BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2020
http://excel.fit.vutbr.cz

Static Deadlock Detection in Frama-C

Tomáš Dacík*

Abstract
Frama-C is a platform for static analysis of source codes written in the C language. It provides a wide range of analysers usually based on EVA – Frama-C's value analysis plugin. Despite some attempts to support analysis of multi-threaded code have been done in Frama-C, the whole platform is currently limited to analysis of sequential code only. In this paper, we present *Deadlock*, a new plugin of Frama-C focused on deadlock detection. Together with the core algorithm of deadlock detection, we present a technique our analyser uses to handle multi-threaded code partially as a sequential one, which allows us to improve the precision of our analysis by using existing plugins of Frama-C. In our experimental evaluation, we show that our tool is able to handle real-world C code with a high precision.
Keywords: Static Analysis — Deadlock Detection — Frama-C

Supplementary Material: Deadlock repository

*xdacik00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In a majority of concurrent programs, some kind of synchronisation is necessary to guarantee consistency of data. However, incorrectly used synchronisation mechanisms may lead to another class of concurrency issues, like, for example, *deadlocks*. In this work, we focus on deadlocks caused by incorrect usage of *locks*, low-level synchronisation primitives, often used in the C language. In this particular case, a deadlock is defined as a situation where each thread from some set is holding a lock and waiting for another lock, that is held by (possibly the same) thread from the set. An example of a simple deadlock is given in Listing 1.

Our implementation of deadlock detection is inspired by the tool RacerX [1] and the deadlock analysis implemented in the CPROVER framework [2]. While RacerX is explicitly designed to handle large code bases and therefore does not do any pointer analysis and resigns on soundness, CPROVER tries to be sound, which leads to its slow running times as well as a lot of false positives. Our goal is to design an analyser that combines both presented solutions, i.e., can use the existing pointer analysis available in Frama-C, but with the stress put on detection of likely deadlocks rather than soundness.

Listing 1. A simple C program with a deadlock between threads *thread1* and *thread2*

```
1   void f() {
2       pthread_mutex_lock(&mutex1);
3       if (...)
4           pthread_mutex_unlock(&mutex1);
5   }
6
7   void g() {
8       pthread_mutex_lock(&mutex1);
9       pthread_mutex_lock(&mutex2);
10  }
11
12  void *thread1 (void *v) {
13      pthread_mutex_lock(&mutex2);
14      f();
15  }
16
17  void *thread2 (void *v) {
18      g();
19  }
```

Another deadlock detection approach for low-level C programs is implemented in the L2D2 (Low-Level Deadlock Detector) plugin [3] of Facebook Infer. It uses a completely different method – an incremental and compositional analysis based on analysing each function without its calling context. While this approach promises to be more scalable, it can also in principle produce more false alarms.

We also note that there exist many more static deadlock analysers, but many of them target higher-level languages (such as Java or C++) or are much more heavyweight. A discussion of such tools is beyond the scope of this paper.

2. Frama-C

Frama-C is an open-source platform for static analysis of source codes written in the C programming language. Frama-C has a modular, plugin-based architecture. Out of the existing plugins, the most heavily used is the EVA (Evolved Value Analysis) plugin [4], which computes an over-approximation of sets of possible values of variables at each program point. Its results can be used for proving the absence of generic errors or assertions written in a specialised assertion language. They also serve as the input for other plugins implementing, for example, program slicing, various code optimisations, or test-input generation. There are also plugins for deductive verification, which are, however not relevant in the context of this paper.

Only sequential code can be analysed by the current version of EVA and consequently by all plugins based on it. As we have already mentioned in the introduction, some attempts to implement analysis of concurrent code have been done in Frama-C, but they were rather experimental and are no longer under active development. Examples of such attempts include the Mthread plugin [5] focused on data race detection, whose source code is unfortunately proprietary, and Conc2Seq [6] for translating concurrent code and its specification into sequential code simulating the original code and checking the given specification. This process is limited only for a subset of the C language.

3. Thread Analysis

Our analysis runs in two phases. In the first phase, we compute possible initial states of different threads (including information about which thread can be started at all). In the second phase, we perform a lockset analysis in which we analyse each thread as a sequential program assuming that it is started from the computed initial state. Here, note that we use the term *thread* as an abstraction representing all threads (instances that could be created during execution of a program) with the same entry point (and hence the same control).

In this section, we concentrate on the first phase of the analysis, namely, the computation of which threads can be created and with which initial states in terms of possible values of global variables and values of arguments passed to threads. The main idea is to use a fixpoint algorithm that runs as long as new threads are discovered. Each iteration of this fixpoint computation employing a nested fixpoint computation that iterates over so-far known threads, analyses them through EVA, and propagate information between them through thread creation statements only. This way, the possibility of creating new threads may be discovered. These threads will then be analysed in another iteration of the outer loop. Note that this approach under-approximates the real behaviour of the threads since no thread interleaving is considered. This is a design decision which we have done for the sake of efficiency of our analysis. While the analysis can indeed under-approximate the real behaviour, in the second phase, we are mainly interested in the parameters of lock/unlock functions, i.e., identifiers of locks, which are usually not that much influenced by thread interleaving in practice.

Algorithm 1: Computation of initial states of threads

```
   Input : create_stmts ... statements where
           threads can be created
 1 function build_graph(threads)
 2     G = empty_graph()
 3     foreach t ∈ threads, s ∈ create_stmts do
 4         set_active_thread(t)
 5         if is_reachable_by_thread(s,t) then
 6             children = get_threads(s)
 7             foreach child ∈ children do
 8                 G.add_edge(t,s,child)
 9             end
10         end
11     end
12     return G
13
14 function analyse_threads()
15     i = 0
16     G^0 = build_graph({main})
17     do
18         i = i + 1
19         Ĝ^i = compute_fixpoint(G^(i−1))
20         threads = Ĝ^i.get_nodes()
21         G^i = build_graph(threads)
22     while G^i ≠ Ĝ^i
23     return G^i
```

Our method of computing initial states is formalised in Algorithm 1. The function `build_graph` is used to construct a graph encoding which thread can create which other threads through which thread-create statements based on the current approximation of the

possible initial states of the threads. The function `set_active_thread` (line 4) is implemented as a wrapper over EVA and used to set its context according the so-far computed initial states of the given thread. For each create statement that is found reachable by EVA from the initial state of the thread being examined, we use EVA to find threads it can create and add corresponding edges to the graph.

The function `analyse_threads` first builds a graph based only on the initial state of the main thread, containing every thread that can be created from the main. Once new initial states are computed, new graphs are iteratively computed on line 21. To update initial states of the threads, we propagate states of their parents in the create statements (line 19). To handle programs with nested or even cyclic dependencies between threads, we compute a fixpoint of a function propagating states over the graph. The fixpoint computation over the graph is implemented using the OCamlgraph library[1]. For programs where threads are created in the main thread only, one iteration of the loop between lines 17 and 22 suffices. However, for more complex programs where the computation of the initial states leads to discovering new threads or dependencies, more iterations of the loop are necessary – we loop until the computed graphs stop changing.

We illustrate the algorithm on the program from Listing 2. It starts with the set $create\_stmt$ containing $stmt6$ and $stmt16$. First, we compute G^0 using the `build_graph` function. Since only $stmt16$ is reachable from the main thread, a graph with the single edge $main \xrightarrow{stmt16} thread1$ will be returned. The fixpoint computation over this graph is trivial – the state of $main$ at $stmt16$ is propagated as the initial state of $thread1$. Afterwards, we check whether a new thread can be discovered based on new initial states. We find that $thread2$ can be created from $thread1$ and add the corresponding edge $thread1 \xrightarrow{stmt6} thread2$ to the graph. The initial state of $thread2$ is computed analogously. Since there is no other thread, we return the initial states computed as follows (thread arguments are ignored):

$$thread1 : \{i \mapsto \{0\}\},\ thread2 : \{i \mapsto \{1\}\}$$

Note that the incrementation on line 17 is not reflected in the initial states of $thread1$ and $thread2$ because it is done after the thread creation.

[1] http://ocamlgraph.lri.fr/index.en.html

Listing 2. A program with nested threads (interfaces of thread-create functions are simplified)

```
1  int i = 0;
2
3  void *thread1 (void *v) {
4      i++;
5      pthread_t t;
6      pthread_create(&t, thread2);
7      return NULL;
8  }
9
10 void *thread2 (void *v) {
11     return NULL;
12 }
13
14 int main() {
15     pthread_t t;
16     pthread_create(&t, thread1);
17     i++;
18     return 0;
19 }
```

4. Lockset Analysis

The key part of our analyser is a lockset analysis inspired by the tool RacerX [1]. In RacerX, however, no pointer analysis is used, and so we had to extend its methods for this purpose. The term *lockset* refers to a set of locks that a thread holds at a particular program point. The result of the lockset analysis is the set of possible locksets for each program statement. Based on this information, we can construct a lock-order graph (further referred to simply as a lockgraph). An edge $a \to b$ in the lockgraph indicates that some thread tries to acquire lock b while already holding lock a.

Our lockset analysis is performed for each thread detected in the previous phase by a depth-first traversal of its control flow graph. The traversal is implemented as path-insensitive, i.e., all conditions are resolved non-deterministically. The analysis is started with the empty lockset, which is modified by the locking and unlocking operations according to the transfer function defined as follows ($[\![p]\!]$ denotes the set of possible values of the variable p):

$$t_{stmt}(ls) = \begin{cases} \{ls \cup \{l\} \mid l \in [\![p]\!]\} & \text{if } stmt \text{ is lock}(p) \\ \{ls \setminus \{l\} \mid l \in [\![p]\!]\} & \text{if } stmt \text{ is unlock}(p) \\ \{ls\} & \text{otherwise} \end{cases}$$

In the transfer function, the evaluation of the parameters (identifiers of locks) ignores the calling context in order to facilitate usage of summaries computed as described in Section 4.1. After applying the transfer function, the analysis is forked for each pair consisting of a successor statement and possible lockset.

Let us consider the following pseudocode as an example:

```
f() {           // entry lockset = {}
    lock(p);    // [[p]] = {m1,m2}
    unlock(p);  // [[p]] = {m1,m2}
}
```

Applying the transfer function on the first statement results into the set of locksets $\{\{m1\},\{m2\}\}$. The rest of the function is then analysed separately for $\{m1\}$ and $\{m2\}$. In both cases, after applying the transfer function on the second statement, we assume that both $m1$ as well as $m2$ can be unlocked despite the fact that one of them was not locked. In other words, we over-approximate the real behaviour considering all combinations of locking and unlocking in such a case. The exit set of locksets of f will then be $\{\{m1\},\{m2\}\}$.

4.1 Function Summaries

Function summaries are an efficient way to speed up interprocedural analysis. In our analysis, function summaries are represented by a mapping from pairs (*function*, *entry lockset*) to a set of exit locksets. The interpretation is the following: if the *function* is called with the *entry lockset*, the result is the union of the sets of locksets at each of its exit points. For example, the analysis of the function f from Listing 1 called from line 14 with the entry lockset containing `mutex2` will produce the following summary:

$$\{(f,\{mutex2\}) \mapsto \{\{mutex1,mutex2\},\{mutex2\}\}\}$$

Since the evaluation of the locks used is done regardless of the calling context, f will produce the same result at every other call site with the given entry lockset. In theory, each function could be analysed with each possible lockset, which means up to 2^n times where n is the number of locks used in the program. However, functions in real programs usually release all locks they acquired, and if the pointer analysis is not too imprecise, only small locksets are created. This implies that the majority of functions are not analysed many times. Our experiments described in Section 5 show that for a subset of programs where we detected some locking operations, functions are on average analysed 1.89 times only.

4.2 Lockgraph Construction

When updating the locksets, whenever a lock l is added to a nonempty lockset ls, a set of edges is added to the lockgraph. The set is computed as $ls \times \{l\}$. To track information of the origin of the edge, each edge is labelled by a set of traces. These traces are created by concatenating call stacks that lead to locking both of the locks as described in Section 4.5. The final step is to check whether there are cycles in the resulting graph, denoting possible deadlocks.

In this step, so-called *self-deadlocks*, i.e., deadlocks caused by a single thread on a single lock are ignored by default because they could lead to many false positives.

4.3 Context Sensitivity

The context insensitive evaluation of locking parameters may cause significant imprecision when wrappers of locking functions are used. An example of such a situation is given in Listing 3. Without context-sensitivity, the evaluation of the variable `m` on line 2 will always be the set of all mutexes used in the program. As a result, the analyser will assume that function `lock_wrapper` can lock any mutex. Then, on line 7, besides the real dependency *mutex1* $\rightarrow$ *mutex2*, the dependency *mutex2* $\rightarrow$ *mutex1* will also be created. Generally, such a situation results in a graph containing all possible edges.

For that reason, we allow such wrapper functions to be analysed in a different way. Namely, during the analysis of such functions, the call stack is taken into account when evaluating variables. A disadvantage is that we can no longer use summaries as described in Section 4.1 for such functions. A list of wrapper functions can be provided by the user of the analysis, but we also try to detect them automatically. To identify them, we check parameters of all functions, and if any of them is either a type representing a lock or a structure containing (possibly recursively) a lock, we mark the function as *context-sensitive*.

Listing 3. An example of a lock wrapper

```
1 void lock_wrapper(pthread_mutex_t *m) {
2       pthread_mutex_lock(m);
3 }
4
5 void *thread (void *v) {
6       lock_wrapper(&mutex1);
7       lock_wrapper(&mutex2);
8 }
```

4.4 Concurrency Checking

To reduce false positives, we check if all edges involved in a detected cycle are concurrent. That is not the case when, for example, both edges of the cycle were created in the main thread or generally in a thread that is not created multiple times. Two threads can also be non-concurrent if the first one is always joined before the second one is created. Checking the first

condition is simple; to decide the second one, we use a graph traversing algorithm that checks whether the first thread is always joined before the second one is created (this excludes deadlocks between threads that can never run simultaneously). Another situation that we currently do not take into account, results from using of the so-called *gatelocks*. This situation happens when both dependencies are created in a critical section protected by a common lock and therefore they cannot be reached simultaneously during the execution of the program.

Even though a cyclic dependency does not lead to a deadlock, such a situation still can be considered as a violation of a lock discipline, which may introduce a deadlock in the future, and is therefore reported as a warning of a lower severity.

4.5 Deadlock Reporting

To be useful in practice, the analyser should be able to provide the user with information that helps him/her to understand the reported issue. Since we do forward traversal and analyse each program path separately, we can easily report a trace of each dependency involved in a cycle and hence a potential deadlock. A trace of the dependency is created by concatenating call stacks of points where the involved locks were acquired. To make the report more succinct, the common prefix is reported only once. A deadlock report for the program in Listing 1 could then look as follows:

```
==== Lockgraph: ====
mutex1 -> mutex2
mutex2 -> mutex1
==== Results: ====
Deadlock between threads thread1 and thread2:

Trace of dependency (mutex2 -> mutex1):
In thread thread1:
    Lock of mutex2 (simple_deadlock.c:13)
    Call of f (simple_deadlock.c:14)
    Lock of mutex1 (simple_deadlock.c:2)

Trace of dependency (mutex1 -> mutex2):
In thread thread2:
    Call of g (simple_deadlock.c:18)
      Lock of mutex1 (simple_deadlock.c:8)
      Lock of mutex2 (simple_deadlock.c:9)
```

4.6 A Heuristic Avoiding EVA

When analysing complex programs using Frama-C and EVA, one usually needs to tune their input parameters to achieve both precision and a reasonable running time. After reporting some classes of alarms, EVA will consider the rest of the code unreachable, and the user must first solve the issue (either by fixing the code, changing parameters of Frama-C/EVA, or providing models for external functions). To provide a fully-automated alternative, we implemented a so-far experimental method that completely avoids using EVA and uses purely syntactic information to identify locks and threads.

The workflow of the analyser remains the same, only the implementation of queries to EVA in the wrapper over it differs. Instead of the value analysis, we use functions of the Frama-C API to extract information which variables are contained in expressions. This is sufficient when only references to global variables are used. If this is not the case, which happens, e.g., for locks that are members of structures frequently passed among functions, the method can lead to both under- and over-approximation. In the case of threads, we also need to find their entry point functions. If this is not possible, we assume that every function with a POSIX threads signature (*void * f (void *)*) can be an entry point of a given thread. For other queries to EVA (mainly related to computation of initial states of threads), top values of the abstract domains corresponding to any possible value are returned.

5. Experiments

We have evaluated our analyser on the benchmark originally used in [2][2]. The benchmark contains 993 programs that are considered to be deadlock-free and 8 programs with deadlocks, which were introduced by the authors of the benchmark. All programs are from the Debian GNU/Linux distribution and use the POSIX thread API. Out of the benchmarks, we could unfortunately use a subset only. A huge fraction of the benchmarks was rejected by Frama-C due to type errors (probably caused by a preprocessing done for the CPROVER tool). Moreover, some of the test cases contain locking operations in non-reachable code only. We compare results our tool achieved on the rest of them with those obtained by CPROVER and L2D2.

The experiments with our tool were conducted on a machine with 2.5GHz Intel Core i5-7300HQ processor and 16 GB RAM, running Ubuntu 18.04. To overcome problems with parametrisation described in Section 4.6, we tried to analyse each program using several combinations of parameters to suppress some errors reported by EVA that stop the analysis. However, this leads in some cases to a slow running time and to timeouts before our deadlock analysis even started.

Programs with deadlocks. When using value analysis, our tool detected deadlocks in all 8 cases that actually contain a deadlock. Both L2D2 and

[2] http://www.cprover.org/deadlock-detection

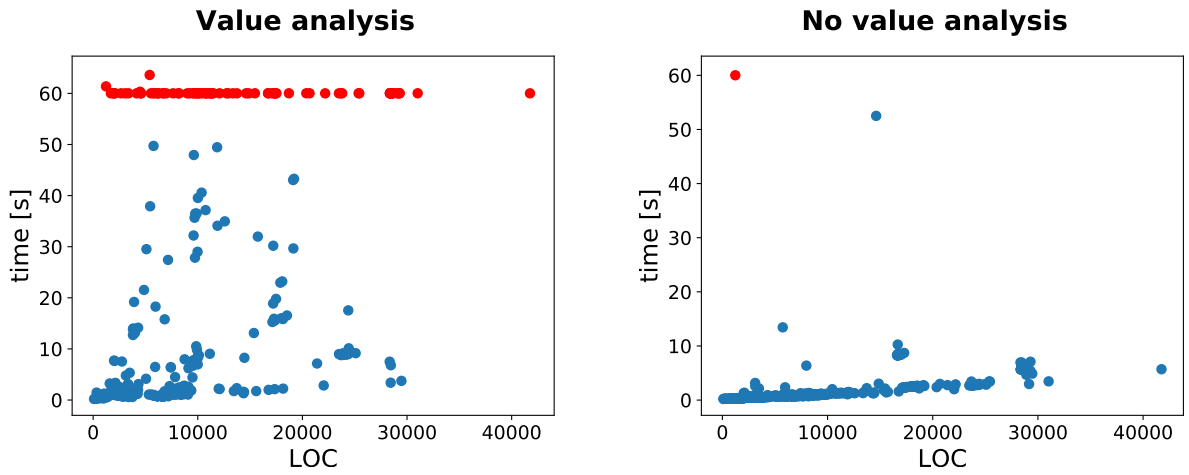

Figure 1. The time needed for the analysis (timeouts after 60 seconds are marked by the red colour)

Table 1. Experimental results on 278 deadlock-free test cases that Deadlock can handle with value analysis

	correct	false positives	no result
Deadlock	181	9	88
L2D2	259	11	8
CPROVER	82	40	156

Table 2. Experimental results on 393 deadlock-free test cases that Deadlock can handle without value analysis

	correct	false positives	no result
Deadlock	357	35	1
L2D2	359	25	9
CPROVER	114	45	234

CPROVER manage to detect them too. Our light-weight version missed one deadlock in a program that uses lock wrappers. This is due to this approach, unlike the solution described in Section 4.3, will see a single lock represented by the formal parameter of the lock wrapper function only, and hence it will not create any locking edge.

Deadlock-free programs. Tables 1 and 2 presents results that our tool with and without using EVA, respectively, achieved on deadlock-free programs that Frama-C could handle and their comparison with results of CPROVER and L2D2. The different numbers of test cases considered in the two tables are caused by the fact that an incorrect parametrisation can lead to considering some locking or thread-creating operations to be unreachable as described at the beginning of this section. The column *no result* includes cases where (a) our tool hit a timeout, (b) CPROVER timeouted or ran out of memory, and (c) L2D2 hit a compilation error.

Figure 1 shows how the running time of our tool grows with the number of lines of code of the programs being analysed when used with and without EVA, respectively. The left part of the graph devoted to the analysis with EVA shows the importance of choosing the right values of parameters of Frama-C and EVA: programs are either analysed quickly (often close to cases when no value analysis is done) or the analysis times out. Note that during the evaluation of CPROVER much higher limits were used: a timeout of 30 minutes and 24 GB of memory.

Further, to verify basic correctness of methods presented throughout the paper, we also prepared a set of crafted programs. These programs are available in the project repository[3].

6. Conclusions

We presented a design of *Deadlock*, a new Frama-C plugin for deadlock detection. The experiments show that it is able to handle real-world C code. However, in some cases, we are limited by the pointer analysis of Frama-C that we use for lock representation. Further work could concentrate on improving precision of method used when value analysis is too demanding.

We are currently also working on a data race detector using a part of our deadlock analysis. Our lockset analysis can be used for checking whether memory accesses to shared variables are protected by locks. In order to do this, some parametrisation needs to be done, because in contrast with deadlocks we need to be more conservative when adding locks to a lockset – an invalid lock generated at the beginning of the analysis would "hide" all following possible races. Differentiating between may- and must- locksets is a possible way to achieve this.

[3] https://github.com/TDacik/Deadlock/tree/master/tests

Acknowledgements

I would like to thank my supervisor Prof. Tomáš Vojnar for his help. The work was supported by the H2020 ECSEL project Arrowhead Tools.

References

[1] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. of SOSP'03*.

[2] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for C/pthreads. In *Proc. of ASE'16*.

[3] Vladimír Marcin. Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2019.

[4] Sandrine Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *18th International Conference on Verification Model Checking and Abstract Interpretation (VMCAI 2017)*.

[5] Boris Yakobowski and Richard Bonichon. Frama-C's Mthread plug-in manual. https://frama-c.com/download/frama-c-mthread-manual.pdf.

[6] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. Conc2Seq: A Frama-C plugin for verification of parallel compositions of c programs. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*.