Static Deadlock Detection in Low-Level C Code*

Dominik Harmim, Vladimír Marcin, Lucie Svobodová, and Tomáš Vojnar
Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract We present a novel scalable deadlock analyser L2D2 capable of handling C code with low-level unstructured lock manipulation. L2D2 runs along the call tree of a program, starting from its leaves, and analyses each function just once, without any knowledge of the call context. L2D2 builds function summaries recording information about locks that are assumed or known to be locked or unlocked at the entry, inside, and at the exit of functions, together with lock dependencies, and reports warnings about possible deadlocks when cycles in the lock dependencies are detected. We implemented L2D2 as a plugin of the Facebook/Meta INFER framework and report results of experiments on a large body of C as well as C++ code illustrating the effectiveness and efficiency of L2D2.

1 Introduction

Nowadays, programs often use *multi-threading* to utilise the many processors of current computers better. However, concurrency does bring not only speed-ups but also a much larger space for nasty errors easy to cause but difficult to find. The reason why finding errors in concurrent programs is particularly hard is that concurrently running threads may *interleave* in many different ways, with bugs hiding in just a few of them. Such interleavings are hard to discover by testing even if it is many times repeated.

Coverage of such rare behaviours can be improved using approaches such as *systematic testing* [22] and *noise-based testing* [7,9,10]. Another way is to use *extrapolating dynamic checkers*, such as [11,12], which can report warnings about possible errors even if those are not seen in the testing runs, based on spotting some of their symptoms. Unfortunately, even though such checkers have proven quite useful in practice, they can, of course, still miss errors. Moreover, monitoring a run of large software through such checkers may also be quite expensive.

On the other hand, 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, which is usually given in the number of context switches. However, so far, the scalability of these techniques is not sufficient to handle truly large industrial code, even when combined with methods such as *sequentialisation* [17,19], which represents one of the most scalable approaches in the area.

An alternative to the above approaches, which can scale better than model checking and can find bugs not found dynamically (though for the price of potentially missing some errors and/or producing false alarms), is offered by approaches based on *static analysis*, e.g., in the form of *abstract interpretation* [5] or *data-flow analysis* [15]. The former approach is supported, e.g., in Facebook/Meta INFER — an open-source framework for creating highly scalable, compositional, incremental, and interprocedural static analysers based on abstract interpretation [4].

INFER provides several analysers that check for various types of bugs, such as buffer overflows, null-dereferencing, or memory leaks. However, most importantly, INFER is

* The work was supported by the project 20-07487S of the Czech Science Foundation and the Brno Ph.D. Talent Scholarship Programme.

a *framework* for building new analysers quickly and easily. As for *concurrency-related bugs*, INFER provides support for finding some forms of *data races* and *deadlocks*, but it is limited to *high-level* Java and C++ programs only and fails for C programs, which use a *lower-level lock manipulation* [1,6].

In this paper, we propose a *deadlock checker* that fits the common principles of analyses used in INFER and is applicable even to *C code* with *lower-level lock manipulation*. Our checker is called L2D2 for "low-level deadlock detector".

As is common in INFER, L2D2 computes function summaries *upwards* along the call tree, starting from its leaves, and analyses every function just once, without knowing anything about its call contexts. The summaries contain various pieces of information about locks that are assumed to be locked/unlocked at the entry of a function, that may be locked/unlocked at the end of the function, that may be both locked and unlocked inside a function, as well as about lock dependencies (saying that some lock is locked while another is still held). If L2D2 detects a loop in the lock dependencies, it warns about possible deadlocks. L2D2 uses multiple heuristics to reduce the number of false alarms, such as detection of locks serving as gate locks.

To show the effectiveness and efficiency of L2D2, we present experiments in which we managed to apply it to 930 programs with 10.3 million lines of code (MLoC) in total, out of which 8 contained known deadlocks. L2D2 rediscovered all the deadlocks, and, out of the remaining 922 programs, it claimed 909 deadlock-free and reported false alarms for 13 of the programs only. The code included benchmarks coming from the CPROVER tool derived from the Debian GNU/Linux distribution, the code of the `grep`, `sort`, `tgrep`, and `memcached` utilities, and the EPROSIMA/FAST-DDS middleware.

Related Work To the best of our knowledge, L2D2 is the only currently existing, publicly available, *compositional static deadlock analyser* for *low-level code*. Below, we briefly discuss approaches that we consider to be the closest to it.

RACERX [8] is a top-down, non-compositional, flow-sensitive and context-sensitive analysis for C programs based on computing so-called *lock sets*, i.e., sets of currently held locks, constructing a *static lock-order graph*, and reporting possible deadlocks in case of cycles in it. It employs various heuristics to reduce false-positive reports. Some of the ideas concerning the lock sets are similar to those used in L2D2, and some of the heuristics used in RACERX inspired those used in L2D2.

The deadlock analyser implemented within the CPROVER framework [16] targets C code with POSIX threads and uses a combination of multiple analyses to create a context-sensitive and sound analysis. It also builds a lock-order graph and searches for cycles to detect deadlocks. Its most costly phase is the pointer analysis used. An experimental comparison with this tool is given in Section 4.

STARVATION [2] is implemented in the INFER framework, and hence it is bottom-up, context-insensitive, and compositional. It detects deadlocks by deriving lock dependencies for each function and checking whether some other function uses the locks in an inverse order. It is thus similar to L2D2, but STARVATION is limited to *high-level* Java and C++ programs with *balanced locks* only. Moreover, it implements many heuristics explicitly tailored for Android Java applications.

GOODLOCK [13] is a well-known *dynamic analysis* for Java programs implemented in Java PathFinder (JPF) [14]. As a representative of dynamic analysers, it inherits their dependence on the concrete execution (or executions) of the given software

seen for detecting possible deadlocks. It monitors the lock acquisition history by creating a *dynamic lock-order graph*, followed by checking the graph for the existence of deadlock candidates by searching for cycles in it. To increase chances of spotting even rarely occurring deadlocks, not directly seen in the given execution, it makes *deadlock predictions* based on an exponential number of permutations of a single execution. A drawback of this approach is that it may produce a high rate of false positives.

AIRLOCK [3] is one of the state-of-the-art dynamic deadlock analysers. It adopts and improves the basic approach from GOODLOCK by applying various optimisations to the extracted lock-order graph. Moreover, AIRLOCK, operating on-the-fly, runs a polynomial-time algorithm on the lock graph to eliminate parts without cycles, followed by running a higher-cost algorithm to detect actual lock cycles.

2 Static Deadlock Detection in Low-Level Concurrent C Code

This section presents the design of the L2D2 analyser. We first introduce the main ideas of the analysis, and then discuss it in more detail.

As already mentioned, L2D2 is designed to handle *C code* with *low-level, unstructured lock manipulation*. It does not start the analysis from the entry code location as done in classical inter-procedural analyses based, e.g., on [20]. Instead, it performs the analysis of a program function-by-function *along the call tree*, *starting from its leaves*. Therefore, each function is analysed just once without any knowledge of its possible call contexts. For each analysed function, L2D2 derives a *summary* that consists of a *pre-condition* and a *post-condition*. The summaries are then used when analysing functions higher up in the call hierarchy. The obtained analysis is *compositional* on the level of functions, and when used in conjunction with some version control system, it allows one to focus on *modified functions* and their dependants only with no need to re-analyse the unchanged functions (which is typically a vast majority of the code).

```
1  void f(Lock *L3') {
2    lock(&L4);
3    unlock(&L3');
4    lock(&L2);
5    ...
6    unlock(&L4); }
7  void *t1(...) {
8    lock(&L1);
9    lock(&L3);
10   ...
11   f(&L3);
12   unlock(&L1); }
13 void *t2(...) {
14   lock(&L2);
15   ...
16   lock(&L1); }
```

Listing 1. A sample low-level code causing a deadlock

L2D2 does not perform a classical *alias analysis*, i.e., a precise analysis for saying whether some pairs of accesses to locks may alias (such an analysis is considered too expensive — no such sufficiently precise analysis works compositionally and at scale). Instead, L2D2 uses *syntactic access paths* [18], computed by the INFER framework, to represent lock objects. Access paths represent heap locations via expressions used to access them. In particular, an access path consists of a base variable followed by a sequence of field selectors. According to [1], the access paths' syntactic equality is a reasonably efficient way to say (in an under-approximate fashion) that heap accesses touch the same address. The mechanism is indeed successfully used, e.g., in the production checker RACERD [1] to detect data races in real-world programs.

We will use Listing 1 to illustrate some ideas behind L2D2. It works in two phases. In the first phase, it computes a summary for each function by looking for lock and unlock events (`lock`/`unlock` calls in the listing) in the function. When a call of a user-defined function appears in the analysed function during the analysis (like on line 11 in the listing), L2D2 uses a summary of the function if available. Otherwise, the function is

analysed on demand, effectively analysing the code bottom-up (when a recursive call is encountered, it is skipped). The summary is then applied to an *abstract state* at the call site. In the listing, the summary of `f` will be applied to the abstract state of `t1`.

In the second phase, L2D2 looks through all computed summaries of the analysed program and focuses on so-called *dependencies* that are a part of the summaries. These dependencies represent possible locking sequences of the analysed program. The obtained set of dependencies is interpreted as a relation. L2D2 computes the transitive closure of this relation and reports a deadlock if some lock depends on itself in the closure. If we run L2D2 on the code in Listing 1, it will report a potential deadlock due to the cyclic dependency between the locks `L1` and `L2` that arises when the thread `t1` holds `L1` and waits on `L2` and the thread `t2` holds `L2` and waits on `L1`.

2.1 Computing Function Summaries

This section outlines the structure and computation of the summaries used by L2D2 when analysing some function f. Intuitively, the pre-condition expresses what states of locks f expects from its callers, and the post-condition reflects the effect of f on the locks. More precisely, the post-condition includes the `lockset` and `unlockset` sets, holding information about which locks *may be locked* and *unlocked*, resp., at the exit of f. The pre-condition consists of the `locked` and `unlocked` sets, stating which locks are *expected to be locked* and *unlocked*, resp., upon a call of f. Note that the `locked`/`unlocked` sets are maintained but not used in the basic algorithm introduced later in this section. They are used to detect possible *double-locking/unlocking*, see Section 3. Next, the summary's post-condition contains the so-called *lock dependencies* (`deps`) in the form of pairs of locks (`L2`, `L1`) where locking of `L1` was observed while `L2` was locked. This exact situation can be seen in Listing 1 on line 16.

```
f:  PRE-CONDITION
 locked={L3'}
 unlocked={L2, L4}
    POST-CONDITION
 lockset={L2}
 unlockset={L3', L4}
 wereLocked={L2, L4}
 deps={(L4, L2)}
 order={(L3', L2)}
t1: PRE-CONDITION
 unlocked={L1, L2, L3, L4}
    POST-CONDITION
 lockset={L2}
 unlockset={L1, L3, L4}
 wereLocked={L1, L2, L3, L4}
 deps={(L1, L2), (L1, L3),
  (L1, L4), (L3, L4)}
t2:  PRE-CONDITION
 unlocked={L1, L2}
     POST-CONDITION
 lockset={L1, L2}
 wereLocked={L1, L2}
 deps={(L2, L1)}
```

Listing 2. Summaries for the functions from Listing 1

Two more sets are a part of the summary's post-condition. First, the `wereLocked` set contains information on which *locks may be locked and then again unlocked* within f. This is needed to detect lock dependencies with such locks in functions higher up in the call hierarchy. Such a situation can be seen in Listing 1. The lock `L4` is locked and then unlocked again within the function `f`. In this case, the lock will not be in `lockset`, and we would have no information that it was locked there. Consequently, we would not create any lock dependencies w.r.t. this lock. However, this lock will appear in `wereLocked`, so we can create dependencies with it (like the dependency (`L1`, `L4`) in the function `t1` when calling `f` on line 11, which could not be created otherwise).

The last sets that are a part of L2D2's post-conditions are denoted as the `order` sets. They comprise pairs of locks (`L3'`, `L2`) where locking of `L2` was seen when `L3'` was unlocked before within the same function. Such a pair is produced, e.g., on line 4 in Listing 1. These sets help L2D2 to better determine the order of operations in functions. Without it, we would create, e.g., the non-existent dependency (`L3`, `L2`) in the function `t1` when calling `f` on line 11. It should not be created because

Algorithm 1: Lock acquisition

```
   Data: lock L being locked; abstract state S
1  def lock(L, S):
2      if L ∉ S.locked ∪ S.unlocked then
3          S.unlocked ← S.unlocked ∪ {L};
4      S.lockset ← S.lockset ∪ {L};
5      S.unlockset ← S.unlockset \ {L};
6      S.wereLocked ← S.wereLocked ∪ {L};
7      S.deps ← S.deps ∪ (S.lockset × {L});
8      S.order ← S.order ∪ (S.unlockset × {L});
```

Algorithm 2: Lock release

```
   Data: lock L being unlocked; abstract state S
1  def unlock(L, S):
2      if L ∉ S.locked ∪ S.unlocked then
3          S.locked ← S.locked ∪ {L};
4      S.unlockset ← S.unlockset ∪ {L};
5      S.lockset ← S.lockset \ {L};
```

Algorithm 3: Integrating a summary of a callee

```
   Data: summary χ of a callee; abstract state S
1  def apply_summary(χ, S):
2      χ ← replace_formals_with_actuals(χ);
3      if ∃L : L ∈ χ.unlocked ∧ L ∉ S.unlockset then S.unlocked ← S.unlocked ∪ {L};
4      if ∃L : L ∈ χ.locked ∧ L ∉ S.lockset then S.locked ← S.locked ∪ {L};
5      S.lockset ← (S.lockset ∪ χ.lockset) \ χ.unlockset;
6      S.unlockset ← (S.unlockset \ χ.lockset) ∪ χ.unlockset;
7      S.wereLocked ← S.wereLocked ∪ χ.wereLocked;
8      S.deps ← S.deps ∪ ((S.lockset × χ.wereLocked) \ χ.order);
```

`L3` is unlocked in `f` on line 3 before `L2` is locked on line 4. Note that the lock `L3` from the function `t1` is passed to `f` as `L3'`. We resolve such situations by replacing the function's formal parameters with the actual ones at the concrete call site.

Listing 2 gives the summaries for the functions in Listing 1, omitting the empty sets.

The high-level algorithm for the summary's computation is given in Algorithms 1–3. Algorithm 1 shows how the abstract state is updated whenever locking occurs during the analysis. First, it updates the pre-condition by adding the lock to the `unlocked` set if this locking is the first operation with that lock in the given function f (lines 2–3). Intuitively, this reflects that the lock should be unlocked before calling f; otherwise, we would encounter double-locking. Next, the lock acquisition takes place, meaning that the lock is added to `lockset` and removed from `unlockset` (lines 4–5). Moreover, the lock is added to `wereLocked` (line 6). Finally, we derive new dependencies and `order` edges by considering all pairs (L', L) where L' is an element of `lockset` and `unlockset`, resp., and L is the acquired lock (lines 7–8). Algorithm 2 then updates the abstract state when some lock is released. It is analogical to the algorithm for locking, but it does not update the `wereLocked`, `deps`, and `order` sets.

Algorithm 3 integrates a callee's summary with the abstract state of an analysed function. Initially, the summary is updated by replacing the formal parameters with the actual ones (line 2). We also check that all the locks that should be locked/unlocked before calling the callee are present in `lockset`/`unlockset`, resp. If they are not, they must be locked/unlocked even before the currently analysed function. Hence, we update the pre-condition (lines 3–4). On lines 5–7, the `lockset`, `unlockset`, and `wereLocked` sets are appropriately modified. At last, new dependencies between the currently held locks and locks acquired in the callee are introduced (line 8). However, we exclude all the dependencies from the `order` set to avoid adding such (L', L) dependencies where L' was unlocked before locking L in the callee.

As L2D2 is based on abstract interpretation, we must further define the *join* operator for combining states along *confluent program paths* (e.g. in `if` statements), the *entailment* operator allowing the analysis to detect it has reached a fixpoint and stop, and the *widening* operator accelerating the analysis of loops. Since we are interested in locking patterns along any possible path, we define the join operator as the union of incoming states' values for all the sets in the summaries. The entailment operator is defined as testing for a subset on all the sets. The widening operator is made equal to the join operator as we are working with summaries on finite and not too large domains.

2.2 Reporting Deadlocks

Checking for deadlocks takes place after the summaries for all functions in the analysed program are computed. L2D2 then merges all of the derived lock dependencies into one set R. This set is interpreted as a relation, and its transitive closure R^+ is computed. If any lock L depends on itself in the closure, i.e., $(L, L) \in R^+$, a potential for a deadlock has been detected. For deadlocks using two locks, L2D2 then looks for dependencies that cause the deadlock. In particular, it looks for a lock L' s.t. $(L, L') \in R^+ \wedge (L', L) \in R^+$ and reports the dependencies (a generalisation to more locks is, of course, possible).

3 Increasing Analysis Accuracy

L2D2 further implements three heuristics intended to decrease the number of possible false alarms. We now introduce the two most important (with the third one being a simple support for recursive locks).

As *double-locking/unlocking* errors are quite rare in practice, the first heuristic uses their detection as an indication that the analysis is over-approximating too much. Instead of reporting such errors, L2D2 resets (some of) the working sets. Namely, if a lock acquisition leads to double-locking, it is assumed that L2D2 followed some non-existent path, and `lockset` is no longer trustworthy. Therefore, it is erased, and the only lock left in it is the currently acquired one as this is the only one about which we can safely say it is locked. For that, the following statement is added to Algorithm 1: **if** $L \in S.lockset$ **then** $S.lockset \leftarrow \{L\}$;. When releasing a lock, we then check whether it may already be unlocked. If so, `lockset` is erased, eliminating any dependencies that the locking error would cause. For that, we add the following to Algorithm 2: **if** $L \in S.unlockset$ **then** $S.lockset \leftarrow \emptyset$;. Finally, we check double-locking/unlocking when a function call is encountered. We ask whether some lock that should be locked/unlocked in the callee is currently released/held, resp. If such a lock is found, it is assumed that L2D2 used a non-existent path to reach the function call, and so `lockset` is discarded, and the `lockset` of the callee will be used instead. We implement this by adding the following to Algorithm 3: **if** $(S.lockset \cap \chi.unlocked \neq \emptyset) \vee (S.unlockset \cap \chi.locked \neq \emptyset)$ **then** $S.lockset \leftarrow \chi.lockset$;.

The second heuristic used in L2D2 is the detection of so-called *gate locks* [13], i.e., locks guarding other locks (upon which deadlocks on the nested locks are not reported). Whenever we detect a possible deadlock — represented by two reverse dependencies $d_1 = (L, L')$ and $d_2 = (L', L)$ — we check whether the same gate lock protects them. If so, we do not report a deadlock. We check this by computing the intersection of the `guards`, i.e., all locks locked before the program points where the dependencies d_1 and d_2 were captured. In particular, we do not report a deadlock for dependencies d_1 and d_2 if $\texttt{guards}(d_1) \cap \texttt{guards}(d_2) \neq \emptyset$.

4 Experimental Evaluation

L2D2 has been implemented in OCaml as a plugin of INFER, and it is publicly available[1]. We now report on various experiments we have performed with it. All of the experiments were run on a machine with the AMD Ryzen 5 5500U CPU, 15 GiB of RAM, 64-bit Ubuntu 20.04.4 LTS, using INFER version v1.1.0-0e7270157.

In our first set of experiments, we have applied L2D2 on a set of 1,002 C programs with POSIX threads derived from a Debian GNU/Linux distribution, originally prepared for evaluating the static deadlock analyser based on the CPROVER framework proposed in [16]. The benchmark consists of 11.3 MLoC. Eight of the programs contain a known deadlock. Like CPROVER, L2D2 was able to detect all the deadlocks. The results for the remaining 994 programs are shown in Table 1 (for L2D2, `mode 1`/`mode 2` refer to using/not using the double-locking-based heuristic), with some more details also in Table 2 discussed below. We can see that, in `mode 1`, L2D2 produced 11 false alarms only (77 programs failed to compile since the INFER's front-end did not support some of the constructions used). We find this very encouraging, considering that the CPROVER's deadlock detector produced 114 false alarms. Moreover, L2D2 consumed 83 minutes only whereas CPROVER needed 4 hours to handle the programs it correctly analysed, producing 453 timeouts (w.r.t. a 30-minute time limit), and ran out of the available 24 GB of RAM in 135 cases (according to [16], the results were obtained on Xeon X5667 at 3 GHz running Fedora 20 with 64-bit binaries).

Table 1. Results of L2D2 and CPROVER on non-deadlocking programs of the CPROVER test-suite

checker	**programs claimed safe**	programs raising alarms	programs failed to analyse
CPROVER	**292**	114	588
L2D2$_{\texttt{mode 1}}$	**906**	11	77
L2D2$_{\texttt{mode 2}}$	**896**	21	77

Table 2 provides our further experimental results. Unlike Table 1, the table gives not only numbers of programs in which an alarm was raised, but it gives concrete numbers of the alarms (more alarms can be raised in a single program). Moreover, it shows how L2D2 behaved on multiple further real-life programs. In particular, EPROSIMA/FAST-DDS 2.6.1 is a C++ implementation of the Data Distribution Service of the Object Management Group. For its analysis, we replaced the C++ guard lock used, which is so far not supported by L2D2, by a normal lock (exploiting the fact that INFER automatically adds all needed `unlock` calls). Next, we analysed `memcached` version 1.6.10, a distributed memory object caching system. The source code of this program was pre-processed by FRAMA-C [21], and we report on the size of the pre-processed code (likewise with all the further mentioned programs). Finally, we also analysed `grep` 3.7, `tgrep` (a multi-threaded version of `find` combined with `grep` by Ron Winacott), and GNU Coreutils `sort` 8.32. The alarms

Table 2. Detailed results on EPROSIMA/FAST-DDS, `sort`, `grep`, `memcached`, `tgrep`

	kLoC	alarms `mode 1`	alarms `mode 2`	dead-locks	runtime (mm:ss)
FAST-DDS	110	3	6	0	06:53
`memcached`	31	6	7	0	00:08
`sort`	7.2	0	0	0	00:02
`grep`	8.7	0	0	0	00:03
`tgrep`	2.4	0	0	0	00:01
CPROVER	10,164	23	80	8	83:23

[1] `https://github.com/svobodovaLucie/infer`

raised for FAST-DDS are false alarms caused by some intricacy of C++ locks for which L2D2 was not prepared. We were not able to check the status of the alarms raised for `memcached`, but we consider them likely false alarms. However, we find the results provided by L2D2 as quite encouraging since the numbers of false alarms are low w.r.t. the number of programs and their extent, and, moreover, we believe that there is space for further improvements (especially, but not only for C++ locks).

References

1. S. Blackshear, N. Gorogiannis, P. O'Hearn, and I. Sergey. RacerD: Compositional Static Race Detection. *Proc. of ACMPL*, 2(OOPSLA):144:1–144:28, 2018.
2. J. Brotherston, P. Brunet, N. Gorogiannis, and M. Kanovich. A Compositional Deadlock Detector for Android Java. In *Proc. of ASE'21*. IEEE, 2021.
3. Y. Cai, R. Meng, and J. Palsberg. Low-Overhead Deadlock Prediction. In *Proc. of ICSE'20*. ACM, 2020.
4. C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, et al. Moving Fast with Software Verification. In *Proc. of NFM'15*, volume 9058 of *LNCS*. Springer, 2015.
5. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approx. of Fixpoints. In *Proc. of POPL'77*. ACM, 1977.
6. D. Distefano, M. Fähndrich, F. Logozzo, and P. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.
7. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *Concur. Computat.: Pract. Exper.*, 15(3–5):485–499, 2003.
8. D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. of SOSP'03*. ACM, 2003.
9. J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-Based Testing of Concurrent Software. *Softw. Test. Verif. Reliab.*, 25(3):272–309, 2015.
10. J. Fiedor, M. Mužikovská, A. Smrčka, O. Vašíček, and T. Vojnar. Advances in the ANaConDA Framework for Dynamic Analysis. . . . In *Proc. of ISSTA'18*. ACM, 2018.
11. C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009.
12. C. Flanagan, S. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of PLDI'08*. ACM, 2008.
13. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
14. K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *Inter. Jour. on STTT*, 2(4):366–381, 2000.
15. G. Kildall. A Unified Approach to Global Program Optimization. In *Proc. of POPL'73*. ACM, 1973.
16. D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound Static Deadlock Analysis for C/Pthreads. In *Proc. of ASE'16*. ACM, 2016.
17. A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV'08*. Springer, 2008.
18. J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbound. Access Paths. In *Proc. of ASE'15*. IEEE, 2015.
19. T. Nguyen, B. Fischer, S. Torre, and G. Parlato. Lazy Sequentialization for the Safety Verif. of Unbound. Concur. Programs. In *Proc. of ATVA'16*, volume 9938 of *LNCS*. Springer, 2016.
20. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of POPL'95*. ACM, 1995.
21. J. Signoles, P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, and B. Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, 27, 2012.
22. 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.