Challenges in Comparing Software Analysis Tools for C*

Florian Merz, Carsten Sinz, and Stephan Falke

Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{florian.merz, carsten.sinz, stephan.falke}@kit.edu
http://verialg.iti.kit.edu

Abstract. Comparing different software verification or bug-finding tools for C programs can be a difficult task. Problems arise from different kinds of properties that different tools can check, restrictions on the input programs accepted, lack of a standardized specification language for program properties, or different interpretations of the programming language semantics. In this discussion paper we describe problem areas and discuss possible solutions. The paper also reflects some lessons we have learned from participating with our tool LLBMC in the TACAS 2012 Competition on Software Verification (SV-COMP 2012).

1 Introduction

There is a growing number of tools focusing on analyzing the correctness of C programs using formal methods, e.g., model checkers such as BLAST [4] and SATABS [7], bounded model checking tools such as LLBMC [10], CBMC [6], or F-Soft [9], and symbolic execution tools like KLEE [5].

While all of these tools have similar design goals, comparing them can be cumbersome (Alglave *et al.* [1] seem to confirm this). Tool comparisons in other fields (like SAT and SMT [2]) suggest that annual competitions and the possibility to quickly and easily compare tools act as a major driving force within a research community for developing better tools. This has also been realized by the organizers of the software verification competition SV-COMP 2012 [3], which took place in March/April 2012 for the first time.

2 Challenges

In the following we discuss challenges in comparing automated software analysis tools for C programs and suggest possible solutions for each of these problems.

* This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

Challenge 1: *What is a correct program?* While many would say that a *specification* is needed in order to determine whether a program is correct or not, we take a slightly different, more general stance here.

Besides errors with respect to a given specification, a second, equally important class of errors is due to underspecification in the programming language definition, e.g. many aspects of the C programming language are underspecified by the C99 standard [8]. Such underspecification is done intentionally to allow for more freedom in implementing the standard on different computer architectures.

As an example, the result of a signed integer overflow in C is deliberately left unspecified in order to allow not only two's complement, but also alternative implementations like, e.g., a sign plus magnitude encoding.

The C standard carefully distinguishes three different kinds of underspecification: *undefined* behavior, *unspecified* behavior, and *implementation-defined* behavior. Implementation-defined behavior indicates that the semantics is defined by the implementation at hand, which usually depends on the architecture and the compiler. In the C99 standard, unspecified behavior is defined as behavior for which the standard "provides two or more possibilities and imposes no further requirements on which is chosen in any instance".

Last but not least, undefined behavior is the least-specified of the three: broadly speaking, if undefined behavior occurs "anything might happen". Or, as Scott Meyers puts it [11]: undefined behavior might even "reformat your disk, send suggestive letters to your boss, fax source code to your competitors, whatever." Because of this malicious nature of undefined behavior we propose to consider it always as a fault in a software implementation.

Proposal: Any program whose behavior depends on undefined or unspecified aspects of a programming language should be considered incorrect. Important errors in C programs, such as buffer overflows (access to not allocated memory), null-pointer dereferences, or integer overflows fall into this class.

Challenge 2: *Which error classes should be supported?* Annex J of the C99 standard defines an extensive list of diverse cases in which undefined behavior occurs, possibly too extensive to support all of these cases in academic tools. On the other hand, for the sake of comparability, a minimal set of error types that should be supported by all verification tools needs to be defined. Most program analysis tools support checking for the following types of errors:

- Signed integer arithmetic overflow.
- Division by zero.
- Undefined bitwise shift behavior.
- Array out of bounds errors.
- Memory access violations.
- Violation of user provided assertions.

This is by no means a complete list of the errors that different tools can detect (e.g., errors in signed integer shift operations are handled by some tools, but not by others). And even if an error class is handled by a tool, its semantics might be interpreted (slightly) differently.

Proposal: One possibility to make checks of such common error classes available in a larger set of tools is to encode them directly into C source code (e.g., by using a preprocessor), as was done in SV-COMP 2012. See Figure 1 for an example. By using such an encoding, the task of bug finding can be reduced to reachability analysis.

```
1  int foo(int x)
2  {
3    return 1/x;
4  }
```

```
1  int foo(int x)
2  {
3    if (x == 0) {
4  error:
5      // unreachable?
6    } else {
7      return 1/x;
8    }
9  }
```

Fig. 1: Encoding a devision-by-zero-error as a reachability check.

This approach helps tool developers to concentrate on the the tools backend, instead of having to implement all different error kinds. It also ensures that the same errors are interpreted in the same way by all tools being evaluated.

Unfortunately, some properties cannot be expressed in simple C code, e.g., memory access correctness, which requires extensive knowledge about the operating system's state. Furthermore, this approach results in intrusive changes to the benchmark, and we therefore suggest to not encode properties as reachability, but require all tools to explicitly support all error classes.

Challenge 3: *How can we assemble suitable benchmark problems?* Assembling benchmark problems to compare the performance of bug-finding and verification tools is an intricate task. Benchmark problems should be of adequate hardness, be as close as possible to real-world programs, and the kind and location of the error should be unambiguous. Thus, a benchmark program should either contain no bug at all or a single, well-known bug. Benchmarks that contain multiple bugs hinder comparison of the results, as different tools might detect different bugs.

Taking undefined behavior as described in Section 2 seriously, we arrive at the following problem: if there is a program location that can exhibit undefined behavior, the program can do anything after having executed the instruction at that location. It could, e.g., jump straight to any position in the code and cause arbitrarily weird follow-up behavior. Thus, a fully conforming verification tool also needs to be able to detect any kind of undefined behavior, even if only a specific class of errors is part of the benchmark suite.

In practice, undefined behavior can have a wide range of different effects. Null-pointer dereferences will typically result in a program crash, whereas signed integer does so rarely, if ever.

No verification tool known to us actually treats undefined behavior according to the standard ("everything is possible"), but opts for a more specific, practical behavior instead. Common choices for such a more practice-oriented treatment are:

- Treat signed integer overflow according to two's complement, assuming that no side effects occur.
- Assume program termination on division by zero.
- Assume program termination on invalid memory access operations.

But the result of an overflowing signed integer operation could also be modeled as a fresh, unconstrained integer, while still assuming the operation to have no side effects. This essentially reduces undefined behavior to undefined results. Reading from invalid memory locations and division by zero could also be modeled using fresh, unconstrained variables, writing to invalid memory locations could be treated as a no-op. As can be seen, there is a wide range of different options on how to treat undefined behavior, all with their own merits and without a clear winner.

Proposal: Benchmark problems should contain exactly one error or no error at all. Moreover, the intended semantics of all cases of undefined behavior should either be clearly specified, or all benchmark problems should be entirely free of undefined behavior.

Challenge 4: *How to treat interface functions?* We call a function for which no implementation is provided an *interface function*. Typical examples of interface functions are (standard) library functions or operating system calls. The problem here is that the semantics of such a function is not directly available, but has to be provided by additional means. If a benchmark problem uses such an interface function the tool has to handle this gracefully. There are several options how to do this:

1. The function can be treated as exhibiting undefined behavior, essentially resulting in a call to the function being an error.
2. The result of each call to the function can be interpreted as a fresh variable of the appropriate type, assuming that the function does not have any side effects.
3. Similarly, the theory of uninterpreted functions can be used to implement functional congruence, while still assuming that the function does not have any side effects.
4. A clearly specified semantics is built into the tool.
5. Finally, short stubs can be provided that mimic essential aspects of the real semantics.

Proposal: Especially for standard library functions (e.g., `memcpy`, `strlen`, ...) and operating system calls the last two approaches seem to be preferable. If interface functions are built into the tool, their semantics has to be specified precisely. For other functions, if they are guaranteed to have no side effects, the third option (uninterpreted functions) is preferable.

Challenge 5: *How to treat the initial state of a benchmark problem?* Many benchmark problems currently in use are small stand-alone programs extracted from larger programs. While in a stand-alone program the initial state of global variables, for example, is defined to be zero by the C standard, this does not seem to faithfully capture the intended semantics of an extracted benchmark, namely to isolate a fraction of the program relevant for the error at hand, while still allowing all possible behaviors of the full program.

Proposal: If a benchmark problem is extracted from a larger program, false initialization effects should be avoided. This can be achieved, e.g., by mimicking the behavior of the missing program parts by stub initialization functions that initialize those global variables that are not initialized explicitly to non-deterministic values.

3 Software Verification or Bug Finding?

When software analysis tools are compared, the aim of the comparison has to be defined succinctly. The primary question is: Does the comparison focus on software verification[1] or on bug finding? While both give an estimate on safety, there is a notable difference in what kinds of results are to be expected.

Software verification tools should be required to not only return "safe" as an answer, but also provide a proof of safety. Bug finding tools, on the other hand, are usually not able to prove safety, but instead are intended to find as many bugs as possible. For each bug they should provide a valid trace that allows to reproduce the bug when running the program.[2] Note that producing such an error-trace is usually not required for software verification tools.

This difference also becomes visible in the way one typically thinks about soundness and completeness.[3] In formal software verification soundness means: if the tool claims the code to be correct then the program is indeed correct. Completeness here means: if the program is correct then the software verification tool can generate a proof. Software verification tools cannot necessarily provide a program trace for bugs. In bug finding it is the other way round: if a sound bug finding tool reports an error, that error really exists in the program (i.e., there are no false positives). A bug finding tool is complete when: if the code contains a bug, then the tool finds it. Bug finding tools usually cannot provide a (sensible) proof of correctness, even when they can guarantee that there is no bug in the code.

In almost all tool-comparisons there seems to be agreement that unsoundness should be punished harder than incompleteness. If software verification tools

[1] In the following, with software verification we mean formal proofs of a software's correctness.

[2] We consider developing a common, standardized format for error traces an important research goal by itself.

[3] If a tool is sound and complete, this distinction is not important, but in reality most tools are indeed incomplete.

and bug finding tools are mixed up, this cannot be distinguished that easily. Should we take the definitions from the verification community or the bug-finding community? But the difference is not only in the tool, but also how the tool is used. For instance, our bounded model checker LLBMC [10] can be used as a verification tool if bounds are high enough, or as a bug finding tool, if bounds are set low and high code coverage is to be achieved.

Considered as a software verification tool, LLBMC can be used with increasingly higher bounds, until it either proves the property or reaches a time-out. On the other hand, if the bounds are set low, LLBMC acts as a bug-finding tool aimed at finding bugs quickly, but sacrificing completeness (in the sense of missing bugs that could only be found with higher bounds).

We thus believe that any comparison of tools (in particular in competitions) should set a clear focus on either software verification or bug finding, depending on the goals of the competition.

References

1. J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *Proc. ATVA 2011*, volume 6996 of *LNCS*, pages 28–42, 2011.
2. C. W. Barrett, L. M. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In *Proc. CAV 2005*, volume 3576 of *LNCS*, pages 20–23, 2005.
3. D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS 2012*, volume 7214 of *LNCS*, pages 504–524, 2012.
4. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.
5. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI 2008*, pages 209–224, 2008.
6. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, 2004.
7. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, pages 570–574, 2005.
8. ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
9. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In *Proc. CAV 2005*, volume 3576 of *LNCS*, pages 301–306, 2005.
10. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE 2012*, volume 7152 of *LNCS*, pages 146–161, 2012.
11. S. Meyers. *Effective C++: 55 Specific Ways to Improve your Programs and Designs*. Addison-Wesley, 2005.