Predicting and Witnessing Data Races Using CSP

Luis M. Carril(✉) and Walter F. Tichy

Institute for Program Structures and Data Organization (IPD), Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany
{luis.carril,walter.tichy}@kit.edu

Abstract. Detecting and debugging data races is a complex task due to the large number of interleavings possible in a parallel program. Most tools can find the data races reliably in an observed execution, but they miss errors in alternative reorderings of events. In this paper we describe an automated approach to generate, from a single program trace, a model in CSP with alternative interleavings. We check for data races patterns and obtain a witness that allows the reproduction of errors. Reproduction reduces the developer effort to correct the error.

Keywords: Data race · Concurrent programs · Debug · CSP

1 Introduction

Finding and debugging synchronization errors such as data races is a daunting task. As multicore processors become common, tools that help developers cope with data races are desperately needed. A data race happens when two threads access the same variable concurrently and at least one performs a write operation.

Usually, dynamic approaches to race detection are based on happens-before or lockset algorithms. Happens-before detectors [2](based on the Lamports relationship [8]) only cover a specific observed interleaving. They are conservative and need to run multiple times to cover unexplored interleavings. Some works [14] relax the happens-before relationship under certain conditions to cover more cases. The lockset algorithm [12] checks for consistent locking of shared objects, but produces a high number of false positives. Hybrid approaches alse exists [7,10], combining advantages of both.

The interleaving presented in listing 1.1 are difficult to detect. Only if the exact timing in listing 1.2 occurs, a happens-before detector reports the race. Additionally, typical approaches only provide the location of the error, but no context about thread state or how they reached that particular point.

Our approach infers alternative interleavings from an observed program trace and finds data race patterns in these reorderings. Once a pattern matches in one

K. Havelund et al. (Eds.): NFM 2015, LNCS 9058, pp. 400–407, 2015.
DOI: 10.1007/978-3-319-17524-9_28

reordering, these reordering is a story on how the program reaches the erroneous state. We infer the interleavings by constructing a model that combines the independent actions of the threads and the semantic behavior of the synchronization operations (e.g.: a mutex can only be held by one thread at a time). This model is described with the process algebra CSP (Communicating Sequential Processes [5]) and the pattern search is done with CSP refinement relationships. This approach not only allows the prediction races, but also provides the steps which lead to that race.

Similar work is found in the maximal causal model [6] and witness generation [11]. These works can also predict races and provide counterexamples from a single observation using a read-write consistency model implemented with SMT.

Listing 1.1. Original captured trace

```
    Thread 1      Thread 2
1:  y++
2:  lock(m)
3:  x++
4:  unlock(m)
5:                lock(m)
6:                x++
7:                unlock(m)
8:                y++
```

Listing 1.2. Reordering with data race

```
    Thread 1      Thread 2
1:                lock(m)
2:                x++
3:                unlock(m)
4:                y++
5:  y++
6:  lock(m)
7:  x++
8:  unlock(m)
```

2 Detecting Races with CSP

CSP is a formal language to describe a system composed by processes (in uppercase), each process is a sequence of atomic events (in lowercase). Then processes communicate with each other and the environment (which can be another process) sharing events synchronously.

Our approach consists of defining a CSP process *PROGRAM*, that represents all alternative interleavings of a captured trace. Using a CSP refinement relationship we check if the *PROGRAM* process matches a data race pattern (defined as a CSP process). When the relationship does not hold, a data race is revealed along with a counterexample, that allows to reproduce the scenario with the data race.

2.1 Modeling Events

A captured trace consists of a sequence of the following events: *read*, *write*, *start*, *end*, *fork*, *join*, *lock* and *unlock*. The events *start* and *end* have a single parameter: the thread identifier. The other events have two parameters: a related thread identifier and a relevant object for the operation: a child thread, a mutex or a variable. We map each captured event directly to a CSP event; e.g. a lock by thread t_1 on lock l_1 is represented $lock.t_1.l_1$. A symbol ? represents any valid value for a parameter; then $lock?t.l_1$ is a lock event on l_1 by any thread.

2.2 Modeling the Program

To construct the process $PROGRAM$ which represents all possible reorderings of the trace, we model the behavior of the different threads and the semantics of the synchronizing operations independently with CSP processes.

Each thread is modeled using the events that it had performed in the captured trace. We only keep one access to a variable per thread between two synchronizing operations. The corresponding events in CSP are concatenated using the prefix operator $\rightarrow$, which creates a total order for the thread trace as it has been observed; e.g. for the thread 1 in the listing 1.1:

$$THREAD_1 = write.t_1.y \rightarrow lock.t_1.m \rightarrow write.t_1.x \rightarrow unlock.t_1.m \rightarrow STOP$$

The process $STOP$ denotes a process which performs no actions (is a reserved CSP process). All these thread processes are combined with the CSP interleaving operator $|||$. The resulting process contains all reorderings of events of the thread processes, but obeying the total order corresponding to each thread:

$$T\_INTER = |||_{i \in Threads}\ T_i$$

This operation is agnostic of the semantic meaning of the events, so the process also contains a lot of traces that would not be possible in the original program, for example the order $..., lock.t_1.m, lock.t_2.m, ...$ We avoid these illegal interleavings modeling the semantic restrictions of synchronizing operations: forking, joining and mutex accesses.

The restrictions are modeled in the process $SYNC$. We compose the previous T_INTER process with the $SYNC$ process using the generalized parallel operator:

$$PROGRAM = T\_INTER \underset{sync\_set}{\parallel} SYNC$$

$$sync\_set = \{|start, end, fork, join, lock, unlock|\}$$

Where $\{|x|\}$ denotes all events derived from x. The parallel operator ensures that if a process wants to execute an event in the set, the other process must be ready to execute it too, otherwise it will be blocked until the second process is also ready. The T_INTER and $SYNC$ processes synchronize in all events with the exception of *read* and *write* events. This combination ensures that all traces of process $PROGRAM$ are traces that follow the order restrictions of T_INTER (total order per thread) and the order restrictions of $SYNC$ (order of the synchronization operations). The process $PROGRAM$ reflects all the interleavings possible by our original program in the path covered by the initial observation.

The $SYNC$ process describes the following synchronizations: thread creation, thread joining, mutex locking and unlocking. We define and interleave a process for each specific synchronization class.

$$SYNC = FORKS \;|||\; JOINS \;|||\; MUTEXES$$
$$FORKS = |||_{i \in Threads}\; FORK(i)$$
$$JOINS = |||_{i \in Threads}\; JOIN(i)$$
$$MUTEXES = |||_{i \in Mutexes}\; MUTEX(i)$$

Each of these processes represent a subgroup of restrictions on the order of the synchronization operations.

The creation of each thread is defined by a process

$$FORK(i) = fork?t.i \rightarrow start.i \rightarrow STOP$$

The *start* event cannot happen if any other thread has not realized a $fork$ on the created thread. This process finally stops without doing anything else.

The *join* of each thread is defined by a process

$$JOIN(i) = end.i \rightarrow join?t.i \rightarrow STOP$$

Similar to the previous process, this is a one-shot process where any *join* on i cannot be complete until the corresponding process executes its *end* event.

A mutex process is described as a recursive process:

$$MUTEX(i) = lock?t.i \rightarrow unlock.t.i \rightarrow MUTEX(i)$$

For a specific mutex i this process is the only one capable of executing *lock* and *unlock* events on it, after one thread process executes the *lock.t.i* operation, the other thread processes are blocked until the corresponding *unlock* event occurs. Afterwards the process returns to the initial point where it accepts the *lock* event for any thread.

2.3 Detecting Races in CSP

With a CSP process representing the observed trace and the alternative reorderings along the same path, we check the model for data races. A race is the concurrent execution of two events from different threads on the same variable, where at least one is a write. Then a trace containing a data race has one of the following subtraces:

$$read.t_1.v, write.t_2.v \qquad write.t_1.v, read.t_2.v \qquad write.t_1.v, write.t_2.v$$
$$read.t_2.v, write.t_1.v \qquad write.t_2.v, read.t_1.v \qquad write.t_2.v, write.t_1.v$$

The two conflicting events can be observed consecutively and without any synchronization between them. We build a process $PATTERN$ which executes an event *race* when it performs one of these subtraces for a specific t_1,t_2 and v. The *race* event works as an indicator that a race has been found. The process $PATTERN$ is defined as:

$$
\begin{aligned}
PATTERN(t_1, t_2, v) = & PATT\_ERR(t_1, t_2, v) \\
& \triangle (\Box\ x : sync\_set \cup rw\_set@x \rightarrow PATTERN(t_1, t_2, v)) \\
rw\_set = & \{read.t_1.v, write.t_1.v, read.t_2.v, write.t_2.v\} \\
PATT\_ERR(t_1, t_2, v) = & read.t_1.v \rightarrow write.t_2.v \rightarrow race \rightarrow STOP \\
& \Box\ read.t_2.v \rightarrow write.t_1.v \rightarrow race \rightarrow STOP \\
& \Box\ write.t_1.v \rightarrow read.t_2.v \rightarrow race \rightarrow STOP \\
& \Box\ write.t_1.v \rightarrow write.t_2.v \rightarrow race \rightarrow STOP \\
& \Box\ write.t_2.v \rightarrow read.t_1.v \rightarrow race \rightarrow STOP \\
& \Box\ write.t_2.v \rightarrow write.t_1.v \rightarrow race \rightarrow STOP
\end{aligned}
$$

The $PATT\_ERR$ process is the combination of the six cases described, it can perform any of them. Only when one of the cases (one option of $PATT\_ERR$) is completed, the *race* event is fired. The process can be restarted in any state (with the $\triangle$ interrupt operator), to permit any other event combination being the prefix of the racy subtraces.

We verify a property in CSP describing it in terms of a refinement relationship $S \sqsubseteq_{\mathrm{T}} I$. A refinement relationship holds if the *behavior* of the implementation process I is a subset of the *behavior* of the specification process S. *Behavior* means the set of all possible traces. We compose the $PATTERN$ process in parallel with the $PROGRAM$ process, so the process $PATTERN$ participates in the execution of all events of $PROGRAM$. But $PATTERN$ is always available to perform any event, so it will not interfere with $PROGRAM$ process orderings. Using the hiding operator $\setminus$ all events become non-observable, with the exception of the *race* event, the one that reveals a matching pattern. We check if the resulting process refines the process $STOP$.

$$
STOP \sqsubseteq_{\mathrm{T}} (PROGRAM \underset{rw\_set \cup sync\_set}{\parallel} PATTERN(t_1, t_2, v)) \setminus \Sigma - \{race\}
$$

Σ denotes all the events in the model. If the process composed of $PROGRAM$ and $PATTERN$ reaches the event *race* then the refinement does not hold and there is a data race between t_1 and t_2 for variable v. But if there is no path to reach the *race* event, then the composition behaves exactly like $STOP$ and the refinement holds.

The corresponding model and assertions for the trace in listing 1.1 is:

$$
\begin{aligned}
THREAD_1 &= write.t_1.y \rightarrow lock.t_1.m \rightarrow write.t_1.x \rightarrow unlock.t_1.m \rightarrow SKIP \\
THREAD_2 &= lock.t_2.m \rightarrow write.t_2.x \rightarrow unlock.t_2.m \rightarrow write.t_2.y \rightarrow SKIP \\
PROGRAM &= (THREAD_1 \,|||\, THREAD_2) \underset{sync\_set}{\parallel} MUTEX(m)
\end{aligned}
$$

$$
STOP \sqsubseteq_{\mathrm{T}} (PROGRAM \underset{rw\_set \cup sync\_set}{\parallel} PATTERN(t_1, t_2, x)) \setminus \Sigma - \{race\} \quad (1)
$$

$$
STOP \sqsubseteq_{\mathrm{T}} (PROGRAM \underset{rw\_set \cup sync\_set}{\parallel} PATTERN(t_1, t_2, y)) \setminus \Sigma - \{race\} \quad (2)
$$

The refinement on x (1) holds but the refinement on y (2) not.

When the refinement is violated, at least one sequence of events in the composition leading to the *race* event exists. We focus only on the trace performed by the *PROGRAM* process, as the process representing the original program. The counterexample takes the form of a sequence of only the synchronization events. For the violated refinement (2) on x in the example, a counterexample is:

$$lock.t_2.m, unlock.t_2.m$$

If we replay the program only allowing the synchronizations operations in the counterexample and in the specified order, we reach a state where all actions performed by the threads are happening concurrently, exposing the data race. For example the case shown in listing 1.2.

3 Preliminary Evaluation

We implemented an automatic tool that generates the corresponding model for a given trace. Our target programs are binaries of multithreaded C programs using the pthread library [1]. We developed a Valgrind [9] plug-in to capture the trace, through a combination of binary instrumentation and library hooks. The trace is post-processed to simplify it and detect shared variables. For each shared variable, a CSP model is built ignoring the events of other variables so it contains the minimum number of events necessary. A refinement check is build for each combination of two threads and shared variable. The model and assertions are coded in CSP_M(machine readable CSP) and fed to the Failures-Divergences Refinement 3 model checker[4]. The evaluation has been done on a dual core machine with 1.4GHz processor and 1GB RAM.

Table 1. Preliminary evaluation

Scenarios	LOC	Real races	Trace size	Checks	Races	HG-Races
20	952	14	20289	28	14	11

Table 1 shows the preliminary evaluation, as the aggregate values of a set of small scenarios. The scenarios are a collection from multiple sources[13–15]. Scenarios from papers have been coded explicitly. Some scenarios are specially complicated for a happens-before detector to reason about, as the cases in [14]. The first column is the number of scenarios, the second column the aggregated lines of code. The third column is the total number of real races in all the scenarios (one per location). The fourth column is the total size (number of events) of the post-processed traces after a single execution. The fifth column is the number of checks (refinements) made. The sixth column shows how many races have been confirmed by the refinements and generated a counterexample. Finally the seventh column shows how many races Helgrind finds in average of 10 executions.

The results show that from a single execution our approach can find more races than Helgrind in multiple executions. A non-predictive race detector relies on reaching a specific timing to be able to see some races. But our solution is time-agnostic finding the races along the same path in the program.

Also race detectors usually provide only the localization of the race, but no information on when and how the program has reached that position. Mixing the race detector with interactive debugging can make the erroneous state difficult to reach, because the probe effect [3]. Our tool provides a step by step counterexample of the synchronization steps that can be use to reproduce the observed data race.

A pure happens-before tool cannot provide false positives. Although not shown in these examples, our approach can provide false positives; if a reordering of the synchronization operations leads to a different path in the program that has not been observed, the race could not exist and the counterexample is infeasible. To tackle this cases, we plan to prune this cases with automatic enforcement and checking of the counterexample. Also improving the model with control flow information as in [6] and implementing more synchronization primitives.

The scalability of the approach is limited by two factors: length of the trace and number of shared variables. A longer trace produces a more complex model, and the model checker needs more time. But our model increases in complexity only with the number of synchronization operations in the trace, which is expected to be a small fraction of the whole program trace. Further improvement can be done by partitioning the trace in windows and checking only one partition at a time. This increases the number of false negatives, as interactions between windows are lost. The number of checks also increases with the number of variables, but we can reduce this cost with a previous filtering step using a cheaper algorithm, that does not produce false negatives but reduces the number of candidates, e.g.: a relaxed happens-before without mutex edges.

4 Conclusion

This paper describes a work-in-progress approach to predict data races and generate a trace witness. We capture a single trace of a multithreaded application and model it in CSP. This model not only includes the observed interleaving but also the alternative reorderings of other possible executions. Using the capabilities of the process algebra we find data race patterns and generate the corresponding counterexamples. These counterexamples reflect how the program reached the erroneous states, and greatly facilitate the debugging process.

Acknowledgments. The authors would like to thank Siemens Corporate Technology for their financial support. We also appreciate the support of the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

1. Barney, B.L.L.N.L.: POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/

2. Flanagan, C., Freund, S.N.S.: FastTrack: efficient and precise dynamic race detection. In: PLDI 2009 Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 121–133. ACM, New York (2009). http://doi.acm.org/10.1145/1542476.1542490, http://dl.acm.org/citation.cfm?id=1542490
3. Gait, J.: A probe effect in concurrent programs. Software: Practice and Experience **16**(3), 225–233 (1986)
4. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014)
5. Hoare, C.: Communicating Sequential Processes. Communications of the ACM **21**(8), 666–677 (1978). http://www.cs.ucf.edu/courses/cop4020/sum2009/CSP-hoare.pdf
6. Huang, J., Meredith, P., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: PLDI 2014 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 337–348 (2014). http://dl.acm.org/citation.cfm?id=2594315
7. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2008, pp. 6:1–6:10. ACM, New York (2008). http://doi.acm.org/10.1145/1390841.1390847, http://www.cs.umd.edu/pugh/ISSTA08/padtad2008/papers/a8-jannesari.pdf
8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978). http://dl.acm.org/citation.cfm?id=359563
9. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamicbinary instrumentation. ACM Sigplan Notices, 89–100 (2007). http://dl.acm.org/citation.cfm?id=1250746
10. Pozniansky, E., Schuster, A.: MultiRace: efficient on the fly data race detection in multithreaded C++ programs. Concurrency and Computation: Practice and Experience **19**(3), 327–340 (2007). http://onlinelibrary.wiley.com/doi/10.1002/cpe.1064/abstract
11. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating data race witnesses by an SMT-based analysis. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 313–327. Springer, Heidelberg (2011)
12. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems **15**(4), 391–411 (1997). http://doi.acm.org/10.1145/265924.265927, http://portal.acm.org/citation.cfm?doid=265924.265927
13. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: data race detection in practice. In: WBIA 2009 Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71 (2009). http://dl.acm.org/citation.cfm?id=1791203
14. Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, p. 387 (2012). http://dl.acm.org/citation.cfm?doid=2103656.2103702
15. Valgrind: Helgrind: a data-race detector (2007). http://valgrind.org/docs/manual/hg-manual.html