2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

SMACK Software Verification Toolchain[*]

Montgomery Carter, Shaobo He,
Jonathan Whitaker, Zvonimir Rakamarić
University of Utah, USA

Michael Emmi
IMDEA Software Institute, Spain

ABSTRACT

Tool prototyping is an essential step in developing novel software verification algorithms and techniques. However, implementing a verifier prototype that can handle real-world programs is a huge endeavor, which hinders researchers by forcing them to spend more time engineering tools, and less time innovating. In this paper, we present the SMACK software verification toolchain. The toolchain provides a modular and extensible software verification ecosystem that decouples the front-end source language details from back-end verification algorithms. It achieves that by translating from the LLVM compiler intermediate representation into the Boogie intermediate verification language. SMACK benefits the software verification community in several ways: (i) it can be used as an off-the-shelf software verifier in an applied software verification project, (ii) it enables researchers to rapidly develop and release new verification algorithms, (iii) it allows for adding support for new languages in its front-end. We have used SMACK to verify numerous C/C++ programs, including industry examples, showing it is mature and competitive. Likewise, SMACK is already being used in several existing verification research prototypes. Our demonstration of SMACK can be found on YouTube at the following address: https://youtu.be/SPPSC1KdRzs

1. INTRODUCTION

Prototyping is a requisite step in presenting innovative software verification techniques to the research community; in order to validate a proposed technique, it must at least be demonstrated to be functional, if not a performance improvement over existing techniques. However, prototyping is often a significant engineering effort. With a traditional monolithic implementation approach, a prototype must provide all of the features of an end-to-end verification tool, from source program parsing and interpretation, to model generation of the interpreted program, and finally verification of the generated model. In contrast with the rapid pace of innovation in the field of software verification, the long development time typical of monolithic tool prototypes is burdensome, requiring researchers to spend more time developing tools and less time developing new techniques. This has impact not only on researchers, but also on the software engineering community that researchers are ultimately trying to support; delayed tool prototyping leads to delayed research publication, which leads to delayed dissemination of new and innovative verification techniques.

Recent advances in *Satisfiability Modulo Theories* (SMT) solvers [3] have led to steady growth in the popularity of automated SMT-based software verification tools. New developments in verification algorithms, in conjunction with improvements in computational efficiency, have brought such tools into industry practice (e.g., Static Driver Verifier [1], SAGE [8]). Most SMT-based tools have to bridge a gap between the complexities of their input language and the simplified first-order-logic-based formats that most modern SMT solvers take as input.

The Boogie *intermediate verification language* (IVL) was designed to alleviate the complexity of modeling new source languages and implementing new verification algorithms [6]. Boogie is well-positioned to replace monolithic prototypes with a more flexible and modular approach. The Boogie IVL serves as a layer of abstraction between the front-end source languages and the back-end SMT-based algorithms for verifying them. This separation of concerns allows input programs to be translated into Boogie IVL, rather than directly proceeding to *verification condition* (VC) generation and SMT solving. Likewise, the Boogie IVL has simple and well-defined syntax and semantics rules, yet expressive enough to provide a common target for modeling the diverse and complex semantics of input languages. In addition to defining the Boogie IVL, the Boogie project also provides a back-end verifier [2], as well as an API for parsing Boogie IVL, VC generation, and interfacing with an underlying SMT solver of choice. The Boogie API allows back-end verification tools to support Boogie IVL with minimal effort.

The LLVM project is a modern compiler infrastructure which has seen rapid adoption both in academia and industry [11]. At its core, the LLVM infrastructure aims to provide a universal *intermediate representation* (IR) for programs written in source languages supported by LLVM-based front-ends. It is an open and extensible framework, containing a number of front-ends that provide support for variety of languages including C, C++, Objective-C, Swift,

[*]Supported in part by the University of Utah Office of Undergraduate Research and NSF CCF 1346756/1421678.

ICSE '16 Companion, May 14–22, 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4205-6/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889163

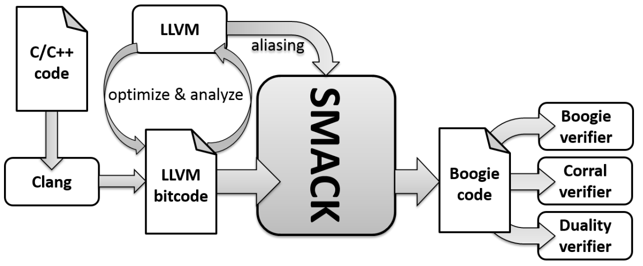

Figure 1: SMACK Toolchain

and Rust. The LLVM IR enables implementation of efficient compiler transformations and analyses, and it provides the methods needed to easily access the resulting programs and information, respectively.

The SMACK software verifier [14] includes a toolchain that leverages Boogie and LLVM to provide an end-to-end verification framework (see Figure 1). The SMACK toolchain is centered around the SMACK tool itself, which takes LLVM IR code as input and translates it into a Boogie IVL program. The toolchain utilizes an LLVM front-end, such as LLVM's Clang, to compile source programs into LLVM IR. The toolchain then passes the resulting *bitcode* to SMACK, which translates it into a Boogie IVL program that models the original input program. The toolchain then passes the Boogie IVL program to the back-end verifier of choice.

The modular nature of SMACK eases the burden on researchers by promoting the development of verification algorithms for the simple Boogie IVL, effectively decoupling the implementations of verification algorithms from the complex details of source languages. Support for new source languages can be integrated easily once an LLVM front-end compiler becomes available for them. Likewise, by embracing Boogie IVL as a canonical program representation, SMACK not only simplifies the development of verification techniques, but also fosters the development of interoperable technology in which verification back-ends can be easily swapped. This enables rapid prototyping, and thus eases the development burden. The consistent, single interface that the SMACK toolchain presents to the developers allows for programs to be verified using existing, well-established back-end verification algorithms. However, this same familiar interface can also be used to test and evaluate the most recent advances in SMT-based verification technology.

SMACK provides benefits to the software verification and software engineering communities alike. The software verification community benefits by reducing the overhead of novel algorithms and techniques prototyping. The software engineering community benefits as cutting-edge advances in verification techniques and technology are made accessible through the consistent, mature toolchain interface provided by the project.

2. SMACK OVERVIEW

SMACK started as a relatively simple tool for translating programs compiled as LLVM IR into Boogie IVL, for consumption by independent back-end verification algorithms. In an effort to further end-user adoption and improve its suitability for real world applications, significant effort has been expended in developing an end-to-end toolchain. The new SMACK toolchain simplifies the application of SMACK and the back-end Boogie verification tools.

As seen in Figure 1, the SMACK toolchain begins with an LLVM front-end compiler. The front-end compiler generates LLVM IR from the original input source program. Targeting LLVM IR enables support for multiple source languages with a single intermediate representation. Further, by utilizing the LLVM APIs, SMACK can leverage the full functionality of the LLVM libraries, including parsing and static analysis tools. Once an LLVM front-end compiler has generated a valid LLVM IR program, the toolchain passes this program to the SMACK tool.

At the heart of the SMACK toolchain is the SMACK tool itself [14]. SMACK translates LLVM IR code into a Boogie IVL program that models the input program. During translation, SMACK leverages LLVM's transformation tools to perform optimizations which improve the performance and accuracy of verification. Chief among these is the memory model employed by SMACK. SMACK leverages LLVM's data structure analysis (DSA) module to perform pointer alias analysis. This allows SMACK to divide the memory heap into distinct regions such that pointers associated with each region can only alias with each other, and not those from other regions. This optimization greatly reduces the verification time for programs with many memory accesses. Other optimizations include handling memory accesses on a byte-size level rather than the word-size level, thus enabling precision for type unsafe operations. SMACK applies these optimizations to the input program as it translates the LLVM bitcode into a valid Boogie IVL program.

Once a Boogie IVL program modeling the original input program is available, it is sent to a back-end verification tool for verification. It is at this stage where the SMACK toolchain really distinguishes itself. Any solver or verifier that accepts Boogie IVL programs as input can easily be included as back-end in the SMACK toolchain. This facilitates very rapid delivery of new verification algorithms and techniques through the consistent interface provided by the SMACK toolchain. Next, we briefly present some of SMACK's distinguishing features.

2.1 Toolchain Front-Ends

SMACK's preliminary implementation has focused on the translation of C programs through LLVM IR into Boogie IVL. We currently support C via the Clang C compiler for LLVM. Our initial experience in verifying C-language programs with SMACK is encouraging. SMACK translates large, full-featured programs — including the entire Contiki operating system, at around 100 KLOC of C code — and has been used on intricate implementations which make extensive use of features such as dynamic memory allocation [14]. Likewise, we have tested SMACK on various aspects of the C++ programming language, including classes, methods, class inheritance, and method overloading. We are currently lacking support for polymorphism in C++ applications, and are working on improving that aspect. Recently, we have implemented preliminary support for Rust, whose compiler is also based on LLVM. As future work, we are looking to support other languages that have an LLVM front-end compiler such as FORTRAN, Go, Objective-C, and Swift to name a few.

2.2 Toolchain Back-Ends

There are several back-ends currently incorporated into the SMACK toolchain (see below). The fact that there are already several supported back-ends demonstrates the ability of researchers to quickly prototype new verification algorithms, and present them to the software verification community without having to develop a full end-to-end verification tool.

2.2.1 *Boogie Verifier*

The Boogie program verifier is the original back-end solver for Boogie IVL programs, and is part of the Boogie project. It provides a sound, base-line solver — its focus has been on soundness and completeness, rather than scalability to large code bases.

2.2.2 *Corral Verifier*

Corral [10] utilizes the Boogie API for parsing of Boogie IVL programs, as well as invocation of the underlying SMT solver. However, Corral implements several new, innovative algorithms for bounded model checking. Corral differentiates itself from the Boogie verifier by implementing novel algorithms and techniques, support for verification of concurrent programs, and enhanced error trace reporting, among others. Corral is currently the default back-end verifier configured for the SMACK toolchain.

2.2.3 *Duality Verifier*

Duality [12] builds upon Corral, and as such, includes all of the innovation delivered by Corral. Duality complements Corral by adding the ability to construct proofs for programs, rather than simply performing SMT solving on a bounded model.

3. USAGE SCENARIOS

In this section we describe two usage scenarios of using SMACK that illustrate its precision and versatility.

3.1 Type Unsafe and Bitwise Operations

SMACK allows users to verify C/C++ programs with bitwise operations and low-level type-unsafe memory accesses. This makes SMACK a competitive tool for verifying real-world low-level programs such as device drivers. Figure 2 gives such an example program that we successfully verified using SMACK. Note that users can specify the intended properties of a program as assertions. To verify this program, run SMACK with the `bit-precise` command line option provided.

The program consists of two parts delimited with the empty line 23. The code segment of the first part (lines 20–22) is legal since a pointer to *char* type can alias with pointers to any other types according to the strict aliasing rules. However, dereferencing pointer variable `p` (line 21) is not type safe because only the least significant byte is written to. The assertion on line 22 should hold because the remaining bytes of `x.j` are all zero, which is reported by SMACK. Note, however, that the verifier which has an imprecise memory model will fail to handle this case which occurs frequently in device drivers. The second part of the program calculates the absolute value of an arbitrary integer using bitwise operations (thus branch-free) and asserts the result is not negative (lines 25–27). Note that the return type of the `absolute` function is *unsigned*, while its return value is assigned to a variable of a *signed* integer type. Therefore, if `x.i` is assigned `INT_MIN` before the `absolute` function is called, the return value (`-INT_MIN`) has the same

```
 1  struct a {
 2    int i;
 3    int j;
 4  };
 5
 6  // implemented with bitwise operations
 7  unsigned absolute(int value) {
 8    unsigned int r;
 9    int mask;
10    mask =
11      value >> sizeof(int)*CHAR_BIT - 1;
12    r = (value + mask) ^ mask;
13    return r;
14  }
15
16  int main(void) {
17    struct a x = {-10, 20};
18
19    // valid yet type unsafe cast
20    char *p = (char *)(&(x.j));
21    *p = 1;
22    assert(x.j == 1);
23
24    // assign nondeterministic value
25    x.i = __VERIFIER_nondet_int();
26    x.i = absolute(x.i);
27    assert(x.i >= 0);
28  }
```

Figure 2: Type Unsafe and Bitwise Operations

```
1  int main(void) {
2    long x = __VERIFIER_nondet_long();
3    long y, z = 0;
4    assume(x > 100);
5    for (y = 0; y < x; ++y) z++;
6    assert(z != x);
7  }
```

Figure 3: Simple Program with an Unbounded Loop

bit-representation as `INT_MIN`, which is the minimum integer value if it is interpreted as a signed integer. Though there are potential integer overflows in the `absolute` function, which are undefined by the C standard, they are applied on purpose by the programmer, and thus modeled by SMACK. Hence, SMACK reports that the assertion does not hold, and the counterexample is the case described above.

3.2 Handling of Loops

Programs containing loops can be verified with SMACK using Boogie, Corral, or Duality as back-end verifiers. These verification tools can statically check user-specified assertions under certain unrolling depths. In addition, SMACK can leverage Duality to *prove* the partial correctness of a program with unbounded loops using techniques based on interpolation.

Consider the program in Figure 3, which increments `z` until its value reaches that of `x`. The terminating condition of the loop on line 5 contains a nondeterministic positive variable `x` that is larger than 100; thus, the assertion fails only if the loop is unrolled 100 times, which creates issues for bounded verifiers. For example, if SMACK is invoked with Corral using an unrolling depth of 50 (specified with the `unroll` command line option), it does not report an assertion violation (false negative). This occurs because Corral attempts to explore loop unrollings only up to the maximum value specified by the user (50 in this case). Therefore, Corral must be given an unroll bound that is large enough to expose the bug (100 in this example), which is often not trivial.

However, by leveraging the power of SMACK to easily switch between back-end verifiers, we can invoke SMACK with Duality (instead of the default Corral) using the `verifier` command line option. Now, SMACK will report the bug and provide the user with the error trace. If we change the assertion to `assert(z == x)`, Duality generates a proof that the assertion holds.

4. EMPIRICAL EVALUATION

SMACK participated in the International Competition on Software Verification (SV-COMP [15]) in 2015 and 2016. It won numerous medals, and it is among the very few verifiers that successfully compete in the overall category. This demonstrates that SMACK, despite its modular design, has comparable performance and scalability to state-of-the-art monolithic program verifiers. Moreover, we publicly release Boogie programs generated by SMACK during SV-COMP as benchmarks, and they have been used by several emerging Boogie back-end tools, such as ICE [7].

5. AVAILABILITY AND INSTALLATION

SMACK is a free and open-source project hosted on GitHub under the standard MIT license:

https://github.com/smackers/smack

The repository is accompanied by a Wiki, which contains a wealth of information such as system requirements, installation instructions, usage instructions, and contribution guidelines. SMACK is typically easy to install using the provided build script, and currently we support Linux, Windows, and Mac OS X environments. In addition, we provide a portable installation environment by leveraging the Vagrant system for creating virtual development environments.

6. RELATED WORK

Even though several existing verification tools leverage the LLVM compiler infrastructure [13, 9], they do not leverage a popular IVL like Boogie, and therefore do not enable the swapping of verification back-ends. As a result, many of the existing tools require significant engineering work to add new features. SMACK is not hindered in this way since cutting-edge verifiers can be easily swapped in, thus enabling SMACK to deliver efficient verification to a wider range of programs. For instance, Seahorn [9] and LLBMC [13] do not support verification of concurrent programs because that feature is not supported by their back-end verification algorithms. We enabled support for concurrency in SMACK with just a little bit of effort since we have access to back-end verifiers that support this feature [5].

HAVOC [4] translates C programs into Boogie IVL programs like SMACK. However, the memory model used in HAVOC is only word-precise, and thus its applicability is limited to type-safe programs without low-level operations.

7. CONCLUSIONS

In this paper, we demonstrate how a modular approach to SMT-based software verification tools can reduce prototyping effort, and rapidly deliver the latest software verification innovations to the software engineering community. The SMACK toolchain is already seeing adoption from other researchers, such as the authors of ICE [7]. In addition, empirical results demonstrate that modularization of the traditional monolithic tool architecture does not incur a performance or precision penalty. Considered collectively, these benefits highlight SMACK as an innovation platform for software verification tools and techniques, to the benefit of the software verification and engineering communities.

8. REFERENCES

[1] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier research platform. In *CAV*, pages 119–122, 2010.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2006.

[3] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *SMT*, 2010.

[4] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS*, pages 19–33, 2007.

[5] P. Deligiannis, A. F. Donaldson, and Z. Rakamarić. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *ASE*, pages 166–177, 2015.

[6] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[7] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, pages 69–87, 2014.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166, 2008.

[9] A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A framework for verifying C programs (competition contribution). In *TACAS*, pages 447–450. 2015.

[10] A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV*, pages 427–443, 2012.

[11] The LLVM compiler infrastructure. http://llvm.org/.

[12] K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[13] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, pages 146–161, 2012.

[14] Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, pages 106–113, 2014.

[15] International competition on software verification (SV-COMP). http://sv-comp.sosy-lab.org.