Cerberus-BMC: A Principled Reference Semantics and Exploration Tool for Concurrent and Sequential C

Stella Lau[1,2(✉)], Victor B. F. Gomes[2], Kayvan Memarian[2], Jean Pichon-Pharabod[2], and Peter Sewell[2]

[1] MIT, Cambridge, USA
stellal@mit.edu
[2] University of Cambridge, Cambridge, UK
{victor.gomes,kayvan.memarian,
jean.pichon-pharabod,peter.sewell}@cl.cam.ac.uk

Abstract. C remains central to our infrastructure, making verification of C code an essential and much-researched topic, but the semantics of C is remarkably complex, and important aspects of it are still unsettled, leaving programmers and verification tool builders on shaky ground. This paper describes a tool, Cerberus-BMC, that for the first time provides a principled reference semantics that simultaneously supports (1) a choice of concurrency memory model (including substantial fragments of the C11, RC11, and Linux kernel memory models), (2) a modern memory object model, and (3) a well-validated thread-local semantics for a large fragment of the language. The tool should be useful for C programmers, compiler writers, verification tool builders, and members of the C/C++ standards committees.

1 Introduction

C remains central to our infrastructure, widely used for security-critical components of hypervisors, operating systems, language runtimes, and embedded systems. This has prompted much research on the verification of C code, but the semantics of C is remarkably complex, and important aspects of it are still unsettled, leaving programmers and verification tool builders on shaky ground. Here we are concerned with three aspects:

1. The Concurrency Memory Model. The 2011 versions of the ISO C++ and C standards adopted a new concurrency model [3,12,13], formalised during the development process [11], but the model is still in flux: various fixes have been found to be necessary [9,14,26]; the model still suffers from the "thin-air problem" [10,15,35]; and Linux kernel C code uses a different model, itself recently partially formalised [7].

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 387–397, 2019.
https://doi.org/10.1007/978-3-030-25540-4_22

2. The Memory Object Model. A priori, one might imagine C follows one of two language-design extremes: a concrete byte-array model with pointers that are simply machine words, or an abstract model with pointers combining abstract block IDs and structured offsets. In fact C is neither of these: it permits casts between pointer and integer types, and manipulation of their byte representations, to support low-level systems programming, but, while at runtime a C pointer will typically just be a machine word, compiler analyses and optimisations reason about abstract notions of the provenance of pointers [27,29,31]. This is a subject of active discussion in the ISO C and C++ committees and in compiler development communities.

3. The Thread-Local Sequential Semantics. Here, there are many aspects, e.g. the loosely specified evaluation order, the semantics of integer promotions, many kinds of undefined behaviour, and so on, that are (given an expert reading) reasonably well-defined in the standard, but that are nonetheless very complex and widely misunderstood. The standard, being just a prose document, is not *executable as a test oracle*; it is not a reference semantics usable for exploration or automated testing.

Each of these is challenging in isolation, but there are also many subtle interactions between them. For example, between (1) and (3), the pre-C11 ISO standard text was in terms of sequential stepwise execution of an (informally specified) abstract machine, while the C11 concurrency model is expressed as a predicate over complete candidate executions, and the two have never been fully reconciled – e.g. in the standard's treatment of object lifetimes. Then there are fundamental issues in combining the ISO treatment of undefined behaviour with that axiomatic-concurrency-model style [10, §7]. Between (1) and (2), one has to ask about the relationships between the definition of data race and the treatment of uninitialised memory and padding. Between (2) and (3), there are many choices for what the C memory object model should be, and how it should be integrated with the standard, which are currently under debate. Between all three one has to consider the relationships between uninitialised and thin-air values and the ISO notions of unspecified values and trap representations. These are all open questions in what the C semantics and ISO standard are (or should be). We do not solve them here, but we provide a necessary starting point: a tool embodying a precise reference semantics that lets one explore examples and debate the alternatives.

We describe a tool, Cerberus-BMC, that for the first time lets one explore the allowed behaviours of C test programs that involve all three of the above. It is available via a web interface at http://cerberus.cl.cam.ac.uk/bmc.html.

For (1), Cerberus-BMC is parameterised on an axiomatic memory concurrency model: it reads in a definition of the model in a Herd-like format [6], and so can be instantiated with (substantial fragments of) either the C11 [3,9,12–14], RC11 [26], or Linux kernel [7] memory models. The model can be edited in the web interface. Then the user can load (or edit in the web interface) a small C program. The tool first applies the Cerberus compositional translation (or elab-

oration) into a simple Core language, as in [29,31]; this elaboration addresses (3) by making many of the thread-local subtleties of C explicit, including the loose specification of evaluation order, arithmetic conversions, implementation-defined behaviour, and many kinds of undefined behaviour. Core computation is simply over mathematical integers, with explicit memory actions to interface with the concurrency and memory object models. However, there is a mismatch between the axiomatic style of the concurrency models for C (expressed as predicates on arbitrary candidate executions) with the operational style of the previous thread-local operational semantics for Core. We address this by replacing the latter with a new translation from Core into SMT problems. This is integrated with the concurrency model, also translated into SMT, following the ideas of [5]. These are furthermore integrated with an SMT version of parts of the PNVI (provenance-not-via-integers) memory object model of [29], the basis for ongoing work within the ISO WG14 C standards committee, addressing (2). The resulting SMT problems are passed to Z3 [32]. The web interface then provides a graphical view of the allowed concurrent executions for small test programs.

The Cerberus-BMC tool should be useful for programmers, compiler writers, verification tool builders, and members of the C/C++ standards committees. We emphasise that it is intended as an executable reference semantics for small test programs, not itself as a verification tool that can be applied to larger bodies of C: we have focussed on making it transparently based on principled semantics for all three aspects, without the complexities needed for a high-performance verification tool. But it should aid the construction of such.

Caveats and Limitations. Cerberus-BMC covers many features of 1–3, but far from all. With respect to the concurrency memory model, we support substantial fragments of the C11, RC11, and Linux kernel memory models. We omit locks and the (deprecated) C11/RC11 consume accesses. We only cover compare-exchange read-modify-write operations, and the fragment of RCU restricted to `read_rcu_lock()`, `read_rcu_unlock()`, and `synchronize_rcu()` used in a linear way, without control-flow-dependent calls to RCU, and without nesting.

With respect to the memory object model, we do not currently support dynamic allocation or manipulation of byte representations (such as with `char*` pointers), and we do not address issues such as subobject provenance (an open question within WG14).

With respect to the thread semantics, our translation to SMT does not currently cover arbitrary pointer type-casting, function pointers, multi-dimensional arrays, unions, floating point, bitwise operations, and variadic functions, and only covers simple structs. In addition, we inherit the limitations of the Cerberus thread semantics as per [29].

Related Work. There is substantial prior work on tools for concurrency semantics and for C semantics, but almost none that combines the two. On the concurrency semantics side, CppMem [1,11] is a web-interface tool that computes the allowed concurrent behaviours of small tests with respect to variants (now somewhat

outdated) of the C11 model, but it does not support other concurrency models or a memory object model, and it supports only a small fragment of C. Herd [6,8] is a command-line tool that computes the allowed concurrent behaviours of small tests with respect to arbitrary axiomatic concurrency models expressed in its `cat` language, but without a memory object model and for tests which essentially just comprise memory events, without a C semantics. MemAlloy [38] and MemSynth [16] also support reasoning about axiomatic concurrency models, but again not integrated with a C language semantics.

On the C semantics side, several projects address sequential C semantics but without concurrency. We build here on Cerberus [28,29,31], a web-interface tool that computes the allowed behaviours (interactively or exhaustively) for moderate-sized tests in a substantial fragment of sequential C, incorporating various memory object models (an early version supported Nienhuis's operational model for C11 concurrency [33], but that is no longer integrated). KCC and RV-Match [19,21,22] provide a command-line semantics tool for a substantial fragment of C, again without concurrency. Krebbers gives a Coq semantics for a somewhat smaller fragment [24].

Then there is another large body of work on model-checking tools for sequential and concurrent C. These are all optimised for model-checking performance, in contrast to the Cerberus-BMC emphasis on expressing the semantic envelope of allowed behaviour as clearly as we can (and, where possible, closely linked to the ISO standard). The former include tis-interpreter [18,36], CBMC [17,25], and ESBMC [20]. On the concurrent side, as already mentioned, we build on the approach of [5], which integrated various hardware memory concurrency models with CBMC. CDSChecker [34] supports something like the C/C++11 concurrency model, but subject to various limitations [34, §1.3]. It is implemented using a dynamically-linked shared library for the C and C++ atomic types, so implicitly adopts the C semantic choices of whichever compiler is used. RCMC [23], supports memory models that do not exhibit Load Buffering (LB), for an idealised thread-local language. Nidhugg [4] supports only hardware memory models: SC, TSO, PSO, and versions of POWER and ARM.

2 Examples

We now illustrate some of what Cerberus-BMC can do, by example.

Concurrency Models. First, for C11 concurrency, Fig. 1 shows a screenshot for a classic message-passing test, with non-atomic writes and reads of `x`, synchronised with release/acquire writes and reads of `y`. The test uses an explicit parallel composition, written `{-{...|||...}-}`, to avoid the noise from the extra memory actions in `pthread_create`. The consistent race-free UB-free execution on the right shows the synchronisation working correctly: after the `i` read-acquire of `y=1`, the `l` non-atomic read of `x` has to read `x=1` (there are no consistent executions in which it does not). As usual in C/C++ candidate execution graphs, `rf` are reads-from edges, `sb` is sequenced-before (program order), `mo` is modification

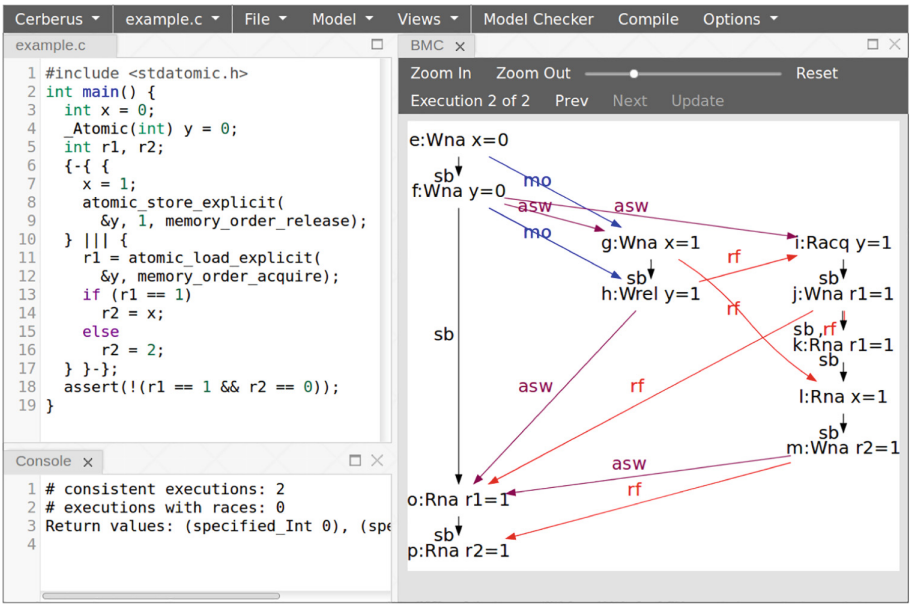


Fig. 1. Cerberus-BMC Screenshot: C11 Release/Acquire Message Passing. If the read of `y` is `1`, then the last thread has to see the write of `1` to `x`.

```
#include "linux.h"
int main() {
  int x = 0, y = 0;
  int r1, r2;
  {-{ {
    WRITE_ONCE(x, 1);
    // synchronize_rcu();
    WRITE_ONCE(y, 1);
  } ||| {
    rcu_read_lock();
    r1 = READ_ONCE(x);
    r2 = READ_ONCE(y);
    rcu_read_unlock();
  } }-}
  assert(!(r1==0&&r2==1));
}
```

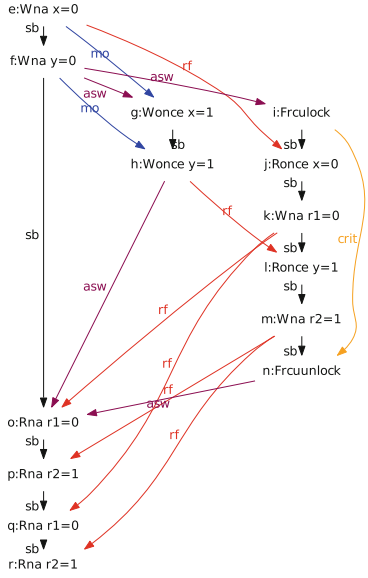


Fig. 2. Linux kernel memory model RCU lock. Without `synchronize_rcu()`, the reads of `x` and `y` can see `0` and `1` (as shown), even though they are enclosed in an RCU lock. With synchronization, after reading `x=1`, the last thread has to see `y=1`.

order (the coherence order between atomic writes to the same address), and asw is additional-synchronised-with, between parent and child threads and vice versa. Read and write events (R/W) are annotated na for non-atomic and rel/acq for release/acquire.

For the Linux kernel memory model, the example in Fig. 2 shows an RCU (read-copy-update) synchronisation.

Memory Object Model. The example below illustrates a case where one cannot assume that C has a concrete memory object model: pointer provenance matters. In some C implementations, x and y will happen to be allocated adjacent (the __BMC_ASSUME restricts attention to those executions). Then &x+1 will have the same numeric address as &y, but the write *p=11 is undefined behaviour rather than a write to y. This was informally described in the 2004 ISO WG14 C standards committee response to Defect Report 260 [37], but has never been incorporated into the standard itself. Cerberus-BMC correctly reports `UB found: source.c:8:5-7, UB043_indirection_invalid_value` following the PNVI (provenance-not-via-integers) memory object model of [29].

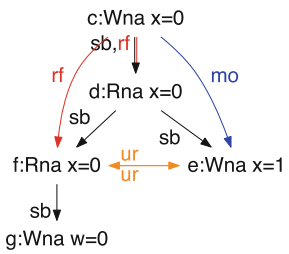
```
#include <stdint.h>
int x = 1, y = 2;
int main() {
  int *p = &x + 1;
  int *q = &y;
  __BMC_ASSUME((intptr_t)p==(intptr_t)q);
  if ((intptr_t)p==(intptr_t)q)
    *p = 11; // does this have UB?
 }
```

ISO Subtleties. Turning to areas where the ISO standard is clear to experts but widely misunderstood, in the example on the right ISO leaves it implementation-defined whether **char** is signed or unsigned. In the former case, the ISO integer promotion and conversion semantics will make the equality test false, leading to a division by 0, which is undefined behaviour.

```
int main() {
 char c1 = 0xff;
 unsigned char c2 = 0xff;
 return 1 / (c1 == c2);
}
```

The example below shows the correct treatment of the ISO standard's loose specification of evaluation order, together with detection of the concurrency model's *unsequenced races* (ur in the diagram): there are write and read accesses to x that are unrelated by sequenced-before (sb), and not otherwise synchronised and hence unrelated by happens-before, which makes this program undefined behaviour.

```
int main() {
  int x=0;
  int w;
  w = x++ + x;
}
```



Treiber Stack. Finally, demonstrating the combination of all three aspects, we implemented a modified Treiber stack (the `push()` function is shown in Fig. 3) with relaxed accesses to struct fields. Although the Treiber stack is traditionally implemented by spinning on a compare-and-swap, as that can spin unboundedly, we instead use `__BMC_ASSUME` to restrict executions to those where the compare-and-swap succeed. Our tool correctly detects the different results from the concurrent relaxed-memory execution of threads concurrently executing the `push` and `pop` functions.

```
struct Node { int data; struct Node *next; };
struct Node * _Atomic T;
void push(struct Node *x, int v) {
   struct Node *t;
   x->data = v;
   t = atomic_load_explicit(&T, memory_order_relaxed);
   x->next = t;
   __BMC_ASSUME(atomic_compare_exchange_strong_explicit(&T, &t, x,
      memory_order_acq_rel, memory_order_relaxed));
}
```

Fig. 3. Treiber stack `push()`

```
1  proc main (): eff loaded integer :=
2    let strong x: pointer = create(Ivalignof('signed int'), 'signed int') in
3    let strong a_437: loaded integer = pure(Specified(1)) in
4    store('signed int', x, conv_loaded_int('signed int', a_437)) ;
5    kill(x) ;
6    (save ret_435: loaded integer (a_436: loaded integer:= Specified(0)) in
7      pure(a_436))
```

Fig. 4. Core program corresponding to `int main(){int x = 1}`. Core is essentially a typed, first-order lambda calculus with explicit memory actions such as **create** and **store** to interface with the concurrency and memory object models.

3 Implementation

After translating a C program into Core (see Fig. 4), Cerberus-BMC does a sequence of Core-to-Core rewrites in the style of bounded model checkers such as CBMC: it unwinds loops and inlines function calls (to a given bound), and renames symbols to generate an SSA-style program.

The explicit representation of memory operations in Core as first-order constructs allows the SMT translation to be easily separated into three components: the translation from Core to SMT, the memory object model constraints, and the concurrency model constraints.

1. Core to SMT. Each value in Core is represented as an SMT expression, with fresh SMT constants for memory actions such as `create` and `store` (e.g. lines `2` and `4`), the concrete values of which are constrained by the memory object and concurrency models. The elaboration of C to Core makes thread-local undefined behaviour (as opposed to undefined behaviour from concurrency or memory layout), like signed integer overflow, explicit with a primitive `undef` construct. Undefined behaviour is then encoded in SMT as reachability of `undef` expressions, that is, satisfiability of the control-flow guards up to them.

2. Memory Object Model. As in the PNVI semantics [30], Cerberus-BMC represents pointers as pairs (π, a) of a provenance π and an integer address a. The provenance of a pointer is taken into account when doing memory accesses, pointer comparisons, and casts between integer and pointer values. Our tool models address allocation nondeterminism by constraining address values based on allocations to be appropriately aligned and non-overlapping, but not constraining the addresses otherwise.

3. Concurrency Model. Cerberus-BMC statically extracts memory actions and computes an extended pre-execution containing relations such as program order. As control flow can not be statically determined, memory actions are associated with an SMT boolean guard representing the control flow conditions upon which the memory action is executed.

Cerberus-BMC reads in a model definition in a subset of the herd `cat` language large enough to express C11, RC11, and Linux, and generates a set of quantifier-free SMT expressions corresponding to the model's constraints on relations. These constraints are based on a set of "built-in" relations defined in SMT such as `rf`. Cerberus-BMC then queries Z3 to extract all the executions, displaying the load/store values and computed relations for the user.

4 Validation

We validate correctness of the three aspects of Cerberus-BMC as follows, though, as ever, additional testing would be desirable. Performance data, demonstrating practical usability, is from a MacBook Pro 2.9 GHz Intel Core i5.

For C11 and RC11 concurrency, we check on 12 classic litmus tests. For Linux kernel concurrency, we hand-translated the 9 non-RCU tests and 4 of the RCU tests of [7] into C, and automatically translated the 40 tests of [2]. Running all the non-RCU tests takes less than 5 min; the RCU tests are slower, of the order of one hour, perhaps because of the recursive definitions involved.

For the memory object model, we take the supported subset (36 tests) of the provenance semantics test suite of [29]. These single-threaded tests each run in less than a second.

For the thread-local semantics, the Cerberus pipeline to Core has previously been validated using GCC Torture, Toyota ITC, KCC, and Csmith-generated test suites [29]. We check the mapping to BMC using 50 hand-written tests and

the supported subset (400 tests) of the Toyota ITC test suite, each running in less than two minutes.

These test suites and the examples in the paper can be accessed via the CAV 2019 pop-up in the File menu of the tool.

Acknowledgments. This work was partially supported by EPSRC grant EP/K008528/1 (REMS), ERC Advanced Grant ELVER 789108, and an MIT EECS Graduate Alumni Fellowship.

References

1. CppMem: Interactive C/C++ memory model. http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/index.html
2. Litmus tests for validation LISA-language Linux-kernel memory models. https://github.com/paulmckrcu/litmus/tree/master/manual/lwn573436
3. Programming Languages — C: ISO/IEC 9899:2011 (2011). A non-final but recent version is available at http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf
4. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
5. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
6. Alglave, J., Maranget, L.: Herd7 (in the diy tool suite) (2015). http://diy.inria.fr/
7. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.S.: Frightening small children and disconcerting grown-ups: concurrency in the Linux kernel. In: Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, 24–28 March 2018, pp. 405–418 (2018). https://doi.org/10.1145/3173162.3177156
8. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: modelling, simulation, testing, and data mining for weak memory. ACM TOPLAS **36**(2), 7:1–7:74 (2014)
9. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCl. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 634–648 (2016). https://doi.org/10.1145/2837614.2837637
10. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 283–307. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_12
11. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: Proceeding POPL (2011)
12. Becker, P. (ed.): Programming Languages — C++, iSO/IEC 14882:2011 (2011). A non-final but recent version is available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf
13. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of PLDI, pp. 68–78. ACM, New York (2008)

14. Boehm, H.J., Giroux, O., Vafeiadis, V.: P0668R2: Revising the C++ memory model. ISO WG21 paper (2018). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0668r2.html
15. Boehm, H., Demsky, B.: Outlawing ghosts: avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC 2014, Edinburgh, United Kingdom, 13 June 2014, pp. 7:1–7:6 (2014). https://doi.org/10.1145/2618128.2618134
16. Bornholt, J., Torlak, E.: Synthesizing memory models from framework sketches and litmus tests. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 467–481 (2017). https://doi.org/10.1145/3062341.3062353
17. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
18. Cuoq, P., Runarvot, L., Cherepanov, A.: Detecting strict aliasing violations in the wild. In: Bouajjani, A., Monniaux, D. (eds.) VMCAI 2017. LNCS, vol. 10145, pp. 14–33. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52234-0_2
19. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: Proceedings of POPL (2012)
20. Gadelha, M.Y.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: an industrial-strength C model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, 3–7 September 2018, pp. 888–891 (2018)
21. Guth, D., Hathhorn, C., Saxena, M., Roşu, G.: RV-Match: practical semantics-based program analysis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016, Part I. LNCS, vol. 9779, pp. 447–453. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_24
22. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 336–345 (2015). https://doi.org/10.1145/2737924.2737979
23. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–17:32 (2018). https://doi.org/10.1145/3158105
24. Krebbers, R.: The C standard formalized in CoQ. Ph.D. thesis, Radboud University Nijmegen, December 2015
25. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
26. Lahav, O., Vafeiadis, V., Kang, J., Hur, C., Dreyer, D.: Repairing sequential consistency in C/C++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, 18–23 June 2017, pp. 618–632 (2017). https://doi.org/10.1145/3062341.3062352
27. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code with twin memory allocation. In: Proceedings of the 2018 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2018, part of SPLASH 2018, Boston, MA, USA, 4–9 November 2018. ACM (2018)
28. Memarian, K., Gomes, V., Sewell, P.: Cerberus (2018). http://cerberus.cl.cam.ac.uk/cerberus

29. Memarian, K., et al.: Exploring C semantics and pointer provenance. In: Proceedings of 46th ACM SIGPLAN Symposium on Principles of Programming Languages, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 67
30. Memarian, K., et al.: Exploring C semantics and pointer provenance. PACMPL **3**(POPL), 67:1–67:32 (2019). https://dl.acm.org/citation.cfm?id=3290380
31. Memarian, K., et al.: Into the depths of C: elaborating the de facto standards. In: PLDI 2016: 37th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (Santa Barbara), June 2016. http://www.cl.cam.ac.uk/users/pes20/cerberus/pldi16.pdf. PLDI 2016 Distinguished Paper award
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Nienhuis, K., Memarian, K., Sewell, P.: An operational semantics for C/C++11 concurrency. In: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. ACM, New York (2016). https://doi.org/10.1145/2983990.2983997
34. Norris, B., Demsky, B.: CDSchecker: checking concurrent data structures written with C/C++ atomics. In: Proceedings of OOPSLA (2013)
35. Ou, P., Demsky, B.: Towards understanding the costs of avoiding out-of-thin-air results. PACMPL **2**(OOPSLA), 136:1–136:29 (2018). https://doi.org/10.1145/3276506
36. TrustInSoft: `tis-interpreter` (2017). http://trust-in-soft.com/tis-interpreter/. Accessed 11 Nov 2017
37. WG14: Defect report 260, September 2004. http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm
38. Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, pp. 190–204. ACM, New York (2017). https://doi.org/10.1145/3009837.3009838