Formal Methods in Computer-Aided Design 2024

CHCs for Weak Memory

Levente Bajczi
Budapest University of Technology and Economics
Department of Measurement and Information Systems

***Abstract*—Memory-model-aware verification commonly relies on using a relaxed happens-before relation to reason about error reachability. However, state-of-the-art tools focus on detecting bugs rather than proving correctness. I propose a solution to this issue by adapting the verification problem to a system of constrained Horn clauses and, therefore, produce inductive invariants to certify safety.**

I. Introduction

Memory-model-aware verification commonly relies on using a relaxed happens-before relation to reason about error reachability [1], [2], [3], [4]. However, current tools can mostly find bugs rather than prove correctness: HERD [5] can generate and check the consistency of candidate executions, DARTAGNAN [6] uses a Bounded Model Checking (BMC) approach to encode the memory model alongside a finite unwinding of the program and its safety property, and Stateless Model Checking tools such as GENMC [7] and NIDHUGG [8] also only fully support finite executions (with some optimizations such as spin-loop support [7]). CBMC [9] and CBMC-derivatives (such as ZORD [10] and DEAGLE [11]) also use a form of BMC for verification.

In this paper, I propose a way to adapt the verification problem to systems of constrained Horn-clauses (CHCs), which can help create inductive invariants for weak memory software and, thus, help prove (and certify) safety.

Instead of a formal definition of the approach (for which there is not enough space in this paper), I would like to take the opportunity to guide the Reader through an example, which encodes Peterson's mutual exclusion algorithm with weak memory semantics as a system of CHCs.

II. Peterson's algorithm

Consider a version of Peterson's algorithm for two threads in Figure 1. If the algorithm is implemented correctly (i.e., respecting the target architecture's memory access guarantees), the two critical sections between the horizontal lines should never be executed concurrently. We want to prove this property given a memory consistency model as the abstraction of the target architecture. The variable $\mathbf{cnt}$ enables us to count the concurrently executing critical sections, and we can assert that its value when read in C_0 is invariantly 0, as any other value would mean another thread is concurrently in its respective critical section.

$$\mathbf{flag_0} := 0, \mathbf{flag_1} := 0, \mathbf{turn} := 0, \mathbf{cnt} := 0$$

$$\begin{array}{ll|l}
L_0 & \mathbf{flag_0} := 1 & \mathbf{flag_1} := 1 \\
L_1 & \mathbf{turn} := 1 & \mathbf{turn} := 0 \\
 & \mathit{do}\ \{ & \mathit{do}\ \{ \\
L_2 & \quad a := \mathbf{flag_1} & \quad a := \mathbf{flag_0} \\
L_3 & \quad b := \mathbf{turn} & \quad b := \mathbf{turn} \\
L_4 & \}\mathit{while}(a \wedge b) & \}\mathit{while}(a \wedge \neg b) \\
\hline
C_0 & c := \mathbf{cnt} & c := \mathbf{cnt} \\
C_1 & \mathbf{cnt} := c + 1 & \mathbf{cnt} := c + 1 \\
C_2 & \mathbf{cnt} := 0 & \mathbf{cnt} := 0 \\
\hline
L_5 & \mathbf{flag_0} := 0 & \mathbf{flag_1} := 0
\end{array} \qquad \text{(Peterson)}$$

Fig. 1: Peterson's algorithm for 2 threads

A. Independent Thread Encoding

For the proof, we first may encode both threads independently as systems of constrained Horn clauses (CHCs) by defining a predicate over the *local* program variables for each program location, treating global memory accesses as either *skip* instructions (in the case of write accesses) or *havoc* instructions (in the case of read accesses):

$$\begin{array}{lcll}
L_0^{T_0}(a,b,c) & \longleftarrow & \top & //\mathit{init} \\
L_1^{T_0}(a,b,c) & \longleftarrow & L_0^{T_0}(a,b,c) & //\mathit{skip} \\
L_2^{T_0}(a,b,c) & \longleftarrow & L_1^{T_0}(a,b,c) & //\mathit{skip} \\
L_3^{T_0}(a',b,c) & \longleftarrow & L_2^{T_0}(a,b,c) & //\mathit{havoc}\ a \\
 & \dots & & \\
L_0^{T_0}(a,b,c) & \longleftarrow & L_5^{T_0}(a,b,c) & //\mathit{repeat} \\
\bot & \longleftarrow & C_1^{T_0}(a,b,c) \wedge \neg(c=0) & //\mathit{error}
\end{array}$$

The other thread mirrors this behavior, except for the guard:

$$\begin{array}{lcll}
L_2^{T_1}(a,b,c) & \longleftarrow & L_4^{T_1}(a,b,c) \wedge (a \wedge \neg b) & //\mathit{fail} \\
C_0^{T_1}(a,b,c) & \longleftarrow & L_4^{T_1}(a,b,c) \wedge \neg(a \wedge \neg b) & //\mathit{succeed}
\end{array}$$

Notice that if all CHCs are independently solvable for each thread, the safety proof will hold given *any* architecture. However, this is not the case here, as the value of c is nondeterministically assigned in $C_1 \longleftarrow C_0$, and thus, the assertion $c = 0$ may fail. Therefore, we need to encode the data regarding global variables as well.

B. Encoding Naive Dataflow

To encode dataflow, we introduce the predicate $W(v, l)$, which means that a write to variable v with value l has happened. To this end, let us add the following rules to the CHC systems (we use the index of the global variables in the header of Figure 1 as the values to v):

$$\begin{array}{lcll}
W(v,0) & \longleftarrow & v \in \{0,1,2,3\} & //\mathit{init} \\
W(0,0) & \longleftarrow & L_0^{T_0}(a,b,c) & //\mathbf{flag_0} := 1 \\
W(2,1) & \longleftarrow & L_1^{T_0}(a,b,c) & //\mathbf{turn} := 1 \\
 & \dots & &
\end{array}$$

Reads may only read such values from global variables that are encoded in W. Therefore, we need to modify the previously *havoc*'d rules:

$$\begin{array}{lcll}
L_3^{T_0}(a',b,c) & \longleftarrow & L_2^{T_0}(a,b,c) \wedge W(1,a') & //a := \mathbf{flag_1} \\
L_4^{T_0}(a,b',c) & \longleftarrow & L_3^{T_0}(a,b,c) \wedge W(2,b') & //b := \mathbf{turn} \\
C_1^{T_0}(a,b,c') & \longleftarrow & C_0^{T_0}(a,b,c) \wedge W(3,c') & //c := \mathbf{cnt} \\
 & \dots & &
\end{array}$$

https://doi.org/00.0000/00000000

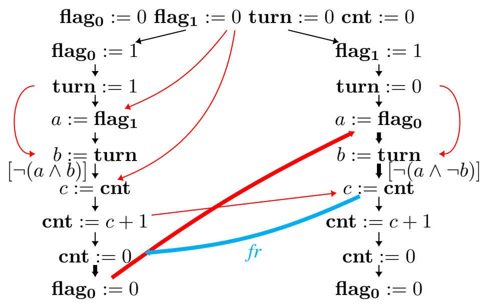

Fig. 2: A spurious counterexample to Peterson's safety

If the systems of CHCs were to be solvable with this encoding, we would get a safety proof that is invariant in the *ordering* guarantees of the memory model, i.e., no instruction reordering would break the proof, as long as the simple guarantee that *only such values can be read from memory that is written to it* is given This even covers non-causal memory models, where reads may receive values from (temporally) later writes.

However, for Peterson's algorithm, this is not the case; the system is unsatisfiable, and therefore, a counterexample to the assertion's safety exists. If we visualize one of the counterexamples on an execution graph (as is conventional for memory-aware verification [5], [6], [7]) in Figure 2, we can see a potential problem: in this counterexample, the program order $\prec_{po}$ (black edges) and read-from relation $\prec_{rf}$ (red edges) both express a form of temporal ordering if we expect every instruction to execute in-order (i.e., over *sequential consistency* (SC)) and additionally, the from-read relation $\prec_{fr}$ (blue edge) also expresses a temporal ordering As there is a cycle in the temporal orderings (shown in **bold** in Figure 2), we must reject this counterexample.

C. Encoding Temporal Relations

To encode temporal relations among memory accesses, we introduce an *event ID* (unique for each memory access) and an ordering relation $\prec$. To preserve causality (i.e., a read must be preceded by its *rf*-related write):

$$A \prec_{rf} B \implies A \prec B \tag{1}$$

, and to make sure a write's value is only read if it has not been overwritten ($\prec_{co}$ is a global order of same-variable writes):

$$\begin{gathered} A \prec_{rf} C \land C \prec_{co} B \implies A \prec_{fr} B \\ A \prec_{fr} B \implies A \prec B \end{gathered} \tag{2}$$

Notice that the source-level instruction order is not encoded in $\prec$. If left as-is, the encoded memory model would allow any two instructions to reorder themselves and, thus, still allow problematic (assertion-violating) program executions. For strict sequentialism, we must also have:

$$A \prec_{po} B \implies A \prec B \tag{SC}$$

, but for more relaxed memory models, we must also relax this constraint in the *preserved program order* (ppo). For example, the Partial Store Order (PSO) memory model relaxes the write-to-read and write-to-write ordering constraints for accesses to different variables:

$$\begin{gathered} A \prec_{po} B \land R(A) \land W(B) \implies A \prec_{ppo}^{PSO} B \\ A \prec_{po} B \land R(A) \land R(B) \implies A \prec_{ppo}^{PSO} B \\ A \prec_{po} B \land var(A) = var(B) \implies A \prec_{ppo}^{PSO} B \\ A \prec_{ppo}^{PSO} B \implies A \prec B \end{gathered} \tag{PSO}$$

To account for these types of memory models (which are globally temporally ordered in $\prec$), we need to modify the CHC encoding of the program. First, every predicate corresponding to a program location will receive a unique *event ID* (e.g., increasing odd and even numbers for the two threads), and every predicate (including W) will receive an SMT array $\prec$, mapping event IDs to natural numbers, corresponding to the events' temporal order.

Thus, any write instruction $l_0 \xrightarrow{\mathbf{x}:=1} l_1$ can be encoded the following way:

$$\begin{array}{lll} W(\prec, \prec_{co}, 0, 0, 0) & \longleftarrow & \top \\ l_0(\prec, \prec_{co}, 1) & \longleftarrow & \top \\ W(\prec, \prec_{co}, e, 0, 1) & \longleftarrow & l_0(\prec, \prec_{co}, e) \land \\ & & W(\prec, \prec_{co}, e', 0, \_) \land \\ & & \prec_{co}[e'] + 1 = \prec_{co}[e] \land \\ & & \prec[e'] < \prec[e] \\ l_1(\prec, \prec_{co}, e) & \longleftarrow & l_0(\prec, \prec_{co}, e') \land \\ & & W(\prec, \prec_{co}, e', \_, \_) \land \\ & & e' + 1 = e \end{array} \tag{W}$$

Informally, we can say that we deduce an initial write with a default value alongside the initial location's predicate, then from the l_0 (because on an outgoing edge there is a write), we deduce a further write with the written value, also asserting that there is an immediately co-before write whom this write succeeds in $\prec$. Then, the next location is deduced from the initial location *and* a successful write with the same event id. In the next location, the event ID (e) is increased by one.

Furthermore, any read instruction $l_0 \xrightarrow{a:=\mathbf{x}} l_1$ can be also be encoded:

$$\begin{array}{lll} W(\prec, \prec_{co}, 0, 0, 0) & \longleftarrow & \top \\ l_0(\prec, \prec_{co}, 1) & \longleftarrow & \top \\ l_1(\prec, \prec_{co}, e) & \longleftarrow & l_0(\prec, \prec_{co}, e') \land \\ & & W(\prec, \prec_{co}, e'', 0, a) \land \\ & & W(\prec, \prec_{co}, e''', 0, \_) \land \\ & & \prec_{co}[e''] + 1 = \prec_{co}[e'''] \land \\ & & e' + 1 = e \land \\ & & \prec[e''] < \prec[e] \land \\ & & \prec[e] < \prec[e'''] \end{array} \tag{R}$$

Again informally, for the system to deduce l_1, (besides l_0) there must be a suitable W to x with an id e'', which must be $\prec$-before e. There must also be an immediate successor to $W(e'')$ in $W(e''')$, which must be $\prec$-after e.

III. Conclusion

In this paper, I showcase an experimental transformation of a program and a corresponding memory model into systems of constrained Horn clauses. Using this encoding, Spacer (Z3) successfully provided a proof for Sequential Consistency and a counterexample for Partial Store Ordering. From the proof, location invariants can be extracted, and thus, safety can be *certified*, and a verification witness can be generated. Therefore, we can successfully prove the safety of some weak memory programs that previously we could only solve up to some finite bound.

References

[1] J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 141–157. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_9

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models (extended version)," *Form. Methods Syst. Des.*, vol. 40, no. 2, p. 170–205, apr 2012. [Online]. Available: https://doi.org/10.1007/s10703-011-0135-z

[3] N. Sinha and C. Wang, "On interference abstractions," *SIGPLAN Not.*, vol. 46, no. 1, p. 423–434, jan 2011. [Online]. Available: https://doi.org/10.1145/1925844.1926433

[4] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't Sit on the Fence," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 508–524.

[5] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, jul 2014. [Online]. Available: https://doi.org/10.1145/2627752

[6] N. Gavrilenko, H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 355–365.

[7] M. Kokologiannakis and V. Vafeiadis, "GenMC: A Model Checker for Weak Memory Models," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I.* Berlin, Heidelberg: Springer-Verlag, 2021, p. 427–440. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_20

[8] P. Agarwal, K. Chatterjee, S. Pathak, A. Pavlogiannis, and V. Toman, "Stateless Model Checking Under a Reads-Value-From Equivalence," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 341–366.

[9] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176.

[10] F. He, Z. Sun, and H. Fan, "Satisfiability modulo ordering consistency theory for multi-threaded program verification," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. Pldi 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1264–1279. [Online]. Available: https://doi.org/10.1145/3453483.3454108

[11] Z. Sun, H. Fan, and F. He, "Consistency-preserving propagation for SMT solving of concurrent program verification," *Proc. ACM Program. Lang.*, vol. 6, no. Oopsla2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563321