

# A Glimpse of a Verifying C Compiler

## – Extended Abstract –

Wolfram Schulte<sup>1</sup>, Songtao Xia<sup>1</sup>, Jan Smans<sup>2</sup>, and Frank Piessens<sup>2</sup>

<sup>1</sup> Microsoft Research, Redmond, USA

<sup>2</sup> Katholieke Universiteit Leuven, Belgium

The goal of the Verifying C Compiler project is to bring design by contract to C. More specifically, we are developing a verifying compiler, code name *vcc*, that takes annotated C programs, generates logical verification conditions from them and passes those verification conditions on to an automatic theorem prover to either prove the correctness of the program or find errors in it.

*C Intricacies.* The *vcc* compiler is designed to support the verification of operating system code. As a consequence it does not only handle the type safe subset of C, but also deals with pointer arithmetic, reinterpretation of data and volatile data access. This flexibility is for example needed to verify low level system code like memory allocators, where data is interpreted in different ways by different parts of the system, or to verify algorithms implemented over polymorphic compare and swap operations.

The *vcc* compiler uses different background axiomatizations to abstract from C's *implementation defined behavior*. For example the size of character type, or how integers are implemented (typically two's complement) is dealt with not by the translation to verification conditions but rather by their interpretation.

The *vcc* compiler rejects C programs with *unspecified behavior*. For example, non-pure functions cannot directly be used as arguments to  $n$ -ary function applications (where  $n > 1$ ), since C does not specify the evaluation order for function arguments.

In addition to verifying developer-defined functional properties, the *vcc* compiler also checks for *undefined behavior*, i.e. null pointer dereferences, dangling pointers, double frees, division by zero, over and underflow. It does so by automatically inserting additional assertions into the verification conditions, which precede the translation of the partial operation. Generating these assertions can selectively be turned on and off to allow the verification of code that uses these features (for example overflow).

*C Memory Model.* The central idea behind *vcc*'s memory model is the subdivision of memory into several disjoint regions. Each region has a fixed size, which is determined at the time it is allocated, and is uniquely identified by a reference. A pointer consists of reference and a byte offset in the corresponding region. So *vcc*'s memory model is in principle byte-oriented.

The memory is represented as a partial updateable map *mem*, mapping (pointer, integer) pairs to values. The integer represents the number of bytes to access starting at the given pointer. So *mem*( $p, n$ ) denotes a value of size  $n$  stored at  $p \dots p + n - 1$ . A value is either a pointer or a bitvector of up to 64 bits (depending on the machine). Note that despite the fact that our memory is byte

oriented, program values such as pointers and shorts, are not stored one byte at a time; instead these values are stored as atomic units. Only when values are not used in a type safe fashion, they are broken down into bytes, and for bitfields even into their bits.

The main advantage of this model is that the theorem prover can work at the proper level of abstraction, i.e. on different sized values when it verifies type safe C code, and on bytes and bits when performing casts and bit-level operations.

Non-heap allocated objects, that is parameters, local and global variables are also stored in the memory. For instance, local variables are modeled by allocating the necessary memory at the start of the method, and freeing the same memory at the end of the method.

*Inspiration.* The *vcc* compiler is Spec#'s little brother [2]. From Spec# we have adopted the idea of flat memory model, ownership and all of Spec's verification machinery [1]. Havoc [3], another C verifier developed at MSR, has inspired the way we handle inductive properties. The architecture of *vcc* is similar to the architecture of Caduceus [5] and Escher's C compiler [4]; among other, we adopted their idea of allowing the user to add mathematical theories to C. Our mathematical lists are influenced by KeY-C [6]. The memory model is similar to the embedding of C in HOL, performed as part of the L4 kernel verification [8]. The latter builds on the embedding of C0 in HOL [7].

*Status.* At the time of writing, the *vcc* compiler prototype only handles sequential C programs, with address arithmetic, memcpy, bitfields, unions and casts. The *vcc* compiler also allows developers to add mathematical theories as well as ghost variables to the C source.

We are currently developing a programming methodology for C that supports (1) inductive properties for pointer structures, (2) abstraction and (3) corresponding frame conditions, and (4) multi-threading, which is prevalent in system software.

*Acknowledgments* The authors would like to thank Rustan Leino, Peli de Halleux, Ernie Cohen, Herman Venter and Nikolaj Bjorner for useful comments, inspiring discussions and for help on implementing the Verifying C Compiler. Jan Smans is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

## References

1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of*

- Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
3. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, Lecture Notes in Computer Science. Springer, March 2007.
  4. David Crocker and Judith Carlton. Verification of c programs using automated reasoning. In *Submitted to SEFM 2007*, 2007.
  5. Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.
  6. Oleg Mürk, Daniel Larsson, and Reiner Hähnle. Key-c: A tool for verification of c programs. In *21st Conference on Automated Deduction (CADE)*, 2007.
  7. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
  8. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.