Static Data Race Detection via Lazy Sequentialization

Bernd Fischer[1], Giulio Garbi[2,3]*, Salvatore La Torre[3], Gennaro Parlato[2], and Peter Schrammel[4]

[1] Stellenbosch University, Stellenbosch, South Africa, `bfischer@sun.ac.za`
[2] University of Molise, Pesche, Italy, `giulio.garbi/gennaro.parlato@unimol.it`
[3] University of Salerno, Fisciano, Italy, `ggarbi/slatorre@unisa.it`
[4] Diffblue Ltd, Oxford, UK `schrammelp@gmail.com`

Abstract. We present a new symbolic static data race detection algorithm, which is defined as a code-to-code translation that injects code to monitor any accesses to shared memory locations. We implemented this algorithm in the LaDR tool as an extension of an existing lazy sequentialization schema which works well when used in tandem with bounded model checkers. We evaluated LaDR on the benchmarks from the data race demonstration category of SV-COMP 2024, and on `safestack`, a lock-free data structure that contains a rare ABA-related bug. LaDR finds more data races than all other tools participating in SV-COMP 2024, and is the only tool that can find a data race in `safestack`.

Keywords: Data races · Static analysis · Lazy sequentialization

1 Introduction

A *data race* occurs if different computations in a concurrent system access the same memory location "at the same time" (i.e., in arbitrary order but immediately following each other) and at least one of the two accesses is a write access. Bugs caused by data races are often extremely difficult to detect manually, due to their non-deterministic nature; therefore, a large number of different data race detection tools have been developed to help developers in debugging.

Traditionally, most tools use *dynamic* techniques, where the target program is instrumented and executed to produce an execution which is then analyzed for races; for example, Eraser [24] tracks the set of locks held for each memory access, while ThreadSanitizer [26] tracks the order of memory accesses and synchronization operations throughout the program execution. Since they observe real program executions over concrete inputs, dynamic techniques only analyze feasible behaviors, and all identified data races can happen in practice (i.e., they report no *false positives*). However, they may miss data races (i.e., report *false negatives*) as a result of poor test suites.

Static techniques construct and analyze an execution model without actually executing the target program. In principle, static techniques can be *sound* (i.e.,

* Corresponding author

never report false positives) or *complete* (i.e., never report false negatives), but in practice precise execution models become too complex for the analysis, and the techniques often use a number of different abstraction methods, balancing soundness and completeness. For example, Locksmith [22] uses lock state analysis, sharing analysis, correlation inference, and escaping and linearity checks.

SAT-based symbolic execution and model checking convert a program into a model in form of a logical formula that is satisfiable if and only if the program violates a given specification. Their performance has improved substantially in the recent past. Retargeting them to data race detection has thus become a viable alternative. We follow this approach here and describe a precise data race detection tool based on bounded model checking; more specifically, we develop and implement on top of the Lazy-CSeq sequentialization [12,13] a code-to-code translation schema that rewrites a concurrent program into a bounded non-deterministic sequential program such that data races can be detected by assertion checking. This sequentialized program is processed by a backend verification tool that constructs a logical formula that is satisfiable iff the original program contains a data race. The non-determinism in the original program is reflected in underconstrained variables in the constructed formula.

Our schema uses a flag for each shared memory location to indicate that location is involved in a data race, and injects control code that manipulates these flags. The flags are non-deterministically set on a write (reflecting the fact that this could be the write-part of the race), and checked on each other write or read (reflecting the fact that this could be the second part of the race). The non-determinism simplifies our design because we do not need to maintain complex data structures, and can be handled efficiently by the SAT solver.

We implemented this schema in the LaDR data race detection tool for C programs that use the `pthreads`-library as concurrency mechanism, using the bounded model-checker CBMC [8,4] as backend verifier. LaDR is sound but incomplete, due to the bounding of programs, but it performs very well over the SV-COMP 2024 data race benchmarks. It finds the highest number (183 out of 232) of data races, outperforming the three best performing tools in this category at SV-COMP 2024. It is also the only tool that can find a data race in `safestack`, a lock-free data structure that contains a rare ABA-related bug.

2 Bounded Multi-Threaded Programs

Like bounded model checking [4], we focus on bug detection. We thus work with *bounded* programs, which are syntactically guaranteed to terminate after a bounded number of steps, e.g., through the absence of loops, recursion, and `goto`-statements representing backward jumps. Unbounded programs can be bounded through a series of simple code-to-code transformations such as loop unwinding, function inlining, and function cloning [6].

LaDR supports the full C language, but we illustrate our code-to-code translation schema using a simple, multi-threaded imperative language with mutex locking and unlocking operations for thread synchronization. Its grammar is

$$
\begin{array}{lcl}
P & ::= & (\mathit{dec}\,\texttt{;})^{*}\ (\texttt{void}\, f_i\ \texttt{()\{}(\texttt{static?}\ \mathit{dec}\,\texttt{;})^{*}\ \mathit{stm}\texttt{\}})_{i=0,\ldots,n} \\
\mathit{dec} & ::= & \mathit{typ}\ v \mid \texttt{thread}\ t \mid \texttt{mutex}\ m \\
\mathit{typ} & ::= & \mathit{typ}\texttt{*} \mid \mathit{typ}\,\texttt{[}\mathit{exp}\texttt{]} \mid \texttt{int} \\
\mathit{stm} & ::= & \mathit{seq}\,\texttt{;} \mid \mathit{con}\,\texttt{;} \mid \texttt{\{}\mathit{stm}^{+}\texttt{\}} \\
\mathit{seq} & ::= & \texttt{assume(}\mathit{exp}\texttt{)} \mid \texttt{assert(}\mathit{exp}\texttt{)} \mid \texttt{goto}\ l \mid l\texttt{:skip} \mid \texttt{if(}\mathit{exp}\texttt{)}\ \mathit{stm}\ \texttt{else}\ \mathit{stm} \mid \mathit{exp} \\
\mathit{con} & ::= & n : \mathit{stm} \mid \texttt{return} \mid t\,\texttt{=}\,\texttt{create}\ f_i\texttt{()} \mid \texttt{join}\ t \mid \texttt{lock}\ m \mid \texttt{unlock}\ m \\
\mathit{exp} & ::= & \mathit{lval}\,\texttt{=}\,\mathit{exp} \mid \mathit{exp}\ \texttt{,}\ \mathit{exp} \mid \mathit{exp}\ \texttt{OP}\ \mathit{exp} \mid \mathit{exp}\ \texttt{?}\ \mathit{exp}\ \texttt{:}\ \mathit{exp} \mid \texttt{\&}\mathit{lval} \mid \mathit{lval} \mid \mathit{const} \mid \texttt{(*)} \\
\mathit{lval} & ::= & \mathit{lval}\ \texttt{[}\mathit{exp}\texttt{]} \mid \texttt{*}\mathit{lval} \mid v
\end{array}
$$

Fig. 1. Syntax of bounded multi-threaded programs

shown in Fig. 1; here m denotes a mutex, t a thread variable, v a program variable, l a non-numeric label, and n a numeric label. Program variables can be integers, pointers, or (multidimensional) arrays, in arbitrary combination.

A *bounded multi-threaded program* comprises a list of declarations of *global* (or *shared*) variables and a list of parameterless *thread functions* f_i that in turn each comprise a list of *local* variable declarations and a (block) statement *stm*.

Shared and `static` local program variables, including pointers and individual array elements, are initialized to `0`; non-static ones remain uninitialized and non-deterministically assume an arbitrary value (denoted by `(*)`) until they are explicitly assigned. Mutexes must be declared globally and are initially free.

A *sequential statement seq* can be an `assume`- or `assert`-statement, a conditional statement, a `goto`- or a labeled `skip`-statement, which serves as jump target, or an expression, which is evaluated for its side effects. A *concurrent statement con* can be a labeled statement of the form $n : stm$ where *stm* is either an unlabeled concurrent statement or a sequential statement that accesses the shared memory, a `return`-statement, a thread creation or join operation, or a mutex lock or unlock operation. A thread creation statement t `= create` $f_i()$ spawns a new thread from f_i. A thread join statement `join` t suspends the executing thread until the thread identified by t has executed its last statement. Thread variables must be initialized by a `create`-statement before they can be accessed by a `join`-statement. Lock and unlock statements respectively acquire and release a mutex, with the usual blocking behavior.

Expressions exp can contain side effects in the form of *assignment expressions* v `=` e, which compute the memory location denoted by the lvalue x, evaluate e and write its value into the computed location, and return the value of e. *Comma expressions* e_1 `,` e_2 explicitly sequence expression evaluation, i.e., they first evaluate e_1 for its side effects but discard its value, then evaluate e_2 and return its value. We also model arbitrary strict operators `OP`, lazy evaluation in the form of *conditional expressions* e_1 `?` e_2 `:` e_3, which first evaluate the guard expression e_1 and, depending on the result, evaluate and return either e_2 or e_3, and the address-of operator `&`.

We assume that a valid program P satisfies the usual well-formedness conditions, and that its execution does not involve the application of any operator to illegal arguments (e.g., division by zero). The last statement in the body of each function must be its single `return`-statement. All concurrent statements in a function as well as its first and last statement must be labeled with a numerical label n, such that the labels in each function start from 0 and increase by 1 according to the statement order; any other label of the program must be non-numerical. We call any statement with a numerical label a *visible statement*. This structure is required by the lazy sequentialization to simulate context-switching and can be established easily by code-to-code transformations.

In the *interleaving semantics*, a concurrent execution is defined as the interleaving of the individual thread executions. A sequence of consecutively executed statements from one thread is called a *context*. A *round* contains one (possibly empty) context from each thread. We assume that all threads are scheduled in each round in the same fixed order; other schedules can be simulated using additional rounds and empty contexts.

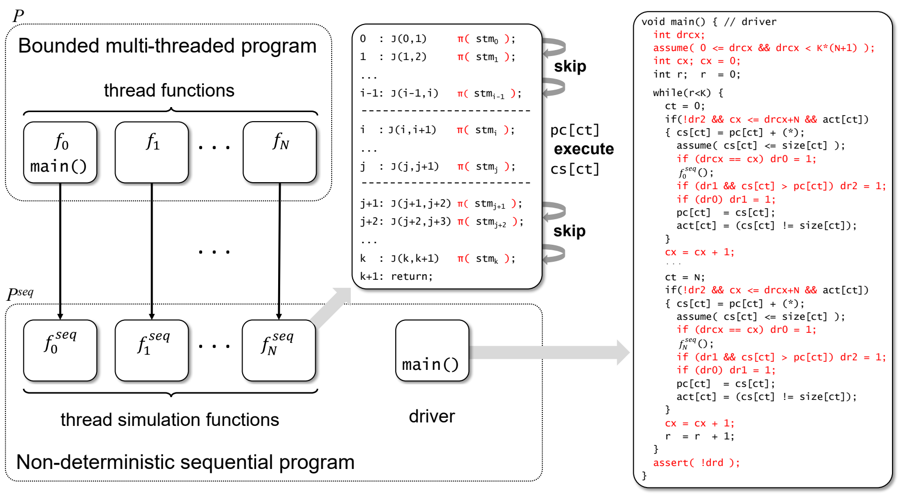
A *data race* occurs in a bounded multi-threaded program P if there is an execution in P where two different threads can access the same shared memory location at the "same time" and at least one of these is a write to this shared location. In the interleaving semantics, a data race is captured if these two accesses occur respectively at the end (i.e., in the last executed visible statement) of one context and at the beginning (i.e., in the first executed visible statement) of the following context in the computation. Since these two accesses are not synchronized there is always another computation that interleaves the two accesses in the reversed order but still keeps one after the other (possibly by increasing the number of contexts). In the following, we thus assume without loss of generality that the write occurs always first in the interleaving.

3 Lazy Sequentialization

Sequentialization [23,17] is a general program analysis technique that transforms concurrent programs into non-deterministic sequential programs such that the reachability of program states is preserved. It can be implemented as a code-to-code translation, which allows us to build tools for the analysis of multi-threaded programs by reusing existing tools for sequential programs.

The *lazy sequentialization* translation (LS) [12,13] has been specifically designed to work well in combination with SAT-based bounded model checking tools (in particular CBMC), and thus fits our approach well. LS was designed for reachability checking, and does not directly support data race detection. The main contribution of this paper is to extend LS to detect data races efficiently through the use of a shadow memory (see Section 4). This section gives an overview of the aspects of LS that are relevant to this extension. Fig. 2 illustrates the complete translation; the code fragments in red are added for the data race detection and can be ignored for now.

We assume that P is a bounded multi-threaded program with $N+1$ thread functions $f_0, f_1, \ldots, f_N$, where f_0 denotes P's `main` function, and that it contains N `create`-statements using each $f_1, \ldots, f_N$ exactly once as thread function; P



```
0  : J(0,1)       π( stm0 );
1  : J(1,2)       π( stm1 );
...
i-1: J(i-1,i)     π( stmi-1 );
------------------------
i  : J(i,i+1)     π( stmi );
...
j  : J(j,j+1)     π( stmj );
------------------------
j+1: J(j+1,j+2) π( stmj+1 );
j+2: J(j+2,j+3) π( stmj+2 );
...
k  : J(k,k+1)     π( stmk );
k+1: return;
```

```
void main() { // driver
  int drcx;
  assume( 0 <= drcx && drcx < K*(N+1) );
  int cx; cx = 0;
  int r;  r  = 0;

  while(r<K) {
    ct = 0;
    if(!dr2 && cx <= drcx+N && act[ct])
    { cs[ct] = pc[ct] + (*);
      assume( cs[ct] <= size[ct] );
      if (drcx == cx) dr0 = 1;
      f0seq();
      if (dr1 && cs[ct] > pc[ct]) dr2 = 1;
      if (dr0) dr1 = 1;
      pc[ct]  = cs[ct];
      act[ct] = (cs[ct] != size[ct]);
    }
    cx = cx + 1;
    ...
    ct = N;
    if(!dr2 && cx <= drcx+N && act[ct])
    { cs[ct] = pc[ct] + (*);
      assume( cs[ct] <= size[ct] );
      if (drcx == cx) dr0 = 1;
      fNseq();
      if (dr1 && cs[ct] > pc[ct]) dr2 = 1;
      if (dr0) dr1 = 1;
      pc[ct]  = cs[ct];
      act[ct] = (cs[ct] != size[ct]);
    }
    cx = cx + 1;
    r  = r  + 1;
  }
  assert( !drd );
}
```

Fig. 2. Code-to-code translation overview

thus spawns at most N threads. The corresponding sequential program P^{seq} is formed by a new function `main` and a *thread simulation function* f_i^{seq} for each thread function f_i in P. LS assumes the interleaving semantics, and simulates the programs in a round-robin fashion up to a given number of rounds $\texttt{K} > 0$.

We use `pc` and `cs` to store the numerical labels respectively at the beginning and the end of each context; `size` to store the largest numerical label for each thread function f_i; `act` to track the active (i.e., created but not yet terminated) threads (initially only `act[0]` is set to `true`, as f_0^{seq} corresponds to the `main` function of P); and `ct` to store the numerical identifier of the thread being currently simulated.

Main Driver. In each loop iteration the new `main` simulates a (possibly empty) context of each active thread $f_{\texttt{ct}}$ of P by calling the corresponding simulation function $f_{\texttt{ct}}^{seq}$ for $\texttt{ct} \in [0, N]$ (see Fig. 2). Specifically, for each active thread, it

– non-deterministically guesses the numerical label of the next context-switch and stores it in `cs[ct]`;
– ensures that this value is within the appropriate bounds;
– calls the thread simulation function $f_{\texttt{ct}}^{seq}$ to simulate the thread `ct` from `pc[ct]` through to `cs[ct]`;
– stores `cs[ct]` in `pc[ct]` (which is used in the next round to resume the computation); and
– sets `act[ct]` to `false` if `cs[ct]` is the label of the `return`-statement, which means that the simulation of the thread is terminated.

Thread Translation. The main driver repeatedly calls each $f_{\texttt{ct}}^{seq}$ to simulate the non-deterministically selected context. Each $f_{\texttt{ct}}^{seq}$ must therefore (i) maintain the thread-local state between the consecutive calls, and (*ii*) execute *only* the statements between `pc[ct]` and `cs[ct]`. The first is achieved simply by changing the storage class of the thread-local variables to `static` (which is more efficient than snapshotting the thread memory), while the second is achieved by injecting control code that suppresses the execution of statements not within the range. More specifically, the control code forces $f^{seq}_{\texttt{ct}}$ to jump over each individual visible statement whose label is smaller than the stored label of the current context's resumption point or larger than its end point. This mechanism can be implemented in a way that avoids complex branching in the control flow by exploiting the consecutive natural numbers used as labels on the visible statements, and so leads to simpler formulas. Specifically, right after each numerical label *i* (except for the last one), a conditional jump of the form

```
if(pc[ct]>i || i>=cs[ct]) goto j;                (J macro)
```

is injected in front of the original statement. Note that the injected condition becomes false iff the control is between `pc[ct]` and `cs[ct]`, so that exactly the same statements as in the original thread are executed by $f^{seq}_{\texttt{ct}}$. The translation always calls the `J`-macro with literal arguments *i* and $i + 1$ (see Fig. 2). In effect, each $f^{seq}_{\texttt{ct}}$ jumps (in multiple hops) to the saved position `pc[ct]` in the code, resumes its execution until the visible statement with the label `cs[ct]` of the next context-switch is reached, and jumps (again in multiple hops) to the final `return`-statement. Fig. 2 illustrates this.

We omit details of LS that are not essential for our data race instrumentation, including the handling of control-flow branching and the simulation of synchronization routines. These details can be found in [12,13].

4 Symbolic Data Race Detection Schema

In this section, we describe our LaDR data race detection scheme. It is implemented as a code-to-code translation on top of LS (see Section 3) and uses the symbolic shadow memory API provided by CBMC-SSM. We first give a high-level overview of our translation before we discuss details of the extension to LS (shown in red in Figure 2).

Overview. In the interleaving semantics, we can split a data race detection into two distinct phases (cf. Section 2):

Phase 1: We non-deterministically select a context C_1 in which the execution of the last visible statement of C_1 updates a shared location ℓ.

Phase 2: We check whether the first visible statement of the next (non-empty) context C_2 of another thread accesses ℓ; if so, a data race involving ℓ occurs.

In contrast to deterministic methods, which manipulate large and complex data structures that store the status of each memory location in order to detect data races, we rely on the schedule non-determinism, which frees us from having to encode these data structures at the formula level, and thus results in a more scalable solution.

We implement the two phases on top of LS by injecting *control code* into the main driver and *rewriting* the statements and expressions in the thread functions. The statement rewriting is represented in Fig. 2 by the function π described below. The injected control code implements a finite state machine that identifies the contexts C_1 and C_2, and tracks the phases. Statement rewriting injects code in the form of comma expressions to set the shadow memory bit for each update to the shared memory when in Phase 1, and to query the shadow memory for each access to the shared memory and to possibly flag that a data race has occurred when in Phase 2.

We use the following auxiliary global variables which are set according to the following invariants:

`dr0` is set to `true` right before C_1 is simulated, indicating that Phase 1 started;
`dr1` is set to `true` right after C_1 has been simulated, indicating that Phase 1 is completed;
`dr2` is initialized to `false` and set to `true` as soon as C_2 is executed, indicating that Phase 2 is completed; and
`drd` is initialized to `false` and set to `true` when a data race is detected.

Only `drd` is modified in the thread simulation functions while `dr0`, `dr1`, and `dr2` are only updated in the main driver. This keeps the generated formulas compact.

Shadow Memory. In the LaDR scheme, we need to determine whether the shared memory location at a given address has been updated by a thread or not. For efficiency reasons, we keep this information as flags in a *shadow memory*, which is a separate data structure that is invisible to the analyzed program but can be accessed efficiently through its original memory addresses.

More specifically, we use the shadow memory as implemented in CBMC-SSM [8], an extension of CBMC [4]. Its shadow memory associates a *bit field* with each byte of memory. We assume that all bit fields are initialized to `false` and then manipulate them via the following API functions:

`set_sm(`*addr*`,` *val*`)` sets to *val* the bit fields of all memory bytes whose memory addresses are between *addr* and *addr* + `sizeof(`T`)`, where T is the type of the object at *addr*.
`get_sm(`*addr*`)` returns the disjunction of all bit field values corresponding to all memory locations whose addresses are between *addr* and *addr* + `sizeof(`T`)`.

Main Driver. We now describe the changes made to the main driver of LS. Since we simulate K contexts for each of the $N + 1$ threads, we identify contexts by integers from 0 to $K \cdot (N + 1) - 1$, increasing in the order in which they are simulated. We declare and constrain a fresh local variable `drcx`, to non-deterministically select the context C_1 through its numerical identifier. The simulation follows the same line as LS, with the difference that a thread is simulated only if it is active, *and* Phase 2 has not been completed yet (i.e., `!dr2` holds). We also require that at most N threads have been scheduled since Phase 1 has been completed (i.e., `cx<=(drcx+N)` holds). This prevents the main driver from executing a function when Phase 1 is completed but Phase 2 has not started within the following N contexts, which allows us to abort schedules that can be captured by others, and avoids false positives (i.e., write and read are contained in the same thread). A final `assert`-statement is injected into the main driver to force failure when a data race is found. The extra code in the main driver maintains the invariants stated above for the auxiliary global variables.

Thread Translation. The thread simulation functions keep the same structure as in LS, since the extension only adds code (in form of comma expressions) at the level of expressions. This is captured by the translation function π in Fig. 2.

Translation Modes. An expression may require different translations, depending on its usage in the code, similar to but not exactly following the usual lvalue/rvalue distinction. For example, the occurrence of a shared variable x requires the injection of data race detection code if the *content* of its memory location is accessed in a read (e.g., in an arithmetic expression $x + y$) or write operation (e.g., in an assignment of the form $x = 0$) but not if only the address of its location is accessed (e.g., in an address-operation $\&x$). Moreover, when we rewrite expressions, we may need to duplicate sub-expressions, but without duplicating any side effects (i.e., memory writes) they can have. We achieve this in the usual way by hoisting the evaluation of side-effecting sub-expressions into comma expressions. We distinguish the four different modes below, and use $[\![e, M]\!]$ to denote the translation of an expression e in mode M:

ACC indicates that the value of the expression is used to access a memory location; if it is an lvalue, it may thus be involved in a data race and we instrument it with data race detection code.

NACC indicates that the value of the expression is not used to access the memory; if the expression itself is an lvalue, we thus do not instrument it with data race detection code. However, its sub-expressions may be instrumented.

WSE is used to access the value of the expression, without executing its side effects, if any. We require that any side effects have already been hoisted out and executed in the preceding part of a comma expression. If the original expression is an lvalue, the resulting expression is also an lvalue that identifies exactly the same memory location as the original one.

PRE is only used to handle the case of the translation of array expressions; it is used essentially to keep track of the ACC/NACC mode in the translation of the sub-expressions.

The Translation Function π. Function π is composed of two parts. The first part, which is injected only for visible statements, checks whether the execution is at the end of the non-deterministically guessed context C_1 (for Phase 1) or at

$$\begin{array}{ll}
[\![\texttt{assume(}e\texttt{)}]\!] & \mathrel{\widehat{=}} \texttt{assume((}[\![e,\mathsf{ACC}]\!]\texttt{ , }[\![e,\mathsf{WSE}]\!]\texttt{))} \\
[\![\texttt{assert(}e\texttt{)}]\!] & \mathrel{\widehat{=}} \texttt{assert((}[\![e,\mathsf{ACC}]\!]\texttt{ , }[\![e,\mathsf{WSE}]\!]\texttt{))} \\
[\![e]\!] & \mathrel{\widehat{=}} [\![e,\mathsf{ACC}]\!] \\
[\![\texttt{goto }l]\!] & \mathrel{\widehat{=}} \texttt{goto }l \\
[\![l\texttt{:skip}]\!] & \mathrel{\widehat{=}} l\texttt{:skip} \\
[\![\texttt{if(}e\texttt{) }\mathit{stm}\texttt{ else }\mathit{stm}]\!] & \mathrel{\widehat{=}} \texttt{if((}[\![e,\mathsf{ACC}]\!]\texttt{ , }[\![e,\mathsf{WSE}]\!]\texttt{))}\ [\![\mathit{stm}]\!]\ \texttt{else}\ [\![\mathit{stm}]\!] \\
[\![\texttt{return}]\!] & \mathrel{\widehat{=}} \texttt{return}
\end{array}$$

Fig. 3. Translation of statements

the begin of the context C_2 (for Phase 2). The result of these checks are stored in the global variables `P1` and `P2`. The second part (denoted by $[\![\mathit{stm}]\!]$) injects the data race control code into the expressions occurring in the statements *stm*; Fig. 3 illustrates this transformation down to the level of the expressions (which include assignments).

Hence, for a visible statement *stm* with the numerical label j of a thread i, $\pi(\mathit{stm})$ is defined as follows:

```
P1 = (j == cs[i]-1) && dr0 && !dr1;
P2 = (j == pc[i])   && dr1;
⟦stm⟧
```

The conditions `j == cs[i]-1` and `j == pc[i]` hold if j is the label of the last and the first visible statement of the current context of thread i, respectively. The other conjuncts in the assignment for `P1` identify the current context of i as C_1 since `dr0 && !dr1` means that Phase 1 has started but not finished yet. Similarly, `dr1` identifies the current context of i as C_2; in fact, as soon as the control returns to the main driver, `dr2` is set to true and no more contexts are simulated (see Fig. 2). `P1` and `P2` are set to `false` at any other visible statements. For non-visible statements, no code is injected and the statement is left unchanged.

The data race detection code that we inject differs for the two phases. In Phase 1, we update the shadow memory and inject code of the form

$$\texttt{P1 \&\& set\_sm(\&}[\![e,\mathsf{WSE}]\!]\texttt{, true)} \qquad \text{(Ph1)}$$

whenever there is a write access to the shared memory location corresponding to e. Note that this relies on C's short-circuit evaluation of the conjunctions, so that the second conjunct is not evaluated if `P1` is false and thus the shadow memory is (correctly) not updated if the statement is not at the end of the non-deterministic selected context C_1.

In Phase 2, we query the shadow memory to update the variable `drd`, and inject code of the form

$$\texttt{P2 \&\& (drd = (drd || get\_sm(\&}[\![e,\mathsf{WSE}]\!]\texttt{)))} \qquad \text{(Ph2)}$$

whenever there is a read or write access to the shared memory location corresponding to e. Again, due to short-circuit evaluation, `drd` is updated only if

P2 holds (i.e., the execution is at the beginning of context C_2); likewise, drd is never updated once it has been set to true. Hence, drd is set to true only if a write access to the location is recorded in the shadow memory. However, since the shadow memory is updated only at the end of the context C_1, this implies that the write access and the second access can happen immediately adjacently in the multi-thread program, i.e., that there is a data race on this location.

Expression Translation. Fig. 4 shows the detailed expression translation rules. For simplicity, we assume that all lvalues identify shared memory locations; for local memory locations, the data race detection code of the form (Ph1) and (Ph2) must be omitted from the translation rules.

ACC and WSE are the only translation modes that are used by the statement translation (see Figure 3). Consequently, these are the only modes where we start the expression translation. The NACC mode comes into play only when no access to the shared memory can occur through the corresponding lvalue (e.g., the operand of a reference expression), or when this is accounted for elsewhere by the translation (e.g., the left-hand side of an assignment). The PRE mode only occurs in the translation of prefixes of array expressions. The cases not shown in Fig. 4 cannot occur in well-formed expressions.

Fig. 4(a) shows the translation for assignment expressions, i.e., expressions of the form e_1 = e_2. In all modes but WSE, the translation unfolds as follows:

1. We generate the side effects of the left part e_1 in NACC mode. Here, the write access to the shared memory is annotated in the next part of the rewriting process, eliminating the need to reconsider this location for data race detection purposes.
2. We then generate the side effects of the right part e_2 in ACC mode, accounting for potential accesses to the shared memory.
3. If in Phase 1, we annotate the shadow memory of the assigned location; for this we use an expression of the form (Ph1).
4. If in Phase 2 and a write access to e_1 was already annotated into the shadow memory, we update drd; for this we use an expression of the form (Ph2).
5. Finally, we simulate the assignment without generating further side effects.

In WSE mode, we just rewrite the expression to $[\![e_1, \mathsf{WSE}]\!]$ since all side effects of the original expression have been accounted for at this point.

Fig. 4(b) shows the translation for the comma expressions of the form e_1 , e_2. In ACC and PRE, we rewrite the entire expression again as a comma expression, with the first sub-expression being e_1 rewritten in ACC mode to account for potential accesses to shared memory. The second sub-expression, e_2, is rewritten in the same mode as the original comma expression. In WSE mode, we simply rewrite the comma expression as e_2 in WSE mode.

Fig. 4(c) shows the translation for expressions the form e_1 OP e_2. These expressions are rewritten as expressions of the same type. Specifically, in ACC and WSE mode, we pass on the sub-expression the same mode since the overall value is computed by the values of the two expressions. Thus, we rewrite both e_1 and e_2 respectively in ACC mode and WSE mode. In PRE mode, we adopt the same

(a)	$\llbracket e_1 \texttt{ = } e_2, \mathsf{ACC/NACC/PRE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{NACC}\rrbracket\texttt{ , }\llbracket e_2, \mathsf{ACC}\rrbracket\texttt{ , P1 \&\& set\_sm(\&}\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{, true) ,}$ $\texttt{P2 \&\& drd = (drd \textbar\textbar{} get\_sm(\&}\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{)) , }\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{ = }\llbracket e_2, \mathsf{WSE}\rrbracket$
	$\llbracket e_1 \texttt{ = } e_2, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{WSE}\rrbracket$
(b)	$\llbracket e_1 \texttt{ , } e_2, \mathsf{ACC}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ , }\llbracket e_2, \mathsf{ACC}\rrbracket$
	$\llbracket e_1 \texttt{ , } e_2, \mathsf{PRE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ , }\llbracket e_2, \mathsf{PRE}\rrbracket$
	$\llbracket e_1 \texttt{ , } e_2, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\llbracket e_2, \mathsf{WSE}\rrbracket$
(c)	$\llbracket e_1 \texttt{ OP } e_2, \mathsf{ACC/PRE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ OP }\llbracket e_2, \mathsf{ACC}\rrbracket$
	$\llbracket e_1 \texttt{ OP } e_2, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{ OP }\llbracket e_2, \mathsf{WSE}\rrbracket$
(d)	$\llbracket e_1 \texttt{ ? } e_2 \texttt{ : } e_3, \mathsf{ACC}\rrbracket$	$\widehat{=}$	$\texttt{(}\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ , }\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{) ? }\llbracket e_2, \mathsf{ACC}\rrbracket\texttt{ : }\llbracket e_3, \mathsf{ACC}\rrbracket$
	$\llbracket e_1 \texttt{ ? } e_2 \texttt{ : } e_3, \mathsf{PRE}\rrbracket$	$\widehat{=}$	$\texttt{(}\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ , }\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{) ? }\llbracket e_2, \mathsf{PRE}\rrbracket\texttt{ : }\llbracket e_3, \mathsf{PRE}\rrbracket$
	$\llbracket e_1 \texttt{ ? } e_2 \texttt{ : } e_3, \mathsf{NACC}\rrbracket$	$\widehat{=}$	$\texttt{(}\llbracket e_1, \mathsf{ACC}\rrbracket\texttt{ , }\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{) ? }\llbracket e_2, \mathsf{NACC}\rrbracket\texttt{ : }\llbracket e_3, \mathsf{NACC}\rrbracket$
	$\llbracket e_1 \texttt{ ? } e_2 \texttt{ : } e_3, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{ ? }\llbracket e_2, \mathsf{WSE}\rrbracket\texttt{ : }\llbracket e_3, \mathsf{WSE}\rrbracket$
(e)	$\llbracket e_1 \texttt{[} e_2 \texttt{]}, \mathsf{ACC}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{PRE}\rrbracket\texttt{ , }\llbracket e_2, \mathsf{ACC}\rrbracket\texttt{ ,}$ $\texttt{P2 \&\& drd = (drd \textbar\textbar{} get\_sm(\&}\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{ [}\llbracket e_2, \mathsf{WSE}\rrbracket\texttt{]))}$
	$\llbracket e_1 \texttt{[} e_2 \texttt{]}, \mathsf{NACC/PRE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{PRE}\rrbracket\texttt{ , }\llbracket e_2, \mathsf{ACC}\rrbracket$
	$\llbracket e_1 \texttt{[} e_2 \texttt{]}, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\llbracket e_1, \mathsf{WSE}\rrbracket\texttt{[}\llbracket e_2, \mathsf{WSE}\rrbracket\texttt{]}$
(f)	$\llbracket v, \mathsf{ACC/PRE}\rrbracket$	$\widehat{=}$	$\texttt{P2 \&\& drd = (drd \textbar\textbar{} get\_sm(\&}v\texttt{))}$
	$\llbracket v, \mathsf{NACC}\rrbracket$	$\widehat{=}$	$\texttt{(void) 0}$
	$\llbracket v, \mathsf{WSE}\rrbracket$	$\widehat{=}$	v
(g)	$\llbracket \texttt{*}e, \mathsf{ACC/PRE}\rrbracket$	$\widehat{=}$	$\llbracket e, \mathsf{ACC}\rrbracket\texttt{ , P2 \&\& drd = (drd \textbar\textbar{} get\_sm(}\llbracket e, \mathsf{WSE}\rrbracket\texttt{))}$
	$\llbracket \texttt{*}e, \mathsf{NACC}\rrbracket$	$\widehat{=}$	$\llbracket e, \mathsf{ACC}\rrbracket$
	$\llbracket \texttt{*}e, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\texttt{*}\llbracket e, \mathsf{WSE}\rrbracket$
(h)	$\llbracket \texttt{\&}e, \mathsf{ACC/NACC}\rrbracket$	$\widehat{=}$	$\llbracket e, \mathsf{NACC}\rrbracket$
	$\llbracket \texttt{\&}e, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$\texttt{\&}\llbracket e, \mathsf{WSE}\rrbracket$
(i)	$\llbracket const, \mathsf{ACC/NACC}\rrbracket$	$\widehat{=}$	$\texttt{(void) 0}$
	$\llbracket const, \mathsf{WSE}\rrbracket$	$\widehat{=}$	$const$

Fig. 4. Translation of expressions

rewriting as in ACC mode, as it represents the scenario when the bottom of the recursion is reached during the rewriting of an array expression.

Fig. 4(d) shows the translation for the conditional expression. In all modes but WSE, we replace expression e_1 with a comma expression in which we evaluate e_1 in ACC mode to accommodate potential memory access within e_1, thus ensuring the production of side effects. Then, we generate the same value as the original expression by a rewriting it in WSE mode. For e_2 and e_3 we rewrite them by keeping the starting translation mode. In WSE mode, we just pass the translation function in WSE mode to the three sub-expressions.

Fig. 4(e) shows the translation for *array expressions* of the form $e_1\texttt{[}e_2\texttt{]}$. According to the grammar given in Fig.1, array expressions are generated by yielding an index at each step and then recursing over the prefix. At the bottom of the recursion an expression evaluating to the starting address of the array elements is generated. This expression gives an lvalue, which can lead to an access to the memory if the expression is not constant. Furthermore, the evaluation of each index can lead to an access to the memory while the intermediate prefixes do not. The translation rules are designed accordingly. In the ACC mode, we translate e_1 (i.e., the prefix of the array expression) in PRE mode, then e_2 (i.e., the index generated at the current recursion step) in ACC mode, and finally we inject code of the form (Ph2) to check a possible data race on e_1`[`e_2`]`. In NACC and PRE modes, e_1`[`e_2`]` does not count as an access to the shared memory and thus the portion of code to check for a possible data race is omitted. In WSE mode, we just pass the translation function in WSE mode to the two sub-expressions.

Fig. 4(f) shows the translation for *identifier expressions* v, e.g., the name of a scalar or pointer variable. Since this cannot have any side-effects, we return in WSE mode the original identifier. In ACC mode we just need to update `drd` by injecting code of the form (Ph2). The same code is injected in PRE mode. In the NACC mode we rewrite v to `(void) 0` (recall that NACC means that v is not used to access the data).

Fig. 4(g-h) shows the translation for the expressions of the forms `*`e and `&`e. In the WSE mode, we simply apply the translation function recursively over the operator. The PRE mode may occur only for `*`e, and for these expressions we apply the same translation both in PRE and ACC modes: first we generate the side-effects of e in ACC mode (since e may contain pointer variables), and then we update `drd` by checking Phase 2 for the location of `*`e. In all remaining cases, we just generate the side-effects of e. For expressions of the form `&`e, the rewriting of e is done in NACC mode since in these expressions the lvalue yield by e is not used to access memory locations.

Fig. 4(i) shows the translation for *constant expressions const*. Since this expression has no side effects and does not contribute to data races, we rewrite it to `(void) 0` in both ACC and NACC modes and to *const* itself in WSE mode.

The given translation rules can be safely simplified by removing the rewriting of a sub-expression in ACC, NACC, and PRE modes if we can deem that there are no side-effects and no shared memory location is accessed. We have adopted this simplification and few more optimizations in our implementation.

5 Experimental Evaluation

We compared our LaDR data race detection tool with the leading contenders from the recent *13th Software Verification Competition* (SV-COMP 2024).[5] SV-COMP is the premier venue for evaluating the performance of verification tools for C and Java programs using a comprehensive suite of established benchmarks. The competition is organized into categories and subcategories based on the specific properties to be verified for the programs. In SV-COMP 2024, the `c/NoDataRace-Main` subcategory concerns data race detection. Three tools emerged as top performers in this subcategory: Sv-sanitizers (based on ThreadSanitizer, see Section 1 for more details), Dartagnan [18], and Deagle [10] (see

[5] `https://sv-comp.sosy-lab.org`

Section 6 for more details). We evaluated LaDR's data race detection capabilities against these three tools. The evaluation covered both the established `c/NoDataRace-Main` benchmarks and a real-world benchmark, `safestack`, known for its particularly rare ABA-related bug.

Experimental Setup. Each experiment was run on a dedicated Google Compute Engine of type `n2-custom-2-16384`. This configuration provides two vCPUs running at 2.80GHz with 16GB of memory, running Ubuntu 22.04. We used the `runexec` tool from the BenchExec suite [1] to manage the individual experiments. Similar to SV-COMP 2024, `runexec` automatically terminates tool executions exceeding pre-defined memory or time limitations. For the experiments, we installed the follwoing tools on each machine: LaDR (replication package available at `https://doi.org/10.5281/zenodo.10826274`), Dartagnan, Deagle, and Sv-sanitizers.[6] We mirrored the competition settings of SV-COMP 2024 by allocating 15 minutes and 12 GB of RAM per experiment.

SV-COMP Benchmarks. The SV-COMP 2024 data race category offers a rich evaluation ground with 1013 benchmarks. These benchmarks average around 60 lines of code (excluding libraries) and represent a diverse set of concurrency constructs and can exhibit significant complexity. We focused our comparison on the 232 benchmarks with data races, since LaDR is designed as a data race detection tool, rather than a data race freedom proving tool.

We ran LaDR with a loop unwinding bound and round bound large enough to expose the data races, according to the usual bounded model checking practices. Other tools were executed using their respective competition scripts. LaDR performs very well on these benchmarks, and returns the highest number of correct results (183 out of 232), followed by Dartagnan (165), Deagle (145) and Sv-sanitizers (144). To validate the soundness of LaDR reports, we additionally ran it on the SV-COMP 2024 benchmarks that were classified as data race free. Notably, LaDR produced only one false data race report, but we believe that this benchmark is in fact incorrectly classified.

Safestack. `safestack` is a real world benchmark implementing a lock-free stack designed for weak memory models [31]. It contains a very rare ABA-related bug that requires at least three threads and five context-switches for exposure under the SC semantics (although only four context-switches under TSO or PSO), while typical real-world concurrency bugs require at most three context-switches to manifest themselves [19]. `safestack`, for this reason, presents a nontrivial challenge for concurrency analysis tools.

LaDR is the only tool we are aware of that can automatically find this data race: it takes 24 minutes and 4 seconds on the test machine to find the data race under the SC semantics, using the minimal necessary loop unwinding and round bounds. Deagle crashes during the analysis, while Dartagnan, PorSE, and Sv-sanitizers did not find the bug in 12 hours.

[6] Reproducibility packages for Dartagnan, Deagle, and Sv-sanitizers can be found on the SV-COMP 2024 web page: `https://sv-comp.sosy-lab.org/2024/systems.php`

6 Related Work

Concurrent Program Sequentialization. The idea of sequentializing concurrent programs for analysis purposes was first proposed by Qadeer and Wu [23] and then generalized to capture an arbitrary number of context switches by Lal and Reps [17]. Several other sequentializations were proposed in the literature [16,7,3,27,20,28]. However, to the best of our knowledge, only two other approaches have used sequentialization to detect data races. Qadeer and Wu [23] adapted their sequentialization to data races involving only one given memory location. The sequentialization we propose here does not have such restriction and works for general aliasing. Like us, Coto et al. [5] have extended Lazy-CSeq to detect data races. However, their implementation stores the target address of each shared variable write in auxiliary variables and explicitly compares these on each read and write to check for a race, leading to large formulas. Moreover, they incorrectly identify some simultaneous read-accesses as data races. In SV-COMP 2024, they found the data race in 39 benchmarks over the 232 ones that had it, while they reported 295 false alarms over the 781 race-free benchmarks.[7]

Bounded Data Race Model Checkers. Dartagnan [18] is a bounded model checker that can perform the analysis under several weak memory models which can be in passed as input parameters. Although it is not a pure data race detector, it has participated in the data race detection category at SV-COMP 2024 by taking as input a sequential consistency model.

Deagle [10] computes a logical formula that captures the happens-before ordering of the shared memory accesses and thus detects data races as loops in such ordering. The inter-thread behaviors are encoded by a logical formula that captures the happens-before relations among the shared memory accesses and the intra-thread behaviors by a propositional formula. The analysis is done by passing the overall formula to an SMT solver.

Automata-Based Tools. Several members of the Ultimate model checker family have also participated successfully in the data race detection category at SV-COMP 2024. Ultimate Automizer [11] is the basic tool in this family. It is based on an automata-theoretic approach to software verification that can check safety properties. Ultimate Taipan [9] combines trace abstraction with abstract interpretation on path programs. Ultimate GemCutter [15] is based on the CEGAR paradigm and integrates the classical CEGAR generalization with orthogonal generalization across interleavings. All these approaches encode data race detection in a more general framework than ours but their underlying automata-based algorithms are quite different from our approach.

Data Race Freedom Provers. Goblint [29] is a static analyzer for multi-threaded C programs that can perform data race detection. It uses a sound concurrency abstraction, based on privatization, and combines different pointer and value analyses which enable it to handle a wide range of locking schemes, including locks allocated dynamically as well as locks stored in arrays.

[7] `https://sv-comp.sosy-lab.org/2024/`

One of the most common techniques for race prevention is to protect any access to a shared memory location with locks. Locksmith [22] is a static analysis tool that discovers data races by checking whether this property is violated. There have also been other techniques developed following this idea [30,14,29,2]. They differ in the locking structures they can address, the accuracy of the detection, the need for user annotations, and the reduction of false alarms. The approach we propose here is more general in the sense that we do not search only for data races that can arise from careless lock usage and we do not require user annotations as some of these approaches do.

Tools Based on Symbolic Execution. PorSE [25] combines partial-order reduction techniques with symbolic execution to handle data non-determinism, and is implemented as an extension of KLEE. This tool has found race conditions in `memcached` and GNU `sort`, showing that the technique scales to industrial-size benchmarks. However, since KLEE explores execution paths individually, it usually struggles to find rare bugs that only manifest themselves in few executions, as shown by our experiments.

7 Conclusions and Future Work

We introduced a new static approach to detect data races in multi-threaded programs based on the LS sequentialization schema. We use a flag for each shared memory location which is non-deterministically set on a write and checked on each other access. The non-determinism allows us to inject simple control code that manipulates these flags.

Our experiments show that our approach outperforms the other static methods on the SV-COMP 2024 benchmarks. LaDR is also the only tool that finds the very rare data race in the `safestack` benchmark. Our results for this benchmark also confirm the general intuition that methods based on bounded model-checking are more suitable for deep and rare bugs in concurrent programs.

We have also tried to extend our approach to perform the analysis in parallel, following the schema from [21]. However, preliminary experiments have shown that the scalability of this approach is only partially improved by our parallelization scheme. We believe that data abstraction techniques can reduce the size of the formulas computed by the underlying bounded model-checker and so improve efficiency. The key will be to balance and reduce the different sources of non-determinism and the number of processors available for the analysis.

Acknowledgments. This work was partially supported by AWS 2021 Amazon Research Awards, INDAM-GNCS 2024, FARB 2021-2024 grants Università degli Studi di Salerno, VITALITY Ecosystem, Spoke 1 MEGHALITIC (E13C22001060006) under the NRRP MUR program funded by the EU - NGEU, and the Google Cloud Research Credits program with the award GCP19980904.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/S10009-017-0469-Y
2. Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD: compositional static race detection. Proc. ACM Program. Lang. **2**(OOPSLA), 144:1–144:28 (2018). https://doi.org/10.1145/3276514
3. Chaki, S., Gurfinkel, A., Sinha, N.: Efficient verification of periodic programs using sequential consistency and snapshots. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. pp. 51–58. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987595
4. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proc. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
5. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A Prototype for Data Race Detection in CSeq 3 - (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany, April 2-7, 2022, Proc., Part II. Lecture Notes in Computer Science, vol. 13244, pp. 413–417. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
6. Currie, D.W., Hu, A.J., Rajan, S.P.: Automatic formal verification of DSP software. In: Micheli, G.D. (ed.) Proc. of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000. pp. 130–135. ACM (2000). https://doi.org/10.1145/337292.337339
7. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Proc. of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 411–422. ACM (2011). https://doi.org/10.1145/1926385.1926432
8. Fischer, B., Torre, S.L., Parlato, G., Schrammel, P.: CBMC-SSM: Bounded Model Checking of C Programs with Symbolic Shadow Memory. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022. pp. 156:1–156:5. ACM (2022). https://doi.org/10.1145/3551349.3559523
9. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace Abstraction and Abstract Interpretation - (Competition Contribution). In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Uppsala, Sweden, April 22-29, 2017, Proc., Part II. Lecture Notes in Computer Science, vol. 10206, pp. 399–403 (2017). https://doi.org/10.1007/978-3-662-54580-5_31
10. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 424–428. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25

11. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the Search for Perfect Interpolants - (Competition Contribution). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_30
12. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proc. Lecture Notes in Computer Science, vol. 8559, pp. 585–602. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_39
13. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded Verification of Multi-threaded Programs via Lazy Sequentialization. ACM Trans. Program. Lang. Syst. **44**(1), 1:1–1:50 (2022). https://doi.org/10.1145/3478536
14. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: van Vliet, H., Issarny, V. (eds.) Proc. of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009. pp. 13–22. ACM (2009). https://doi.org/10.1145/1595696.1595701
15. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate GemCutter and the Axes of Generalization - (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Munich, Germany, April 2-7, 2022, Proc., Part II. Lecture Notes in Computer Science, vol. 13244, pp. 479–483. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
16. La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proc. Lecture Notes in Computer Science, vol. 5643, pp. 477–492. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_36
17. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods Syst. Des. **35**(1), 73–97 (2009). https://doi.org/10.1007/s10703-009-0078-9
18. de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Dublin, Ireland, April 25-30, 2020, Proc., Part II. Lecture Notes in Computer Science, vol. 12079, pp. 378–382. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_24
19. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785
20. Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. In: Artho, C., Legay,

A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 174–191 (2016). https://doi.org/10.1007/978-3-319-46520-3_12

21. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Rosu, G., Penta, M.D., Nguyen, T.N. (eds.) Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. pp. 753–764. IEEE Computer Society (2017). https://doi.org/10.1109/ASE.2017.8115686
22. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical static race detection for C. ACM Trans. Program. Lang. Syst. **33**(1), 3:1–3:55 (2011). https://doi.org/10.1145/1889997.1890000
23. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Pugh, W.W., Chambers, C. (eds.) Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004. pp. 14–24. ACM (2004). https://doi.org/10.1145/996841.996845
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.E.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997). https://doi.org/10.1145/265924.265927
25. Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K.: Symbolic Partial-Order Execution for Testing Multi-Threaded Programs. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proc., Part I. Lecture Notes in Computer Science, vol. 12224, pp. 376–400. Springer (2020). https://doi.org/10.1007/978-3-030-53288-8_18
26. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic Race Detection with LLVM Compiler - Compile-Time Instrumentation for ThreadSanitizer. In: Khurshid, S., Sen, K. (eds.) Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7186, pp. 110–114. Springer (2011). https://doi.org/10.1007/978-3-642-29860-8_9
27. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Verifying Concurrent Programs by Memory Unwinding. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proc. Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_52
28. Tomasco, E., Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Using shared memory abstractions to design eager sequentializations for weak memory models. In: Cimatti, A., Sirjani, M. (eds.) Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proc. Lecture Notes in Computer Science, vol. 10469, pp. 185–202. Springer (2017). https://doi.org/10.1007/978-3-319-66197-1_12
29. Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: the Goblint approach. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. pp. 391–402. ACM (2016). https://doi.org/10.1145/2970276.2970337

30. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Crnkovic, I., Bertolino, A. (eds.) Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007. pp. 205–214. ACM (2007). https://doi.org/10.1145/1287624.1287654
31. Vyukov, D.: Bug with a context switch bound 5. `https://social.msdn.microsoft.com/Forums/en-US/91c1971c-519f-4ad2-816d-149e6b2fd916/bug-with-a-context-switch-bound-5?forum=chess` (2010), accessed: 2022-08-17