RacerF: Lightweight Static Data Race Detection for C Code

Tomáš Dacík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Tomáš Vojnar
Faculty of Informatics, Masaryk University, Brno, Czech Republic
Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract

We present RacerF, a novel static analyser for thread-modular data race detection. The approach behind RacerF exploits static analysis of sequential program behaviour whose results are generalised for multi-threaded programs using a combination of lightweight under- and over-approximating methods. The tool is implemented as a plugin of the Frama-C platform and can leverage several analysis backends, most notably the Frama-C's abstract interpreter EVA. Although our methods are mostly heuristic without providing formal guarantees, our experimental evaluation shows that even for intricate programs, RacerF can provide very precise results competitive with more heavyweight approaches while being faster than them.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases concurrency, data race detection, static analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.37

Category Experience Paper

Supplementary Material
Software (Source Code): `https://github.com/TDacik/Deadlock_and_Racer` [11]
archived at `swh:1:dir:fc33aed78b227ea0738a89894b3dfa91b4f46844`

Funding The research was supported by the Czech Science Foundation project 23-06506S and the FIT BUT internal project FIT-S-23-8151. Tomáš Dacík was also supported by the Brno PhD Talent Scholarship of the Brno City Municipality. The research involved discussions with Julien Signoles from the Frama-C team at CEA that were supported by the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

Acknowledgements We appreciate discussions on the subject with Julien Signoles from the Frama-C team at CEA, France and with Adam Rogalewicz from FIT BUT, Czechia.

1 Introduction

Data races in concurrent programs belong among the most nasty bugs in computer programs: they are easy to cause, hard to spot during code inspection, and hard to catch by testing as they can manifest only very rarely. Due to this, a lot of effort has gone into designing as safe as possible libraries and languages for writing concurrent programs, but, on one hand, many programs are written using languages without such safety features (with the C language being a prominent example), and on the other hand, even in the safety-oriented languages such as Rust, unsafe programming is usually allowed and sometimes used, e.g., for performance reasons.

A lot of work has also been invested in designing various static as well as dynamic methods of finding data races and/or proving their absence. However, the complex behaviour of concurrent programs reflects in that such methods do often have problems with scalability

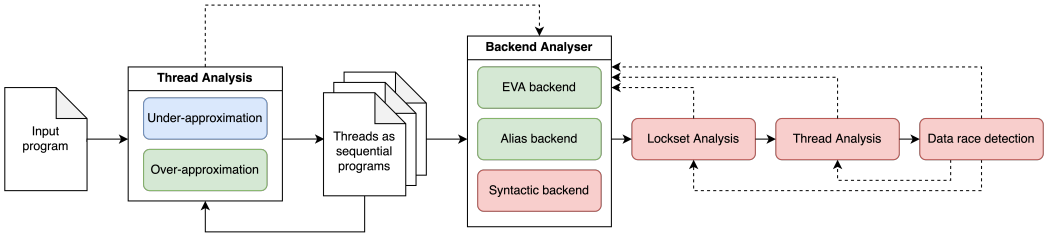

Figure 1 An overview of the workflow of RacerF. The phases with under- and over-approximation guarantees are marked by blue and green colour, respectively. The phases designed without guarantees are marked by red colour. The solid lines represent the workflow of individual analysis phases. The dashed lines represent queries on the results of other phases.

or precision. In this paper, we present a novel tool RacerF for thread-modular data race detection with the aim of providing a suitable balance between the scalability and precision of the analysis. We then also present our experience from running the tool and comparing it with several other data race detectors.

The approach implemented in RacerF relies on leveraging results of static analysis of the sequential behaviour of program threads for finding data races in multi-threaded programs. We use two strategies of combining information resulting from the analysis of the sequential behaviour of program threads: one under-approximates and one over-approximates the concurrent behaviour of the threads running in parallel. We combine these strategies to obtain a more precise analysis such that the under-approximation is used to find data races while the over-approximation is used to show their absence.

RacerF is implemented as a plugin of the Frama-C platform [10, 1] and is in particular designed for finding data races in C programs with POSIX threads (pthreads). RacerF can use several analysis backends implementing the underlying analysis of the sequential behaviour of program threads and differing in their precision and scalability. One of them is built on top of the Frama-C's value analysis plugin EVA based on abstract interpretation [6], one is purely syntactic, and the last one is based on Alias, Frama-C's plugin for may-alias and points-to analysis [5].

We have performed experiments with RacerF on a number of benchmarks coming both from the SV-COMP competition in software verification, which includes both synthetic benchmarks as well as benchmarks derived from real-world code, as well as on a number of student projects in an advanced course of operating systems. Although the approach behind RacerF is mostly heuristic without providing formal guarantees, our experimental evaluation shows that even for intricate programs, RacerF can provide very precise results competitive with more heavy-weight approaches while being faster than them.

Approach Overview

An overview of the workflow of RacerF is illustrated in Fig. 1. The core component is a *backend analyser* which answers several kinds of queries based on analysing individual threads as sequential programs (the general interface and several different backend analysers are described in Section 4). The *thread analysis* (Section 5) incrementally discovers new program threads and analyses them using the chosen backend analyser. Since the discovery of a new thread may add additional information, the process of thread discovery and their analysis using the backend analyser is repeated until a fixpoint is reached. Then, two helper

analyses are run to deal with checking whether two memory accesses may happen in parallel – a *lockset analysis* checks which accesses are guarded by which locks and an *active-threads analysis* checks when threads are created and joined (Section 6). Those analyses query results of the backend analysis, mainly for possible values of parameters of locking and threading functions. Finally, *data race detection* (Section 7) is performed. This phase checks memory accesses (using information provided by the backend analyser) and tracks whether they need to be protected and, if so, whether they indeed are. The detected issues are then classified into may- and must-data-races. Must-data-races are reported to the user, while an absence of may-data-races is used to claim program race-free.

Fig. 1 also depicts the approximation guarantees provided by the individual phases. Note that the combination of under- and over-approximating methods may seem problematic from the point of view of what can be guaranteed to hold for the produced results. However, we do not aim at providing guarantees in the sense of formal verification. Our aim is to achieve high efficiency and precision in practice even though some false alarms may be missed or some bugs not identified.

2 Related Work

RacerX [15] is a static analyser for C based primarily on a lockset analysis. The tool has been successfully used to find bugs in the kernels of several operating systems. Compared to our approach, RacerX uses a very simple pointer analysis for the representation of locks only. Further, RacerX uses many heuristics in individual analysis phases to deal with analysis mistakes while RacerF tries to achieve precision using careful ranking. RacerX is unfortunately not available for an experimental comparison. RacerD [4] is a compositional analyser implemented in the Infer framework. It provides scalability by analysing each function only once. Unlike us, RacerD targets Java programs with structural locking, assuming that all locking operations are balanced. Both RacerX and RacerD focus on under-approximation only. O2 [26] is another static checker building on the notion of origins unifying threads and events. The approach was implemented for C, C++, and Java/Android; and used to find many previously unknown bugs in real-world code. However, as we show in our experiments, it can still miss many data races in smaller but real-world programs.

On the other side, there are tools focused on soundness, rather than data race detection. Goblint [32] and Astrée [24, 27] rely on a thread-modular abstract interpretation to soundly over-approximate thread behaviour. The work [32] further proposes a way to handle common locking schemes in Linux driver devices. In our approach, we are not interested in the analysis of multithreaded programs in general and we can consequently afford more drastic approximations that seem to be sufficient for data race detection. We also focus on under-approximation to detect bugs.

As an alternative to static data race detection, various approaches based on *testing* and *dynamic analysis* can be used. A big challenge for these approaches is the fact that errors like data races can manifest in very rare behaviours only, which are difficult to catch, even through many repeated test runs. Coverage of such rare behaviours can be improved using approaches such as *systematic testing* [33] or *noise-based testing* [13, 17]. Another way is to use *extrapolating dynamic checkers*, such as [30, 14, 19], which can report warnings about possible errors even if those are not seen in the testing runs, based on spotting their symptoms. However, even though such checkers have proven quite useful in practice, they can, of course, still miss rarely occurring errors. Moreover, monitoring a run of large software

through such checkers may be quite expensive too.

Approaches based on *model checking*, i.e., exhaustive state-space exploration, can guarantee the discovery of all potentially present errors – either in general or at least up to some bound, often given in the number of context switches. However, so far, the scalability of these techniques is not sufficient to handle large industrial code, even when combined with methods such as *sequentialisation* [25, 28] or advanced techniques based on *automata* [12, 23] or *SMT encodings* [20, 21, 16]. Indeed, this shows up in our experiments too.

3 Preliminaries

We now introduce several basic notions that we use when describing the analysed programs and the analyses proposed.

Concurrency model. We primarily target C programs using threads and mutexes in the style of the POSIX threads (pthreads) execution model. We use the term *thread* to mean thread's entry point, i.e., an identifier of the function from which the given thread is started. In the examples, we shorten the calls of locking, unlocking, thread-creating, and thread-joining functions as `lock`(...), `unlock`(...), `create`(...), and `join`(...), respectively. We further implicitly assume that all locks are properly initialised and omit the arguments that are not relevant, e.g., we write just `create`(`t1`), where `t1` is an entry point, when the thread id, attributes, and argument are not relevant for the example.

The CValue domain and memory addresses. Throughout the paper, we frequently refer to the CValue abstract domain [6, 10] implemented in Frama-C and used by the abstract interpretation of EVA. We are primarily interested in the representation of memory addresses. A memory address is represented as a pair (b, o) where b is a *base address* (in the following referred to simply as a *base*) and o is an offset. Each variable (global, local, and also formal) has its associated base address. Dynamic allocations are modelled by creating so-called *dynamic bases* which come in two flavours – a *strong* dynamic base corresponds to the result of precisely one call to an allocation function; on the other hand, *weak* memory bases may represent multiple allocated addresses (typically, they are the consequence of allocations in loops). Offsets are represented by a dedicated integer domain, the details of which are not important for this paper. We use CValue.val for the domain representing possible values of a single variable and CValue.state for the domain representing program state, i.e., possible values for multiple variables. We use Addresses to denote the aforementioned set of memory addresses. In the rest of the paper, the operations *join* and *widening* as well as *top values* refer to those on the CValue domain.

Analysis contexts. We call an *analysis context* a triple $(thread, calls, stmt)$ consisting of a thread, a sequence of the latest n calls (each of them formed by a call site and the function called), and a program statement. Our data race analysis relies on various underlying analyses presented below to compute an abstract representation of thread states reachable by performing the given statement for each analysis context. Notice that a context holds no information about how the thread was created nor how its execution was interleaved with other threads. The empty call sequence is denoted by ε. The number n is a parameter of the analysis set to 1 by default to handle simple lock/unlock wrapper functions. We use Context to denote the set of all possible contexts.

$$\begin{aligned}
\mathsf{analyse_thread} &: \mathsf{Function} \times \mathsf{CValue.state} \to \mathsf{Result} && \text{(sequential analysis of a thread)}\\
\mathsf{state} &: \mathsf{Result} \times \mathsf{Context} \to \mathsf{CValue.state} && \text{(state before a statement)}\\
\mathsf{value} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to \mathsf{CValue.val} && \text{(non-pointer expression values)}\\
\mathsf{value_ptr} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to 2^{\mathsf{Address}} && \text{(pointer expression values)}\\
\mathsf{functions} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to 2^{\mathsf{Function}} && \text{(function pointer resolution)}\\
\mathsf{accesses} &: \mathsf{Result} \times \mathsf{Context} \times \mathsf{Expr} \to (2^{\mathsf{Address}})^2 && \text{(memory reads and writes)}
\end{aligned}$$

Figure 2 The interface of the backend analysis used by the proposed analysis of data races. The function analyse_thread performs the analysis of a thread as a sequential program given by its entry point, starting from the given initial state. The other functions answer queries based on the results computed by analyse_thread (specific for different backends). Note that the signatures do not include statements as those are already included in contexts.

4 Backend Analysis

Our data race analysis relies on a backend analysis of sequential program behaviour to get answers for several kinds of queries. Those include *function pointer resolution* (to determine which functions are to be analysed when and to determine thread entry points passed to thread create functions), getting *points-to information* (to determine targets of (un)lock operations, to perform escape analysis, and, most importantly, to determine on which addresses to look for data races), getting information on *memory accesses*, and characterizing *values of integer variables* (to determine whether offsets of two accesses overlap, and to check whether a trylock operation succeeds in the given branch of code).

To offer a selection of multiple backend analyses differing in their scalability, precision, and ease of use, we have implemented several such analyses. The interface a backend needs to implement is shown in Fig. 2. The function analyse_thread runs the analysis of a thread as a sequential program starting from the provided state of global variables and the thread's argument. Based on the results, the different queries are then answered.

Evolved Value Analysis (EVA) backend. This backend is based on EVA [6], a value analysis plugin of Frama-C based on abstract interpretation. EVA provides a combination of multiple abstract domains and an API to answer all the needed queries, and our backend is just a thin wrapper over EVA, which implements caching, simplification, and answering some trivial queries on its own. The interface in Fig. 2 is in fact inspired by the API of EVA, and the other backends mimic it. In particular, the EVA's API implements queries for call stacks which can be easily obtained from our contexts.

Syntactic backend. This backend provides a lightweight analysis based only on syntactical information in the style of RacerX [15] or RacerD [4]. The implementation of analyse_thread is simply an empty function and the functions state and value always return the top value. The function value_ptr searches the expression and extracts bases of the expected type. This is inspired by the insight that in some programs only global locks are accessed using `lock(&m)` and `unlock(&m)` (where `m` is a global variable), and it is thus sufficient to simply extract the base of `m`. The function memory_accesses is implemented similarly with a more detailed analysis of an expression to determine whether a base is being read or written to it. Finally,

```
1  void *t1(void *arg1) {
2    int *x = (int *)arg1;
3    (*x)++;
4  }
5
6  void *t2(void *arg2) {
7    int *y = (int *)arg2;
8    (*y)++;
9  }
10 int main() {
11   int data = 0;
12
13   create(t1, &data);
14   create(t2, &data);
15 }
```

Figure 3 An example of a program for which the Syntactic backend fails to find a data race.

function pointers are over-approximated by taking each function assigned in the program. In the case of thread entry points, this can be further refined by taking only functions matching the thread signature (i.e., `void` $*$ (*`fn`)(`void` $*$ `arg`) for pthreads).

Alias backend. The aim of the third backend is to provide a middle ground between the precision of the EVA and the scalability of the syntactic backend. It relies on another Frama-C plugin called Alias [5] which performs a may-alias analysis using a variant of the classical Steensgaard's algorithm [31]. For queries not related to pointer expressions (state and value), the top values are returned. For the other queries, since Alias does not have an API to answer our queries directly, we take the result of the syntactic backend and replace each base by a canonical representative of its alias equivalence class. In particular, for memory accesses, we ensure that a selected canonical address for an alias class is vulnerable to data races if such an address exists (e.g. because its base corresponds to a global variable or a local variable that escaped; see Section 7 for more details). Alias also does not allow starting with a user-supplied set of aliases (as expects the signature of our analyse_thread function). We therefore need to perform an additional saturation of the aliases set for all queries, as demonstrated by the example in the next paragraph. The Alias plugin is still relatively new (it lacks support, e.g., for the dynamic memory allocation) and thus our backend over it is also experimental.

▶ **Example 1.** The advantage of the Alias backend over the syntactic backend is demonstrated in Fig. 3. The program contains a data race on an address of the variable `data` that escapes its scope in the `main` function by passing it as an argument to the threads `t1` and `t2`. Both threads share the same code, which creates a local alias for this address. This is a very common pattern when thread arguments are used – since the type of the argument is generic `void`$*$, it needs to be casted to its actual type. The address is then written to by both threads using their local aliases. While the syntactic backend would not report any relevant access for data race detection, the Alias backend would compute aliases for `x` and `y` as $\{$`x`, `arg1`$\}$ and $\{$`y`, `arg2`$\}$, respectively. Our backend then performs a saturation of both sets by adding aliases $\{$`arg1`, &`data`$\}$ and $\{$`arg2`, &`data`$\}$, respectively. Finally, &`data` will be selected as a canonical base, because it is vulnerable to a data race.

5 Thread Analysis

In this section, we propose our approach on how to leverage an analyser for sequential programs to analyse programs with multiple threads. The design of our approach is based on the observation that answers to our queries described in Section 4 on objects relevant for our

Algorithm 1 Thread analysis.

Input : Program entry point main
Input : Initial values of global variables GlobalsVals ∈ Cvalue.state
Data : Initial states of threads $I : T \to$ Cvalue.state
Output : Thread-creation graph $G = (T, \to)$
Output : Results of analysis of sequential behaviour of the threads $R : T \to$ Result

1 $i = 0$
2 $G_0 = (\{\mathsf{main}\}, \emptyset)$ `// Initial graph containing main only`
3 $I_0 = \{\mathsf{main} \mapsto \mathsf{GlobalsVals}\}$ `// Initial state of the main thread`
4 $R_0 = \{\mathsf{main} \mapsto \mathsf{Backend.analyse\_thread}(\mathsf{main}, \mathsf{GlobalsVals})\}$ `// Analyse main`
5 **do**
6 $\quad i = i + 1$
7 $\quad G_i = G_{i-1}$
$\quad$ `/* Update the graph with reachable threads. */`
8 $\quad$ **foreach** thread represented as a vertex p in G_{i-1} **do**
9 $\quad\quad$ **foreach** stmt s of the form $\mathsf{thread\_create}(f)$ syntactically reachable by p **do**
10 $\quad\quad\quad children = \mathsf{Backend.functions}(R_{i-1}(p), (p, \varepsilon, s), f)$
11 $\quad\quad\quad G_i = G_i \cup \{p \xrightarrow{s} c \mid c \in children\}$
$\quad$ `/* Build equations wrt. the chosen approx. strategy. */`
12 $\quad \Phi_i = \mathsf{ConstructEquations}(G_i)$
$\quad$ `/* Solve the system of equations. */`
13 $\quad I_i = \mathsf{SolveEquations}(\Phi_i, I_{i-1})$
$\quad$ `/* Analyse each thread as a sequential program. */`
14 $\quad R_i = \{t \mapsto \mathsf{Backend.analyse\_thread}(t, I_i(t)) \mid t \in \mathsf{dom}(I_i)\}$
15 **while** $G_{i-1} \neq G_i$
16 **return** G_i, R_i

analysis are often not too much affected by thread interleavings. We design two strategies for combining results of the analyses of the sequential behaviour of individual threads that under- and over-approximate the overall behaviour of the concurrent program consisting of the threads (not including behaviour possibly caused by errors in the code, i.e., for example undefined behaviour after a data race). Furthermore, these strategies can subsequently be combined to increase the precision of the resulting analysis by using under-approximation to find races while using over-approximation to show their absence. However, as we show in our experimental evaluation in Section 8, both of our strategies are often sufficiently precise on their own.

The core idea of our thread analysis is similar to that of the Frama-C's proprietary Mthread plugin described in [34]. Starting with the `main` thread, we iteratively build a thread-creation graph that captures the parent-child relationship among the so-far discovered threads. In each iteration, we add newly discovered threads and perform the analysis (potentially multiple times) of each thread as a sequential program (using the `analyse_thread` function of the selected backend). This is done by solving a system of equations over the initial states of the threads given by whose construction we describe below. However, before that, we first describe the graph construction algorithm.

The section is relevant mainly for our EVA backend and partly for our Alias backend. For the Syntactic backend, only the thread-creation graph construction is relevant (because no analysis of threads is performed).

5.1 Construction of the Thread-Creation Graph

The method is given in Algorithm 1. The output of the algorithm is (1) a thread-creation graph $G = (T, \to)$ where T is the set of all discovered threads and $\to$ is a labelled edge relation such that $p \xrightarrow{s} c$ means that the parent p creates a child c at the program statement s, and (2) a mapping $R : T \to \mathsf{Result}$ containing results of the analysis of the behaviour of each thread. The algorithm further operates with the mapping $I : T \to \mathsf{CValue.state}$ representing the initial states of discovered threads[1]. Once we derive a stable mapping I (i.e., a fixpoint of the thread's initial states), we have also reached a fixpoint of the results R, needed to answer all the queries issued by our analysis.

The algorithm starts with a graph that contains only the main thread whose initial state is given by the initial values of global variables. With this initial state, main is analysed on line 4.

In the main loop of the algorithm starting on line 5, the graph is updated by adding threads created from (syntactically) reachable statements. Notice that the evaluation of an expression f on line 10 is performed without using any calling context. This is because our analysis, in particular the system of equations in Sections 5.2 and 5.3, is not very sensitive to thread-create wrappers. Consequently, we can construct smaller graphs by distinguishing edges only by statements and not by calling contexts.

After a new iteration of the graph G_i is constructed, we build a system of equations, one for each discovered thread, given by G_i. Each of these equations describes how an initial state of a thread can be computed using initial states of other threads. Then, the system is solved using initial states from the previous iteration I_{i-1} to obtain new initial states I_i. The solution may need an application of widening operators – we provide more details on our implementation of the equation solving in Section 8. Based on the new initial states, we compute new results R_i on line 14.

This loop is repeated until the thread-creation graph stabilises. Overall, the algorithm runs two fixpoint computations – the outer updates the graph with new threads, while the inner (equation solving on line 13) performs analysis on the current graph, and can thus help to discover previously unknown threads in new iterations.

Below, we discuss how the aforementioned equations are constructed.

5.2 Under-Approximating Thread Analysis

We use the following under-approximated semantics of thread creation. A thread is started with the initial state corresponding to the join of initial states of its create statements, and its writes to global variables are not propagated to other threads. The creation of threads then roughly corresponds to calling its entry point in a "sandbox" that makes all of its actions invisible for other threads. Formally, given a thread-create graph G_i with edges $\to$, we compute the initial state of a thread t (other than `main` – the initial state of `main` is given by the initial states of global variables) as follows:

$$I(t) \triangleq \bigsqcup_{p \xrightarrow{s} t} \mathsf{let}\ R = \mathsf{Backend.analyse\_thread}(p, I(p))\ \mathsf{in}\ \mathsf{Backend.state}(R, (p, \varepsilon, s)),$$

[1] In the implementation, the product $\mathsf{CValue.state} \times \mathsf{CValue.val}$ is used as the domain of the initial states to represent the values of global variables and the value of the thread's argument. However, for simplicity of the presentation, we may assume that the argument is simply included among the global variables.

```
1  int *g;
2  void *t1() {
3    g = malloc(...);
4  }
5  void *t2() {
6    (*g)++;
7  }
8  int main() {
9    create(t1); join(t1);
10   create(t2); create(t2);
11 }
```

(a)

```
1  int g = 1;
2  void *thread() {
3    lock(L);
4    if (g == 1) {
5      unlock(L); return NULL;}
6    g = 2;
7  }
8  int main() {
9    create(t1);
10   lock(L); g = 0; g = 1; unlock(L);
11 }
```

(b)

Figure 4 Examples of programs on which (a) the under- and (b) over-approximating thread analysis reports a false negative and a false positive, respectively.

i.e., we take all statements s creating t, and for each of them, we recompute the corresponding parent p and take its state at the statement s, returning the join of all those values. In this way, the creation of a thread does not affect its parent at all.

The motivation for this strategy is straightforward – for acyclic thread-creation graphs, it is enough to analyse each entry point only once (provided caching is used for the `analyse_thread` function). On the other hand, there are two main sources of possible imprecisions – thread interleavings are not taken into account, and parents do not see effects of their children. The latter can sometimes result in seemingly weird false negatives when performing data race detection, as demonstrated in Listing 4a. In the program, the `main` thread first creates a thread `t1` which allocates an integer value and then a thread `t2` (created twice) accesses this value without any synchronization. However, we cannot detect this data race using EVA[2] since in our under-approximated setting, `t2` will never see the allocation, and in such a situation, EVA will mark the memory access on line 6 as surely invalid and will not report any memory access for it.

5.3 Over-Approximating Thread Analysis

While for the under-approximating strategy, we assumed that no changes done by threads are propagated, in our over-approximating strategy, we assume that every thread can see all possible changes (even those that should be invisible due to some synchronization method). For this method to work correctly, we also need to perform a simple transformation of the analysed program, which we discuss, together with a motivation example, later in this section.

We compute the initial state of a thread t (here also including the `main` thread) as the join of the states our analysis has encountered so far. Formally, given a thread graph G_i with a set T of currently known threads, the initial values of global variables $V = \mathsf{GlobalsVals}$, and a set of all program statements Stmt, we proceed as follows:

$$I(t) \triangleq V \sqcup \bigsqcup_{s \in \mathsf{Stmt},\ t' \in T} \mathsf{let}\ R = \mathsf{Backend.analyse\_thread}(t', I(t'))\ \mathsf{in}\ \mathsf{Backend.state}(R, (t', \varepsilon, s)),$$

i.e., we take the join of the states of all statements for each currently discovered thread.

[2] We could detect the memory access with the other backends that do not care about memory initialization.

```
1  atomic_int g;
2
3  void *t() {
4    g++;
5  }
6
7  int main() {
8    g = 0;
9    create(t);
10   if (g == 1)
11     ...
12 }
```

(a) Original program.

```
atomic_int g;

void *t() {
  g = (*) ? g : g+1;
}

int main() {
  g = (*) ? g : 0;
  create(t);
  if (g == 1)
    ...
}
```

(b) Transformed program.

Figure 5 An example of a program for which the over-approximating strategy fails to compute a sound over-approximation unless the program is transformed ($*$ denotes non-deterministic condition).

To solve such a system, more than one sequential analysis of each entry point and an application of widening is needed to converge[3] (cf. Section 8). Observe that the equations do not depend on edges of the thread-creation graph, just on its vertices.

As we also consider context switches that cannot happen due to synchronization (even those that cannot happen because a child was not yet created, e.g., when computing the initial state of the main thread and already using states of its children), we may later report false data races as demonstrated in Listing 4b. In this example, we will compute the initial state of `thread` to be $\{\mathsf{g} \mapsto \{0, 1, 2\}\}$. However, the value 0 is never visible to `thread` because the sequence of updates on line 10 (as well as the condition on line 4) is performed atomically, resulting in no change of g. Thus, the memory access on line 6, which would be flagged as a data race (together with one of the accesses on line 10), is in fact unreachable.

Although both examples in Fig. 4 are taken from publicly available benchmarks, we believe that they are quite rare in practice. Moreover, our strategies can be combined by allowing the under-approximation to only report errors and the over-approximation to only report that no error has been found. Otherwise, an unknown verdict will be reported, which would be also the case in both examples in Fig. 4.

5.3.1 Program transformation

The presented over-approximating strategy is not in itself sufficient to compute a sound over-approximation of multi-threaded behaviour as demonstrated on the program in Figure 5. In the example, we are particularly interested in the block inside the condition on line 10, which may contain some important event for our analyser. This code is unreachable when the main thread is analysed as a sequential program regardless of the initial state. Indeed, our over-approximation strategy would compute the initial state as $\{\mathsf{g} \mapsto [0, \mathsf{max\_int}]\}$, but already on line 8, this state would be modified to $\{\mathsf{g} \mapsto [0, 0]\}$ and the over-approximation would be lost.

[3] We apply widening quite aggressively by performing it already after the second iteration to reach a fixpoint as soon as possible.

We solve the above problem using a simple source code transformation rather than modifying the internal state of the sequential analyser (so we do not need to make further requirements on the interface of our backends). The idea behind our solution is that it is sufficient to ensure that each program instruction is *extensive*, i.e., if S and S' are the sets of concrete states before and after the instruction, respectively, then $S \subseteq S'$. We modify all program instructions to be extensive using the following rules.

- An assignment $\mathtt{x} = \mathtt{y}$ is transformed into a non-deterministic choice between the old and the new value as $\mathtt{x} = (*)\ ?\ \mathtt{x} : \mathtt{y}$.
- A function call $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ where at least one of the arguments x_i is passed by reference is transformed into a condition which non-deterministically skips the call as $\mathtt{if}(*)\ \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$.

In the rest of the paper, we implicitly assume that all backend queries for a thread t are evaluated over the results $R_i(t)$ where R_i is the value returned by Algorithm 1.

6 Lockset and Active-Threads Analyses

This section presents two helper analyses for data race detection. Both are used to check whether a pair of contexts can happen in parallel – the *lockset analysis* checks this based on locks being held in the given contexts, and the *active-thread analysis* deals with creating and joining of threads.

6.1 Lockset Analysis

Lockset analysis is traditionally used to compute must- or may-locksets being held for each statement. Our lockset analysis is context-sensitive, path-insensitive, and it computes both must- and may-locksets simultaneously. It is inspired by the approach of RacerX described in [15], which, however, does not employ any advanced alias analysis and is not context-sensitive.

We represent locks by their concrete addresses, i.e., $\mathsf{Lock} = \mathsf{Address}$, and we define $\mathsf{Lockset} = 2^{\mathsf{Lock}}$. Our lockset analysis is implemented as a forward data flow analysis with the domain of locksets, computed for each thread in separation, starting with the empty lockset. On the intraprocedural level, we start to analyse a function with an entry lockset L_{entry} coming from the caller, and gradually update it using the following transfer function that, for a context c with a statement s, transforms the lockset L_{old} reached before executing s into a set of locksets $\mathcal{L}_{new}$ resulting from the execution of s as follows:

$$\mathcal{L}_{new} = \begin{cases} \{L_{old} \cup \{\ell\} \mid \ell \in \mathsf{Backend.value\_ptr}(c, \mathtt{p})\} & \text{if s is } \mathtt{lock}(\mathtt{p}), \\ \{L_{old} \setminus \{\ell\} \mid \ell \in \mathsf{Backend.value\_ptr}(c, \mathtt{p})\} & \text{if s is } \mathtt{unlock}(\mathtt{p}), \\ \{L_{old}\} & \text{otherwise.} \end{cases}$$

After applying the transfer function, if $\mathcal{L}_{\text{new}}$ is not a singleton set, we fork the analysis and continue with each element of $\mathcal{L}_{\text{new}}$ separately. As a result, we obtain the summary of a function f analysed in a context c in the form $(c, L_{entry}) \mapsto \mathcal{L}_{exit}$ where $\mathcal{L}_{exit}$ is the set of locksets at the return statements of all forked analysis runs inside f. Such summaries are then used for the interprocedural analysis. We also construct *statement summaries* $\mathcal{S}$ of the same form as for functions, i.e., mappings from pairs of an input context and a lockset to output sets of locksets. These are used for caching at the statement level and to implement the must- and may-lockset queries for a context c as

$$\mathsf{may\_ls}(c) = \bigcup_{(c,L)\in\mathsf{dom}(\mathcal{S})} L \qquad \mathsf{must\_ls}(c) = \bigcap_{(c,L)\in\mathsf{dom}(\mathcal{S})} L$$

Dealing with path explosion. As we represent locks using memory addresses and fork the analysis whenever we cannot precisely evaluate the parameter of a (un)locking operation, the set of traversed paths can easily explode. To tackle this problem, our transfer function selects only a small number of locks when the set $\mathsf{Backend.value\_ptr}(c, \mathtt{p})$ is too large (currently, the threshold is set to three locks). This works in many practical cases because it is usually enough to differentiate the may- and must-cases through a few paths only. In the future, we would like to provide support for symbolic treatment of locks combined with may- and must-equality reasoning.

Our analysis does not support reentrant mutexes/semaphores but can handle non-blocking lock functions and read-write locks as discussed below.

Non-blocking locking functions. For non-blocking lock functions such as `trylock`, which does not block when the mutex is already locked, or `timedlock`, which blocks just for a limited time, we update the lockset with relevant locks for which we also track the variable used to store the return value of the corresponding call. Evaluation of a query $\mathsf{may\_ls}(c)$ or $\mathsf{must\_ls}(c)$ will then ask the backend for the state s computed for the context c through the query $s = \mathsf{Backend.state}(c)$, and filter out the locks for which the state s does not guarantee that the locking was (possibly or surely, depending on the query) successful. This guarantee will only be provided if the success of the locking was explicitly checked in the context c. This way we obtain a limited path-sensitivity for branching over expressions containing variables storing return values of lock operations. For the function `pthread_mutex_lock`, we assume that it is blocking (i.e., that it never fails), but the user can easily change the configuration to treat it as non-blocking.

Read-write locks. To support read-write locks, a lock object keeps additional information about whether it was obtained in the read- or write-mode. Based on this information, we modify the intersection of locksets, presented later on as the main operation used for data race detection, so that it contains only those locks which are present in both operands and at least in one of them in the write-mode. In other words, if two accesses are guarded only by a lock obtained in both cases in the read-mode, we will treat them as unprotected.

6.2 Active-Threads Analysis

The goal of this analysis is to deal with situations when two memory accesses cannot happen in parallel either because one happens before the thread of the other one is created, or because one happens after the thread of the other one is surely joined. We again compute this information in a thread-modular way by computing a *set of sets of threads* that may or must be active in parallel for each context, respectively. Two contexts are then may-parallel (must-parallel) if the thread of one of them is in the union (intersection) of sets computed for the other, and vice versa.

The sets of active threads are computed in a very similar way to how the lockset analysis is performed – just working with sets of threads instead of sets of locks, with the transfer functions for thread creation and joining analogous to those for locking/unlocking, and with the same way of handling the may- and must-queries. The set of thread entry points is usually small, and we thus do not need any special treatment to tackle path explosion unlike in the lockset analysis. One difference between the analyses is in the initial states. The lockset analysis of each thread is started with the empty lockset, but the active-threads analysis of a thread t starts with the union of all threads that may be active together with t

in some context of the parent of t (the analysis is thus run in the order given by a topological order on thread-creation relation, and for the very rare case of cyclic thread-creation graphs, we simply skip the analysis, providing trivial answers for queries based on it).

Thread identifiers. Another significant difference is the way in which threads are joined. While our analysis abstracts threads using their entry points, in the concrete semantics, they are manipulated using their *identifiers*. In many programs, however, the manipulation of identifiers is straightforward, and one can easily map identifiers to abstract threads and vice versa. Based on this observation, we assume that if an identifier expression of a created and joined thread is the same, then the thread is also the same. We have never encountered a situation when this would lead to a wrong analysis verdict.

7 Data Race Detection

Our data race detection does a forward traversal of the program and collects memory accesses consisting of a context (used for querying the results of lockset and active-thread analyses), a base address, and a symbolic offset (a set of intervals in the Cvalue domain that may be some simple value for backends different from EVA). We cluster memory accesses according to their bases to later check for data races only inside each of those clusters. This is justified by the following hypothesis about the memory model: it is not possible to get from one base to another by adding an offset. This hypothesis might not hold for some low-level C programs, but it is important for efficiency. Moreover, analysis of programs breaking this assumption is not even supported by EVA [8].

The tracking of accesses is inspired by [30]. For each cluster represented by its base, we track a state $s \in \{\mathsf{ReadOnly}\ o, \mathsf{Exclusive}\ o, \mathsf{Shared}, \mathsf{SharedMod}\}$ representing that a memory base is either read-only (ReadOnly) or modified (Exclusive) by a single thread o, read by several threads with none of them writing to it (Shared), or modified by distinct threads such that at least one of them writes to it (SharedMod). With each access to a memory base b by a thread t, the state of b is updated according to the transition graph in Fig. 6. When the analysis is finished, we keep only the memory bases marked as SharedMod or Exclusive o (when o is a thread created multiple times), and for each of them, we check each pair of accesses for a data race. The information about which threads are unique and which are created multiple times is computed during the thread analysis based on whether its create statement was visited once or multiple times.

Escape analysis. An important part of data race detection is discovering local bases that may escape their scope. Our syntactic backend assumes that dynamically allocated bases can always escape and static bases escape when their address is taken. Using the Alias and EVA backends, we enhance this by testing (for both static and dynamic bases) not only whether their addresses were taken, but also whether they were assigned to global variables.

May- and must-data-races. We distinguish two types of data races based on the confidence we have in them. For *may-data-races*, the below conditions need to be satisfied:

1. At least one of the accesses is a write access.
2. The accessing threads are either distinct or the same non-unique thread.
3. The offsets of the addresses may overlap (checked by a query over the CValue domain of intervals).

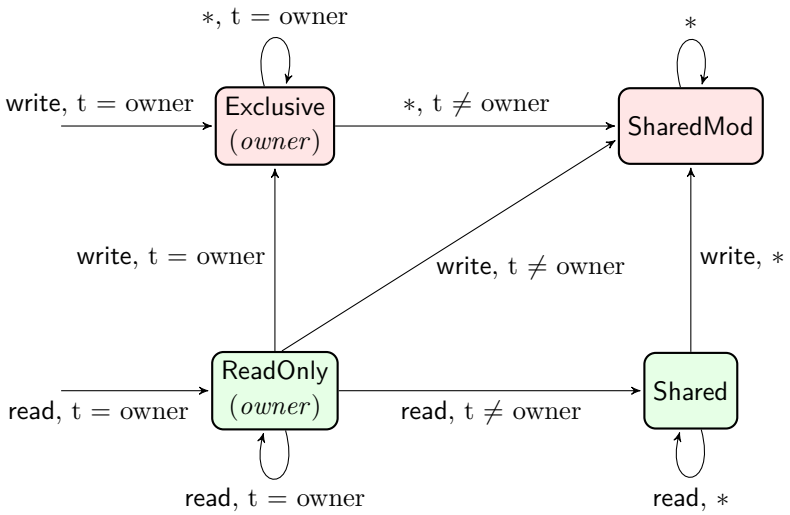

Figure 6 The transition graph for tracking states of memory bases. Each transition corresponds to a read or write memory access by a thread t. The symbol $*$ represents any access kind/thread. The states vulnerable to data races are marked in red.

4. The intersection of the may-locksets of the concerned contexts is empty (checked using the results of the lockset analysis).
5. The threads of both accesses may run in parallel (checked using the results of the active-threads analysis).

For *must-data-races*, we require the *must*-variants of Conditions 3–5 to hold. In addition, the following conditions are required to rule out races involving a memory access representing abstraction of multiple memory accesses:

6. The memory base is not *weak* (i.e., if it is dynamic, then it represents precisely one dynamic allocation) and it is not allocated in a non-unique thread.
7. The size of the access does not exceed the size of the type of the base.
8. The base is the only base being read/written to in its context.

Condition 6 ensures that we do not report must-races on bases obtained by abstracting and hence merging different calls to allocation functions. Condition 7 rules out races on possibly different indices to arrays (an access with an offset bigger than the size of the type of the base usually abstracts multiple accesses). Finally, Condition 8 handles situations when our context-sensitivity level is not sufficient to distinguish accesses to multiple bases.

Arrays and loop unfolding. To slightly relax the restrictiveness of Condition 7, we can perform a small number of unfoldings of each loop. This can help us report must data races on the accesses in unfolded cases. The loop unfolding is also useful for dealing with the creation of threads in loops – since our analyses are path-insensitive, they will also consider the infeasible path on which the cycle is never entered, and no thread is created. For memory accesses following the loop, the must variant of Condition 5 will thus not hold. Unfolding such loops at least once will fix this problem.

8 Implementation and Experimental Evaluation

The RacerF[4] analyser based on the above presented principles is implemented as a plugin of the Frama-C platform as a part of our bigger project for detection of concurrency bugs. The individual analyses are written as OCaml functors parametrised by a module implementing the backend analysis. We also provide a Python wrapper which runs a combination of under- and over-approximating analyses and also performs preprocessing (currently, the preprocessing is implemented outside of our analyser for technical reasons, thus the analyser is slowed a bit by the need of repeated parsing). To solve the systems of equations from Sections 5.2 and 5.3 we use the *chaotic iteration strategy with widenings* [7] implemented in the Ocamlgraph library [9]. The widening operators are those provided by the CValue domain.

Our implementation provides support for atomic and thread-local variables. The threading and locking functions (and their relevant properties) as well as atomic functions (such as gcc's `__atomic_store`) may be defined using configuration files.

We have conducted a series of experiments to support the following two hypotheses:

- Despite having no formal guarantees, RacerF can efficiently analyse intricate programs without sacrificing a precision. We validated this hypothesis on a collection of benchmarks from SV-COMP containing 1029 programs, many of them being intricate examples and corner cases contributed by participants to showcase capabilities of their tools.
- RacerF can detect data races in real-world programs for which other existing static analysers do not provide precise results (either because of too coarse approximation or running out of resources). We validated this hypothesis on a set of student programs from [18].

Experimental setup. Some data presented in experiments were taken from the results of SV-COMP 2025 [3]. All other experiments were run on a machine with 2.5 GHz Intel Core i5-7300HQ CPU and 16 GiB RAM, running Ubuntu 22.04 in a virtual machine limited to 10.7 GiB RAM. Our experiments were conducted using BenchExec [2], a framework for reliable benchmarking.

8.1 SV-COMP Benchmarks

The SV-COMP sub-category *NoDataRace-Main* is, to our knowledge, the largest publicly available data race benchmark. Many of its programs are small but intricate tests provided by the participants. Our main goal is to demonstrate the scalability of our approach and to also show that its heuristic nature does not compromise its precision even on those intricate examples. The benchmark also contains implementations of lock-free algorithms, which are out of the scope of our tool. We have implemented a single heuristic that detects active waiting (essentially a loop with an empty body) as a characteristic pattern for lock-free algorithms, and when it is present, returns an unknown result.

We compare with the five best-performing tools participating in SV-COMP 2025 [21, 22, 23, 29] (including $\text{RacerF}_{\text{sv-comp}}$, an older version of RacerF) and we also include $\text{RacerF}_{\text{under}}$ and $\text{RacerF}_{\text{over}}$ representing RacerF running with under- and over-approximating thread analysis only, respectively. We wanted to include also the O2 checker [26], but since its results were not good and all the other tools were configured specifically for this benchmark by their authors, we do not consider such a comparison fair.

[4] RacerF is available under the MIT license at `https://github.com/TDacik/Deadlock_and_Racer`.

Table 1 Results for the SV-COMP sub-category *NoDataRace-Main*. The columns represent true negatives, false positives, true positives, false negatives, out-of-resource (time or memory), score (as defined by SV-COMP rules, but excluding witness validation), and times (excluding timeouts) on local and competition machine, respectively.

	No race (794)		Races (235)					
Analyser	TN	FP	TP	FN	RO	Score	Time$_1$ [s]	Time$_2$ [s]
RacerF	674	4	105	0	0	1389	4620	–
RacerF$_{\text{over}}$	674	6	83	0	0	1335	3970	–
RacerF$_{\text{under}}$	680	4	105	6	0	1209	2710	–
RacerF$_{\text{sv-comp}}$	674	4	98	0	0	1382	3100	1300
Goblint	712	0	0	0	3	1424	–	2320
Deagle	685	0	203	1	43	1541	–	10k
UGemCutter	578	0	161	0	263	1317	–	152k
UAutomizer	609	0	121	0	274	1339	–	120k

The results are summarised in Table 1. Unlike the competition results [3], we present the results ignoring the witness validation. RacerF produced 4 false alarms, all of them on programs where a data race is ruled out by some intricate condition. Two of them are variants of a program where a data race does not manifest because an array being searched in parallel contains only zeros and a certain condition is thus never triggered which makes the potential racy access unreachable (which is the kind of behaviour that our tool cannot handle). The other two contain locking patterns that are beyond the scope of our lockset analysis.

We first focus on a comparison of the versions of RacerF. The difference between its newest version and the version competing in SV-COMP'25 lies mainly in improved escape analysis and preprocessing (loop unrolling in the under-approximating mode and assignment modification in the over-approximating mode). The changes show a positive impact on more reported data races but also a slowdown which is a consequence of preprocessing done in over-approximating mode. The slowdown mostly manifests for the active-thread analysis, which could be a performance bug in the implementation.

Interestingly, running RacerF in either the over- or under-approximating mode does not compromise its precision too much, leading to an additional 2 false positives (and 22 must-races classified just as may- and thus not reported) and 6 false negatives, respectively.

Compared to participants of SV-COMP'25, RacerF placed second in the competition ranking and third in our evaluation. Compared to Goblint, which never reports races, RacerF can produce correct results for more programs. Compared to model checker Deagle, which benefits from the fact that many programs are small and their loops can be fully unrolled, no version of RacerF ever runs out of time or memory.

Where RacerF shines is the ldv–linux–3.14–races benchmark subset containing 5 programs derived from the Linux kernel (all of them are race-free). None of the participants with a positive score can provide correct verdicts for those programs (except Goblint which handles one of them), but RacerF can correctly analyse each of them under two minutes.

8.2 Student Programs

To test our tool on more non-crafted programs, we performed an evaluation on a set of 116 student programs from [18]. The programs are rather short (around 100 – 300 LoC) and implement the same heavily concurrent algorithm parametric in the number of threads, but

Table 2 Results for student programs. The columns are true positives, false negatives, no race reported, race reported, out-of-resources, and time (excluding timeouts).

	Races (30)		Others (86)			
Analyser	TP	FN	NR	RR	OOR	Time [s]
RacerF	16	3	46	0	0	688
O2	7	23	86	0	0	54
Goblint$_{sv}$	0	3	46	0	0	41
Goblint	27	3	46	40	0	41
Deagle	3	1	6	0	78	1644

they differ in various intricate bugs (for their detailed discussion see [18]). We have identified 29 data races using dynamic analysers. However, there may be more that those tools missed. One additional race was discovered by all RacerF, O2, and Goblint. The issue could manifest only in a very rare situation when two threads simultaneously failed and called a clean-up function that assigns a global pointer without synchronization. The time limit was set to 60 seconds and the memory limit to 1GB.

We have compared with Goblint[5] and Deagle (version from SV-COMP'25) that performed better than us in SV-COMP, and with O2 (implemented in Coderrect[6]) that we selected as a recent representative of lightweight and scalable race detection tools.

The results are given in Table 2. As expected, since all the programs contain unbounded loops and also an unbounded number of threads is created, Deagle ran out of the resources on almost all programs and provided an answer in only 10 cases. Interestingly, Goblint and RacerF claimed *the same* 46 programs as race-free (but RacerF without providing any false positives). The three missed races by RacerF and Goblint are also the same and are caused by an undefined behaviour related to re-initialisation and destroying of mutexes that still may be used by some threads (neither of the tools supports detection of such problems). A detailed inspection suggests that both tools struggle with programs that use an array where each thread accesses only the element at its dedicated index.

Overall, RacerF achieves excellent precision on both sides. While O2 misses many races and Goblint reports many false alarms, RacerF is able to discover more than half of the races while not reporting any false alarms.

9 Conclusions and Future Work

We have presented a novel data race detector RacerF and performed a series of experiments indicating that RacerF achieves a nice compromise between precision and scalability both on crafted benchmarks and real-world programs. In the future, we would like to target other classes of concurrency bugs. We already have an implementation for deadlock detection based on top of our lockset analysis, which is not discussed in this paper. Another interesting direction is combination with dynamic approaches, e.g., guiding insertion of noise injection by results of static analysis.

[5] We use Goblint in the version from SV-COMP'25 but with the default configuration as the other available configurations did not improve the results and led to timeouts. Further, to provide more detailed results, we run Goblint via its SV-COMP wrapper that never reports detected races (Goblint$_{sv}$) as well as without it.

[6] The link `https://coderrect.com` from which we got version 1.1.3 is no longer available.

References

1 Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs: The Frama-C Software Analysis Platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

2 Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable Benchmarking: Requirements and Solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. doi:10.1007/s10009-017-0469-y.

3 Dirk Beyer and Jan Strejček. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Proc. of TACAS'25*, LNCS, 2025.

4 Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: Compositional static race detection. *Proc. of ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276514.

5 Allan Blanchard, Loïc Correnson, Tristan Le Gall, Jan Rochel, and Julien Signoles. Alias, Efficient May-alias and Points-to Analysis. https://www.frama-c.com/fc-plugins/alias.html.

6 Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. In *Proc. of VMCAI*, pages 112–130, 2017. doi:10.1007/978-3-319-52234-0_7.

7 François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications*, 1993.

8 David Bühler, Pascal Cuoq, and Boris Yakobowski. *The Eva Plug-in (For Frama-C 30.0)*. Available from https://frama-c.com/download/frama-c-eva-manual.pdf.

9 Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a Generic Graph Library using ML Functors. In *Symposium on Trends in Functional Programming*, 2007.

10 Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proc. of SEFM*, pages 233–247, 2012. doi:10.1007/978-3-642-33826-7_16.

11 Tomáš Dacík and Tomáš Vojnar. TDacik/Deadlock_and_Racer. Software, swhId: swh:1:dir:fc33aed78b227ea0738a89894b3dfa91b4f46844 (visited on 2025-06-18). URL: https://github.com/TDacik/Deadlock_and_Racer, doi:10.4230/artifacts.23614.

12 Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Mehdi Naouar, Andreas Podelski, and Claus Schätzle. Verification of Concurrent Programs Using Petri Net Unfoldings. In *Proc. of VMCAI*, pages 174–195, 2021. doi:10.1007/978-3-030-67067-2_9.

13 Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003. doi:10.1002/cpe.654.

14 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM, 2007. doi:10.1145/1273442.1250762.

15 Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003. doi:10.1145/1165389.945468.

16 Hongyu Fan, Zhihang Sun, and Fei He. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.*, 45(1), 2023. doi:10.1145/3579835.

17 J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-Based Testing of Concurrent Software. *Software Testing Verification and Reliability*, 25(3):272–309, 2015. doi:10.1002/stvr.1546.

18 Jan Fiedor and Tomas Vojnar. Noise-based Testing and Analysis of Multi-Threaded C/C++ Programs on the Binary Level. *2012 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012 - Proceedings*, 2012. doi:10.1145/2338967.233681.

19 C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009. doi:10.1145/1543135.1542490.

20 Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Proc. of CAV*, pages 355–365, 2019. doi:10.1007/978-3-030-25540-4_19.

21 Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). In *Proc. of TACAS'22*, pages 424–428, 2022. doi:10.1007/978-3-030-99527-0_25.

22 M. Heizmann, M. Bentele, D. Dietsch, X. Jiang, D. Klumpp, F. Schüssele, and A. Podelski. Ultimate Automizer and the Abstraction of Bitwise Operations (Competition Contribution). In *Proc. of TACAS'24*, LNCS 14572, pages 418–423, 2024. doi:10.1007/978-3-031-57256-2_31.

23 Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. Ultimate GemCutter and the Axes of Generalization. In *Proc. TACAS'22*, pages 479–483, 2022. doi:10.1007/978-3-030-99527-0_35.

24 Daniel Kästner, Antoine Miné, Laurent Mauborgne, X. Rival, Jérôme Feret, Patrick Cousot, Stephan Wilhelm, and Christian Ferdiand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *In ERTS'16*, 2016.

25 A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV'08*, pages 37–51, 2008. doi:10.1007/978-3-540-70545-1_7.

26 B. Liu, P. Liu, Y. Li, C.-C. Tsai, D. Da Silva, and J. Huang. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proc. of PLDI'21*. ACM, 2021. doi:10.1145/3453483.345407.

27 Antoine Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *Proc. of VMCAI*, pages 39–58, 2014. doi:10.1007/978-3-642-54013-4_3.

28 T. Nguyen, B. Fischer, S. Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *Proc. of ATVA'16*, volume 9938 of *LNCS*, 2016. doi:10.1007/978-3-319-46520-3_12.

29 S. Saan, J. Erhard, M. Schwarz, S. Bozhilov, K. Holter, S. Tilscher, V. Vojdani, and H. Seidl. Goblint: Abstract Interpretation for Memory Safety and Termination (Competition Contribution). In *Proc. of TACAS'24*, LNCS 14572, pages 381–386, 2024. doi:10.1007/978-3-031-57256-2_25.

30 Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997. doi:10.1145/265924.265927.

31 Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of POPL'96*, pages 32–41, 1996. doi:10.1145/237721.237727.

32 Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static Race Detection for Device Drivers: the Goblint Approach. In *Proc. of ASE*, pages 391–402, 2016. doi:10.1145/2970276.2970337.

33 J. Wu, Y. Tang, H. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012. doi:10.1145/2345156.2254090.

34 Boris Yakobowski and Richard Bonichon. *The Mthread plugin*. Available from `https://frama-c.com/download/frama-c-mthread-manual.pdf`.