SMACK: Decoupling Source Language Details from Verifier Implementations*

Zvonimir Rakamarić[1] and Michael Emmi[2]

[1] School of Computing, University of Utah, USA
zvonimir@cs.utah.edu
[2] IMDEA Software Institute, Spain
michael.emmi@imdea.org

Abstract. A major obstacle to putting software verification research into practice is the high cost of developing the infrastructure enabling the application of verification algorithms to actual production code, in all of its complexity. Handling an entire programming language is a huge endeavor that few researchers are willing to undertake; even fewer could invest the effort to implement a verification algorithm for many source languages. To decouple the implementations of verification algorithms from the details of source languages, and enable rapid prototyping on production code, we have developed SMACK. At its core, SMACK is a translator from the LLVM intermediate representation (IR) into the Boogie intermediate verification language (IVL). Sourcing LLVM exploits an increasing number of compiler front ends, optimizations, and analyses. Targeting Boogie exploits a canonical platform which simplifies the implementation of algorithms for verification, model checking, and abstract interpretation. Our initial experience in verifying C-language programs is encouraging: SMACK is competitive in SV-COMP benchmarks, is able to translate large programs (100 KLOC), and is being used in several verification research prototypes.

1 Introduction

A major obstacle to putting software verification research into practice is the high cost of developing the infrastructure enabling the application of verification algorithms to actual production code, in all of its complexity. Each high-level programming language brings a diverse assortment of statements and expressions with varying semantics. Handling an entire language is a huge effort which few researchers are willing to undertake; even fewer could invest the effort required to implement their verification algorithms for multiple source languages.

To address this problem, we introduce SMACK: a translator from the LLVM compiler's popular *intermediate representation* (IR) [27,24] into the Boogie *intermediate verification language* (IVL) [19,26]. SMACK's primary function is to precisely and efficiently translate the rich set of LLVM-IR features, including dynamic memory allocation and pointer arithmetic, to the comparatively-simple

---

* Partially supported by NSF award CCF 1346756.

Boogie IVL, which does not include such features. SMACK thus promotes the development of verification algorithms on simple IVLs, effectively decoupling the implementations of verification algorithms from the details of source languages, and enabling rapid prototyping on production code. Sourcing LLVM IR exploits a rapidly-growing frontier of LLVM frontends, encompassing a diverse set of languages including C/C++, Java, Haskell, Erlang, Python, Ruby, Ada, and Fortran. In addition, SMACK benefits from code simplifications made by LLVM's optimizer, including constant propagation and dead-code elimination, as well as readily-available analyses, including LLVM's pointer analyses. Targeting Boogie IVL exploits a canonical platform which simplifies the implementation of verification algorithms due to Boogie's minimal syntax and mathematically-focused expression language, which is easily rendered into the satisfiability modulo theories (SMT) format of automated theorem provers [6]. By embracing Boogie IVL as a canonical program representation, SMACK not only simplifies the development of program verification technology, but also fosters the development of interoperable technology in which verification backends can be easily swapped.

Our initial experience in verifying C-language programs with SMACK, using Microsoft Research's Boogie and Corral [23] as backends, is encouraging. SMACK has eased the development of our research prototypes by enabling IVL-level, rather than C-level or LLVM-level, implementations. In doing so, it appears that our approach does not significantly compromise performance, as SMACK (with Boogie and Corral backends) is competitive on SV-COMP [33] benchmarks. Furthermore, SMACK translates large, full-featured programs — including the entire Contiki operating system [15], 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.

While our experience with SMACK has thus far been centered on SMT-based *bounded verification*, i.e., validation of program assertions up to recursion-depth and loop-unroll bounds, our prior experience [10,30] suggests that SMACK can also be applied straightforwardly to *deductive verification*, i.e., validation of assertions in programs adequately annotated with loop invariants and procedure pre- and post-conditions. While in theory SMACK is equally applicable for fully automatic unbounded verification methods (e.g., based on computing fixed points), in practice such applications may require powerful reasoning engines capable of generating quantified invariants over the unbounded maps which SMACK uses to model dynamically-allocated memory; it remains to be seen whether such applications are feasible.

SMACK is an open source project available on GitHub[3] implemented in roughly 4K lines of C++ code, and is integrated into the `rise4fun` website.[4] Currently, SMACK is supported on Linux, OSX, and Windows, and is used in several projects, including Microsoft Research's Q program verifier.[5]

---

[3] `http://github.com/smackers/smack`

[4] `http://rise4fun.com/SMACK`

[5] `http://research.microsoft.com/en-us/projects/verifierq`

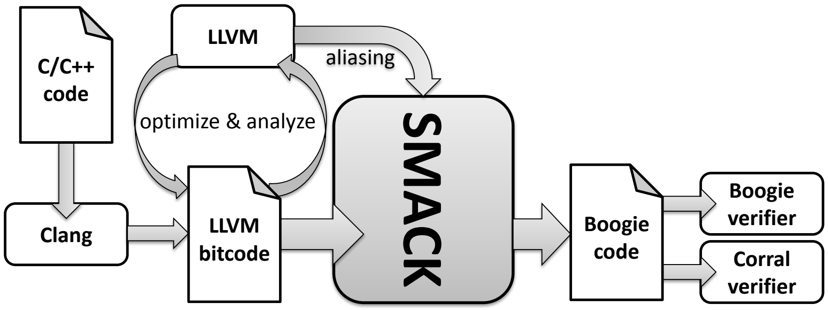

Fig. 1. Design of the SMACK modular software verification ecosystem.

Related Work. Automatic verification using automated theorem provers, and in particular SMT solvers, is an active area of research. Many tools are available with various capabilities, features, and trade-offs, including Caduceus [21], Calysto [4], Cascade [34], CBMC [13], CPAchecker [7], ESBMC [16], Frama-C [17], GraVy [2], HAVOC [10], Joogie [3], KLEE [9], LLBMC [28], SATABS [12], Symbolic PathFinder [29], TASS [32], UFO [1], and VCC [14]. Our SMACK effort stands alone, since none of these tools combine the language independence of leveraging a popular IR with the ease of implementation provided by IVLs. Furthermore, SMACK has been designed to accommodate a diverse set of extensions, from supporting new source language features to generating alternate IVL encodings.

2 Translation from LLVM IR to Boogie IVL

We have developed SMACK as one essential component of the software verification ecosystem depicted in Fig. 1. Currently, the other components include the Clang compiler [11], the LLVM compiler infrastructure [27,24], and the Boogie [19,26] and Corral [23] verification engines. Beginning from a program written in C/C++, we use Clang to emit LLVM bitcode in an intermediate representation (IR) used by LLVM. LLVM IR is a typed, static single assignment (SSA), and platform-independent assembly language, and an ideal representation for LLVM's code optimizer/analyzer.

Following LLVM code optimizations, such as constant propagation and dead-code elimination, SMACK translates LLVM bitcode to code in Boogie's intermediate verification language (IVL). Boogie IVL is typed, imperative, and procedural, includes a rich mathematical expression language, and is an ideal representation for program verifiers. The Boogie programs which SMACK generates are essentially control-flow graphs with very few statements — they have `goto`, assignment, procedure call & return, and `assume`/`assert` statements — which manipulate global and procedure-local variables over very few types — only integers and maps from integers to integers. For the most part, SMACK's translation is tight, in the sense that LLVM data and instructions correspond closely to Boogie data and instructions, modulo representing fixed-width integers with mathematical integers.[6]

While there are many syntactic differences between LLVM IR and Boogie IVL, a key fundamental difference which SMACK addresses is memory representation: while LLVM IR performs dynamic allocation on the memory heap, programs in Boogie IVL have only a fixed number of global variables, albeit over unbounded types including mathematical integers and maps (i.e., arrays). Although in theory the entire heap could be represented with one single map, experience indicates that this strategy is not efficient; a verifier which represents map-type variables with array-theory expressions would suffer as the map is updated across many addresses. Instead, SMACK uses static analyses in LLVM to infer a set of memory regions which are *disjoint*, in the sense that two distinct regions are never accessed by the same program expression; each region of the heap is then given its own map, and each heap access translates to an expression using the accessed region's map [31]. SMACK's modular design facilitates the implementation of alternate memory models by, for example, redefining: (1) the Boogie-code implementations of `malloc` and `free` to describe alternate allocation policies (which does not require recompiling SMACK), or (2) the translation of **`load`** and **`store`** operations to model heap accesses at byte-sized granularity (currently requires recompilation).

SMACK passes the resulting Boogie-IVL program to either the Boogie or Corral verifier; both function by generating verification conditions [5] which are discharged using satisfiability modulo theories (SMT) solvers, such as Z3 [18].

3 An Example Translation

We illustrate our verification workflow step-by-step on the program listed in Fig. 2. The C program (top left) is first compiled with Clang into the LLVM IR program shown on the right. In the process, calls to `malloc` in C are compiled into the respective invocations in the LLVM IR. Structure field accesses are compiled into a combination of **`getelementptr`** and **`load`**/**`store`** instructions, where **`getelementptr`** performs the structure field address computation that is subsequently accessed using **`load`**/**`store`**. Note that while the LLVM IR is a simple representation, it does include dynamic memory allocation, pointer arithmetic, and complex data types — none of which are included in the Boogie IVL.

From the LLVM IR program, SMACK generates the Boogie IVL program by leveraging LLVM's static *data structure analysis* (DSA) [25] to split memory into a set of disjoint regions so that pointers to two distinct regions can never alias [31]. Each such region is then statically assigned its own map, and each memory access translates to an expression using the accessed region's map. In Fig. 2, based on the fact that DSA accurately reported that LLVM IR pointer variables `%5` and {`%6`, `%7`} cannot alias, SMACK statically introduced memory

---

[6] While our current implementation uses *unbounded* integers and maps thereof, in principle we could also use bit-vectors to model, e.g., 32-bit integers precisely.

```
// original C code
typedef struct { int f; int g; } S;

void main() {
  S *x = malloc(sizeof(S));
  S *y = malloc(sizeof(S));
  x->f = 1;
  y->f = 2;
  y->g = 3;
  assert(x->f == 1);
}
```

```
// Boogie IVL code from SMACK
var $M.0, $M.1: [int] int;

procedure main() {
  var $p, $p1, $p2, .., $p6: int;
$bb0:
  call $p := $malloc(8);
  call $p1 := $malloc(8);
  $p2 := $pa($pa($p, 0, 8), 0, 1);
  $M.0[$p2] := 1;
  $p3 := $pa($pa($p1, 0, 8), 0, 1);
  $M.1[$p3] := 2;
  $p4 := $pa($pa($p1, 0, 8), 4, 1);
  $M.1[$p4] := 3;
  $p5 := $pa($pa($p, 0, 8), 0, 1);
  $p6 := $M.0[$p5];
  assert($p6 == 1);
  return;
}
```

```
// LLVM IR code from Clang/LLVM
define void @main() #0 {
  %1 = call i8* @malloc(i64 8)
  %2 = bitcast i8* %1 to %struct.S*
  %3 = call i8* @malloc(i64 8)
  %4 = bitcast i8* %3 to %struct.S*
  %5 = getelementptr inbounds
       %struct.S* %2, i32 0, i32 0
  store i32 1, i32* %5, align 4
  %6 = getelementptr inbounds
       %struct.S* %4, i32 0, i32 0
  store i32 2, i32* %6, align 4
  %7 = getelementptr inbounds
       %struct.S* %4, i32 0, i32 1
  store i32 3, i32* %7, align 4
  %8 = getelementptr inbounds
       %struct.S* %2, i32 0, i32 0
  %9 = load i32* %8, align 4
  %10 = icmp eq i32 %9, 1
  ... assertion omitted ...
  ret void
}
```

Fig. 2. An example program in C, along with its LLVM IR and Boogie IVL translations.

maps `$M.0` and `$M.1` in Boogie code, respectively. While not shown, our translation defines the `$pa` function to model **`getelementptr`**, and the `$malloc` procedure to model memory allocation, by keeping track precisely of allocated and unallocated sections of memory. The **`load`** and **`store`** instructions are then translated as accesses into the appropriate region's map. Finally, assertions in C are ultimately translated into Boogie assertions, and checked using our backend verifiers.

4 Our Experience with SMACK

Our experience in using SMACK for developing research prototype verification tools has benefited from increased productivity without prohibitive performance sacrifices. One example is the `c2s` project[7] which implements various concurrent-to-sequential Boogie code translations — so called "sequentializations" — for delay-bounded verification [20], and which has been used in several of the authors' research projects. The authors of the CSeq tool [22], which implements a related sequentialization directly in C code rather than in a simple IVL, admit a telling limitation:

---

[7] `http://github.com/michael-emmi/c2s`

Table 1. Comparison of SMACK, CPAchecker, CBMC, and UFO on SV-COMP benchmarks. #B is the number of benchmarks (both correct and buggy) in a suite. No-Reuse and Reuse correspond to two distinct memory models currently provided by SMACK. Experiments were performed on an Intel Core i7-3930K 3.20 GHz machine with 32 GB of memory running Ubuntu 12.04. All runtimes are in seconds.

Benchmark Suite	#B	KLOC	SMACK				SV-COMP 2014		
			No-Reuse		Reuse				
			Boogie	Corral	Boogie	Corral	CPAchecker	CBMC	UFO
locks	13	2.3	9.1	9.3	9.0	9.3	365.1	1.4	2.9
ntdrivers-simpl	10	18.1	12.3	85.7	12.3	86.4	43.5	4.6	3.4

> "CSeq does not support [heap-allocated memory] yet. Lifting these restrictions, and in particular supporting dynamic memory . . . will require significant efforts."

In contrast, the Boogie IVL-based `c2s` tool was simple to implement, and has been used for the analysis of intricate C-language concurrent data structure implementations which make extensive use of dynamic memory allocation [8].

Despite the threat to performance incurred by separating backend verifiers from source languages, SMACK-based tools are competitive with state-of-the-art verifiers. While a truly-meaningful comparison is difficult, since different verifiers generally provide different guarantees, Table 1 makes an attempt, comparing SMACK with 3 competitive verifiers (CPAchecker [7], CBMC [13], UFO [1]) on 2 benchmark suites from the SV-COMP [33] annual software verification competition. Both suites contain both correct and buggy benchmarks, and all verifiers categorize them correctly: neither false positives nor negatives are reported.[8]

Note that since these are preliminary results mixing tools aimed at bug-finding (SMACK, CBMC) with those aimed at verification (CPAchecker, UFO), a direct comparison of runtimes is somewhat unfair. However, the table does illustrate that even though SMACK has not been optimized for SV-COMP benchmarks — thus far we have spent minimal effort in optimization — its performance is comparable to established verifiers which regularly participate in SV-COMP. As future work, we plan to expand these preliminary results with more benchmarks, and enroll SMACK in a future SV-COMP.

As expected, the current version of SMACK does have some limitations. First, integer datatypes are modeled with unbounded mathematical integers; this limitation can be lifted by leveraging Boogie's support for bit-vectors. Floating point datatypes pose a more serious challenge, as they are not widely supported by current software verifiers and automated theorem provers. Finally, SMACK currently precisely handles word-aligned memory accesses only.

---

[8] To make our results readily reproducible, we created a virtual machine profile in the Apt testbed facility containing all used tools, scripts, and benchmarks. It is available at `https://www.aptlab.net/p/fmr/smack-cav2014`.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *International Conference on Computer Aided Verification (CAV)*, pages 672–678, 2012.
2. S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NASA Formal Methods Symposium (NFM)*, pages 313–327, 2014.
3. S. Arlt, P. Rümmer, and M. Schäf. Joogie: From Java through Jimple to Boogie. In *ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis (SOAP)*, pages 3–8, 2013.
4. D. Babić and A. J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering (ICSE)*, pages 211–220, 2008.
5. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 82–87, 2005.
6. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 2010.
7. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification (CAV)*, pages 184–190, 2011.
8. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *European Symposium on Programming (ESOP)*, pages 290–309, 2013.
9. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.
10. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 19–33, 2007.
11. Clang: A C language family frontend for LLVM. `http://clang.llvm.org`.
12. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 570–574, 2005.
13. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176, 2004.
14. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42, 2009.
15. Contiki: The open source OS for the Internet of things. `http://www.contiki-os.org`.
16. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 137–148, 2009.
17. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pages 233–247, 2012.

18. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
19. 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.
20. M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 411–422, 2011.
21. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 15–29, 2004.
22. B. Fischer, O. Inverso, and G. Parlato. Cseq: A sequentialization tool for C (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 616–618, 2013.
23. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *International Conference on Computer Aided Verification (CAV)*, pages 427–443, 2012.
24. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.
25. C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.
26. K. R. M. Leino. This is Boogie 2, 2008.
27. The LLVM compiler infrastructure. `http://llvm.org`.
28. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 146–161, 2012.
29. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008.
30. Z. Rakamarić and A. J. Hu. Automatic inference of frame axioms using static analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 89–98, 2008.
31. Z. Rakamarić and A. J. Hu. A scalable memory model for low-level code. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.
32. S. F. Siegel and T. K. Zirkel. TASS: The toolkit for accurate scientific software. *Mathematics in Computer Science*, 5(4):395–426, 2011.
33. International competition on software verification (SV-COMP). `http://sv-comp.sosy-lab.org`.
34. W. Wang, C. Barrett, and T. Wies. Cascade 2.0. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 142–160, 2014.