BMuzz: Combining Bounded Model Checking and Fuzzing to Enhance Code Coverage

Markus Krahl
University of Applied Sciences Munich
Tasking Germany GmbH
Munich, Germany
markus.krahl@hm.edu
markus.krahl@tasking.com

Matthias Güdemann
University of Applied Sciences Munich
Munich, Germany
matthias.guedemann@hm.edu

Stefan Wallentowitz
University of Applied Sciences Munich
Munich, Germany
stefan.wallentowitz@hm.edu

***Abstract*—In safety-critical domains, extensive software testing is required to validate functional properties and meet standards such as ISO-26262 and DO-178C, which mandate strict code coverage levels. Uncovered code sections may indicate insufficient testing or unreachable code, leaving latent defects undetected.**

Traditional coverage tools reveal such gaps but cannot determine their cause. Fuzzing effectively discovers bugs but cannot prove unreachability, while bounded model checking (BMC) can formally prove unreachability and generate test cases but inherently underapproximates program behavior due to bounded exploration.

We present a hybrid testing framework, *BMuzz*, that combines fuzzing and BMC in a concurrent, automated workflow. It measures baseline coverage, instruments uncovered regions, and applies both techniques to either generate additional inputs or prove unreachability. The resulting tests and proofs help identify untested requirements, requirement violations, and dead code. Applied to a subset of an industrial-grade C standard library for embedded automotive systems, our approach achieves more efficient coverage than standalone fuzzing or BMC, while also identifying unreachable code and specific decision constellations, demonstrating its potential for broader adoption in safety-critical domains.

I. Introduction

Software testing is a fundamental stage of the development lifecycle, where coverage metrics serve as key indicators of test suite adequacy. In safety-critical domains, standards such as ISO-26262 [1] for automotive software and DO-178C [2] for aeronautic software mandate specific coverage levels and prescribe strategies for test case derivation. For instance, ISO-26262 *strongly recommends*, whereas DO-178C *requires*, deriving tests from software requirements. Yet, executing all available tests may still not achieve the desired coverage metrics. Portions of the code may remain unexecuted, or the mandated Modified Condition/Decision Coverage (MC/DC) [3] level may not be fully met. These gaps create ambiguity over whether the uncovered code reflects insufficient testing or genuinely unreachable logic, leading either to redundant testing efforts or to latent defects that remain undetected.

Model checking is a fully automated formal verification technique that systematically explores program states to verify properties such as safety, liveness, and reachability. A prominent variant, bounded model checking (BMC), restricts analysis to a finite number of execution steps and employs Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solvers. Advances in SMT solving have greatly improved efficiency, enabling practical software verification tools such as the C Bounded Model Checker (CBMC) [4]. In contrast, fuzzing relies on repeatedly supplying randomized inputs to a program until erroneous states are triggered, and has uncovered numerous bugs in widely used applications [5]. While BMC offers exhaustive and sound reasoning within bounded domains, fuzzing excels in scalability and practicality. Their complementary strengths suggest significant potential for hybrid approaches, motivating further exploration in software testing and verification.

In this paper, we propose a portfolio-based testing framework, *BMuzz*, that integrates fuzzing and BMC in a concurrent, automated workflow to determine whether uncovered regions of C code – arising from if and else-if statements – can be exercised by inputs or are provably unreachable. Newly generated inputs may expose gaps in requirement-based testing or reveal requirement violations, while proofs of unreachability can identify dead code requiring refinement.

BMuzz leverages the complementary strengths of fuzzing and BMC. While BMC may struggle with unbounded or deeply nested loops, its SMT solvers can prove unsatisfiability, thereby establishing code unreachability. Fuzzing, in contrast, effectively handles unbounded loops but cannot provide formal guarantees for unreachability. To enhance robustness and effectiveness, *BMuzz* executes multiple BMC instances in parallel alongside fuzzing, each configured with a distinct SMT solver that may perform better for different program characteristics, such as floating-point operations or bitvectors.

The key contributions of this paper are:

- *BMuzz*, a novel portfolio-based framework that integrates fuzzing and BMC in a concurrent workflow for combining coverage-driven instrumentation with formal guarantees of unreachability,
- involving methods for unrolling loops or utilizing wrapper functions suitable for BMC, and
- share results of evaluating *BMuzz* on a subset (61 functions involving roughly 6800 lines of code) of an embedded C standard library that demonstrate its potential for further use.

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00036

II. Related Work

Previous research has explored the use of model checking techniques to improve code coverage [6]. In particular, BMC has become a cornerstone of automated test generation. Tools such as CBMC employ BMC by encoding C and C++ programs into SAT or SMT formulas, producing counterexamples that can serve as test cases to exercise specific execution paths. While CBMC itself supports the automatic generation of test vectors for a given source file, some approaches – similar to ours – first instrument the source code with assert statements and then apply CBMC to extract counterexamples that yield new test vectors, e.g. [7]. [8] investigates the use of BMC specifically to identify missing test vectors with the aim of increasing the coverage of an existing test suite. Although these studies share certain similarities with our proposed *BMuzz* approach, they rely on a single SAT solver, whereas *BMuzz* leverages multiple SMT solvers in a portfolio setting. Related work also includes BlueCov [9], a BMC-based framework for automatically generating test vectors for Java programs. BlueCov is built on JBMC [10], a bounded model checker for Java that shares the same underlying logic as CBMC.

In addition, symbolic execution, often combined with model checking, has emerged as a prominent approach to coverage-driven test generation. A widely used tool in this domain is KLEE [11], which automatically generates test cases for programs compiled to LLVM bitcode through symbolic execution. Building on KLEE, UnitTestBot [12] was developed to integrate with commonly used IDEs and to automatically generate unit tests for C code.

In [13], several test-case generation tools based on BMC, along with a fuzzer, are compared with respect to their performance and effectiveness in producing new test inputs that reveal bugs.

Our approach combines BMC, via the CBMC tool, with fuzzing in a portfolio setting to either generate test vectors that increase the coverage of existing test suites or provide proofs of code unreachability. In this setting, multiple SMT solvers are employed alongside CBMC and compete not only with the fuzzer but also with each other. This competition exploits the strengths of different solvers, allowing the approach to adapt more effectively to varying program characteristics.

III. Motivation

Although code coverage has been extensively studied in academia, its suitability as a metric for evaluating test adequacy remains debated. Prior work has reported conflicting findings: For instance, [14] argues that coverage should not be used as an indicator of test suite effectiveness, whereas [15] suggests that statement coverage can serve as a suitable metric for test quality. Similarly, [16] observes that the correlation between coverage and effectiveness varies across projects, while [17] shows that combining multiple control-flow criteria, and further incorporating data-flow coverage, improves the ability of coverage metrics to identify bug-revealing tests. Despite these divergent perspectives, industry has largely converged on code coverage – particularly the criteria mandated by the industry standards ISO-26262 and DO-178C – as a practical and standardized basis for assessing test adequacy. These standards complement coverage requirements with requirement-based testing, reinforcing their applicability in safety-critical software development.

In particular, they require statement coverage, branch (decision) coverage, and MC/DC. Statement coverage ensures that every executable statement is exercised at least once. Branch coverage requires that each outcome of a control-flow decision (e.g., the true and false branches of an if statement) is tested. MC/DC further demands that each atomic condition within a decision independently influences the decision's overall outcome.

In practice, even well-designed test suites may achieve only limited code coverage. As discussed in [8], several factors can account for uncovered code. Missing coverage may indicate that certain requirements were overlooked during test creation or that unspecified features were implemented. Ambiguous requirements can also prevent the derivation of precise test cases. In other cases, uncovered code may be inherently unreachable by any program input.

This could also be related to software aging. As introduced in [18], software aging refers to the gradual deterioration of a system over time, either due to failure to adapt to evolving requirements or through the accumulation of unstructured incremental changes. The latter often introduces bugs, reduces performance and reliability, and may manifest as decreasing code coverage.

Identifying the root causes of low code coverage is often complex and time-consuming, and if left unresolved, can exacerbate weaknesses in both the code base and the test suite. To address this challenge, we propose *BMuzz*, which either generates inputs to exercise uncovered code sections or employs SMT solvers to prove unreachability. Both outcomes support verification of test-requirement alignment, reveal the need for code refactoring, and may strengthen test adequacy and overall software quality.

IV. Coverage Evaluation with BMuzz

In the following, we present the key aspects of the *BMuzz* approach. An overview of its operation process is provided in Figure 1.

BMuzz involves the following automated steps controlled by Python code:

- Examining code coverage of the original test suite
- Instrumenting uncovered code sections with assert statements
- Analyzing the instrumented source files with BMC and fuzzing in parallel
- Collecting and interpreting results (new test vectors or indication of unreachability)

While both approaches can generally identify new test vectors that increase code coverage, BMC can additionally indicate the unreachability of certain code regions. However, both approaches may also terminate due to timeout constraints,

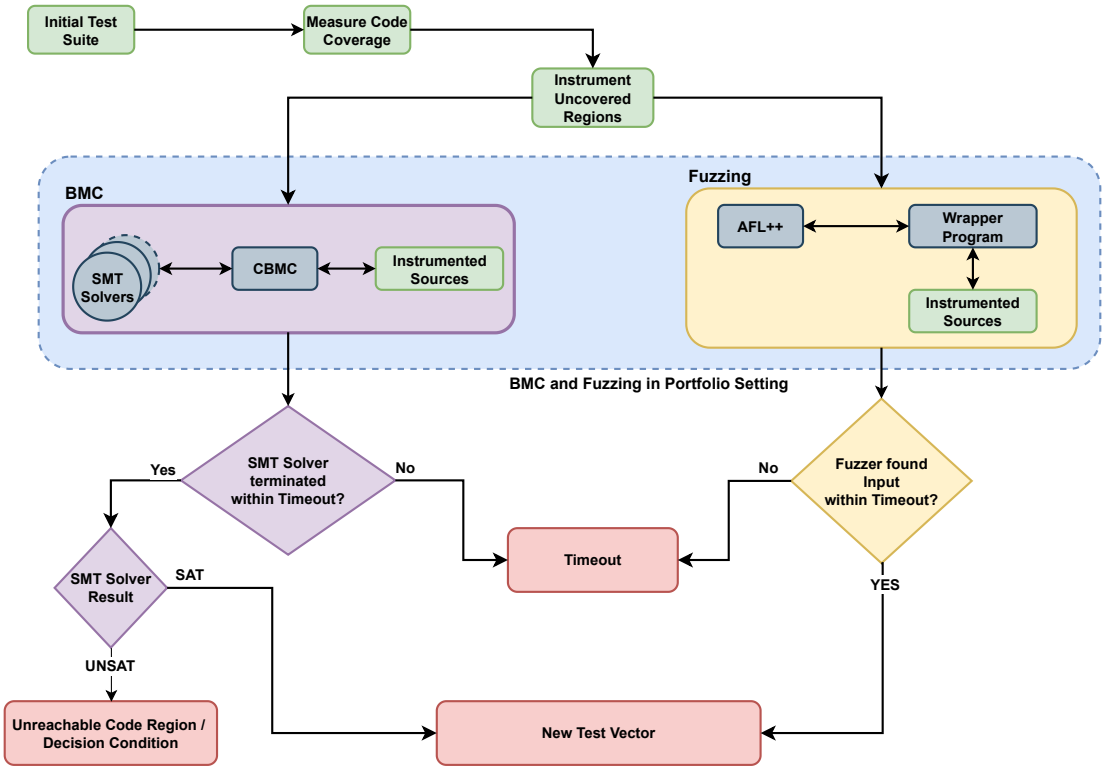

Fig. 1: Overview of the *BMuzz* framework

in which case it remains unknown whether the investigated uncovered code region is reachable by any program input.

Each building block of *BMuzz* is described in more detail in the following.

A. Coverage Measurement and Instrumentation

Code coverage of the initial test suite is measured using the `tinycover` tool [19]. The tool instruments the code under test and produces a coverage report after execution. In addition to statement and branch coverage, the report also provides information on the level of MC/DC coverage for decisions in the analyzed sources. We slightly modified `tinycover` to generate coverage reports in `JSON` format, thereby simplifying the extraction of coverage information.

Based on this report, *BMuzz* identifies conditions and decisions missing specific truth-value assignments required to increase coverage. We refer to these as Coverage Verification Conditions (CVCs), not to be confused with the unrelated SMT solver CVC5. A single CVC represents a particular truth-value assignment of the corresponding condition or decision in the code. *BMuzz* analyzes if a CVC can be reached – that is, whether a valid execution path exists from the function's entry point to the location of the CVC in the code that satisfies its specific assignment. An input satisfying a CVC corresponds to a test vector that increases code coverage. To verify whether such an input exists, the uncovered code sections are instrumented with assertions negating the corresponding CVCs, under the assumption that these sections may be unreachable. The instrumented sources are then subsequently analyzed by fuzzing or BMC: fuzzing attempts to find an input, and BMC a counterexample, such that the inserted assertions are violated; i.e., the corresponding CVC can be satisfied.

For illustration, a code fragment containing an *if(A)* statement shall be considered. If, after executing the original test suite, the coverage report shows that the body of this statement remains uncovered – i.e., condition *A* evaluated to false (F) in every test – then *BMuzz* inserts the negated CVC as `assert(!A)` before the if statement.

Similarly, achieving full MC/DC coverage for an AND-decision $a \wedge b$ may require test cases where $a = F$. In such a situation, `tinycover` records in the coverage report that $a = F; b = X$ is not covered, where X indicates that the specific truth value of b is irrelevant. To address this, *BMuzz* inserts two assertions: `assert(!(!a && b))` and `assert(!(!a && !b))`. An input that triggers one of these assertions to fail corresponds to the missing test vector required to satisfy full MC/DC coverage.

B. Bounded Model Checking

1) General: BMC is a formal verification technique that checks software properties by exhaustively exploring behavior within a fixed bound. It translates the program (up to a bound

k) and its properties into verification conditions (VCs) solved by SAT or SMT solvers. BMC is effective at finding counterexamples to safety or liveness properties and mitigates the state-space explosion inherent in unbounded model checking, but it cannot prove correctness beyond the bound.

BMC models the analyzed program as state transition system $M = (S, T, I)$ where S is the set of states, $I \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is the transition relation [20], [21]. For a property ϕ, and a bound k, BMC unrolls the system k times and generates a VC ψ that is satisfiable iff a counterexample exists within bound k. ψ is expressed in a quantifier-free, decidable subset of first-order logic and submitted to an SMT solver:

$$\psi = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg\phi(s_i)$$

This formula is `SAT` if there is a path $\pi = (s_0, s_1 ..., s_k)$ from I to $\neg\phi$, representing a valid counterexample; `UNSAT` means ϕ holds for the program up to bound k.

In *BMuzz*, we employ CBMC as the BMC engine to check the instrumented assertions. The resulting VCs are solved using a portfolio of state-of-the-art SMT solvers, including `bitwuzla` [22], `CVC5` [23], and `z3` [24]. SMT solvers are employed, as they provide efficient reasoning over specific domains such as floating-point, integers, arrays, and bit-vectors, making them particularly well suited for analyzing embedded C code. Running multiple solvers in parallel may both increase the likelihood of resolving a given verification condition within a given time and broaden the range of program behaviors that can be effectively analyzed. Each violated assertion – i.e., a counterexample identified by BMC – directly yields a new test vector. Conversely, if BMC in combination with the SMT solvers reports that no counterexample exists for a given assertion, then the corresponding CVC cannot take truth values beyond those observed in the initial tests. This implies that the uncovered code section is unreachable. If, however, the solvers reach a predefined timeout, it remains unknown whether a counterexample exists, and thus whether the corresponding uncovered code section can be covered by any program input.

2) Handling Loops: A limitation of BMC arises when analyzing programs with long or nested loops. If the bound k does not capture all possible iterations, BMC under-approximates the search space and may become incomplete.

In *BMuzz*, we address this either through a dedicated BMC wrapper for the analyzed function (see the next section) or by applying the following strategy (illustrated in Figure 2):

CBMC allows users to specify a bound for loop unwinding and provides the `--unwind-assertions` option to check whether all involved loops were fully unrolled within that bound. By default, *BMuzz* instructs CBMC to unwind loops a fixed number of times (e.g., 10 iterations). If the program contains no loops, or if all loops are fully unrolled within this bound, CBMC proceeds with the analysis. Otherwise, if CBMC detects that one or more loops remain partially unrolled (i.e., the assertions generated by the `--unwind-assertions` option are violated) then *BMuzz* repeatedly restarts the analysis with increasing unwind bounds until either all loops are fully unrolled or the corresponding BMC process terminates – either due to a timeout or because a concurrent portfolio component (e.g., fuzzing) has already completed.

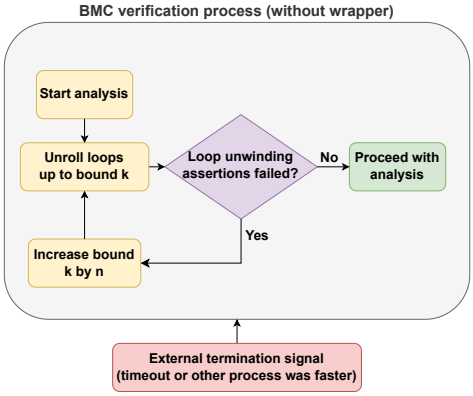

Fig. 2: Loop unwinding strategy in *BMuzz*

3) Applying BMC Wrapper: While CBMC can directly handle primitive data types such as integers and floats, we observed limitations when analyzing functions with pointer parameters. To address this, *BMuzz* allows the configuration of BMC wrapper functions that invoke the function under test. Depending on the function signature, the wrapper can allocate memory for pointer parameters, initialize them with appropriate values, and then call the target function.

An example is shown in Listing 1, which demonstrates how to test a function with a `const char*` parameter. The wrapper first allocates a character array of size `BOUND`, representing a null-terminated string. To ensure null termination, the string length `L` is chosen nondeterministically to be less than `BOUND`, and the array is filled with characters up to that length, with the last element set as the null terminator. If arbitrary character values were desired instead of controlled initialization, CBMC's built-in `__CPROVER_havoc_object` function could be used in place of the loop. Finally, the target function is invoked with the constructed string.

Depending on the body of the analyzed function, wrapper functions can also be used to set limits for loop unwinding by initializing variables or arrays and integrating corresponding assumptions. For example, if the target function in Listing 1 contains a loop that iterates over an array until a null terminator is reached (common in string processing), the wrapper can implicitly bound the number of iterations to `L` by placing a null terminator at position `L` and assuming all preceding elements are valid string characters. In this case, *BMuzz* only needs to be configured to unwind loops by at least `BOUND` iterations when using this wrapper. However, it should be noted that assumptions included in BMC function wrappers – such as bounding loops or enforcing a specific structure on input arrays – may underapproximate the potential inputs to the

Listing 1 Exemplary BMC Wrapper Function

```
void bmc_wrapper() {
  extern RET_TYPE TARGET_FUNC_NAME(const char *);
  char buf[BOUND];

  unsigned int L;
  __CPROVER_assume(L < BOUND);

  for (unsigned int i = 0; i < L; ++i)
  {
    char c;
    buf[i] = c;
    __CPROVER_assume(buf[i] != '\0');
  }
  buf[L] = '\0';
  TARGET_FUNC_NAME(buf);
}
```

target function and thus model its behavior only up to such an implicit bound.

Such optional wrappers have to be defined manually, as they depend on the specific signature of the function under test. If a wrapper is configured, *BMuzz* first instructs CBMC to analyze the target function without it, allowing the analysis with uninitialized argument types such as *NULL* pointers. Only if this analysis fails, the wrapper – potentially enforcing more restrictions on the input arguments – is applied.

C. Fuzzing

In *BMuzz*, we use fuzzing as a complementary technique to BMC. Fuzzing is an automated testing approach that feeds a program under test with large volumes of semi-random, malformed, or unexpected inputs to uncover vulnerabilities, logic errors, and stability issues. By systematically generating inputs, fuzzing is particularly effective at detecting memory corruption, crashes, and security flaws that may be missed by conventional testing. We employ `AFL++` [25], a coverage-guided fuzzer that instruments the program to provide runtime feedback, thereby optimizing input selection and maximizing code path exploration. As with BMC, the tested source files are instrumented with assertions enforcing that the corresponding CVCs cannot assume the missing truth values required by the targeted coverage criteria.

As shown in Figure 3, the instrumented source files are compiled (currently without any optimization) and linked with a fuzzing wrapper program that provides a `main` function and helper routines for reading input from `stdin` (supplied by the fuzzer) and converting it into valid parameter types for the tested function. Since fuzzing inputs are mapped directly to function parameters, assigning `NULL` pointers as arguments is currently not supported. Since the wrapper reads fuzzing input from `stdin` and converts it into the required types, only fixed-length arrays are supported as function parameters. Consequently, if the analyzed function exhibits behavior that depends on array lengths exceeding this fixed size, the fuzzer cannot generate inputs to trigger such behavior. If the fuzzer creates an input that causes the evaluated function to abort – indicating that one of the instrumented assertions was violated – a new test vector has been found that increases code coverage. If, however, the program does not abort within a specified timeout, it remains unknown whether any input exists that can cover the evaluated code regions. Unlike BMC, fuzzing cannot prove the unreachability of uncovered code sections. Nevertheless, fuzzing is particularly well suited for analyzing constructs with variably bounded or nested loops, where BMC might encounter limitations.

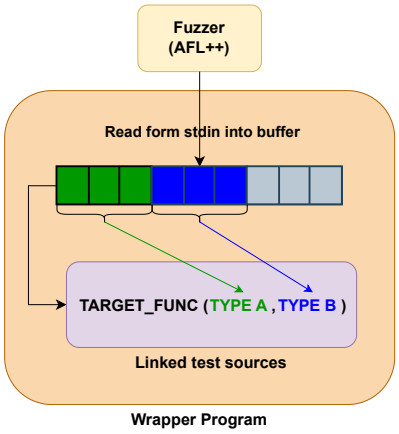

Fig. 3: Fuzzing Wrapper in *BMuzz*

V. Evaluation

In this section, we evaluate *BMuzz* within the development process of a commercial product: a standard C library for embedded systems. The library was developed in compliance with automotive domain standards, and its existing test suite already achieves high code coverage. Although neither the source code nor the test suite can be made publicly available, the library is representative of real-world embedded software, and its high baseline coverage provides a meaningful context for assessing *BMuzz*'s effectiveness. In addition, the library currently supports only a limited set of architectures and contains relatively few assembly statements, which increases the applicability of CBMC or `AFL++`.

For the evaluation, we selected a subset of functions from the library, measured their code coverage after removing 50% of the associated tests, and then applied *BMuzz* to determine whether it could find the missing test vectors to re-establish the previous coverage level. Functions that depend on specific hardware features or rely heavily on platform-specific assembly code – such as dynamic memory allocation functions (e.g., `malloc`) or exit functions – were excluded. The final evaluation set consisted of 61 functions including approximately 6800 lines of code drawn from various C headers, ranging from floating-point-intensive routines in `math.h` to string operations in `string.h`. Based on the reduced test suite, the analysis of partly covered or uncovered conditions and decisions in these functions yielded a total of 314 CVC goals.

During this evaluation, we encountered a limitation in CBMC: `union` statements for converting between floating-point numbers and integers are currently not supported. To mitigate this issue, we manually adapted the source code prior to applying *BMuzz*.

We begin by evaluating the performance of *BMuzz* as a standalone approach and subsequently compare its results against using either BMC or fuzzing alone.

A. Standalone Evaluation of BMuzz

First, we applied *BMuzz* to evaluate the CVCs of the selected functions, using a timeout of 600 seconds per verification task. The portfolio configuration comprised three BMC instances (with CBMC (6.7.1) and either `bitwuzla` (0.7.0), `cvc5` (1.2.1), or `z3` (4.13.4)) and one fuzzing instance (with `AFL++` (4.34a)). All instances were executed in parallel, and *BMuzz* terminated the remaining instances as soon as one instance resolved the respective CVC or the timeout was reached.

Figure 4 summarizes the overall results, showing the number of CVCs for which new test vectors were generated, BMC with SMT solvers produced an unreachable verdict, or the task remained unsolved within the provided time limit. In total, *BMuzz* successfully resolved 312 out of 314 CVCs (99.36%), with 275 (87.58%) reachable by newly generated test vectors and 37 (11.78%) proven unreachable. Only 2 CVCs (0.64%) could not be resolved within the time bound. A particularly noteworthy outcome is the performance of `bitwuzla`, which alone resolved 233 CVCs as reachable and 36 as unreachable – 269 CVCs in total (85.67%) – before any of the other approaches. In contrast, `cvc5` was first to resolve only one reachable CVC, while `z3` produced 8 reachable and 1 unreachable verdicts ahead of the others. Fuzzing with `AFL++` discovered 33 reachable CVCs, corresponding to 10.51% of the total.

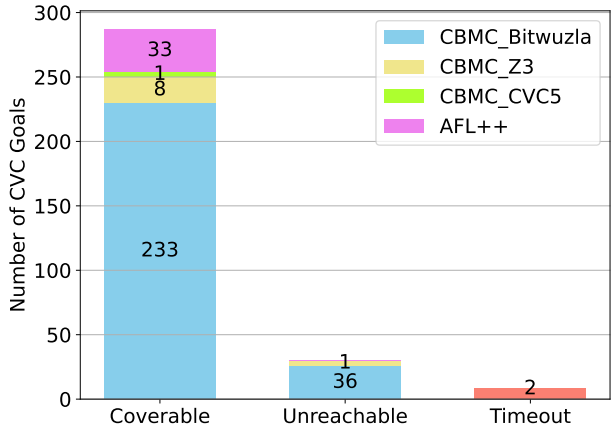

Fig. 4: Overall results of *BMuzz*

Figures 5 (logarithmic scale) and 6 show the time taken by the first verification process within *BMuzz* to resolve each CVC, distinguishing between those found to be reachable and those proven unreachable. Red crosses indicate `AFL++`, green `cvc5`, blue `bitwuzla`, and orange `z3`. The scatter plots reveal that `bitwuzla` is most often the fastest solver for both satisfiable and unsatisfiable CVCs, typically resolving them in under one second. Notably, `bitwuzla` also tends to be the first to resolve more complex CVCs that require longer runtimes.

Figure 5 further displays that fuzzing with `AFL++` has a relatively narrow distribution of runtimes, most often resolving CVCs between roughly 0.5 and 1 seconds. Its performance lies roughly between the fast and slow CVC resolutions achieved by the SMT solvers. Although `cvc5` was the first to resolve only a single reachable CVC, it is noteworthy that this presumably more complex case was solved in approximately 10 seconds. `z3`, in comparison, shows a distribution of runtimes in the range of 0.1 to 1 seconds.

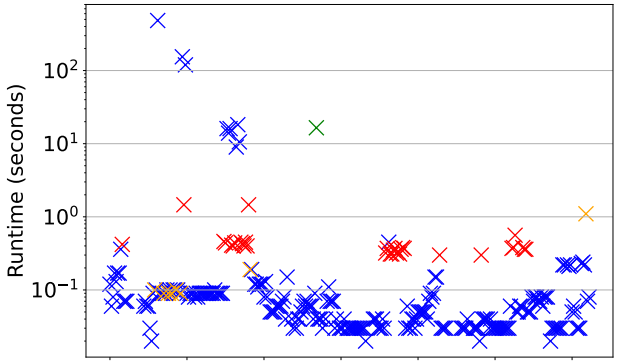

Fig. 5: Runtime for each satisfiable CVC by first method (blue for `bitwuzla`, red for `AFL++`, orange for `z3`, green for `cvc5`)

Figure 6 shows that `bitwuzla` again demonstrates strong effectiveness in proving unreachability, outperforming the other SMT solvers and resolving all CVCs in under 0.15 seconds. `z3` proved a single CVC unreachable first, requiring approximately 0.1 seconds. In contrast, `cvc5` did not prove any CVCs unreachable before the other SMT solvers, and `AFL++` cannot decide on unreachability at all.

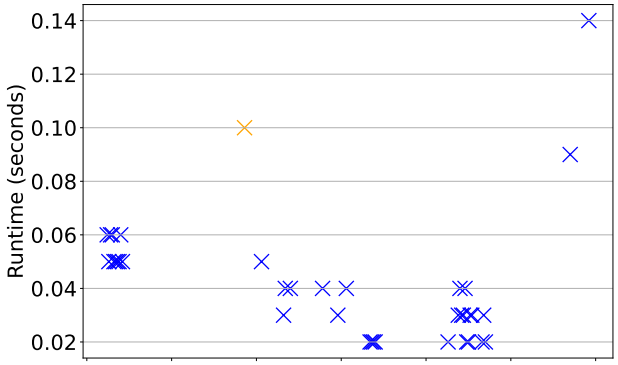

Fig. 6: Runtime for each unreachable CVC by first method (blue for `bitwuzla`, orange for `z3`)

B. Comparison of BMuzz with BMC and Fuzzing as Separate Approaches

In the following, we compare the results of *BMuzz* with those obtained by using either BMC or fuzzing alone. For this purpose, we applied the same set of CVCs but with a shorter timeout of 30 seconds per task. The BMC-only configuration used the same portfolio of SMT solvers as *BMuzz* but without fuzzing, while the fuzzing-only configuration relied solely on `AFL++`.

Figure 7 summarizes the overall results of the three approaches. *BMuzz* resolved 272 CVCs as reachable, 37 as unreachable, and left 5 unsolved due to reaching the time limit. The higher number of unsolved CVCs compared to the previous evaluation is explained by the reduced timeout of 30 seconds. Using BMC alone, 259 CVCs were resolved as satisfiable, 37 as unsatisfiable, and 18 remained unsolved. Fuzzing alone achieved 197 satisfiable CVCs, while 117 timed out. Overall, compared to *BMuzz*, the BMC-only configuration resolved 13 fewer CVCs, and fuzzing-only resolved 112 fewer.

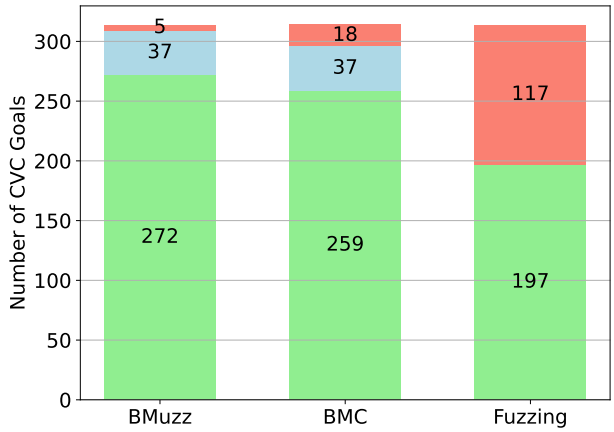

Fig. 7: Overall results of *BMuzz*, BMC-only, and fuzzing-only (green for satisfiable CVCs, blue for CVCs proven unreachable, and red for unsolved CVCs due to timeouts)

Figure 8 presents a cumulative line plot comparing the number of resolved CVCs over the total runtime of each approach. The green line represents *BMuzz*, the blue line corresponds to BMC alone, and the red line indicates fuzzing alone. The plot shows that *BMuzz* consistently resolves more CVCs over time than the other two approaches. Although the BMC-only approach follows a trend similar to *BMuzz*, and fuzzing alone shows slower progress, the integration of fuzzing into *BMuzz* leads to an overall reduction in solving time. This demonstrates the advantage of a portfolio-based strategy that combines fuzzing and BMC.

VI. Strengths and Limitations

Our results indicate that combining BMC with multiple SMT solvers and fuzzing in a portfolio setting within *BMuzz* yields faster results. While the SMT solvers – particularly `bitwuzla` – showed notable performance advantages, fuzzing was in some cases faster at identifying new test vectors. Moreover, *BMuzz* supports seamless integration of any C code that can be compiled independently and is accompanied by an initial test suite for measuring baseline coverage. Compared to writing conventional unit tests, this setup requires only minimal effort: wrapper functions for fuzzing must be (and for BMC may be) provided, but they can be reused across multiple functions with same signatures, further reducing setup overhead. Besides addressing coverage gaps, the CBMC component – and to some extent the fuzzing component – integrated into *BMuzz* can also be applied to verify arbitrary assertions in the source code and to detect runtime errors such as overflows or null-pointer dereferences during coverage evaluation, thereby extending the scope of *BMuzz* beyond test adequacy assessment.

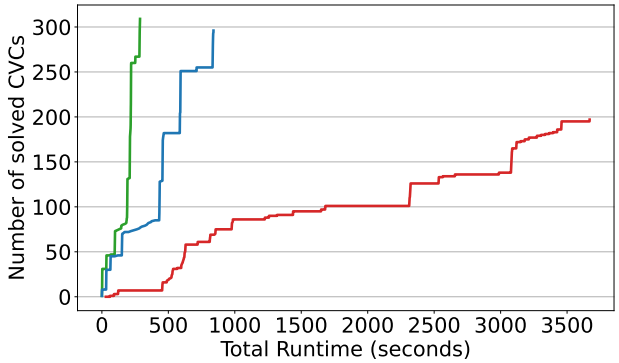

Fig. 8: Resolved CVCs over total runtime by *BMuzz* (green), BMC-only (blue), and fuzzing-only (red)

However, the current implementation of *BMuzz* also has limitations. Currently, instrumentation is performed only for evaluating conditions and decisions in if- or else-if statements at the source-code level, which may miss certain coverage aspects. Furthermore, *BMuzz* is currently limited to analyzing code that assumes type sizes compliant with common 32- or 64-bit architectures; for example, it cannot handle 16-bit floating-point numbers. Another restriction is its inability to accurately model hardware-specific behavior, particularly when code interacts directly with peripherals or depends on platform-specific features. In addition, inline assembly statements, which are common in low-level or performance-critical code, are not supported.

VII. Conclusion and Outlook

This paper demonstrates that *BMuzz* complements test coverage analysis in software verification workflows. Its portfolio-based approach – running multiple BMC instances with different SMT solvers alongside fuzzing – effectively leverages their complementary strengths. Empirical results show that this strategy achieves faster results than using a single solver or technique alone, thereby improving the overall efficiency of the verification process. By employing BMC to provide formal guarantees of code unreachability, *BMuzz* reduces ambiguity in test adequacy evaluations and supports

decisions related to code maintenance and refactoring. While the techniques implemented in *BMuzz* to guide CBMC in handling variably bounded loops showed strong potential in the evaluated source code, the integration of fuzzing further enables efficient analysis without explicitly requiring loop-unwinding bounds.

Future work will focus on several directions. One is improving the usability and robustness of *BMuzz*. This includes developing more sophisticated instrumentation techniques to handle a broader range of uncovered code sections, such as switch- or loop-iteration-conditions, by employing advanced approaches based on abstract syntax tree instrumentation. This also involves using specialized dictionaries for fuzzing to better adapt input generation to specific parameter types, as well as the incorporation of additional SMT solvers into the portfolio to further enhance BMC effectiveness. Another is the full integration of *BMuzz* into CI/CD pipelines to enable continuous, automated verification of coverage gaps, providing results alongside routine build and test feedback. A key aspect for further investigation is the evaluation of *BMuzz* on a broader range of real-world software projects, including open-source applications, to assess its generalizability and effectiveness. In this context, benchmarking *BMuzz* against other state-of-the-art test generation and coverage analysis tools would provide a more comprehensive assessment of its performance.

REFERENCES

[1] *ISO26262 Road Vehicles — Functional Safety — Part 6: Product Development at the Software Level*, International Organization for Standardization (ISO) Std., 2018.

[2] *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Inc. Std. DO-178C, 2011.

[3] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," NASA, Tech. Rep., 2001.

[4] D. Kroening and M. Tautschnig, "CBMC–C Bounded Model Checker: (Competition Contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. Springer, 2014, pp. 389–391.

[5] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[6] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.402

[7] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina, "Using Bounded Model Checking for Coverage Analysis of Safety-Critical Software in an Industrial Setting," *Journal of Automated Reasoning*, vol. 45, no. 4, pp. 397–414, Dec. 2010.

[8] A. Nellis, P. Kesseli, P. R. Conmy, D. Kroening, P. Schrammel, and M. Tautschnig, "Assisted coverage closure," in *NASA Formal Methods*, S. Rayadurgam and O. Tkachuk, Eds. Cham: Springer International Publishing, 2016, pp. 49–64.

[9] M. Güdemann and P. Schrammel, "Bluecov: Integrating test coverage and model checking with jbmc," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1695–1697. [Online]. Available: https://doi.org/10.1145/3555776.3577829

[10] L. Cordeiro, D. Kroening, and P. Schrammel, "Jbmc: Bounded model checking for java bytecode," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 219–223.

[11] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, pp. 209–224.

[12] D. Ivanov, A. Babushkin, S. Grigoryev, P. Iatchenii, V. Kalugin, E. Kichin, E. Kulikov, A. Misonizhnik, D. Mordvinov, S. Morozov, O. Naumenko, A. Pleshakov, P. Ponomarev, S. Shmidt, A. Utkin, V. Volodin, and A. Volynets, "Unittestbot: Automated unit test generation for c code in integrated development environments," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 380–384.

[13] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Hardware and Software: Verification and Testing*, O. Strichman and R. Tzoref-Brill, Eds. Cham: Springer International Publishing, 2017, pp. 99–114.

[14] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 435–445. [Online]. Available: https://doi.org/10.1145/2568225.2568271

[15] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 72–82. [Online]. Available: https://doi.org/10.1145/2568225.2568278

[16] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 560–564.

[17] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 151–156.

[18] D. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, 1994, pp. 279–287.

[19] "tinycover," Aug. 2025, original-date: 2023-11-19T02:03:11Z. [Online]. Available: https://github.com/rmbishop/tinycover

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[21] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, Jul. 2001.

[22] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds. Cham: Springer Nature Switzerland, 2023, pp. 3–17.

[23] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "Cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, vol. 13243, pp. 415–442.

[24] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. R. Ramakrishnan, and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340.

[25] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi