Int J Softw Tools Technol Transfer (2007) 9:505–525
DOI 10.1007/s10009-007-0044-z

SPECIAL SECTION FASE'04/05

The software model checker BLAST

Applications to software engineering

Dirk Beyer · Thomas A. Henzinger · Ranjit Jhala · Rupak Majumdar

Published online: 13 September 2007

Abstract BLAST is an automatic verification tool for checking temporal safety properties of C programs. Given a C program and a temporal safety property, BLAST either statically proves that the program satisfies the safety property, or provides an execution path that exhibits a violation of the property (or, since the problem is undecidable, does not terminate). BLAST constructs, explores, and refines abstractions of the program state space based on lazy predicate abstraction and interpolation-based predicate discovery. This paper gives an introduction to BLAST and demonstrates, through two case studies, how it can be applied to program verification and test-case generation. In the first case study, we use BLAST to statically prove memory safety for C programs. We use CCURED, a type-based memory-safety analyzer, to annotate a program with run-time assertions that check for safe memory operations. Then, we use BLAST to remove as many of the run-time checks as possible (by proving that these checks never fail), and to generate execution scenarios that violate the assertions for the remaining run-time checks. In our second case study, we use BLAST to automatically generate test suites that guarantee full coverage with respect to a given predicate. Given a C program and a target predicate p, BLAST determines the program locations q for which there exists a program execution that reaches q with p true, and automatically generates a set of test vectors that cause such executions. Our experiments show that BLAST can provide automated, precise, and scalable analysis for C programs.

Keywords Model checking · Software verification · Software specification · Memory safety · Test-case generation

D. Beyer (✉)
Simon Fraser University, Surrey, BC, Canada

T. A. Henzinger
EPFL, Lausanne, Switzerland

R. Jhala
University of California, San Diego, USA

R. Majumdar
University of California, Los Angeles, USA

1 Introduction

Model checking is an algorithmic technique to verify a system description against a specification [20,22,75]. Given a system description and a logical specification, the model-checking algorithm either proves that the system description satisfies the specification, or reports a counterexample that violates the specification. *Software model checking*, the application of algorithmic verification to implemented code, has been an active area of recent research [4,17,28,35,42,52,66]. The input to a software model checker is the program source (system description) and a temporal safety property (specification). The specification is usually given by program instrumentation that defines a monitor automaton [5,7,41,79], which observes if a program execution violates the desired property, such as adherence to a locking or security discipline. The output of the model checker is ideally either a proof of program correctness that can be separately validated [47,68], or a counterexample in the form of a specific execution path of the program.

A key paradigm behind some of the new software verification tools is the principle of *counterexample-guided abstraction refinement* (CEGAR) [1,3,4,21,49,61,78]. In this paradigm, the model checker attempts to verify the property starting with a coarse abstraction of the program, which

tracks only a few relations (called *predicates*) between program variables, instead of the full program state. By the conservative nature of this abstraction, if the verification succeeds, then one is guaranteed that the concrete program satisfies the specification. If the verification fails, then it produces a path that violates the specification in the abstract program. This path may either correspond to a concrete program execution (*feasible* path) which violates the specification, or arise due to the imprecision of the abstraction, and thus not correspond to a concrete program execution (*infeasible* path). In the former case, a program bug has been found. In the latter case, the infeasibility of the abstract error path is used to automatically deduce additional predicates which encode relevant facts about the program variables. By tracking the values of these additional predicates, the abstraction of the program is refined in a way that guarantees that subsequent verification attempts, which use the refined abstraction, will not produce the previously encountered infeasible error path. The entire process is repeated, by discovering and tracking an ever increasing number of predicates, until either a feasible path that witnesses a program bug (the so-called *counterexample*) is found, or the abstraction is precise enough to prove the absence of such paths (or, since the verification problem is undecidable, the iteration of refinements never terminates). The result is a sound (no violations of the specification are missed) and precise (no false alarms are generated) program verification algorithm.

The scheme of counterexample-guided predicate abstraction refinement was first implemented for verifying software by the SLAM project [4], and applied successfully to find bugs in device drivers. The basic scheme was improved by the BLAST model checker. First, relevant predicates are discovered locally and independently at each program location as interpolants between the past and the future fragments of an infeasible error path (*interpolation-based predicate discovery*) [46]. Second, instead of constructing an abstraction of the program which tracks all relevant predicates, the discovered new predicates are added and tracked locally in some parts of a tree that represents the abstract executions of the program, namely, in those parts where the infeasible error path occurred (*lazy predicate abstraction*) [49]. The resulting program abstraction is nonuniform, in that different predicates are tracked at different program locations, possibly even at different visits to the same location. This emphasis on parsimonious abstractions renders the analysis scalable for large programs: BLAST has been used to verify temporal safety properties of C programs with up to 50 K lines of code [57].

In this article we provide a tutorial introduction to the BLAST model checker, and demonstrate its use in program analysis and software testing through two case studies. The first study [9] uses BLAST to check run-time assertions on the safe usage of pointers, which ensures a form of *memory safety*. The second study [6] uses the abstract and symbolic state exploration capabilities of BLAST to generate *test cases* that meet a certain coverage goal. We proceed with a brief overview of software verification tools that are related to BLAST, and then we explain the two case studies.

Software model checking. Program verification has been a central problem since the early days of computer science [34,50,63,64]. The area has received much research attention in recent years owing to new algorithmic techniques (such as CEGAR) and faster computers, which have renewed the promise of providing automatic tools for checking program correctness and generating counterexamples [51].

Automatic software verification tools can be broadly classified into *execution-based* and *abstraction-based* approaches. Execution-based model checkers [2,35,42,52,66] exhaustively search the concrete state space of a program. The emphasis of this approach is on finding bugs rather than proving correctness, and the main challenge is scaling the search to large state spaces. Abstraction-based verifiers [12,28,31,62] compute an abstraction of the state space. The emphasis is on proving correctness by demonstrating the absence of bugs in the abstract domain, and the main challenge is improving the precision of the analysis. Traditionally, abstractions are specified manually by providing a lattice of dataflow facts to be tracked.

CEGAR techniques combine aspects of both execution-based and abstraction-based approaches. The CEGAR loop tries to automate the process of finding a suitable abstraction by stepwise abstraction refinement, which is guided by searching for abstract error paths and relating them to the concrete state space. The promise of CEGAR is that it will automatically adjust the precision of an abstraction for the purpose of proving the property of interest, while at the same time keeping the abstract state space small enough to be efficiently searched. Implementations of CEGAR include SLAM, BLAST, MAGIC, MOPED, SATABS, and F- SOFT [4,17,23,33,54]. The integration of execution-based and abstraction-based techniques has been carried even further in several recent projects: abstraction-based model checkers can be accelerated by exploring concrete program executions [38,60], and execution-based tools can be augmented to propagate symbolic inputs [36,80].

Checking memory safety. Invalid memory access is a major source of program failures. If a program statement dereferences a pointer that points to an invalid memory cell, the program is either aborted by the operating system or, often worse, the program continues to run with an undefined behavior. To avoid the latter, one can perform checks before every memory access at run time. For some programming languages (e.g., Java) this is done automatically by the compiler/run-time environment, at considerable performance cost. For C, neither the compiler nor the run-time

environment enforces memory-safety policies. As a result, C programmers often face program crashes (or worse, security vulnerabilities) whose cause can be traced back to improper access of memory.

Memory safety is a fundamental correctness property, and therefore much recent research interest has focused on proving the memory safety of C programs statically, possibly coupled with additional run-time assertions when the static analysis is inadequate [13,26,43,69,81]. For example, CCURED [26,71] is a program-transformation tool for C which transforms any given C program to a memory-safe version. CCURED uses a type-based program analysis to prove as many memory accesses as possible memory safe, and inserts run-time checks before the remaining memory accesses, which it could not prove safe statically. The resulting, "cured" C program is memory safe in the sense that it raises an exception if the program is about to execute an unsafe operation. We identify two directions in which a model checker can extend the capabilities of CCURED. First, the remaining run-time checks consume processor time, so a deeper analysis that can prove that some of these checks will never fail allows the removal of checks, leading to performance gains. Second, instead of providing late feedback, just before the program aborts, it is better to know at compile time if the program is memory safe, and to identify execution scenarios that can break memory safety.

We address these two points by augmenting CCURED with the more powerful, path-sensitive analysis performed by BLAST. For each memory access that the type-based analysis of CCURED fails to prove safe, we invoke the more precise, more expensive analysis of BLAST. There are three possible outcomes. First, BLAST may be able to prove that the memory access is safe (even though CCURED was not able to prove this). In this case, no run-time check needs to be inserted, thus reducing the overhead in the cured program. Second, BLAST may be able to generate a feasible path to an invalid pointer dereference at the considered program location, i.e., a program execution along which the run-time check inserted by CCURED would fail. This may expose a program bug, which can, based on the counterexample provided by BLAST, then be fixed by the programmer. Third, BLAST may time-out attempting to check whether or not a given memory access is always safe. In this case, the run-time check inserted by CCURED remains in the cured program. It is important to note that BLAST, even though often more powerful than CCURED, is invoked only after a type-based pointer analysis fails. This is because where successful, the CCURED analysis is more efficient, and may succeed in cases that overwhelm the model checker. However, the combination of CCURED and BLAST guarantees memory-safe programs with less run-time overhead than the use of CCURED alone, and it provides useful compile-time feedback about memory-safety violations to the programmer.

Generating test cases. Model checking requires a specification. In the absence of particular correctness assertions, the software engineer may still be interested in a set of inputs that exercise the program "enough," for example, by meeting certain coverage goals. An example coverage goal may be to find the set of *all* program locations q that program execution can reach with the value of a predicate p being true, or false, when q is reached. For example, when checking security properties of programs, it is useful to find all instructions that the program may execute with root privileges. Furthermore, instead of a particular path through the program, it is often more useful to obtain a *test vector*, that is, a map from program inputs to values that force the program execution along the given path. This is because the program structure may change, eliminating the path, while the input vector can still act as a regression test for newer versions of the program.

We have extended BLAST to provide test suites for coverage goals. In the special case that p is *true*, BLAST can be used to find the reachable program locations, and by complementation, it can detect dead code. Moreover, if BLAST claims that a certain program location q is reachable such that the target predicate p is true at q, then from a feasible abstract path that leads to p being true at q, we can automatically produce a test vector that witnesses the truth of p at q. This feature enables a software engineer to pose reachability queries about the behavior of a program, and to automatically generate test vectors that satisfy the queries [73]. Technically, we symbolically execute the feasible abstract path produced by the model checker, and extract a test vector from a satisfying assignment for the symbolic path constraints. In particular, given a predicate p, BLAST automatically generates for each program location q, if p is always true at q, a test vector that exhibits the truth of p at q; if p is always false at q, a test vector that exhibits the falsehood of p at q; and if p can be both true and false at q, depending on the program path to q, then two test vectors, one that exhibits the truth of p at q, and another one that exhibits the falsehood of p at q.

Often a single test vector covers the truth of p at many locations, and the falsehood of p at others, and BLAST heuristically produces a small set of test vectors that provides the desired information. Moreover, in order to scale, BLAST uses incremental model checking [48], which reuses partial proofs as much as possible. We have used our extension of BLAST to query C programs about locking disciplines, security disciplines, and dead code, and to automatically generate corresponding test suites.

There is a rich literature on test-vector generation using symbolic execution [24,37,40,56,58,59,76]. Our main insight is that given a particular target, one can guide the search to the target efficiently by searching only an *abstract* state space, and refining the abstraction to prune away infeasible paths to the target found by the abstract search.

This is exactly what the model checker does. In contrast, unguided symbolic execution-based methods have to execute many more program paths, resulting in scalability problems. Therefore, most research on symbolic execution-based test generation curtails the search by bounding, e.g., the number of iterations of loops, the depth of the search, or the size of the input domain [15,36,55,58]. Unfortunately, this makes the results incomplete: if no path to the target is found, one cannot conclude that no execution of the program reaches the target. Of course, once a suitable program path to the target is found, all previous methods to generate test vectors still apply.

This is not the first attempt to use model-checking technology for automatic test-vector generation. However, the previous work in this area has followed very different directions. For example, the approach of Hong et al. [53] considers fixed boolean abstractions of the input program, and does not automatically refine the abstraction to the degree necessary to generate test vectors that cover all program locations for a given set of observable predicates. Peled proposes three further ways of combining model checking and testing [74]. Black-box checking and adaptive model checking assume that the actual program is not given at all, or not given fully. Unit checking is the closest to our approach in that it generates test vectors from program paths; however, these paths are not found by automatic abstraction refinement [39].

Organization. Section 2 introduces the techniques implemented in the software model checker BLAST. We explain how the algorithm finds abstract paths to specified program locations, and how infeasible abstract paths are used to refine the abstraction. Section 3 details the first case study on checking memory safety using BLAST. Section 4 shows how test vectors are generated from feasible abstract paths, and how sufficiently many such paths are obtained to guarantee coverage for the resulting test suite. We conclude each of the two application sections by presenting experimental results. Section 5 summarizes the current state of the BLAST project.

2 Software model checking with BLAST

The software model checker BLAST is based on the principle of counterexample-guided abstraction refinement. We illustrate how BLAST combines lazy abstraction and interpolation-based, localized predicate discovery on a running example.

2.1 Example

The example program (shown in Fig. 1) consists of three functions. Function `altInit` has three formal parameters: `size`, `pval1`, and `pval2`. It allocates and initializes a global array `a`. The size of the allocated array is given by `size`.

```
#include <stdio.h>
#include <stdlib.h>
int *a;

void myscanf(const char* format, int* arg) {
  *arg = rand();
}

int altInit(int size, int *pval1, int *pval2) {
 1:  int i, *ptr;
 2:  a = (int *) malloc(sizeof(int) * (size+1));
 3:  if (a == 0) {
 4:    printf("Memory exhausted.");
 5:    exit(1);
 6:  }
 7:  i = 0; a[0] = *pval1;
 8:  while(i < size) {
 9:    i = i + 1;
10:    if (i % 2 == 0) {
11:      ptr = pval1;
12:    } else {
13:      ptr = pval2;
14:    }
15:    a[i] = *ptr;
16:    printf("%d. iteration", i);
17:  }
18:  if (ptr == 0) ERR: ;
19:  return *ptr;
}

int main(int argc, char *argv []) {
20:  int *pval = (int *) malloc(sizeof(int));
21:  if (pval == 0) {
22:    printf("Memory exhausted.");
23:    exit(1);
24:  }
25:  *pval = 0;
26:  while(*pval <= 0) {
27:    printf("Give a number greater zero: ");
28:    myscanf("%d", pval);
29:  }
30:  return altInit(*pval, pval, pval);
}
```

Fig. 1 Example C program

The array is initialized with an alternating sequence of two values, pointed to by the pointers `pval1` and `pval2`. After the initialization is completed, the last value of the sequence is returned to the caller. Function `main` is a simple test driver for function `altInit`. It reads an integer number from standard input and ensures that it gets a value greater than zero. Then it calls function `altInit` with the read value as parameter for the size as well as for the two initial values. Finally, the stub function `myscanf` models the behavior of the C library function `scanf`, which reads input values. The stub `myscanf` models arbitrary user input by returning a random integer value.

2.2 Control-flow automata

Internally, BLAST represents a program by a set of *control-flow automata* (CFAs), one for each function of the program.

A CFA is a directed graph, with locations corresponding to control points of the program (program-counter values), and edges corresponding to program operations. An edge between two locations is labeled by the instruction that executes when control moves from the source to the destination; an instruction is either a *basic block* of assignments, an *assume predicate* corresponding to the condition that must hold for control to proceed across the edge, a *function call* with call-by-value parameters, or a *return* instruction. Any C program can be converted to this representation [70]. Figures 2 and 3 show the CFAs for the functions `main` and `altInit`, respectively. In Fig. 3 the error location with label `1#22` is depicted by a filled ellipse. We now describe how BLAST checks that the label `ERR` (or equivalently, the error configuration consisting of error location `1#22` and the error predicate *true*) is not reached along any execution of the program. While the actual reason for correctness is simple, the example shows that the analysis to prove safety must be interprocedural and path-sensitive.

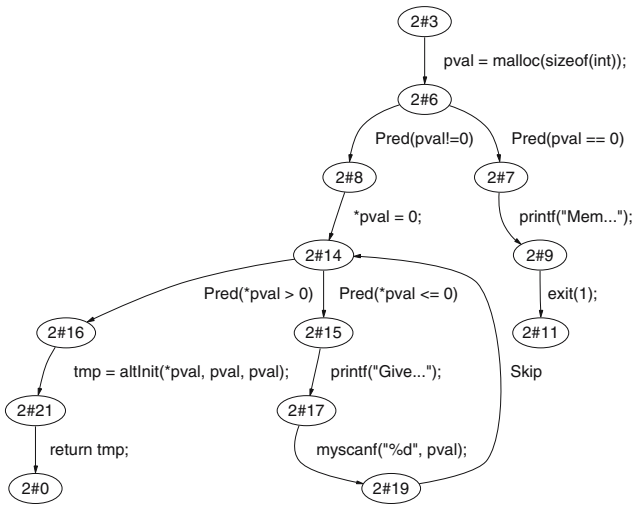

Fig. 2 CFA for function `main`

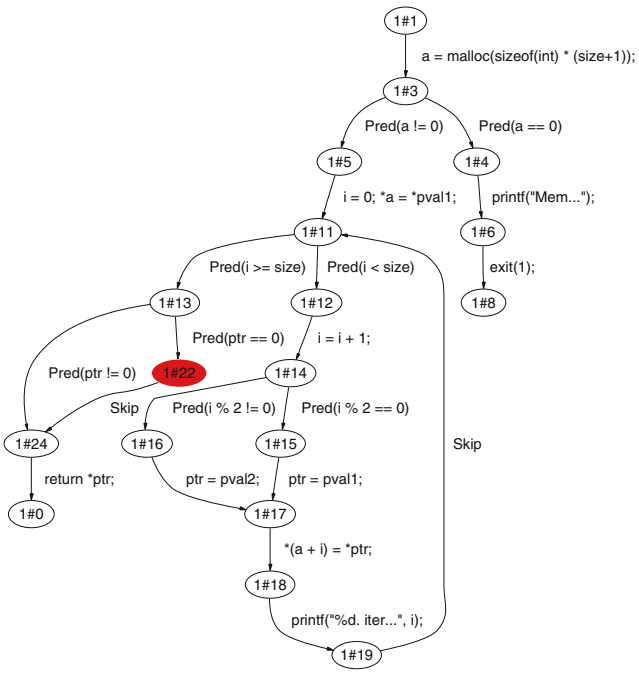

Fig. 3 CFA for function `altInit`

2.3 Abstract reachability trees

The implementation of BLAST uses a context-free reachability algorithm [77] to compute an approximation of the reachable set of states. For ease of exposition, we illustrate the algorithm with a simpler version of reachability, where function calls are inlined (which, unlike context-free reachability, does not handle recursive calls). While a simplification, this version already illustrates the basic features of the algorithm, namely, abstract reachability, counterexample analysis, and predicate discovery, while avoiding some technical aspects of context-free reachability.

In order to prove that the error configuration is never reached, BLAST constructs an *abstract reachability tree* (ART). An ART is a labeled tree that represents a portion of the reachable state space of the program. Each node of the ART is labeled with a location of a CFA, the current call stack (a sequence of CFA locations representing return addresses), and a boolean formula (called the *reachable region*) representing a set of data states. We denote a labeled tree node by $\mathtt{n} : (q, s, \varphi)$, where `n` is the tree node, q is the CFA location, s is the call stack, and φ is the reachable region. Each edge of the tree is marked with a basic block, an assume predicate, a function call, or a return. A path in the ART corresponds to a program execution. The reachable region of a node `n` describes an overapproximation of the reachable states of the program assuming execution follows the sequence of operations labeling the path from the root of the tree to `n`.

Given a region (set of data states) φ and program operation (basic block or assume predicate) op, let $post(\varphi, \mathsf{op})$ be the region reachable from φ by executing the operation op. For a function call op, let $post(\varphi, \mathsf{op})$ be the region reachable from φ by assigning the actual parameters to the formal parameters of the called function. For a return instruction op and variable `x`, let $post(\varphi, \mathsf{op}, \mathtt{x})$ be the region reachable from φ by assigning the return value to `x`. An ART is *complete* if (1) the root is labeled with the initial states of the program; (2) the tree is closed under postconditions, that is, for every internal node $\mathtt{n} : (q, s, \varphi)$ of the tree with φ is satisfiable,

(2a) if $q \xrightarrow{\mathsf{op}} q'$ is an edge in the CFA of q and op is a basic block or assume predicate, then there is a successor node $\mathtt{n}' : (q', s, \varphi')$ of `n` in the tree such that the edge $(\mathtt{n}, \mathtt{n}')$ is marked with op and $post(\varphi, \mathsf{op}) \Rightarrow \varphi'$;

(2b) if $q \xrightarrow{\mathsf{op}} q'$ is a CFA edge and op is a function call, then there is an op-successor $\mathtt{n}' : (q'', s', \varphi')$ in the tree such that q'' is the initial location of the CFA of the called function, the call stack s' results from pushing the return location q' together with the left-hand-side variable of the function call onto s, and $post(\varphi, \mathsf{op}) \Rightarrow \varphi'$;

(2c) if $q \xrightarrow{\mathsf{op}} q'$ is a CFA edge and op is a return instruction, then there is an op-successor $\mathtt{n}' : (q'', s', \varphi')$ in the tree such that $(q'', \mathtt{x})$ is the top of the call stack s, the new call stack s' results from popping the top of s, and $post(\varphi, \mathsf{op}, \mathtt{x}) \Rightarrow \varphi'$;

and (3) for every leaf node $\mathtt{n} : (q, s, \varphi)$ of the tree, either q has no outgoing edge in its CFA, or φ is not satisfiable, or there exists an internal tree node $\mathtt{n}' : (q, s, \varphi')$ such that $\varphi \Rightarrow \varphi'$. In the last case, we say that n is *covered* by $\mathtt{n}'$, as every program execution from n is also possible from $\mathtt{n}'$. A complete ART overapproximates the set of reachable states of a program. Intuitively, a complete ART is a finite unfolding of a set of CFAs whose locations are annotated with regions and call stacks, and whose edges are annotated with corresponding operations from the CFAs. A complete ART is *safe* with respect to a *configuration* (q, p), where q is a CFA location (the *error* location) and p is a predicate over program variables (the *error* region), if for every node $\mathtt{n} : (q, \cdot, \varphi)$ in the tree, we have $\varphi \wedge p$ is not satisfiable. A complete ART that is safe for configuration (q, p) serves as a certificate (proof) that no execution of the program reaches a state where the location is q and the data state satisfies p.

Theorem 1 [49] *Let C be a set of CFAs, T a complete ART for C, and p a predicate. For every location q of C, if T is safe with respect to* (q, p)*, then no state that has q as location and whose data state satisfies p is reachable in C.*

Figure 4 shows a complete ART for our example program. We omit the call stack for clarity. Each node of the tree is labeled with a CFA location (we use primed labels to distinguish different nodes of the same CFA location), and the reachable region is depicted in the associated rectangular box. The reachable region is the conjunction of the list of predicates in each box (our example does not contain any disjunctions). Notice that some leaf nodes in the tree are marked COVERED. Since this ART is safe for the error location 1#22, this proves that the label ERR cannot be reached in the program. Notice that the reachable region at a node is an overapproximation of the concretely reachable states in terms of some suitably chosen set of predicates. For example, consider the edge $\mathtt{1\#16} \xrightarrow{\mathtt{ptr=pval2}} \mathtt{1\#17}$ in the CFA. Starting from the region

$$\mathtt{pval1} \neq 0 \wedge \mathtt{pval2} \neq 0 \wedge \mathtt{size} \geq 1 \wedge \mathtt{i} \neq 0,$$

the set of states that can be reached by the assignment ptr=pval2 is

$$\mathtt{pval1} \neq 0 \wedge \mathtt{pval2} \neq 0 \wedge \mathtt{size} \geq 1 \wedge \mathtt{i} \neq 0 \wedge \mathtt{ptr} = \mathtt{pval2}.$$

However, the tree maintains an overapproximation of this set of states, namely,

$$\mathtt{pval1} \neq 0 \wedge \mathtt{pval2} \neq 0 \wedge \mathtt{size} \geq 1 \wedge \mathtt{i} \neq 0 \wedge \mathtt{ptr} \neq 0,$$

which loses the fact that ptr now contains the same address as pval2. This overapproximation is precise enough to show that the ART is safe for the location 1#22.

Overapproximating is crucial in making the analysis scale, as the cost of the analysis grows rapidly with increased precision. Thus, the safety-verification algorithm must (1) *find* an abstraction (a mapping of control locations to predicates) which is precise enough to prove the property of interest, yet coarse enough to allow the model checker to succeed, and (2) efficiently *explore* (i.e., model check) the abstract state space of the program.

2.4 Counterexample-guided abstraction refinement

BLAST solves these problems in the following way. It starts with a coarse abstraction of the state space and attempts to construct a complete ART with the coarse abstraction. If this complete ART is safe for the error configuration, then the program is safe. However, the imprecision of the abstraction may result in the analysis finding paths in the ART leading to the error configuration which are infeasible during the execution of the program. BLAST runs a counterexample-analysis algorithm that determines whether the path to the error configuration is feasible (i.e., there is a program bug) or infeasible. In the latter case it refines the current abstraction using an *interpolation-based predicate-discovery* algorithm, which adds predicates locally to rule out the infeasible error path. For a given abstraction (mapping of control locations to predicates), BLAST constructs the ART on-the-fly, stopping and running the counterexample analysis whenever a path to the error location is found in the ART. The refinement procedure refines the abstraction locally, and the search is resumed on the nodes of the ART where the abstraction has been refined. The parts of the ART that have not been affected by the refinement are left intact. This algorithm is called *lazy abstraction*; we now describe how it works on our example.

Constructing the ART. Initially, BLAST starts with no predicates, and attempts to construct an ART. The ART construction proceeds by unrolling the CFAs and keeping track of the reachable region at each CFA location. We start with the initial location of the program (in our example the first location of main), with the reachable region *true* (which represents an arbitrary initial data state). For a tree node $\mathtt{n} : (q, s, \varphi)$,

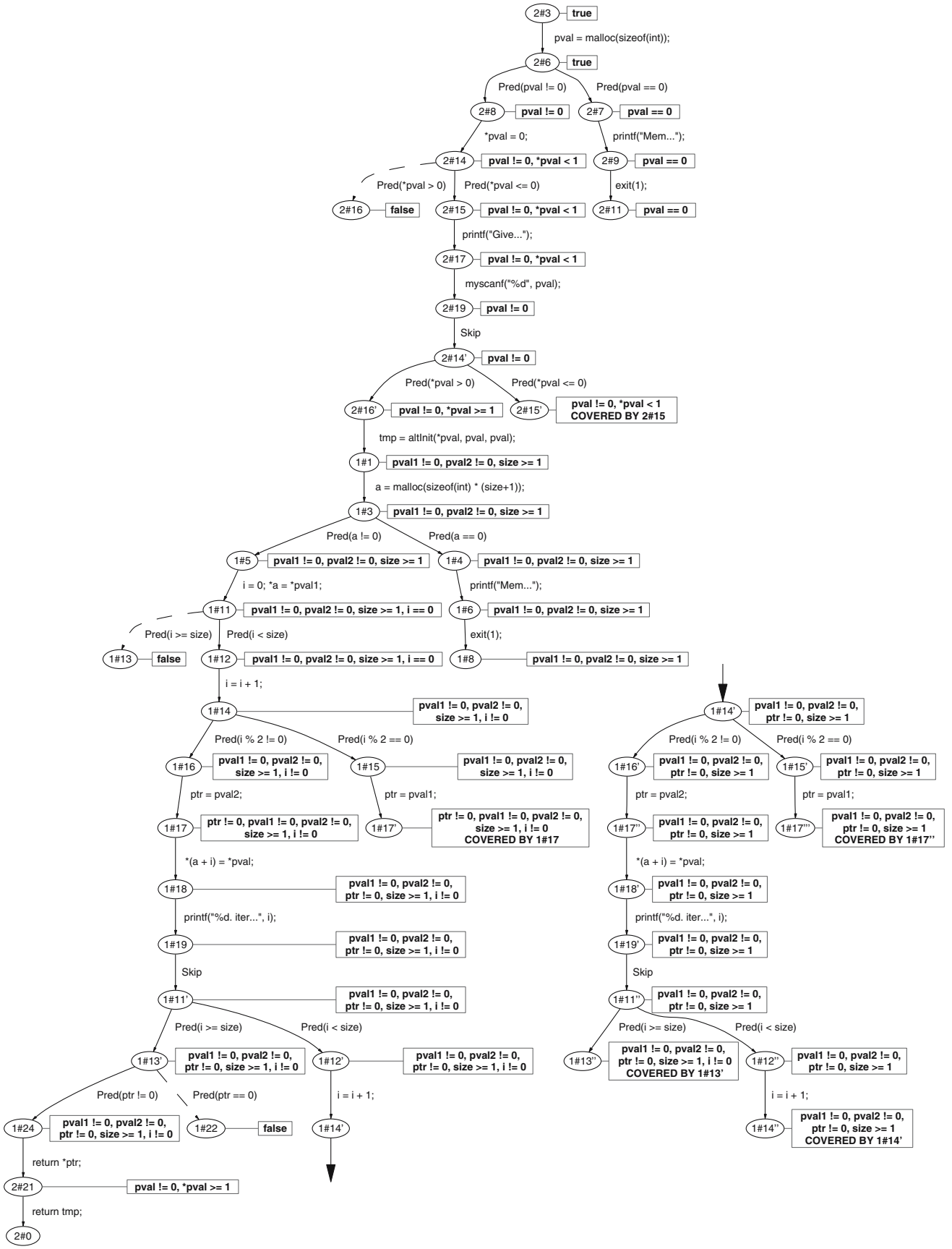

Fig. 4 Complete ART

we construct successor nodes of n in the tree for all edges $q \xrightarrow{\mathsf{op}} q'$ in the CFA. The successor nodes are labeled with overapproximations of the set of states reachable from (q, s, φ) when the corresponding operations op are performed. For the first iteration, since we do not track any facts (predicates) about variable values, all reachable regions are overapproximated by *true* (that is, the abstraction assumes that every data state is possible). With this abstraction, BLAST finds that the error location may be reachable in the example. Figure 5 shows the ART when BLAST finds the first path to the error location. This ART is not complete, because some nodes have not yet been processed. In the figure, all nodes with incoming dotted edges (e.g., the node 2#7) have not been processed. However, the incomplete ART already contains an error path from node 2#3 to 1#22 (the error node is depicted as a filled ellipse).

Counterexample analysis. At this point, BLAST invokes the counterexample-analysis algorithm, which checks if the error path is feasible in the concrete program (i.e., the program has a bug), or whether it arises because the current abstraction is too coarse. To analyze the abstract error path, BLAST creates a set of constraints—called the *path formula* (PF)—which is satisfiable if and only if the path is feasible in the concrete program [25,59]. The PF is built by transforming the path into static single-assignment form [30] (every variable is assigned a value at most once, which is achieved by introducing new special variables), and then generating constraints for each operation along the path.

For the abstract error path of the example, the path formula is shown in Fig. 6. Note that in this example, each program variable occurs only once at the left-hand-side of an assignment; if, for instance, the program variable pval were assigned a value twice along the path, then the result of the first assignment would be denoted by the special variable $\langle \mathtt{pval}, 1\rangle$ and the result of the second assignment would be denoted by the special variable $\langle \mathtt{pval}, 2\rangle$. The path formula is unsatisfiable, and hence the error path is not feasible. There are several reasons why this path is not feasible. First, we set $*\mathtt{pval}$ to 0 in main, and then take the branch where $*\mathtt{pval} > 0$. Furthermore, we check in main that $*\mathtt{pval} > 0$, and pass $*\mathtt{pval}$ as the argument size to altInit. Hence, $\mathtt{size} > 0$. Now, we set i to 0, and then check that $\mathtt{i} \geq \mathtt{size}$. This check cannot succeed, because i is zero, while size is greater than 0. Thus, the path cannot be executed and represents an infeasible error path.

Predicate discovery. The predicate-discovery algorithm takes the path formula and finds new predicates that must be added to the abstraction in order to rule out the infeasible error path. The key to adding predicates is the notion of an *interpolant*. For a pair of formulas φ^- and φ^+ such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a *Craig interpolant* [29] ψ is a formula such that (1) the implication $\varphi^- \Rightarrow \psi$ is valid, (2) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and (3) ψ only contains symbols that are common to both φ^- and φ^+. For the theory of linear arithmetic with uninterpreted functions, which is implemented in BLAST, such a Craig interpolant is guaranteed to exist [65]. The refinement algorithm cuts the infeasible error path at every node. At each cut point, the part of the path formula corresponding to the path fragment up to the cut point is φ^-, and the part of the formula corresponding to the path fragment after the cut point is φ^+ (the cuts are more complicated for function calls; see [46] for details).

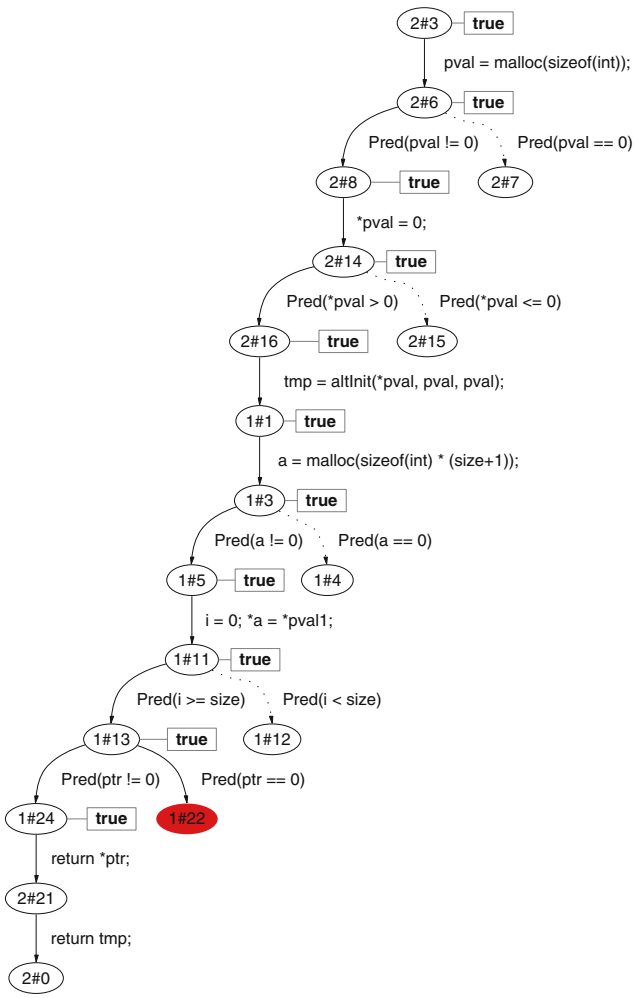

Fig. 5 ART when the first infeasible error path is found

$$\left.\begin{array}{l}\langle \mathtt{pval},1\rangle = \mathtt{malloc}_0 \wedge \langle \mathtt{pval},1\rangle \neq 0 \wedge \\ \langle *(\langle \mathtt{pval},1\rangle),1\rangle = 0 \wedge \langle *(\langle \mathtt{pval},1\rangle),1\rangle > 0 \wedge\end{array}\right\} \text{function main}$$

$$\left.\begin{array}{l}\langle \mathtt{size},1\rangle = \langle *(\langle \mathtt{pval},1\rangle),1\rangle \wedge \\ \langle \mathtt{pval1},1\rangle = \langle \mathtt{pval},1\rangle \wedge \\ \langle \mathtt{pval2},1\rangle = \langle \mathtt{pval},1\rangle \wedge\end{array}\right\} \text{formals assigned actuals}$$

$$\left.\begin{array}{l}\langle \mathtt{a},1\rangle = \mathtt{malloc}_1 \wedge \langle \mathtt{a},1\rangle \neq 0 \wedge \\ \langle \mathtt{i},1\rangle = 0 \wedge \langle \mathtt{i},1\rangle \geq \langle \mathtt{size},1\rangle \wedge \\ \langle \mathtt{ptr},1\rangle = 0\end{array}\right\} \text{function altInit}$$

Fig. 6 Path formula for the infeasible error path of Fig. 5

Then, the interpolant at the cut point represents a formula over the live program variables that contains the reachable region after the path up to the cut point is executed (by property (1)), and is sufficient to show that the rest of the path is infeasible (by property (2)). The live program variables are represented by those special variables which occur both up to and after the cut point (by property (3)). In order to eliminate the infeasible error path from the ART, the interpolation procedure constructs all interpolants from the same proof of unsatisfiability in such a way that the interpolant at node n_{i+1} is implied by the interpolant at node n_i and the operation from n_i to n_{i+1}, that is, the constructed interpolants are inductive.

For example, consider the cut at node `2#16`. For this cut, formula φ^- (resp. φ^+) is the part of the formula in Fig. 6 above (resp. below) the horizontal line. The common symbols across the cut are $\langle$`pval`, 1$\rangle$ and $\langle *(\langle$`pval`, 1$\rangle)$, 1$\rangle$, and the interpolant is $\langle *($`pval`, 1$)$, 1$\rangle \geq 1$. Relating the special variable $\langle *($`pval`, 1$)$, 1$\rangle$ back to the program variable `*pval`, this suggests that the fact `*pval` ≥ 1 suffices to prove the error path infeasible. This predicate is henceforth necessary at location `2#16`. Similarly, at locations `1#1`, `1#3`, and `1#5`, the interpolation procedure discovers the predicate `size` ≥ 1, and at location `1#11`, the predicates `size` ≥ 1 and `i` $= 0$ are found. After adding these predicates, BLAST refines the ART, now tracking the truth or falsehood of the newly found predicates at the locations where they are needed. In the example shown in this paper, we add the new predicate about variable `x` not just at the location dictated by the interpolant, but at all locations that are in the scope of `x`. This heuristic avoids adding the same predicates in multiple refinement steps; it is implemented in BLAST as an option.

Refining the ART. When BLAST refines the ART with the new abstraction, it only reconstructs subtrees that are rooted at nodes where new predicates have been added. In the example, a second abstract error path is found. Figure 7 shows the ART when this happens. Notice that this time, the reachable regions are not all *true*; instead they are overapproximations, at each node of the ART, of the reachable data states in terms of the predicates that are tracked at the node. For example, the reachable region at the first occurrence of location `2#14` in the ART is `*pval` < 1 (the negation of the tracked predicate `*pval` ≥ 1), because `*pval` is set to 0 when proceeding from `2#8` to `2#14`, and `*pval` < 1 is the abstraction of `*pval` $= 0$ in terms of the tracked predicates. This more precise reachable region disallows certain CFA paths from being explored. For example, again at the first occurrence of location `2#14`, the ART has no left successor with location `2#16`, because no data state in the reachable region `*pval` < 1 can take the program branch with the condition `*pval` > 0 (recall that `*pval` is an integer).

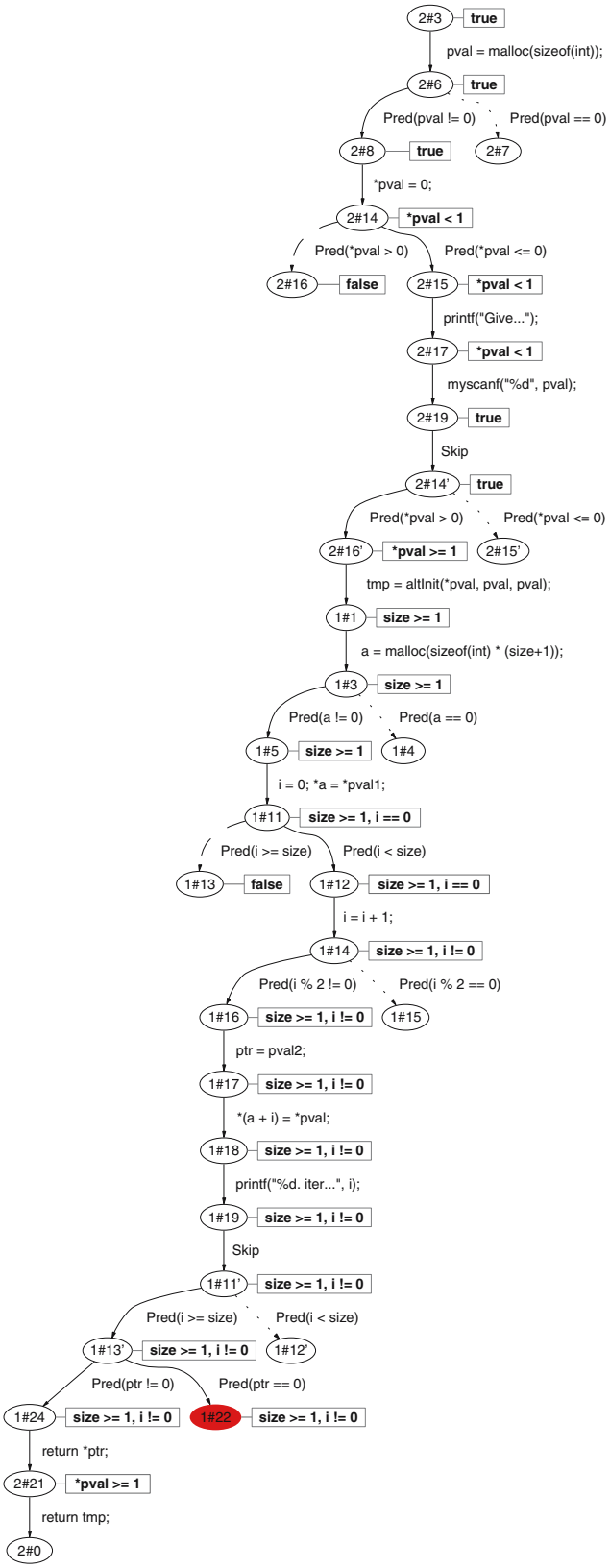

Fig. 7 ART when the second infeasible error path is found

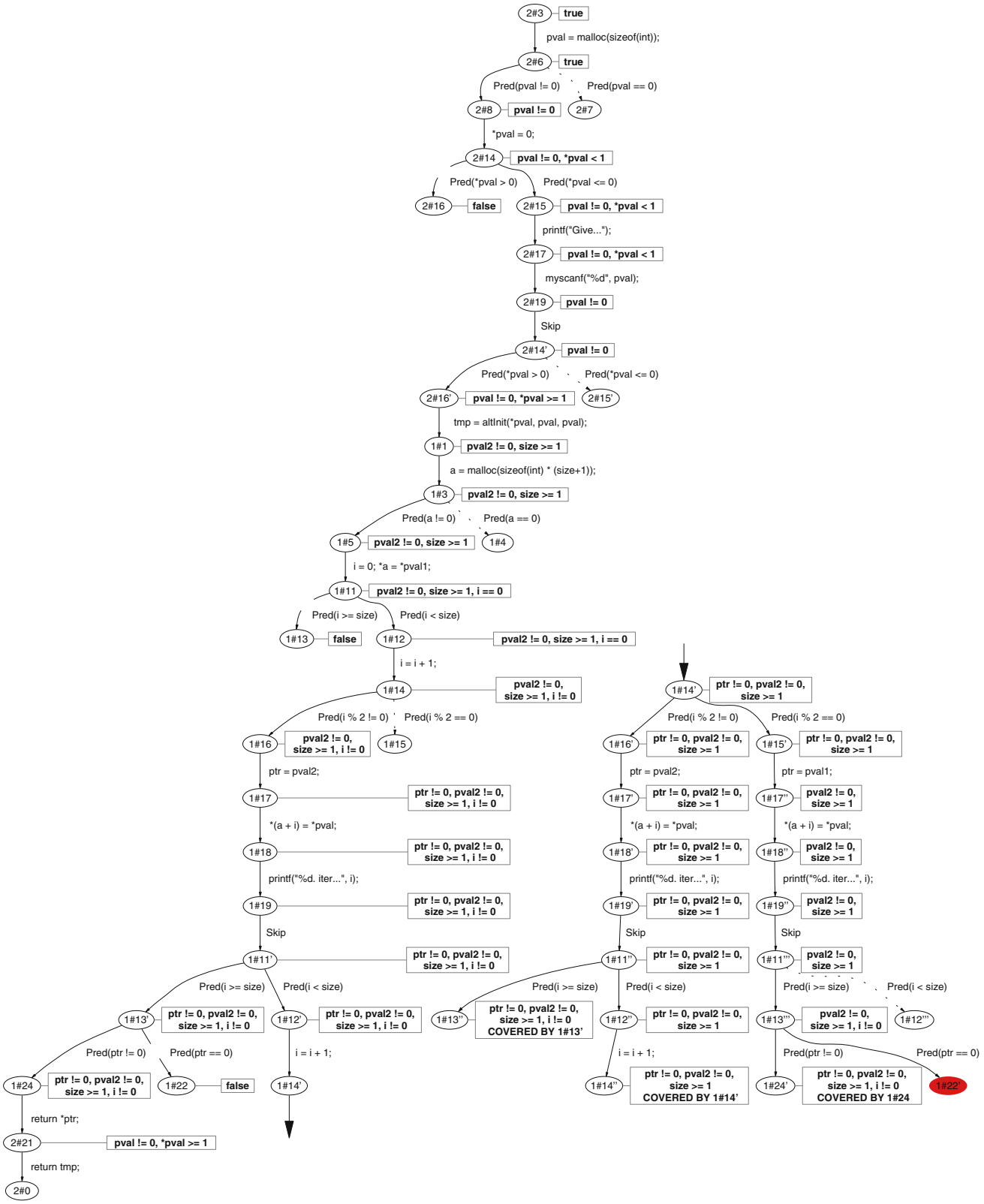

Fig. 8 ART when the third infeasible error path is found

The second error path is again found to be infeasible, and counterexample analysis discovers the new predicates `pval` = 0, `pval2` = 0, and `ptr` = 0. In the next iteration, BLAST finds a third infeasible error path, shown in Fig. 8, from which it discovers the predicate `pval1` = 0.

With these predicates, BLAST constructs the complete ART shown in Fig. 4. Since this tree is safe for the error location `1#22`, this proves that `ERR` can never be reached by executing the program. Note that some leaf nodes in the tree are covered: as no new states can be reached by exploring states from covered nodes, BLAST stops the ART construction at such nodes, and the whole process terminates.

Limitations of BLAST**.** There are some technical reasons why BLAST may give false alarms or unsound results. The decision procedures underlying BLAST, which are used for computing reachable regions and interpolants of path formulas, implement linear arithmetic and uninterpreted functions [32, 65,72]. Operations such as multiplication and bit-level manipulations are conservatively treated as uninterpreted functions. This may cause BLAST to return false alarms, because it may not be able to prove the infeasibility of an infeasible error path. More seriously, the decision procedures used by BLAST model integers as mathematical integers and do not account for overflows. Therefore, certain errors that depend on arithmetic overflow may not be caught. These limitations can be overcome by using decision procedures with a more accurate (bit-level) model of C semantics as a back-end [27].

BLAST assumes that the memory is laid out as a logical array, where the types of variables are preserved (i.e., we do not write data of one type to an address, and read it back as a different type). BLAST furthermore assumes that all pointer arithmetic is safe (i.e., within array bounds). These assumptions can either be ensured independently by tools such as CCURED, or added explicitly to the code as assertions, which BLAST can then try to prove or disprove (as explained in the case study of Sect. 3). BLAST uses a flow-insensitive may-alias analysis to deal with pointer updates. The may-analysis ignores pointer arithmetic and does not distinguish between different locations of an array. This causes conservative approximations in the analysis, because a write to one address may invalidate predicates about a different address if these addresses are not distinguished by the alias analysis. Again, the result may be false alarms.

BLAST treats library calls by assuming that the return values can be arbitrary. However, BLAST assumes that the library does not change any global heap data. If a more accurate model of a library function is required, the user must provide a stub implementation.

The predicates tracked by BLAST do not contain logical quantifiers. Thus, the language of invariants is weak, and BLAST is not able to reason precisely about programs with arrays or inductive data structures whose correctness involves quantified invariants. In such cases, BLAST may loop, discovering an ever increasing number of relevant predicates, without being able to discover the quantified invariant.

3 Checking memory safety

A program is *memory safe* if it only accesses memory addresses within the bounds of the objects it has allocated or to which it has been granted access. Memory safety is a fundamental correctness requirement for most applications. In the following, we consider one particular aspect of memory safety: *null-pointer dereferencing*. The value 'null' is used for a pointer in C programs to indicate that the pointer is not pointing to a valid memory object. Dereferencing a null pointer can cause an arbitrary value to be read, or the program to crash with a segmentation fault. Thus, the absence of null-pointer dereferences is a safety property.

3.1 Example

We focus on one particular pointer dereference in the program from Sect. 2: the dereference of the pointer `ptr` at the end of the function `altInit` (on line 19). We wish to prove that along all executions of the program, this pointer dereference is valid, that is, the value of `ptr` is not null. Notice that this property holds for our program: along every feasible path to line 19, the pointer `ptr` equals either `pval1` or `pval2`. Moreover, when `altInit` is called from `main`, the actual arguments passed to `pval1` and `pval2` are both `pval` (line 30). We have allocated space for `pval` in `main` (line 20), and we have already checked that the allocation succeeded (the test on line 21 and the code on lines 22–23 ensures that the program exits if `pval` is null).

We have instrumented the program to check for this property (line 18), by checking whether the pointer `ptr` is null immediately before the dereference. In the next subsection, we will describe how such instrumentations are inserted automatically by a memory-safety analysis. With the instrumentation, the label `ERR` on line 18 is reached if and only if the pointer `ptr` is null and about to be dereferenced at line 19. With this instrumentation we have reduced the memory-safety check to a reachability check, which can be solved by our model checker.

3.2 Program instrumentation for model checking

In principle, we can annotate every dereference operation in the program with a check that the dereferenced pointer is not null, and run BLAST on the annotated program to verify that no such check fails. However, this strategy does not scale well. First, many accesses can be proved memory safe using

an inexpensive type-based approach, and using an expensive analysis like BLAST is unnecessary. Second, each annotation should be checked independently, so that the abstractions required to prove each annotation do not interfere and result in a large state space while model checking. Therefore, we use CCURED [26,71], a type-based memory-safety analysis, to classify the pointers according to usage and annotate the program with run-time checks.

CCURED analyzes C programs with respect to a sound type system which ensures that well-typed programs are memory safe. When the type system cannot prove that a pointer variable is always used safely, CCURED inserts run-time checks in the program which monitor correct pointer usage at execution time. In particular, each dereference of a potentially unsafe (i.e., not proved safe by the type system) pointer is annotated with a check that the pointer is non-null. The run-time checks abort the program safely, instead of running into undefined configurations. However, each run-time check constitutes overhead at execution time, and CCURED implements many optimizations that remove redundant run-time checks based on simple data-flow analyses. Typically, the CCURED optimizations remove over 50% of the run-time checks inserted by the type system, and the optimized programs run within a factor of two of their original execution time. We wish to check how many of the remaining run-time checks can be removed by the more sophisticated analysis implemented in BLAST.

For each potentially unsafe pointer dereference `*p` in the program, CCURED introduces a call `__CHECK_NULL(p)` which checks that the pointer `p` is non-null. The function `__CHECK_NULL` terminates the program if its argument is null, and simply returns if the argument is non-null. Thus, if the actual argument `p` at a call site is non-null along all feasible paths, then this function call can be removed without affecting the behavior of the program. To check if a call to `__CHECK_NULL` can be removed from the program, BLAST does the following. First, it replaces the call to `__CHECK_NULL` with a call to `__BLAST__CHECK_NULL` with the same argument, where `__BLAST__CHECK_NULL` is the following function:

```
void __BLAST__CHECK_NULL(void *p) {
 if (!p) { __BLAST_ERROR: ; }
}
```

Second, BLAST checks if the location labeled with `__BLAST_ERROR` is reachable. Both steps are performed independently for each call to `__CHECK_NULL` in the program body. Each call of BLAST has three possible outcomes.

The first outcome is that BLAST reports that the label `__BLAST_ERROR` is not reachable. In this case, the function call can be removed, since the corresponding check will not fail at run time.

The second possible outcome is that BLAST produces an error path that represents a program execution in which `__BLAST__CHECK_NULL` is called with a null argument, which indicates a situation where the run-time check fails. In this case, the check must remain in the program to terminate the program safely should the check fail. This may also indicate a program error, in which case the feedback provided by BLAST (the error path) provides useful information for fixing the bug. We often encountered error paths of the form that the programmer forgot to check the return value of `malloc`: if the memory allocation fails, then the next dereference of the pointer is unsafe. BLAST assumes that `malloc` may return a null pointer and discovers the problem. However, not every error path found by BLAST necessarily indicates a program error, because BLAST makes several conservative assumptions; see the discussion of its limitations at the end of Sect. 2.

There is a third possible outcome, namely, that BLAST fails to declare whether the considered run-time check is superfluous or necessary, due to time or space limitations. In this case, we say that BLAST *fails*, and we will provide the failure rate for the experiments below. If BLAST fails on a run-time check, then the check must of course remain in the program. Notice that by changing each call to `__CHECK_NULL` separately, BLAST analyzes if a run-time check is necessary independently from all other checks. These analyses can be run in parallel and often lead to different program abstractions.

3.3 Experiments

We ran our method on several examples. The first seven programs are from the Olden v1.0 benchmark suite [16]. We included the programs for the Bitonic Sort algorithm (bisort), the Electromagnetic Problem in Three Dimensions (em3d), the Power Pricing problem (power), the Tree Add example (treeadd), the Traveling Salesman problem (tsp), the Perimeters algorithm (perimeter), and the Minimum Spanning Tree problem (mst). Finally, we processed the scheduler for Unix systems fcron, version 2.9.5, and the Lisp interpreter (li) from the Spec95 benchmark suite. We ran BLAST on each run-time check inserted by CCURED separately, and fixed a time-out of 200 s for each check; that is, a run of the model checker is stopped after 200 s with *failure*, and the studied run-time check is conservatively declared necessary.

Table 1 presents the results of our experiments. The first column lists the program name, the second and third columns give the number of lines of the original program ("LOC orig.") and of the instrumented program after preprocessing and CCURED instrumentation ("LOC cured"). The three columns of "Run-time checks" list the number of run-time checks inserted by the CCURED type system (column "inserted"), the number of remaining checks after the CCURED optimizer removes redundant checks (column "optim."), and finally the number of remaining checks after BLAST is used to remove run-time checks (column "BLAST").

Table 1 Experimental results for Sect. 3

Program	LOC		Run-time checks			Proved safe	Potential
	Orig.	Cured	Inserted	Optim.	BLAST	by BLAST	errors found
bisort	684	2510	51	21	6	15	6
em3d	561	2831	33	20	9	11	9
power	763	2891	149	24	24	0	24
power-fixed	763	2901	149	24	12	12	12
treeadd	370	2246	11	7	6	1	6
tsp	565	2560	93	59	44	15	4
perimeter	395	2292	49	18	8	10	5
mst	582	2932	54	34	19	15	18
fcron 2.9.5	11994	38080	877	455	222	233	74
li	6343	39289	1715	915	361	554	11

The column "Proved safe by BLAST" is the difference between the "optim." and "BLAST" columns: it shows the number of checks remaining after the CCURED optimizer which BLAST proves will never fail.

The remaining checks, which cannot be removed by BLAST, fall into two categories. First, the column "Potential errors found" lists the number of checks for which BLAST found an error path leading to a violation of the run-time check; those are potential bugs and the error paths give useful information to the programmer. For example, we took the program with the most potential errors found, namely power, and analyzed its error paths. In many of them, a call to `malloc` occurs without a check whether there is enough memory available. So we inserted after each call to `malloc` a null-pointer check to ensure that the program execution does not proceed in such a case. Analyzing the fixed program (with null-pointer checks inserted after each `malloc`), we can remove 12 more run-time checks. To give an example of the performance of BLAST, in the case of power-fixed, the cured program was checked in 15.6 s of processor time on a GNU/Linux machine with a 3.06 GHz Intel P4 Xeon processor and 4 GB memory.

Second, the difference between the columns "BLAST" and "Potential errors found" gives the number of run-time checks on which the model checker fails (times out) without an answer. The number of these failures is not shown explicitly in the table; it is zero for the first five programs. Since BLAST gives no information about these checks, they must remain in the program.

4 Test-case generation

Section 2 introduced the model-checking algorithm, which finds program paths that lead to particular program locations. We now show how we can generate *test vectors* from these paths, that is, sequences of input values that cause the desired paths to be executed. We start with an overview of the method using a small example.

4.1 Example

Consider the program of Fig. 9, which should compute the middle value of three integers (but has a bug). The program takes three inputs and invokes the function `middle` on them. A test vector for this program is a triple of input values, one for each of the variables `x, y, z`. Figure 10 shows the control-

```
#include <stdlib.h>
#include <stdio.h>

int readInt(void);

int middle(int x, int y, int z) {
L1:   int m = z;
L2:   if(y < z)
L3:     if(x < y)
L5:       m = y;
L6:     else if(x < z)
L9:       m = x;
      else
L10:    if(x > y)
L12:      m = y;
L13:   else if(x > z)
L15:      m = x;
L7:  return m;
}

int main() {
  int x, y, z;
  printf("Enter the 3 numbers: ");
  x = readInt();
  y = readInt();
  z = readInt();
  printf("Middle number: %d", middle(x,y,z));
}
```

Fig. 9 Program `middle`

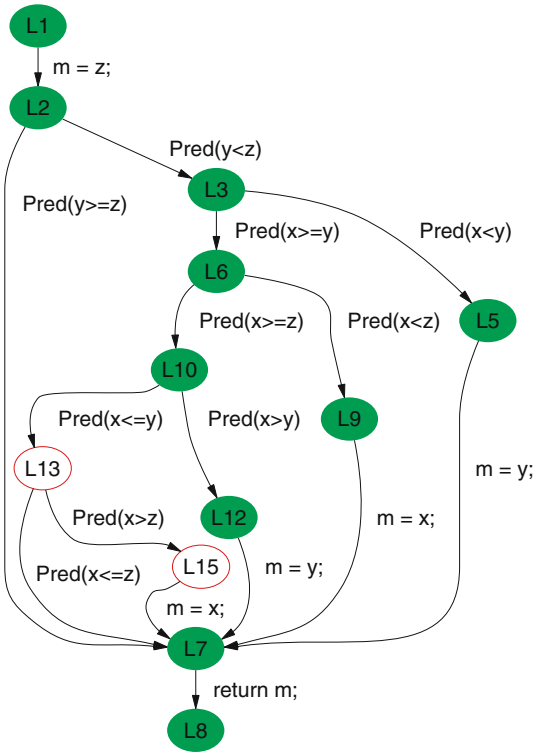

Fig. 10 CFA for program `middle`

flow automaton (CFA) for function `middle`. For brevity, we omit the CFA for function `main`. We first consider the problem of location coverage, i.e., we wish to find a set of test vectors such that for each location of the CFA, there is some test vector in the set that drives the program to that location.

Phase 1: Model checking. To find a test vector that takes the program, e.g., to location `L5`, we first invoke BLAST to check if `L5` is reachable. If `L5` is reachable, BLAST produces a feasible error path in the corresponding abstract reachability tree (ART). In our example, the path to `L5` is given by the following sequence of operations: `m=z;assume(y < z);` `assume(x < y)` where the first operation corresponds to the assignment upon entry, and the second and third (`assume`) operations correspond to the first two branch conditions being taken.

Phase 2: Tests from counterexamples. In the second step, we use the feasible error path from the model-checking phase to find a test vector, i.e., an initial assignment for `x,y,z` that takes the program to location `L5`. This is done as follows. First, we build the path formula (PF) for the error path, which in this case is $\texttt{m} = \texttt{z} \wedge \texttt{y} < \texttt{z} \wedge \texttt{x} < \texttt{y}$. Second, we find a satisfying assignment for the formula, e.g., "$\texttt{x} \mapsto 0, \texttt{y} \mapsto 1, \texttt{z} \mapsto 2, \texttt{m} \mapsto 2$", which after ignoring the value for `m`, gives a test vector that takes the program to `L5`. Notice that since the path to `L5` is feasible, the PF is guaranteed to be satisfiable.

x	y	z	Error Path
0	0	0	⟨L1,L2,L7,L8⟩
0	1	2	⟨L1,L2,L3,L5⟩
0	0	1	⟨L1,L2,L3,L6,L9⟩
1	0	1	⟨L1,L2,L3,L6,L10,L12⟩

Fig. 11 Generated test vectors for program `middle`

We repeat these two phases for each location, noting that one input takes us to several locations—those along the path—until we have a set of test vectors that covers all locations of the CFA. Along with each test vector, BLAST also produces a path in the corresponding ART that is exercised by the test. A set of test vectors for location coverage of `middle` is shown in Fig. 11. Each row in the table gives an input test vector—initial values for `x,y,z`—and the corresponding path as a sequence of locations. For example, the vector of test values for the target location `L12` is `(1,0,1)`, and BLAST reports the path ⟨`L1,L2,L3,L6,L10,L12`⟩, which is easy to understand with the help of the CFA in Fig. 10. The path is a prefix of a complete program execution for the corresponding test vector.

The alert reader will have noticed that the tests do not cover all locations; `L13` and `L15` remain uncovered, as denoted by the absence of shading for two locations in Fig. 10. It turns out that BLAST proves that these locations are not reachable—i.e., they are not visited for any initial values of `x,y,z`—and hence there exists dead code in `middle`. A close analysis of the source code reveals that a pair of braces is missing, and that the indentation is misleading for the code without braces: the `if` on `L6` matches the `else` after `L9`, which is meant for the `if` on `L2`.

4.2 The testing framework

Testing is usually carried out within a framework comprising (1) a suitable representation of the *program*, (2) a representation of *test vectors*, and a set of test vectors called a test suite, (3) an *adequacy* criterion that determines whether a test suite adequately tests the program, (4) a test generation procedure that generates an adequate *test suite*, and (5) a *test driver* that executes the program with a given test vector by automatically feeding input values from the vector.

Programs and tests. As before, we use CFAs as our representation of programs. We represent a test vector by a sequence of input data required for a single run of the program. This sequence contains the initial values for the formal parameters of the function to be tested, and the sequence of values supplied by the environment whenever the program asks for input. In other words, in addition to input values, the test vector also contains a sequence of return values for external function calls. For example, when testing device drivers, the test vector would contain a sequence of suitable return

values for all calls to kernel functions made by the driver, and a sequence of values for data read off the device.

Target predicate coverage. A test adequacy criterion is a set of *coverage goals* that determine when the program has been tested thoroughly. Ideally, one would like the test suite to exercise all program paths ("path coverage"), and thus expose most errors that the program might have. As such test suites would be infinitely large for most programs, weaker notions of test adequacy, for example, node and edge coverage, are used to approximate when a program has been tested sufficiently [67,82]. A test suite is adequate w.r.t. an adequacy criterion if it contains enough tests to satisfy each coverage goal in the criterion.

We use the following notion of *target predicate coverage*: given a C program in the form of a set of CFAs, and a target predicate p, we say a test vector *covers* a location q of some CFA w.r.t. p if the execution resulting from the test vector takes the program into a state where it is in location q and the variables satisfy the predicate p. We deem a test suite adequate w.r.t. the target predicate coverage criterion with predicate p if all p-reachable CFA locations are covered w.r.t. p by some test vector in the test suite. A location q is p-reachable if some program execution reaches a state where the location is q and the variables satisfy p.

As a special case, if the target predicate p is *true*, the test-generation algorithm outputs test vectors for all *reachable* CFA locations. Furthermore, BLAST reports all CFA locations that are (provably) unreachable by any execution as dead locations (they correspond to dead code). If we run BLAST on a program with both predicates p and $\neg p$, then for all CFA locations q that can be reached with p either true or false, we obtain *two* test vectors—one that causes the program to reach q with p evaluating to *true*, and another one that causes the program to reach q with p evaluating to *false*.

The notion of target predicate coverage corresponds to *location coverage* ("node coverage") if $p = true$. For *edge coverage*, for an edge e that represents a branch condition p_e, we can find a test that takes the program to the source location of e with the state satisfying the predicate p_e, thus causing the edge e to be traversed in the subsequent execution step. We can similarly adapt our technique to generate tests for other testing criteria [53,82]; we omit the details.

Test flow. The overall testing framework as implemented in BLAST is shown in Fig. 12. The *test-suite generator* takes as input a program and a target predicate p, and produces a test suite that is adequate w.r.t. p. The *test-driver generator* takes as input a program, and produces a test driver. During a testing run, the test driver reads the test suite, and executes the program being tested once on each test vector, using the values from the vector as input. It can be run on each individual test vector separately, and the user can study the resulting dynamic behavior, for example by using a debugger.

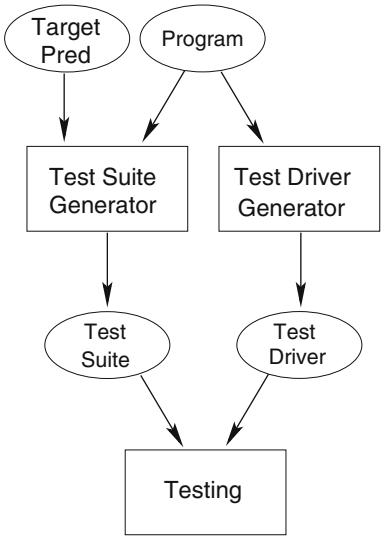

Fig. 12 Test flow

Example of target predicate coverage. The use of target predicate coverage is illustrated on the simple program in Fig. 13, which manipulates Unix security privileges using `setuid` system calls. Unix processes can execute in several privilege levels; higher privilege levels may be required to access restricted system resources. Privilege levels are based on process user id's. Each process has a real user id, and an effective user id. The system call `seteuid` is used to set the effective id, and hence the privilege level of a process. The user id `0` (or root) allows a process full privileges to access all system resources. We assume for our program that the real user id of the process is not zero, i.e., the real user does not have root privileges. This specification is a simplification of the actual behavior of `setuid` system calls in Unix [19], but is sufficient for exposition.

The `main` routine first saves the real user id and the effective user id in the variables `saved_uid` and `saved_euid`, respectively, and then sets the effective user id of the program to the real user id. This last operation is performed by the function call `seteuid`. The function `get_root_privileges` changes the effective user id to the id of the root process (id `0`), and returns `0` on success. If the effective user id has been set to root, then the program does some work (in the function `work_and_drop_privileges`) and sets the effective user id back to `saved_uid` (the real user id of the process) at the end (`L9`). To track the state changes induced by the `setuid` system calls, we instrument the code for the relevant system calls as follows. The user id is explicitly kept in a new integer variable `uid`; the function `getuid` is instrumented to return a nonzero value (modeling the fact that the real user id of the process is not zero); and the function `geteuid` is instrumented to nondeterministically return either a zero or a nonzero value. Finally, we change the system call `seteuid(x)` so that the variable

```
int saved_uid, saved_euid;

int get_root_privileges () {
L1: if (saved_euid!=0) {
L2:   return -1;
    }
L3: seteuid(saved_euid);
L4: return 0;
}

work_and_drop_priv() {
L5: FILE *fp = fopen(FILENAME,"w");
L6: if (!fp) {
L7:   return;
    }
L8:   // work
L9: seteuid(saved_uid);
}

int main(int argc, char *argv[]) {
L10:saved_uid = getuid();
L11:saved_euid = geteuid();
L12:seteuid(saved_uid);
L13:  // work under normal mode
L14:if (get_root_privileges ()==0){
L15:  work_and_drop_priv();
    }
L16:execv(argv[1], argv + 1);
}
```

Fig. 13 Program `setuid`

`uid` is updated with the argument `x` passed to `seteuid` as a parameter. The instrumented versions are omitted for brevity.

Secure programming practice requires that certain system calls that run untrusted programs should not be made with root privileges [18], because the privileged process has full permission to the system. For example, calls to `exec` and `system` must never be made with root privileges. Therefore it is useful to check which parts of the code may run with root privileges.

This check is performed by generating a test suite with the target predicate `uid` $= 0$. Perhaps surprisingly, the test suite contains a test vector where the call to `execv` can be executed while this predicate holds. Such a test vector can now be used to explore the erroneous behavior of the program. Closer inspection identifies a bug in the function `work_and_drop_privileges`: if the call to `fopen` fails, the function returns without dropping root privileges.

In the following subsections we describe how to generate an adequate test suite, and how to generate a test driver that can execute the program on the test vectors in the suite.

4.3 Test-suite generation

Recall that the model-checking algorithm described in Sect. 2 takes as input a set of CFAs and a configuration (q, p) of target location and target predicate. Provided it terminates, the algorithm returns either with outcome O_1, a complete ART T that is safe w.r.t. (q, p), or with outcome O_2, a path from the root node to a node $\mathtt{n} : (q, \cdot, \varphi)$ such that $\varphi \wedge p$ is satisfiable. Given a program and a target predicate p, the test-suite generation now proceeds as follows.

Step 1. The locations of the CFAs are numbered in depth-first order, and put into a *worklist* in decreasing order of the numbering (i.e., the location numbered last in DFS order is first on the worklist). We create an initial ART that consists of a single node $\mathtt{n} : (q_0, s_0, \mathit{true})$, where q_0 is the initial location of the entry function and s_0 is the empty stack. The initial test suite is the empty set.

Step 2. If the worklist is empty, then we return the current test suite; otherwise let q be the first CFA location in the worklist. We invoke the model checker with the current ART and the configuration (q, p).

Step 3. If the model checker returns with outcome O_1, then we conclude that for all locations q' such that the new ART is safe w.r.t. (q', p), no test vector exists, and so we delete all such locations from the worklist. Otherwise, if the model checker returns with outcome O_2, then we have found a p-reachable location q. We use the path to q to compute a test vector that covers the location q w.r.t. p using a procedure described below. We add this vector to the test suite, and remove q from the worklist. In either case, we proceed with **step 2**.

It can be shown that upon termination, the above procedure returns a test suite that is adequate w.r.t. p according to our criterion of target predicate coverage.

We incorporate several optimizations to the above loop. First, the model checking of a location is not started from scratch in **step 2**. Instead, the (incomplete) ART from the previous model-checking phase is reused to start the next run of the model-checking algorithm. The current ART is searched and updated to find p-reachable occurrences of location q. Second, when a test vector is found, we can additionally check (by symbolically executing the program on the vector) which other locations it covers, and we remove those locations from the worklist. Third, the model-checking algorithm uses heuristics to choose the next node to unfold in the current ART. The nodes that need to be unfolded are partitioned into those whose location has been covered by a vector in the current test suite, and those whose location is still uncovered. The model checker unfolds nodes of uncovered locations first, and it unfolds nodes of covered locations only if there remain no nodes of uncovered locations. A node whose location has been covered by a previous test may still need to be unfolded, because a path to an (as yet)

uncovered location may go through it. Forth, the user has the option to give a time-out for the model checking. Thus in **step 3**, if instead of O_1 or O_2, the model checker times out, then we give up on the location q, by deleting it from the worklist and going back to **step 2**. We have found these optimizations to be essential for the algorithm to work on large programs.

Example. Consider the example `middle` from Sect. 4.1. The test-generation algorithm first prioritizes the locations according to depth-first search order, which is ⟨L1, L2, L7, L8, L3, L6, L10, L13, L15, L12, L9, L5⟩. The test-generation algorithm runs the model checker starting with L5 as target location and going downward in this order. For the location L5, the model checker builds the ART in breadth-first order (since no location is covered so far, all locations have equal priority). At this point, the model checker finds a feasible path in the ART to location L5 (namely, the path ⟨L1, L2, L3, L5⟩). Since the predicate p is *true*, this does not require extra work in this case. However, for a general predicate p, a location q on the path is declared covered only if the model checker finds a path to q such that p holds at location q. The first test vector is generated for this path as explained in the next section. The model checker marks the locations on the path as covered by some test, and moves on to the next uncovered location, L9.

When running the model checking procedure for location L9, BLAST uses the ART that was already built in the previous stage, and continues to explore reachable locations. Notice that this time, the locations L1, L2, L3, and L5 are treated as locations of lower priority. Hence, the algorithm decides to expand L6 before L5. It then finds the path ⟨L1, L2, L3, L6, L9⟩, which is feasible. The current ART is now used to search for the next uncovered location in the list, L12. In this way, all remaining locations will be covered.

Generating tests from paths. When model checking in **step 2** ends with outcome O_2, the resulting ART contains a path to a node $\mathtt{n} : (q, \cdot, \varphi)$ such that the path ends at q and $\varphi \wedge p$ is satisfiable. We now describe how to extract from this path a test vector that, when fed to the program, takes it to location q satisfying the target predicate p.

First, we construct the PF for the path t in the ART leading to q, and additionally conjoin the predicate p renamed with the current names of variables at the end of the path. Next, we use a decision procedure to produce a satisfying assignment for the variables of the new formula. From the satisfying assignment we build a test vector that drives the program to the target location and target predicate. For a satisfiable formula φ, let $S(\varphi)$ be a satisfying interpretation of all special variables that occur in the formula. A test vector that exercises the path t is obtained by setting every input variable x of the program to the initial value $S(\varphi)(\langle \mathtt{x}, 1 \rangle)$, and then providing, through the test driver, the return value of any call to an external function from the corresponding assignment to a variable in S.

Figure 14a shows a small program, and Fig. 14b and 14c show, respectively, a path to the program location LOC, and the PF for that path. The constraint for each atomic operation of the path is shown to the right of the operation; the PF is the conjunction of all constraints. Figure 14d shows a satisfying interpretation for the special variables of the PF of Fig. 14c. It is easy to check that if we set the inputs initially to "$\mathtt{x} = 0, \mathtt{y} = 0, \mathtt{z} = 2$," then the program follows the path of Fig. 14b. The generated test vector is shown in Fig. 14e.

If the constraints in the PF only contain linear arithmetic, then a satisfying assignment can be obtained by using a solver for integer linear programming. The method can be extended in the presence of disjunctions and uninterpreted functions that model memory aliasing relationships. Of course, there are programs for which our constraint-based test-generation strategy fails, because the constraint language understood by the constraint solver that is used to generate tests is not expressive enough.

Library calls. If a program path contains library calls whose source code is not available for analysis, or asks for user input, the constraint generation assumes that the library call or the user can return any value. Thus, some of our tests may not be executable if the library calls always return values from some subset of possible values. In this case, the user can model postconditions on library calls by writing stub functions that restrict the possible return values.

4.4 Test-driver generation

Recall that a test vector generated by BLAST is a sequence of integer values (our test-vector generation is currently restricted to integer inputs): these are the values that are fed to the program by the test driver during the actual test; they include the initial values for all formal parameters and the return values for all external function calls.

The test-driver generator takes as input the original program and instruments it at the source-code level to construct a test driver, which consists of the following components: (1) a wrapping function, (2) a test-feeding function, and (3) a modified version of the original program. The test-driver generator modifies the code of the original program by replacing every call to an external function with a call to the test-feeding function. The test driver can then be compiled and run to examine the behavior of the original program on the test suite.

The wrapper is the main procedure of the test driver: it reads a test vector and then calls the entry function of the modified program, passing it initial values for the parameters from the test vector. The test-feeding function reads the

Fig. 14 Generating a test vector. **a** Program, **b** Path, **c** Path formula, **d** Assignment, **e** Test vector

(a)

```
Example() {
  if (y == x)
    y ++ ;
  if (z <= x)
    y ++ ;
  a = y - z;
  if (a < x)
    LOC:
}
```

(b)

```
assume(y=x)
y = y+1
assume(¬ z≤x)

a = y−z
assume(a<x)
```

(c)

$$\langle y^{\lhd}1\rangle = \langle x^{\lhd}1\rangle \ \wedge$$
$$\langle y^{\lhd}2\rangle = \langle y^{\lhd}1\rangle + 1 \ \wedge$$
$$\neg\langle z^{\lhd}1\rangle \leq \langle x^{\lhd}1\rangle \ \wedge$$
$$\langle a^{\lhd}1\rangle = \langle y^{\lhd}2\rangle - \langle z^{\lhd}1\rangle \ \wedge$$
$$\langle a^{\lhd}1\rangle < \langle x^{\lhd}1\rangle$$

(d)

$$\langle x^{\lhd}1\rangle \mapsto 0$$
$$\langle y^{\lhd}1\rangle \mapsto 0$$
$$\langle y^{\lhd}2\rangle \mapsto 1$$
$$\langle z^{\lhd}1\rangle \mapsto 2$$
$$\langle a^{\lhd}1\rangle \mapsto -1$$

(e)

$$x \mapsto 0$$
$$y \mapsto 0$$
$$z \mapsto 2$$

Table 2 Experimental results for Sect. 4

Program	LOC	CFA locations	Locations			Tests	Predicates		Time
			Live	Dead	Fail		Total	Average	
kbfiltr	5,933	381	298	83	0	39	112	10	5 min
floppy	8,570	1,039	780	259	0	111	239	10	25 min
cdaudio	8,921	968	600	368	0	85	246	10	25 min
parport	12,288	2,518	1,895	442	181	213	509	8	91 min
parclass	30,380	1,663	1,326	337	0	219	343	8	42 min
ping	1,487	814	754	60	0	134	41	3	7 min
ftpd	8,506	6,229	4,998	566	665	231	380	5	1 day

next value from the test vector and returns it. We are guaranteed that the vector will have taken the program to the target when the test-feeding function has consumed the entire vector. Hence, once the test vector is consumed, the feeder returns arbitrary values.

4.5 Experiments

We ran BLAST to generate test suites for several benchmark programs. The first five programs are Microsoft Windows device drivers: kbfiltr, floppy, cdaudio, parport, and parclass. The program ping is an implementation of the ping utility, and ftpd is a GNU/Linux port of the BSD implementation of the ftp daemon. The experiments were run on a GNU/Linux machine with a 3.06 GHz Intel P4 Xeon processor and 4 GB memory.

Table 2 presents the results of our experiments for checking the reachability of code. The target predicate was always *true*: we checked which program locations are *live* (reachable on some feasible path) and *dead* (not reachable on any feasible path), and we generated test vectors that cover all live locations. Syntactically plausible paths (for example, control-flow paths, or data flows) may not be semantically possible, for example, due to correlated branching [14]. This is called the *infeasibility problem* in testing [56,82]. The usual approach to deal with infeasible error paths is to argue manually on a case-by-case basis, or to resort to adequacy scores (the percentage of all static paths covered by tests). By using BLAST we can automatically detect dead code, and generate tests for live code.

In the table, the first column lists the program name, the second column ("LOC") lists the number of lines in the program. CFAs represent programs compactly; each basic block is a single edge. The third column ("CFA locations") shows the number of locations of the CFAs which are syntactically reachable by exploring the corresponding call graph of the program. The column "live" shows the number of reachable locations, "dead" the number of unreachable locations, and "fail" the number of locations on which our tool failed. Ideally, the total number of CFA locations is equal to the sum of the live and dead locations. However, we set a time-out of 10 min per location in our experiments. So in practice, the tool fails on a small percentage of locations. The failure is due both to time-outs, and to not finding suitable predicates for abstraction.

The column "Tests" gives the number of generated test vectors. The implementation does not run the model checker for a location that is already covered by a previous test vector. Thus, the number of test vectors is usually much smaller than the number of reachable locations. This is especially apparent for the larger programs. The column "total" is the total number of predicates, over all locations, generated by the model-checking process. The column "average" is the average number of predicates active at any one program location. The average number of predicates at any location is much smaller than the total number of predicates, thus confirming our belief that local and precise abstractions can scale to large programs. Finally, the column "Time" is the running time rounded to minutes (except for `ftpd`, where the tool ran for two overnight runs).

We found many locations that were not reachable because of correlated branches. For example, in `floppy`, we found the following code:

```
driveLetterName.Length = 0;
// cut 15 lines
...
if (driveLetterName.Length != 4 ||
 driveLetterName.Buffer[0] < 'A' ||
 driveLetterName.Buffer[0] > 'Z') { ...
}
```

Here, the predicate `driveLetterName.Length != 4` is true; so the other tests are never executed. Another reason we get dead code is that certain library functions (like `memset`) make many comparisons of the size of a structure with different built-in constants. If called from a proper client, most of these comparisons fail, giving rise to many dead locations.

Further, the Windows device drivers were already annotated with a safety specification that deals with I/O request packet (IRP) completion. Hence, the percentage of unreachable locations in these drivers is high, because statements for error handling are not reached. For example, the driver checks in certain situations if the observing state machine was in the proper state, and called an error function otherwise. On all feasible paths, the state machine was in a proper state, and therefore the error functions are unreachable.

5 State of the BLAST project

The challenge. Regardless of their competence and experience, programmers make mistakes while programming, and they spend much of their time on hunting and fixing bugs in their programs. Automatic proof- and bug-finding tools can aid programmers in the search for bugs; they hold the potential both for reducing the time and cost of the development cycle, and for producing more robust and reliable code. The development of such *semantic* code-checking tools, which extend simple syntax, type, and style checkers available today, is an important research goal for software engineering (cf. [51] for an overview).

The convergence of technologies. Over the past decades, several research communities worked on improvements of techniques that were fundamentally different in how they approach the problem of program verification. Recently, however, we have seen a convergence of the different verification approaches. The most promising current solutions to program verification consist of ingredients from *model checking*—for effective techniques to exhaustively search state spaces and to represent boolean data—from *abstract interpretation*—for techniques to formalize and symbolically execute abstract representations of programs—and from *theorem proving*—for techniques to solve constraints and decide logical theories. BLAST is a witness to this convergence: it uses ideas that originated in the model-checking community for representing and searching state spaces, and for counterexample-guided abstraction refinement; it is most naturally formalized as a path-sensitive static analysis over an adjustable abstract domain; and it calls on automatic theorem provers for constructing abstract state transitions, for checking the feasibility of abstract error paths, and for computing interpolants. It is thus mostly for historical reasons that we call BLAST a "model checker [11]."

The results. Our experiments show that CEGAR-based software model checkers like BLAST are able to analyze programs of medium size (thousands of lines of code), and that they can be successfully applied to software engineering practices such as checking temporal safety properties and generating test cases. BLAST is released as open source under the BSD license and available online.[1] The software includes an ECLIPSE plug-in to provide model-checking techniques within an integrated development environment [8]. The BLAST web site provides also a supplementary web page for this paper, including the example C programs, and the version of BLAST that we have used for the experiments in this paper.

Current and future directions. Our recent research in the BLAST project has focused on three topics: first, to make BLAST more usable in the software engineering process; second, to extend the algorithmic capabilities of BLAST to handle larger programs and deeper properties; and third, to handle multi-threaded software. To improve the usability of BLAST, we provide a high-level language to specify queries about a code base as well as proof decomposition strategies [7]. We also provide *incremental* verification capabilities so that BLAST can be incorporated within a regression testing/verification framework, without restarting the verification process from scratch after every program modification [48]. While BLAST has been used successfully for checking *control*-dominated programs, such as device drivers and protocols, it is still inadequate for reasoning about large programs that use heap data structures. We are currently augmenting BLAST with more expressive abstract domains to support a more precise heap analysis within the CEGAR framework [10,11]. We are also exploring the automatic synthesis of temporal interfaces for software components, as well as the verification of code against such interfaces [45]. Such techniques for summarizing the correct usage and effects of procedures and libraries are crucial for obtaining a modular, scalable approach to verification, and for the analysis of partial programs. Finally, a main strength of model checking is its ability to explore nondeterministic program behavior resulting from the interleaving of multiple threads.

[1] http://mtc.epfl.ch/blast.

To control verification complexity, assumptions about the interference between threads, like component interfaces, need to be synthesized and discharged. An extension of BLAST in this direction has been used to check concurrent programs for the presence of data races [44].

References

1. Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing verification by successive approximation. Inf. Comput. **118**(1), 142–157 (1995)
2. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: ZING: A model checker for concurrent software. In: Proc. CAV, LNCS, vol. 3114, pp. 484–487. Springer, Berlin (2004)
3. Balarin, F., Sangiovanni-Vincentelli, A.L.: An iterative approach to language containment. In: Proc. CAV, LNCS, vol. 697, pp. 29–40. Springer, Berlin (1993)
4. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM, New York (2002)
5. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research (2002)
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE, pp. 326–335. IEEE, Los Alamitos (2004)
7. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The BLAST query language for software verification. In: Proc. SAS, LNCS, vol. 3148, pp. 2–18. Springer, Berlin (2004)
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: An ECLIPSE plug-in for model checking. In: Proc. IWPC, pp. 251–255. IEEE, Los Alamitos (2004)
9. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with BLAST. In: Proc. FASE, LNCS, vol. 3442, pp. 2–18. Springer, Berlin (2005)
10. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Proc. CAV, LNCS, vol. 4144, pp. 532–546, Springer, Berlin (2006)
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS, vol. 4590, pp. 504–518. Springer, Berlin (2007)
12. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones, LNCS, vol. 2566, pp. 85–108. Springer, Berlin (2002)
13. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: Proc. PLDI, pp. 321–333. ACM, New York (2000)
14. Bodik, R., Gupta, R., Soffa, M.L.: Interprocedural conditional branch elimination. In: Proc. PLDI, pp. 146–158. ACM, New York (1997)
15. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: Proc. ISSTA, pp. 123–133. ACM, New York (2002)
16. Carlisle, M.C.: Olden: parallelizing programs with dynamic data structures on distributed memory machines. Ph.D. thesis, Princeton University (1996)
17. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Trans. Softw. Eng. **30**(6), 388–402. Wiley, New York (2004)
18. Chen, H., Wagner, D.: MOPS: An infrastructure for examining security properties of software. In: Proc. CCS, pp. 235–244. ACM, New York (2002)
19. Chen, H., Wagner, D., Dean, D.: Setuid demystified. In: Proc. USENIX Security Symp., pp. 171–190 (2002)
20. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Proc. Logic of Programs 1981, LNCS, vol. 131, pp. 52–71. Springer, Berlin (1982)
21. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV, LNCS, vol. 1855, pp. 154–169. Springer, Berlin (2000)
22. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT, Cambridge (1999)
23. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS, LNCS, vol. 3440, pp. 570–574. Springer, Berlin (2005)
24. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2**(3), 215–222 (1976)
25. Clarke, L.A., Richardson, D.J.: Symbolic evaluation methods for program analysis. In: Program Flow Analysis: Theory and Applications, pp. 264–300. Prentice-Hall, Boston (1981)
26. Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W.: CCURED in the real world. In: Proc. PLDI, pp. 232–244. ACM, New York (2003)
27. Cook, B., Kroening, D., Sharygina, N.: COGENT: accurate theorem proving for program verification. In: Proc. CAV, LNCS, vol. 3576, pp. 296–300. Springer, Berlin (2005)
28. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., Zheng, H.: BANDERA: Extracting finite-state models from Java source code. In: Proc. ICSE, pp. 439–448. ACM, New York (2000)
29. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. Symbolic Logic **22**(3), 250–268 (1957)
30. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadek, F.K.: Efficiently computing static single-assignment form and the program dependence graph. ACM Trans. Program. Languages Systems **13**(4), 451–490 (1991)
31. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proc. PLDI, pp. 57–68. ACM, New York (2002)
32. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
33. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Proc. TACAS, LNCS, vol. 3920, pp. 489–503. Springer, Berlin (2006)
34. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, pp. 19–32. AMS (1967)
35. Godefroid, P.: Model checking for programming languages using VERISOFT. In: Proc. POPL, pp. 174–186. ACM, New York (1997)
36. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI, pp. 213–223. ACM, New York (2005)
37. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: Proc. ISSTA, pp. 53–62. ACM, New York (1998)
38. Gulavani, B., Henzinger, T.A., Kannan, Y., Nori, A., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Proc. FSE. ACM, New York (2006)
39. Gunter, E., Peled, D.: Temporal debugging for concurrent systems. In: Proc. TACAS, LNCS, vol. 2280, pp. 431–444. Springer, Berlin (2002)
40. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Proc. ASE, pp. 219–228. IEEE, Los Alamitos (2000)

41. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific static analyses. In: Proc. PLDI, pp. 69–82. ACM, New York (2002)
42. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. STTT **2**(4), 366–381 (2000)
43. Henglein, F.: Global tagging optimization by type inference. In: Proc. LISP and Functional Programming, pp. 205–215. ACM, New York (1992)
44. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Proc. PLDI, pp. 1–13. ACM, New York (2004)
45. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proc. ESEC/FSE, pp. 31–40. ACM, New York (2005)
46. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM, New York (2004)
47. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Proc. CAV, LNCS, vol. 2404, pp. 526–538. Springer, Berlin (2002)
48. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: International Symposium on Verification: Theory and Practice, LNCS, vol. 2772, pp. 332–358. Springer, Berlin (2003)
49. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM, New York (2002)
50. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
51. Hoare, C.A.R.: The verifying compiler: a grand challenge for computing research. J. ACM **50**(1), 63–69 (2003)
52. Holzmann, G.J.: The Spin model checker. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
53. Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: Proc. ICSE, pp. 232–243. IEEE, Los Alamitos (2003)
54. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P.: F- Soft: Software verification platform. In: Proc. CAV, LNCS, vol. 3576, pp. 301–306. Springer, Berlin (2005)
55. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Proc. ISSTA, pp. 14–25. ACM, New York (2000)
56. Jasper, R., Brennan, M., Williamson, K., Currier, B., Zimmerman, D.: Test data generation and infeasible path analysis. In: Proc. ISSTA, pp. 95–107. ACM, New York (1994)
57. Jhala, R., Majumdar, R.: Path slicing. In: Proc. PLDI, pp. 38–47. ACM, New York (2005)
58. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proc. TACAS, LNCS, vol. 2619, pp. 553–568. Springer, Berlin (2003)
59. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
60. Kroening, D., Groce, A., Clarke, E.M.: Counterexample guided abstraction refinement via program execution. In: Proc. ICFEM, LNCS, vol. 3308, pp. 224–238. Springer, Berlin (2004)
61. Kurshan, R.P.: Computer Aided Verification of Coordinating Processes. Princeton University Press (1994)
62. Lev-Ami, T., Sagiv, M.: Tvla: A system for implementing static analyses. In: Proc. SAS, LNCS, vol. 2280, pp. 280–301. Springer, Berlin (2000)
63. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill, New York (1974)
64. McCarthy, J.: Towards a mathematical science of computation. In: Proc. IFIP Congress, pp. 21–28. North-Holland (1962)
65. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. **345**(1), 101–121 (2005)
66. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: Proc. OSDI. USENIX (2002)
67. Myers, G.J.: The Art of Software Testing. Wiley (1979)
68. Necula, G.C.: Proof carrying code. In: POPL 97: Principles of Programming Languages, pp. 106–119. ACM, New York (1997)
69. Necula, G.C., Lee, P.: Efficient representation and validation of proofs. In: Proc. LICS, pp. 93–104. IEEE, Los Alamitos (1998)
70. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proc. CC, LNCS, vol. 2304, pp. 213–228. Springer, Berlin (2002)
71. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe retrofitting of legacy code. In: Proc. POPL, pp. 128–139. ACM, New York (2002)
72. Nelson, G.: Techniques for program verification. Tech. Rep. CSL81-10, Xerox Palo Alto Research Center (1981)
73. Peled, D.: Software Reliability Methods. Springer, Berlin (2001)
74. Peled, D.: Model checking and testing combined. In: Proc. ICALP, LNCS, vol. 2719, pp. 47–63. Springer, Berlin (2003)
75. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. Symposium on Programming, LNCS, vol. 137, pp. 337–351. Springer, Berlin (1982)
76. Ramamoorthy, C.V., Ho, S.B.F., Chen, W.T.: On the automated generation of program test data. IEEE Trans. Softw. Eng. **2**(4), 293–300 (1976)
77. Reps, T.W., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. POPL, pp. 49–61. ACM, New York (1995)
78. Saidi, H.: Model-checking-guided abstraction and analysis. In: Proc. SAS, LNCS, vol. 1824, pp. 377–396. Springer, Berlin (2000)
79. Schneider, F.B.: Enforceable security policies. Tech. Rep. TR98-1664, Cornell (1999)
80. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for C. In: Proc. ESEC/FSE, pp. 263–272. ACM, New York (2005)
81. Suzuki, N., Ishihata, K.: Implementation of an array bound checker. In: Proc. POPL, pp. 132–143. ACM, New York (1977)
82. Young, M., Pezze, M.: Software Testing and Analysis: Process, Principles and Techniques. Wiley, New York (2005)