramid. BMC in its original presentation begins by unwinding the transition system 1 step and looking for violations of the specification within 1 step, and then it unwinds the transition system 2 steps, and so on until it reaches k steps. However, many popular tool implementations will instead unroll the entire system minus any loops and recursion to their limits, and then unwind the loops to k steps, as shown in Fig. 4.

```
 1 (int, int)
 2 count (Array a, int target)
 3 {
 4   int found = 0, last = -1, i=0;
 5   /** First unwinding **/
 6   if (i < a.length()) {
 7     assert(0≤i<a.length());
 8     if (a[i] == target) {
 9       found = found + 1;
10       last = i; }
11     i = i + 1;
12
13     /** Second unwinding **/
14     if (i < a.length()) {
15       assert(0≤i<a.length());
16       if (a[i] == target) {
17         found = found + 1;
18         last = i; }
19       i = i + 1;
20
21     if(i < a.length())
22       /** Unwinding assertion **/
23       assert(0);
24     }
25   }
26   assert(0≤last<a.length());
27   assert(found≠0⇒a[last]=target);
28   return (found, last);
29 }
```

Fig. 4. BMC applied to the example program. The loop is unwound 2 times. The assertion at line 23 checks whether the loop is unrolled sufficiently for the input values

A significant development in BMC was the use of SAT-solvers [15] instead of BDDs. Once the formula representing reachable states has been constructed, a SAT solver can efficiently check whether a counterexample exists using this formula. This is similar to symbolic execution, but instead of formulae representing subsets of paths, we have one formula that represents all of the paths. Modern software model checking tools typically take as input either source code, or some intermediate compiled representation of the source code such as LLVM [63], and convert this into an LTS, and then use SAT-based model checking to check this LTS against the specification.

There are many model checking algorithms beyond those mentioned, such as IC3/PDR [16] and k-induction [39], as well as techniques for reducing the size of the state space, such as program slicing [82] and predicate abstraction [10], and many hybrid techniques that use combinations of symbolic and explicit representations for different elements of the program. We refer the reader to [40, 54] for a comprehensive survey.

Industry: Bounded model checkers for software, such as CBMC [60] have been applied to automotive software [75], verifying bootcode [28] and other industrial software at Amazon Web Services [27].

Running Example: BMC will be able to find a counterexample to assertion 2 with a bound of $k = 1$, which will show that if the array is size 1 and does not contain the target, `last` remains set to -1. It cannot prove assertion 1 and assertion 3 are true, although it can say that they hold up to a bound k.

3.5 Deductive Verification

Robert Floyd (working on flow-charts [46]) and Tony Hoare (working on programs [52]) developed equivalent approaches for manually constructing proofs of program correctness. Tony Hoare's presentation of the ideas as a logic was more widely adopted, leading the approach to be known as *Hoare logic*. Their approach contained two key ideas. First logical formulae are used to represent sets of states at a particular point in the program. The set contains all of the states that make the formula true. This gives the fundamental building block of Hoare logic; the triple:

$$\{Pre\}\ \texttt{Program}\ \{Post\}$$

where Pre and $Post$ are formulae and `Program` is a part of a program. The triple denotes the statement "If the state of the program meets the precondition (Pre is true) then after `Program` has been run the state will meet the postcondition ($Post$ is true)". Hoare logic gives a series of proof rules for how these triples can be constructed. One of these proof rules is the second key idea; that an inductive argument about an invariant set can be used to prove properties of loops:

$$\frac{\{Inv \wedge Cond\}\ \texttt{Body}\ \{Inv\}}{\{Inv\}\ \texttt{while (Cond) Body}\ \ \{Inv \wedge \neg Cond\}}$$

This rule formalises the argument : if the body of a loop takes a state in (the set described by) Inv to another state in Inv, and Inv is true before the loop, then it must be true after the loop. Inv is referred to as an *inductive invariant*. Inductive invariants are both the main strength of Hoare logic and its main cost. They allow a finite, small proof to reason about the behaviour of an unbounded number of traces. However, invariants often have to be created by humans as there is no way of creating suitable invariants automatically for all programs and specifications. This is why deductive verification is on the human assisted corner

of the pyramid. Once the candidate invariants have been provided both they and the specification can be checked automatically. Tools such as Daikon [41] and Houdini [44] have had some success in suggesting routine invariants. The choice of loop invariant is dependent on both the program and the specification. If the invariant is too weak (describes a set with too many elements), it may not be sufficient to prove parts of the specification after the loop. If it is too strong (describes a set with too few elements) then it may not hold before the loop or may not be inductive. Devising a suitable loop invariant is a skilled task and is one of the reasons for the higher skill rating in Table 1.

Dijkstra [36] contributed various ideas to the field of deductive verification. He showed that some of Hoare's rules for constructing tuples could be replaced with an algorithm that transformed one formula into the other. The best known of these *predicate transformers* are the *strongest postcondition* which use the Pre condition and `Program` to compute the most precise $Post$ and the *weakest precondition* which uses the $Post$ condition and `Program` to compute the least restrictive Pre condition. Dijkstra also showed that these techniques could be used to build software from a specification so that it was provable correct and argued forcefully that these *formal methods*[4] were the only professional approach to software engineering. In doing so he provided not only the means but also the motivation for Hoare logic to be used as a verification technology rather than a purely theoretical construct.

Table 4. Verification conditions generated to check the verification conditions in Fig. 3, using the inductive invariant given in Sect. 3.5. For readability, we do not include the formulae that reason about variables which do not change.

$found = 0 \wedge last = -1 \wedge i = 0$	strongest post condition
$(found \neq 0 \implies a[last] = target)$	check invariant
$(found \neq 0 \implies a[last] = target)$	invariant
$i < a.length()$	loop body run
$a[i] = target \Rightarrow found' = found + 1 \wedge last' = i$	execute body
$i' = i + 1$	loop counter update
$0 \leq i < a.length()$	check assertion 1
$(found' \neq last' \implies a'[last'] = target')$	check invariant
$(found \neq 0 \implies a[last] = target)$	invariant
$\neg(i < a.length())$	loop exit
$0 \leq last < a.length()$	check assertion 2
$(found \neq 0 \implies a[last] = target)$	check assertion 3

[4] Software verification is a technique that is used by some formal methods. However there are formal methods which do not use it and uses of software verification in development methodologies which are not traditionally considered formal.

Early uses of Hoare logic were proving the correctness of algorithms. However, there are now many tools in existence that apply deductive verification to actual software. The early tools, such as SPARK [8], were labour intensive and required manual annotations to write pre- and post-conditions. Later tools, such as ESC/Java [45], use weakest precondition/strongest postcondition alongside Hoare's inductive rule for loops to generate assertion s, and then deployed independent theorem provers [45] or SMT solvers [11,43] to check the conditions.

Industry: SPARK [8] has been used in civil and military avionics, railway signalling and cryptographic solutions; Why3 has been used for proof of smart contracts [67]; Boogie [11] is developed and maintained by Microsoft, and has, amongst other things, been used for smart contract verification [80].

Running Example: The first thing we need to do is provide a loop invariant for the loop. We will use $(found \neq 0 \implies a[last] = target)$ (this is the same as assertion 3, and often invariants may be guessed from the properties we wish to prove). We then generate the verification conditions in Table 4, which correspond to the path from the start of the function to the invariant at the top of the loop, the path from the invariant around the loop once, and the path from the invariant exiting the loop via the loop condition. Assertions that cannot be proven mean either the specification is not met or that the invariant is too weak. However, there is no automatic technique that can tell the difference between the two in all cases. In this instance, it is possible to prove assertion 3 always holds but not that assertion 2 fails or assertion 1 always holds. But, if we make the invariant stronger, and use $(found \neq 0 \implies a[last] = target) \wedge i > 0$, we can now prove assertion 1 always holds as well.

3.6 Functional Verification

Functional verification comprises techniques that use mathematical reasoning to show equivalence between functional programs and constructive proofs. The result that it builds on is the connection between function application (β-reduction) in typed lambda calculus and modus ponens in intuistic logic:

$$\frac{t : A \quad \lambda x.E : A \to B}{E[x := t] : B} \qquad \frac{A \quad A \Rightarrow B}{B}$$

This can be seen as giving a logical character to programs; showing that a function $f : A \to B$ is well-typed is equivalent to proving that if t meets the precondition (is of type A), then $f(t)$ meets the postcondition (is of type B). This allows type checking and type inference algorithms to be used as verification tools.

This school of verification is a branch of functional programming as it is limited to programming languages and type systems that have an equivalence

with a suitable logic. These languages tend to be functional as the logical equivalent of mutable state and pointers remain open research questions. As a result, functional verification is most effectively applied to *build* code that is correct by construction rather than to verifying code that already exists. Types play a similar role to annotations in deductive verification systems, giving the specification to be proven and the intermediate steps used to assist the verification tool / type checker. From this point of view, inductive invariants in loops are equivalent to type declarations for recursive functions and the repetition between proofs and programs found in deductive verification is avoided. However, the tight link between types and logic means that the specification must be expressible in the type system. It also requires a high level of skill as both the program and proof must be constructed simultaneously.

Culturally aligned with functional verification but with distinct foundations, there are a number of approaches to verification that use *Interactive Theorem Provers* (ITPs). The user constructs a mathematical proof that the program meets the specification and the ITP then checks this proof.

Industry: Despite the high skill level required, functional verification has been used for various projects with complex functional specifications that required non-trivial proof, e.g., the seL4 project [59] and the CompCert C Compiler [65].

Running Example: As our example program in Figure 3 is written in C we cannot directly demonstrate this style of verification. However, we can consider what would be needed to verify a functional implementation of the same system. To show that all of the array accesses were in bounds we would need an array type that included the length (i.e. (*array* 10) could be a valid type) and we would need an integer type that could express bounds, or a type inference algorithm that could determine them automatically.

4 Specifications

The specification is the set of criteria we check the program against. If you are building or maintaining a system then you need a specification – an understanding of what the system should do. Otherwise, you have no way of saying if the system is working as intended!

In this paper we are interested in specifications that are or can be formalised, i.e., expressed in a language with formal semantics.

There are many different kinds of specifications ("the program must be able to", "the program must always", "the program must never", etc.) and the difficulty of verifying them can vary significantly. Tools and techniques for formal verification are often only applicable or are most suitable for certain kinds or parts of specifications. For example, showing that the program has a print feature is (hopefully!) fairly simple and testing may give sufficient evidence for the verification case. On the other hand, showing that there are no executions of the program with buffer over-runs is harder and will likely need software verification tools to achieve a reasonable degree of certainty.

4.1 Ways of Expressing Specifications

Which specifications can be used and how they are represented depends on the verification tool.

Builtins. One approach is to have a number of specifications built in to the tool and to have the user pick which one(s) they wish to verify against, for example, "no trace executes undefined behaviour". This approach is the easiest from the point of view of the user, setting the specification is ticking a few boxes or setting command-line flags. It is also convenient for tool developers as the verification tool can be specialised to handle the particular specifications supported. However, it is limited; if the tool does not support a relevant specification then it will be of little use, even if the core analysis that it is performing is relevant to the task.

Annotation. Another approach to specification is annotation. *Annotations* are statements in the program that describe a set of traces with reference to the location of the annotation. These may be written in comments, library calls (such as `assert`) or specific language constructs (such as pragmas). One way of classifying annotations is by how they describe the set of traces. They can refer to the state of the program when(ever) it reaches the annotation, for example giving constraints on values (`0 <= i && i < n`). These describe all of the traces that have the required state when they reach the annotation. They can refer to the future behaviour of the traces after they have reached the annotation, for example, termination. They can refer to the past behaviour of the traces before reaching the annotation, for example, taint (this parameter must not be influenced by user input). Implicit in the idea of annotations is a notion of reachability; annotations only apply when a trace reaches their location. This means that the verification tool must determine which parts of the program can be executed and so Rice's Theorem applies (see Sect. 2).

Annotations are more flexible than fixed specifications, they can be developed and maintained in parallel with the software and they can be used in a modular fashion and re-used along with the software. They can also be used to assist the verification tool by providing predicates that the tool can use for modular reasoning (see Sect. 3.5). Some tools such as Dafny [64] make a distinction between annotations that express specifications and those for assisting the proof. Describing specifications by annotations has some disadvantages. It interleaves the specification and implementation, meaning that they often have to be developed together or at least by teams who understand both aspects. They are also harder to review independently of the implementation.

External Objects. A third popular way of providing a specification is an external object, written in some formal language. Examples of this include providing another program as a specification and verifying the *equivalence* of the two (the specification should represent the same set of traces as the program) or providing a more abstract program as a specification and showing that the concrete

program is a *refinement* of it (the traces in the program are a subset of the traces permitted by the specification). These approaches can be very useful if a reference model, protocol or implementation is available or in a hierarchical development approach where one language is used to create a series of progressively more detailed implementations by showing that each is a refinement of the previous one. One recent and promising direction in verification is to use *a previous version* of the software as the specification. This is *differential verification* [62] and allows us to check specifications about the *changes* between versions; "this modification does not introduce new bugs", "the change only affects a bounded amount of the program" or even "this patch definitely blocks a given exploit". Specifications as a separate object allow the most flexibility and reuse as well as giving a good separation between the development of the program and the specification. However, it may require significant extra development (differential verification is the notable exception to this requiring almost no extra effort) potentially as much as developing the program itself.

5 Using the Pyramid

The pyramid model allows us to classify and contrast techniques by which of the three key properties they guarantee and which they use computation to work towards. It is also a useful model for designing verification work-flows and selecting and evaluating appropriate tools.

For a particular project, there will only be a small (finite) number of programs and specifications of interest, thus Turing's result is not directly applicable. It would be possible (in theory) to create a verification tool that could achieve all three properties *for the programs and specifications in that project*. Developing such a project-specific tool is not financially viable for most organisations, so a process needs to be created using existing (or customised) tools. Given a particular program and specification, the key question is whether a reasonable amount of computation will reduce the missing attribute (false alarms, unexplored areas or human effort) by an acceptable amount. This question can only be answered by considering the wider context of the project; what role does the verification of the software play in the correctness, safety, security or performance of the system? What happens if the system is wrong? How much software already exists and how much can it be modified?

For example, if the verification is used to make a claim of code quality then using an over-approximate technique might be fine if the number of defects (including false alarms) is below the required threshold. An under-approximate technique might be suitable for a component with redundancy or fail-safe mechanisms as software defects will cause a loss of service rather than failure. Reducing the number of these is clearly beneficial but it is not necessary to remove all of them as low probability defects have a small impact. If defective software would cause a catastrophic system failure then a human-assisted technique might be most appropriate as a proof of correctness of the software can strengthen the system-wide verification case.

By prioritising the three attributes (automatic, no missed bugs, no false alarms) with respect to the project's goals and the need for software verification, the pyramid model can help select the right kind of tool.

5.1 Process

The human side of using software verification tools covers three aspects; developing the software, developing the specification and dealing with the missing third attribute. The first two of these are common to all approaches and are covered by both formal and non-formal development methodologies. The pyramid model can help inform the third area:

- If an over-approximate technique is used, then human effort will be required to deal with the output which will be a mix of false alarms and genuine defects. This was widely regarded as tedious but tractable.
  Unfortunately there is evidence [38] that even skilled developers are not able to reliably distinguish between false and real alarms.
- If an under-approximate analysis is used then any defects that are found are definitely real. Most tools of this kind will produce a trace or test-case that demonstrates the issue. These are often of considerable value for developers in fixing the problem [26]. For these approaches the missing attribute is 'no missed bugs' and so effort has to be put into increasing coverage. One way of achieving this is to create more fine-grained specifications; similar to unit tests. If checking the whole program leaves areas of the state-space unexplored, these can be used to increase the coverage. For example, if a monolithic under-approximate analysis does not check a particular function, the function can be checked independently. To do this requires a specification that includes the range of values of input variables over which the function is to be verified. The developer experience is likely to be similar to writing unit tests but using constraints to describe a space of possible inputs rather than fixed values.
- Finally if a human-assisted tool is used then the human effort in the process will be in producing the parts of the proof the tool is unable to directly infer. These will typically be pre and post-conditions for functions and loop invariants (see Sect. 3.5) or types (see Sect. 3.6). This requires developers with relevant training or experience.

5.2 Understanding Tool Evaluation

The pyramid model also helps understand how tools are evaluated in academia and industry. If tools are viewed as having two of the three attributes then evaluation is a question of measuring or approximating how each unit of computational effort reduces the number of missed bugs, false alarms or the amount of human-written proof across the set of all programs. As discussed earlier in this section, most tool users only care about the specific programs and specifications that they have. Combined with the obvious difficulty of running experiments

over the (infinite) set of all programs and specifications means that most evaluations are conducted with *benchmarks*, sets of programs and specifications which are claimed to be representative.

Experimental evaluations of over-approximate tools tend to focus on the relative alarm rates or the relative difference between approximations given by different techniques. Underapproximate techniques tend to compare the speed at which different tools solve the same problem(s) (i.e., finding known bugs) or the number solved with given human effort. This is because measuring missed bugs is hard. Human-assisted tools use proxies to estimate the amount of effort needed, for example number of lines of proof, or ratio of lines of proof to lines of code, or the number of human-hours it takes to produce. Note that many of these tools take very different inputs, so it is very hard to compare directly and measuring effort/expertise is very subjective.

6 Conclusion

We have found the pyramid of verification to be an invaluable framework for classifying and choosing verification techniques, for teaching, and for bridging the gap between academics and potential users of verification tools. We hope that this paper enables the reader to do the same.

References

1. Coverity Scan: Static analysis. https://scan.coverity.com/. Accessed 10 Apr 2024
2. Cppcheck: A tool for static C/C++ code analysis. https://cppcheck.sourceforge.io/. Accessed 10 Apr 2024
3. CREST: Concolic test generation tool for C. https://www.burn.im/crest/. Accessed 20 July 2020
4. FindBugs. http://findbugs.sourceforge.net/. Accessed 22 July 2020
5. Fortify static code analyzer. https://www.opentext.com/products/fortify-static-code-analyzer. Accessed 10 Apr 2024
6. MALPAS software static analysis toolset. http://malpas-global.com/. Accessed 10 Apr 2024
7. PolySpace Code Prover. https://www.mathworks.com/products/polyspace-code-prover.html. Accessed 22 July 2020
8. SPARK. https://www.adacore.com/about-spark. Accessed 10 Apr 2024
9. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
10. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Burke, M., Soffa, M.L. (eds.) Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001, pp. 203–213. ACM (2001)
11. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17

12. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-662-07964-5
13. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
14. Biere, A.: Bounded model checking. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481. IOS Press (2009)
15. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Irwin, M.J. (ed.) Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999, pp. 317–320. ACM Press (1999)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
17. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^20 states and beyond. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990, pp. 428–439. IEEE Computer Society (1990)
18. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. USENIX Association (2008)
19. Cadar, C., et al.: Symbolic execution for software testing in practice: preliminary assessment. In: ICSE, pp. 1066–1071. ACM (2011)
20. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
21. Cassé, H., Féraud, L., Rochange, C., Sainrat, P.: Using the abstract interpretation technique for static pointer analysis. SIGARCH Comput. Architect. News **27**(1), 47–50 (1999)
22. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
23. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
24. Chen, D., Huang, R., Qu, B., Jiang, S.: Improving static analysis performance using rule-filtering technique. In: Reformat, M. (ed.) The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013, pp. 19–24. Knowledge Systems Institute Graduate School (2014)
25. Clarke, E., Kroening, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_9

26. Clarke, E., Veith, H.: Counterexamples revisited: principles, algorithms, applications. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_9
27. Cook, B.: Formal reasoning about the security of amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
28. Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. Formal Methods Syst. Des. **57**(1), 34–52 (2021)
29. Coppa, E., D'Elia, D.C., Demetrescu, C.: Rethinking pointer reasoning in symbolic execution. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 613–618. IEEE Computer Society (2017)
30. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
31. Cousot, P., Cousot, R.: Static determination of dynamic properties of generalized type unions. In: Language Design for Reliable Software, pp. 77–94. ACM (1977)
32. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282. ACM Press (1979)
33. Cousot, P., et al.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
34. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
35. D'Abruzzo Pereira, J., Vieira, M.: On the use of open-source C/C++ static analysis tools in large projects. In: 2020 16th European Dependable Computing Conference (EDCC), pp. 97–102 (2020)
36. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453-457 (1975)
37. Dijkstra, E.W.: EWD 1308: What Led to "Notes on Structured Programming". In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 340–346. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-59412-0_19
38. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, pp. 181–192. ACM (2012)
39. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 351–368. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_26
40. D'Silva, V.V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(7), 1165–1178 (2008)
41. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
42. Ferdinand, C.: Worst case execution time prediction by static program analysis. In: IPDPS. IEEE Computer Society (2004)
43. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8

44. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45251-6_29
45. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI, pp. 234–245. ACM (2002)
46. Floyd, R.W.: Assigning meanings to programs. In: Colburn, T.R., Fetzer, J.H., Rankin, T.L. (eds) Program Verification. Studies in Cognitive Systems, vol. 14. Springer, Dordrecht (1993). https://doi.org/10.1007/978-94-011-1793-7_4
47. Gibson-Robinson, T.: FDR3: the future of CSP model checking. In: Welch, P.H., Barnes, F.R.M., Broenink, J.F., Chalmers, K., Pedersen, J.B., Sampson, A.T. (eds.) 35th Communicating Process Architectures, CPA 2013, Edinburgh, Scotland, UK, August 25, 2013, pp. 321–322. Open Channel Publishing Ltd. (2013)
48. Godefroid, P.: Software model checking improving security of a billion computers. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 1–1. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02652-2_1
49. Gopinath, R., Jensen, C., Groce, A.: Code coverage for suite evaluation by developers. In: Jalote, P., Briand, L.C., van der Hoek, A. (eds.) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India, May 31 - June 07, 2014, pp. 72–82. ACM (2014)
50. Havelund, K.: Java PathFinder a translator from Java to Promela. In: Dams, D., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680, pp. 152–152. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48234-2_11
51. Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST software verification system. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 25–26. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_4
52. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
53. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
54. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4), 21:1–21:54 (2009)
55. Johnson, S.C.: Lint, a C program checker, pp. 78–1273 (1978)
56. Kästner, D., Wilhelm, R., Ferdinand, C.: Abstract interpretation in industry - experience and lessons learned. In: In: Hermenegildo, M.V., Morales, J.F. (eds) Static Analysis. SAS 2023. Lecture Notes in Computer Science, vol 14284. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-44245-2_2
57. Kettl, M., Lemberger, T.: The static analyzer infer in SV-COMP (competition contribution). In: TACAS 2022. LNCS, vol. 13244, pp. 451–456. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_30
58. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
59. Klein, G., Elphinstone, K., et al.: seL4: formal verification of an OS kernel. In: SOSP, pp. 207–220. ACM (2009)
60. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
61. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China, June 11 - 16, 2012, pp. 193–204. ACM (2012)

62. Lahiri, S.K., Vaswani, K., Hoare, C.A.R.: Differential static analysis: opportunities, applications, and challenges. In: FoSER, pp. 201–204. ACM (2010)
63. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis and transformation. In: CGO, pp. 75–88. IEEE Computer Society (2004)
64. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
65. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
66. Logozzo, F.: Practical specification and verification with code contracts. In: HILT, pp. 7–8. ACM (2013)
67. Nehaï, Z., Bobot, F.: Deductive proof of industrial smart contracts using Why3. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 299–311. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_22
68. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
69. Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 230–266. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04652-0_5
70. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
71. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: an analysis and survey. Adv. Comput. **112**, 275–378 (2019)
72. Pasareanu, C.S., et al.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: ISSTA, pp. 15–26. ACM (2008)
73. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Trans. Am. Math. Soc. **74**, 358–366 (1953)
74. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: MacQueen, D.B., Cardelli, L. (eds.) POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19–21, 1998, pp. 38–48. ACM (1998)
75. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 62–77. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_5
76. Shen, H., Fang, J., Zhao, J.: EFindBugs: effective error ranking for findBugs. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011, pp. 299–308. IEEE Computer Society (2011)
77. Tillmann, N., de Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
78. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. Proc. London Math. Soc. **s2-42**(1), 230–265 (1937)
79. Vernier-Mounier, I.: Symbolic executions of symmetrical parallel programs. In: 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), January 24-26, 1996, Portugal, pp. 327–335. IEEE Computer Society (1996)

80. Wang, Y., et al.: Formal verification of workflow policies for smart contracts in Azure Blockchain. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE 2019. LNCS, vol. 12031, pp. 87–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_7
81. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. In: Deusen, M.S.V., Galil, Z., Reid, B.K. (eds.) Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985, pp. 291–299. ACM Press (1985)
82. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)
83. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6