Stateless Model Checking for the C/C++11 Memory Model

Ishita Jaju
School of Computer Science and Engineering
Vellore Institute of Technology, Chennai, Tamil Nadu, India
ishita.jaju2016@vitstudent.ac.in

Dr. B V A N S S Prabhakar Rao
School of Computer Science and Engineering
Vellore Institute of Technology, Chennai, Tamil Nadu, India
prabhakarrao@vit.ac.in

Abstract— The C11 memory model is a relaxed model that works on top of an architecture. This model has many intriguing properties which help in the efficiency of a program but these may also cause certain unexpected behaviors. Such behavior violations are a cause of annoyance to programmers. There exist some C11/C++11 model checking tools which analyze the given program and produce examples in the form of execution traces when such behavior violation may take place. In this paper, we discuss four such model checking tools and compare them on the basis of their performance on some example programs.

Keywords- C11 Memory Model, Model Checking, Program Verification, Relaxed Memory

I. INTRODUCTION

The C11 relaxed memory model was a topic of great discussion among academics, since it came out with two new features on top of the pre-existing C model – atomic operations and memory fences. The novel relaxed nature of the C11 model tended to have certain properties, such as instruction re-orderings between threads and inter-thread instruction interleavings. A read instruction in concurrent or parallel programming may read a value of a variable form any of the previously set values. This relaxed model was introduced in order to increase efficiency and speed.

However, this relaxed model also caused certain behaviors that proved to be unexpected for the programmers, such as an incorrect value of a variable being set. Such behaviors would be an impossibility in the sequential consistency model and hence were unexpected and undesirable. In such a case, the programmer may add a statement such as an "assertion" which can ensure the prevention of such behavior.

A single run with the order of instructions is called an execution trace. This "assert" statement will be violated in some cases with instructions reordering and will be satisfied in others. A model checker will provide the programmer with counter examples of entire execution traces where the assertion will get violated. The programmer can then use these counter examples to obtain the scenarios which should thus be avoided.

In this paper, six well-known multi-threaded algorithms have been used with varying number of threads to compare four model checking tools in the C11 relaxed memory model based on the time they take to figure out buggy execution traces.

II. BACKGROUND

A. A. Sequential Consistency and Relaxed Memory Models

A program is said to have sequential consistency if the order and result of the concurrent program comes out as if operations of the execution were run in some sequential order. The order of each individual process appears in the sequence that was determined by the program.

For instance, let's take a look at Fig. 1. A sequentially consistent model would execute in only the following orders:

```
init A = 0, B = 0

      Thread 1          Thread 2
(1) A = 1          (3) B = 1
(2) print(B)       (4) print(A)
```

Figure 1. Simple two-threaded program

1 -> 2 -> 3 -> 4 output: 01
3 -> 4 -> 1 -> 2 output: 01
1 -> 3 -> 2 -> 4 output: 11
1 -> 3 -> 4 -> 2 output: 11

In the above example, a sequentially consistent model would be able to print the values, “01” and “11” only. It would be impossible for it to print, say, “00.”

On the other hand, even though these models would preserve the rules for being sequentially consistent, the program would become extremely slow since only one instruction would execute at a time. This is where relaxed memory models come in, where instruction A would not need to wait for instruction B to complete and both would run in parallel. Examples of relaxed memory models include Total Store Ordering (TSO), Processor Consistency (PC), Partial Store Ordering (PSO), Weak Ordering (WO), Release Consistency (RC). In these models, there is some allowance in operation reordering, such as read-write reordering etc.

In the case of these relaxed memory models, it may become a possibility to get an output of “00” when two or more threads concurrently access the same memory location. For example, if R1 and R2 both read the init values of y and x instead of the updated values in instructions (1) and (3), then the output would be printed as “00”.

This would be inconsistent with the sequential model and hence this behavior would not be expected by the programmer.

```
init A = 0, B = 0

Thread 1          Thread 2
(1) A = 1         (3) B = 1
(2) print(B)      (4) print(A)

assert(A != 0 and B != 0)
```

Figure 2. Simple two-threaded program with “assert” statement

To prevent such behavior while running relaxed consistency, the programmer might add an “assert” statement (Fig. 2), ensuring that such unexpected behavior does not take place in the program.

B. Backgound on C11

In 2011, a new ISO standard for C/C++ was sanctioned which included some new features and additions to the pre-existing C model. This was informally known as C11 or C++11. This new standard defines the memory model for C/C++, which describes the behavior and interaction of threads in a concurrent program through shared data.

The memory model introduced in the new standard basically introduces two new portable ways for memory access synchronization in concurrent or multi-threaded programming, namely atomic operations and fences.

Atomic operations consist of a load operation and a store operation. These atomic operations can be executed to or from a particular memory location without causing data races or undefined behavior There are also modes to these operations, which are relaxed, release, acquire, consume, acquire-release and sequential consistency. These have different degrees of strength and weakness, depending upon how relaxed the mode is. The order of these models are described in Fig. 3.

non-atomic < relaxed < release = acquire < sequential consistency

Figure 3. Memory orders in increasing order of strength

Fences introduced in C11 allow memory operations to be ordered or reordered between threads [1]. A fence, on the other hand, can either be a sequential fence, a release fence or an acquire fence. There are synchronizations among these fences which ensure synchronization between instructions. ensuring that such unexpected behavior does not take place in the program.

C. Model Checking

Model checking is a verification method for finite state concurrent systems, first developed in the 1980s. It exhaustively searches the state space of the design with certain specifications. These specifications may be written in propositional logic or temporal logic.

There are many properties of model checking that programmers can take advantage of. It is a fast process with no proofs involved. A programmer will also receive a counter example of a certain execution, describing the order and trace of each instruction and their metadata. Many concurrency properties can be expressed with the model checking logic. A major disadvantage is that there might be a state explosion problem, with too many

processes and data paths being explored. A lot of progress has been made on this problem. Some of these solutions include binary decision diagrams, partial order reduction, abstraction, compositional reasoning, symmetry, cone of influence reduction, semantic minimization.

The performance of model checkers today has drastically improved, with model checking tools being able to routinely handle systems with a hundred to three hundred state variables. Even systems with an essentially infinite number of states can be checked by using appropriate techniques.

This paper talks about stateless model checking. Stateless model checking entails testing concurrent programs explosively by verifying and exploring all possible scenarios and thread interleavings. This is as opposed to a variation such as bounded model checking which unrolls the finite state machine for only a fixed number of steps and keeps checking for a certain condition to be met. The motivation behind stateless model checking was that programmers and developers often need unit tests to test their code or part of their code to figure out what behavior the underlying memory model allows. Stateless model checkers explore the state space of a program's possible behaviors with different combinations of thread interleavings in the program. Such an exhaustive search of a program does result in computational intractability and programs may even grow to any unreasonable lengths. This issue is solved by many methods such as Partial-Order Reduction (POR) mechanisms

III. LITERATURE REVIEW

C11 has been accused of supporting "racy" memory accesses [2] within different levels of consistency. This 2017 paper discusses some of the innate flaws in the C11 memory model and proposes a repaired version, namely RC11 (for Repaired C11). This proposed model is a repaired version of C11 which gets rid of the notion of dependency cycles.

A concurrent program's semantics depend upon the underlying architecture and the memory model of the system. This property has generated a lot of interest among academia.

Reference [10] presents a framework for applications of Stateless Model Checking of programs running in the POWER architecture. Reference [11] implements model checking in a range of different memory models which include SC, Intel x86 and IBM Power. On the other hand, [12] and [13] conduct testing and verification for weak memory models.

Pioneers in computer science provided verification tools and formal proof methods which are helpful in verification and checking for program correctness. However, these techniques were hardly used in practice until model checking tools were introduced [8]. Reference [9] discusses bounded model checking as with state equivalence under two memory models with Dartagnan and Porthos and compares it to other model checking tools. It also discusses factors such as user interface, reachability and inclusion.

Model checking is an industrial-level verification technique. It has applications in integrated circuit design [15], where the algorithm for model checking can conduct circuit verification with a huge number of states. Another application includes the analysis of biochemical networks [5]. Database backend web applications also use model checking [16] to interpose in the middle of the program layer and backend layer and track the effects of the queries made to the database. Other applications of model checking include situations with natural description for concurrent and reactive systems such as automobiles and aero-space control systems, nuclear control systems, network management systems etc.

IV. MODEL CHECKERS

A. CDS Checker

CDS Checker [4] was proposed in 2013. It is a stateless model checking tool for C/C++ code. This tool exhaustively unit tests several relaxed behaviors of the given concurrent code in the C/C++ memory model. Made in University of California, Irvine, the authors have developed novel techniques and methods for this extensive exhaustive search and also to minimize the number of execution behaviors that need to be explored.

CDS Checker provides counter examples and the complete execution trace for each example. The issue of explosive search space is solved by using a class of optimization techniques called dynamic partial-order reduction (DPOR) in CDS Checker.

A limitation of CDS Checker comes in unbounded loops. Exploring while loops is not a possibility for this tool. If a user needs to test a program containing a while loop, it would first have to be converted to a for loop with limited iterations. CDS Checker also fails to provide the user with the line number from the source code of the execution trace since the tool works on the generated binary object file of the code, as opposed to using the main source code.

CDS Checker was one of the first model checkers for the C11 model and it was the basis for many of the new and other model checkers explored in this paper. One difference between this and the others is that CDS Checker uses C++ files converted into binary object files which are then tested and explored.

B. RCMC

RCMC [6] is a stateless model checking tool which came out in 2018. It verifies concurrency under the repaired version of C11, RC11. This tool states that it works directly on execution graphs, as compared to other earlier methods where threads enumerate interleavings up to some partial order reduction improvements, like the way used by CDS Checker.

The advantage to this is that it avoids redundant exploration. This tool constructs execution graphs by marking the nodes revisitable, such as revisiting multiple reads or writes. They have also created a mechanism to rule out incoherent executions. The authors claim that using this approach makes RCMC significantly faster, making it more efficient than other model checking tools. Another feature is that it is also more resilient to small changes in the benchmarks.

A limitation to RCMC is that the provided output stops when it finds the first violation, as opposed to CDS Checker which provides all the possible violations. RCMC also fails to provide certain important properties of the instructions in the output, such as the read-from (rf) value. It does, however, give the line number from the original source code, which CDS Checker failed to provide.

The paper finds out that model checking under weak memory constraints is much more easily done than doing the same for sequential consistency. This was in contrast to thread-interleaving method which creates difficulties in model checking with weak memory effects. The evaluation of this tool shows that other tools consider a number of infeasible executions, thus increasing the runtime, whereas RCMC manages to outperform the others in most of the cases.

C. GenMC

A model checker for weakly consistent libraries is GenMC [7]. This tool claims to be optimal and parametric in choosing and using the memory models and it also can verify concurrent library clients. This tool does not explore inconsistent execution paths and explores each program execution according to the model exactly once.

Reference [7] talks about tackling multiple challenges and discusses their solutions. The first challenge is related to the memory model that the program has to follow, since the model gives the outcome of the program. They implement their tool in multiple memory models such as SC, TSO, PSO, RC11. Problems arise since some assertions hold under the SC model and the TSO model but not under PSO or RC11 because of differences in handling instructions among the models. They solve this in their extensive algorithm, which works under any memory model and satisfies the four basic assumptions - acyclicity, extensibility, prefix-closedness, well-blocking.

GenMC is the most parametric tool when it comes to memory models out of all the tools discussed, since it can be used for models other than traditional memory models, such as those including high-level libraries like those that include mutex locks as primitive operations. They solve the problem of numerous trivial execution graphs through the partial order reduction method.

A drawback to GenMC is that just like RCMC, it stops at the first buggy execution trace it finds. Apart from that, GenMC is a fast and reliable tool for C11 model checking.

D. Tracer

Tracer [14] is based on symbolic execution. It works mainly for the safety properties of sequential C programs. Tracer basically builds an execution tree which over approximates the arrangement of all solid reachable states. The symbolic path tries to reach the error and in case it cannot reach it, the program is reported as “safe”. Otherwise, the program is said to contain a bug or is non-terminating.

The idea of symbolic execution is that there are multiple executions based on which inputs cause the program to run in what kind of way. This may not be very apparent in simple programs, but multi-threaded programs with multiple ways of thread interleavings along with if-else branch statements and loops can procure a myriad of different outputs and execution traces. Symbolic execution actually has many benefits, which are mentioned as part of why it was chosen for Tracer. These include not exploring infeasible paths and avoiding expensive refinements, recovery from overly specific abstractions. However, there are some challenges involved as well. As it turns out, there are an exponential number of paths and in case of unbounded loops, an infinites number of paths.

Tracer creates trees of the symbolic execution traces of the given input program. If and when the symbolic execution reaches the error node, it reports the error. If the error is real then it aborts and in case of spurious errors, it refines it. In case of no error, it continues the symbolic execution normally. After the symbolic execution, if all of Tracer’s assertions are violated then the program is reported as safe. If it is unsafe, then a counter example is shown as output.

V. METHODOLOGY AND ARCHITECTURE

Experiments for this paper were conducted using four of the aforementioned model checking tools, namely CDS Checker, Tracer, GenMC and RCMC. These tools were chosen since they work in the C11/C++11 memory model, as opposed to general relaxed memory models implemented in other tools such as Nidhugg [3].

Each tool has varying input formats and file types - CDS Checker has a C++ file input with a .cc file extension. It first converts this into a binary object file using its makefile, which is then given as input to the tool. Tracer takes a C++ file input with extension .cpp which it turns into an executable file before giving it as input to the tool. GenMC takes a C file input which is distributed between two files - a file with thread functions and a main file. RCMC input works the same as GenMC with a distributed C file.

Each tool also has its own different libraries, defining functions such as thread functions, assertion statements etc. In some cases, even the main function is specially defined by the tool and the default main function is not used. For example, CDS Checker defines its own main function as “user_main” so it is easier for the tool to conduct its own tests.

The examples used are common algorithms in concurrency and most were easily found in the examples and benchmarks of the tools. There were cases where some examples were not present in one or two of the tools, in which case the program was re-written according to the particular tool’s format and then executed.

For each tool, the time was an important measure. The tools GenMC and RCMC had built-in functions and provided the time taken for execution in their respective outputs. On the other hand, both CDS Checker and Tracer did not provide the execution time in their outputs. Therefore, a script was written for each in python, measuring the time taken to receive the output and also to count the relevant lines of code.

Another measure to be taken into consideration was the lines of code for each tool and each program. The lines of code were measured as separate instructions divided by lines, excluding empty lines - just new-line characters, or just brackets. In the case of GenMC and RCMC where the input file is distributed, lines of code from both files are taken into consideration and added.

The programs lamport, casw and n_reads_m_writes are actually algorithmic programs in which the number of child threads created can be varied in order to obtain varying outputs. Hence multiple values for each were tested and the number of threads used for each is mentioned inside the brackets.

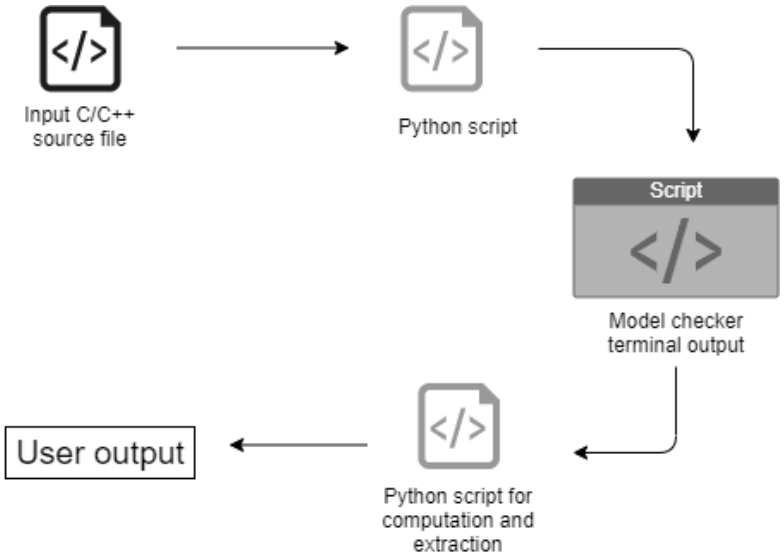

Figure 4. Architecture of the proposed system

TABLE I. Results of the performance of the four model checking tools

Input file (with no. of threads)	CDS Checker				Tracer				GenMC				RCMC			
	LOC	BTE	TCEE	Time	LOC	BTE	TCEE	Time	LOC	BTE	TCEE	Time	LOC	BTE	TCEE	Time
dekker_rlx (2)	26	2	10	0.0057	26	2	6	0.0061	27	1	0	0.05	27	1	2	0.02
peterson (2)	48	24	241	0.036	51	22	106	0.024	47	1	0	0.03	44	1	0	0.06
lamport (2)	105	16	404	0.05	102	16	282	0.061	46	10	13	0.03	40	1	28	0.03
szymanski (2)	91	∞	∞	∞	91	∞	∞	∞	71	904	201	0.06	53	0	384	0.09
casw (3)	40	0	23	0.007	19	0	30	0.005	21	0	24	0.03	21	0	66	0.03
casw (4)	40	0	243	0.046	19	0	336	0.021	21	0	200	0.03	21	0	1200	0.08
n_writers m_readers (10,3)	27	0	1765	0.31	23	0	1331	0.18	27	0	20736	1.44	24	0	21457	1.7
n_writers m_readers (15,3)	27	0	4965	1.16	23	0	4096	0.7	27	0	83521	9.17	24	0	105439	24.3
n_writers m_readers (20,3)	27	0	10715	3.34	23	0	9261	2.15	27	0	234256	44.03	24	0	582467	92.6

all time is in seconds

LOC: Lines of code BTE: Buggy Traces Explored TCEE: Total Complete Executions Explored

Architecture of the proposed model is described in Fig. 4. The C/C++ source file with the required assertions is provided as input to the python script, which then runs the code on the model checker. The script times the tool and extracts information such as the total number of explored executions, the number of buggy executions explored etc and gives the results as an output to the user.

VI. RESULTS

Using these tools, the major deduction that can be made is that CDS Checker and Tracer match in many aspects and are similar to each other, whereas GenMC and RCMC are similar to each other. Both CDS Checker and Tracer work on C++ files and convert them into executables before providing it to the tool. On the other hand, GenMC and RCMC have distributed C files with thread and main functions in different files.

The example file, casw is a special file, containing RMW (read-modify-write) function. The RMW function is not supported by all tools, for instance Nidhugg does not support such a function. The szymanski algorithm is also unsupported by Tracer and CDS Checker, hence producing a non-terminating output with infinite counter examples of buggy executions. This is because of the unbounded loop present in the program.

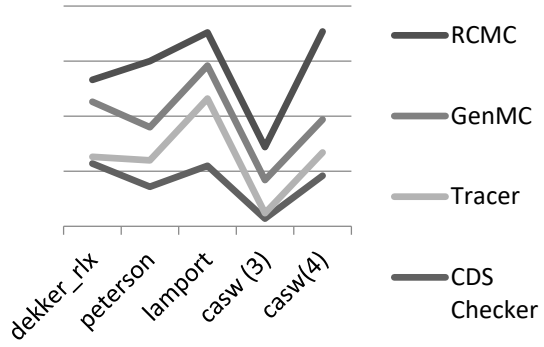

Figure 5. Performance of the tools on different programs based on time taken by each

Fig. 5. compares the performance of the four tools based on only the time factor. We notice that the tools cannot be compared based on certain factors such as "number of total execution traces" or "number of buggy execution traces," since the tools do not behave in the same way. Tool CDS Checker provides all possible execution traces, while RCMC stops at the first buggy trace found.

VII. CONCLUSION AND FUTURE SCOPE

The most comprehensive model checking tool for programs in the C11 concurrency model is CDS Checker. It provides the programmer with all the possible execution traces, giving the number of infeasible as well as possible buggy execution traces. The output contains all the counter examples as well as their metadata such as reads-from values, memory order etc. It also is fast and efficient, with the only seeming drawback being that unbounded loops are not handled in it. There are many applications for this type of comprehensive tool.

A tool like RCMC or GenMC, which stops the output at the first found buggy execution has other applications. However, these tools fail to provide certain required information about execution traces and their metadata. This in turn might result in lack of information and can be unusable in some cases.

C2x is the latest standard revision in the C memory model. Model checkers are already being made for the C20 model. It adds one new principle to the principles of C, relating to API documentation and arrays.

REFERENCES

[1] "C - C 11 Memory Model: c Tutorial," C - C 11 Memory Model | c Tutorial. [Online]. Available: https://riptutorial.com/cplusplus/topic/7975/cplusplus11-memory-model. [Accessed: 20-Mar-2020].

[2] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in C/C 11," ACM SIGPLAN Notices, vol. 52, no. 6, pp. 618–632, 2017.

[3] P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas, "Stateless model checking for TSO and PSO," Acta Informatica, vol. 54, no. 8, pp. 789–818, Jul. 2016.

[4] B. Norris and B. Demsky, "CDSchecker," ACM SIGPLAN Notices, vol. 48, no. 10, pp. 131–150, Dec. 2013.

[5] M. Carrillo, P. A. Góngora, and D. A. Rosenblueth, "An overview of existing modeling tools making use of model checking in the analysis of biochemical networks," Frontiers in Plant Science, vol. 3, 2012.

[6] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis, "Effective stateless model checking for C/C concurrency," Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, pp. 1–32, 2018.

[7] M. Kokologiannakis, A. Raad, and V. Vafeiadis, "Model checking for weakly consistent libraries," Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019, 2019.

[8] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to model checking," Handbook of Model Checking, pp. 1–26, 2018.

[9] H. Ponce-De-Leon, F. Furbach, K. Heljanko, and R. Meyer, "BMC with memory models as modules," 2018 Formal Methods in Computer Aided Design (FMCAD), 2018.

[10] P. A. Abdulla, M. F. Atig, B. Jonsson, and C. Leonardsson, "Stateless model checking for POWER," Computer Aided Verification Lecture Notes in Computer Science, pp. 134–156, 2016.

[11] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient bounded model checking of concurrent software," Computer Aided Verification Lecture Notes in Computer Science, pp. 141–157, 2013.

[12] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats," Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 14, 2013.

[13] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi, "On the verification problem for weak memory models," ACM SIGPLAN Notices, vol. 45, no. 1, p. 7, 2010.

[14] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," Computer Aided Verification Lecture Notes in Computer Science, pp. 758–766, 2012.

[15] J. Burch, E. Clarke, D. Long, K. Mcmillan, and D. Dill, "Symbolic model checking for sequential circuit verification," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 4, pp. 401–424, 1994.

[16] M. Gligoric and R. Majumdar, "Model checking database applications," Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science, pp. 549–564, 2013.