BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS IN THE FRAMA-C ENVIRONMENT FOCUSED ON DEADLOCK DETECTION

STATICKÁ ANALÝZA V PROSTŘEDÍ FRAMA-C ZAMĚŘENÁ NA DETEKCI UVÁZNUTÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ DACÍK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Dacík Tomáš**
Programme: Information Technology
Title: **Static Analysis in the Frama-C Environment Focused on Deadlock Detection**
Category: Software analysis and testing

Assignment:

1. Get acquainted with static analyses suitable for discovering synchronization problems in concurrent programs.
2. Study the Frama-C framework, its support of various forms of static analysis, as well as existing static analyzers implemented in Frama-C.
3. Design and implement an analyzer based on the Frama-C framework for detecting deadlocks in concurrent programs.
4. Experimentally evaluate the proposed analyzer on suitably chosen non-trivial programs.
5. Summarize the achieved results and discuss possibilities of their further future improvements.

Recommended literature:

- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005.
- Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD: Compositional Static Race Detection, PACMPL, 2(OOPSLA), 2018.
- Engler, D.R., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks, In: Proc. of SOSP'03, ACM, 2003.
- Yakobowski, B., Bonichon, R.: Frama-C's Mthread Plug-in, CEA LIST, Saclay, France, 2012.
- Marcin, V.: Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí, bakalářská práce, FIT VUT v Brně, 2019.

Requirements for the first semester:

- Items 1, 2 and initiation of the proposal from item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: May 8, 2020

Abstract

This thesis presents a design of a new static analyser focused on deadlock detection, implemented as a plugin of the Frama-C platform. Together with the core algorithm of deadlock detection, we also present a light-weight method that allows one to analyse (not only for the purposes of deadlock detection) multi-threaded programs using sequential analysers of Frama-C. Results of experiments show that our tool is able to handle real-world C code with high precision. Moreover, we demonstrate its extensibility by so-far experimental implementation of data race detection.

Abstrakt

Tato práce se zabývá návrhem nového statického analyzátoru pro detekci uváznutí, implementovaného jako plugin platformy Frama-C. Kromě samotného algoritmu pro detekci uváznutí představuje také odlehčené řešení, které umožňuje využít platformu Frama-C pro analýzu vícevláknových programů s využitím analyzátorů Frama-C podporujících pouze sekvenční programy. Výsledky experimentů ukazují, že implementovaný nástroj je schopný analyzovat reálné programy s vysokou přesností. Pro demonstraci další rozšiřitelnosti je představeno experimentální rozšíření umožňující detekovat také časově závislé chyby nad daty.

Keywords

deadlock, data race, static analysis, abstract interpretation, analysis of multi-threaded programs, Frama-C

Klíčová slova

uváznutí, časově závislá chyba nad daty, statická analýza, abstraktní interpretace, analýza vícevláknových programů, Frama-C

Reference

DACÍK, Tomáš. *Static Analysis in the Frama-C Environment Focused on Deadlock Detection.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Rozšířený abstrakt

Testování je v dnešní době neodmyslitelnou součástí procesu vývoje softwaru. *Dynamická analýza* a *automatizované testování* mohou být využity k detekci široké škály chyb, ale z principu je obecně nelze použít k prokázání jejich absence. Na druhou stranu statické analyzátory založené na formálních metodách mohou dokázat korektnost programu vzhledem ke specifikaci, ale jejich škálovatelnost je často omezená a může se projevit například velkým množstvím falešně nahlášených chyb. Jako kompromis lze navrhnout statické analyzátory, které nejsou korektní (neodhalí všechny reálné chyby) ani úplné (můžou hlásit falešné chyby), ale dokáží dobře škálovat, na rozdíl od dynamických nástrojů nevyžadují kompletní spustitelný program a v některých případech mohou být také rychlejší.

Tento přístup lze využít například při analýze vícevláknových programů, které vzhledem k jejich nedeterministické povaze, mohou obsahovat chyby těžko odhalitelné dynamickými přístupy, a zároveň pokud se jedná o komplexní programy, jejich korektní analýza může být příliš náročná. Na druhou stranu odlehčené statické analyzátory mohou umožnit rychlou analýzu schopnou odhalit i vzácně se objevující chyby.

V tomto duchu je navržený i *Deadlock*, analyzátor představený v této práci. Jak jeho jméno naznačuje, je zaměřený na detekci *uváznutí*. Konkrétně se zaměřuje na uváznutí způsobená nesprávným používáním zámků, nízkoúrovňových synchronizačních mechanismů často používaných v programovacím jazyce C. V tomto konkrétním případě lze uváznutí definovat jako situaci, kdy pro každý proces z dané množiny procesů platí, že vlastní zámek a čeká na uvolnění dalšího zámku, který vlastní proces z dané množiny.

Deadlock je implementován jako plugin prostředí Frama-C, které nabízí širokou škálu analyzátorů, zaměřených ovšem především na analýzu sekvenčních programů. Společně s algoritmem pro detekci uváznutí, je tedy představena i metoda, která nejprve vypočítá počáteční stavy vláken programu tak, že pak mohou být analyzovány jako samostatné sekvenční programy. Tato metoda pak může být použita i pro další analýzy vícevláknových programů, nejen uváznutí, jak ukazuje i naše prototypová analýza pro detekci časově závislých chyb nad daty (*data race*). Za účelem vyšší efektivity tento přístup zanedbává veškerou komunikaci mezi vlákny. Vzhledem k tomu, že další analýzu zajímají především možné zámky použité v programu, o kterých předpokládáme, že v praxi příliš mezivláknovou komunikací ovlivněny nejsou, lze tuto pod-aproximaci tolerovat.

V další fázi pak *Deadlock* počítá množiny vlastněných zámků pro všechna místa v programu a konstruuje graf, který zachycuje pořadí, ve kterém jsou zamykány. Tento algoritmus vychází z existujícího nástroje RacerX [12] a v některých ohledech se inspiruje nástrojem CPROVER [16]. Z cyklů detekovaných ve výsledném grafu, které reprezentují potenciální uváznutí, jsou poté vyfiltrovány ty, které s vysokou pravděpodobností nemohou nastat v paralelní exekuci programu. Následně je provedena jejich klasifikace a hlášení uživateli.

Deadlock byl ověřen na rozsáhlé sadě programů z linuxové distribuce Debian. Na těchto programech dosáhl vysoké přesnosti, ale ukázalo se, že je v některých případech limitován dlouhou dobou běhu pluginu EVA, který počítá možné hodnoty proměnných, nad nimiž *Deadlock* poté pracuje. Z tohoto důvodu je představena také heuristika, které analýzu hodnot nevyžaduje. Za cenu snížení přesnosti tak lze dosáhnout mnohem rychlejší analýzy, která navíc nevyžaduje manuální nastavení uživatelem.

Static Analysis in the Frama-C Environment Focused on Deadlock Detection

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of prof. Ing. Tomáš Vojnar, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Dacík
May 28, 2020

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar for numerous advice to this thesis. Furthermore, I would like to thank my family for their support. I also thank for the support received from the H2020 ECSEL project Arrowhead Tools.

Contents

Chapter 1

Introduction

Nowadays, various forms of testing are an inseparable part of the software development process. *Dynamic analysers* and *automated testing* can be used to detect a wide range of bugs, but in principle, they cannot prove their absence, at least not in the general case. On the other hand, *static analysers* based on formal methods can be used to prove the correctness of a program with respect to a specification, but often with a limited scalability that can result in a huge number of false alarms, making results of the analysis unusable. As a compromise, static tools that are neither *sound* (can miss some bugs) nor *complete* (can report false alarms) may be designed.

Such static tools can be particularly advantageous for the analysis of concurrent programs. Due to the non-deterministic nature of concurrent programs, some concurrency-related bugs may remain hidden for dynamic tools, and for sound tools, the analysis can be too demanding. However, light-weight static analysis that does not directly reason about all possible interleavings can be used to find some classes of such bugs, potentially much more faster than dynamic tools would.

In this thesis, we present an analyser designed in such a way, called *Deadlock*. As its name suggests, it focuses on detection of deadlocks, synchronisation errors caused by incorrect usage of locks, low-level synchronisation primitives, often used in the C programming language. *Deadlock* is implemented as a new plugin of the Frama-C framework. Its design is inspired by analyses underlying two existing tools: RacerX [12] primary designed for the analysis of huge code bases and therefore resigning on soundness, and the CPROVER framework [16] that, on the other hand, targets primarily soundness. *Deadlock* tries to combine both approaches using existing analyses of Frama-C to improve precision, but with the stress put on detection of likely deadlocks rather than soundness. The experimental evaluation of *Deadlock* shows its capabilities to handle real-world C programs with a high precision.

Structure of the thesis The rest of the thesis is structured as follows. Chapter 2 gives a brief introduction into concurrency-related errors and abstract interpretation. Existing static and dynamic tools for deadlock detection are presented as well as the Frama-C framework. In Chapter 3, the design of a new deadlock analyser in Frama-C is presented, followed by its implementation details in Chapter 4. Chapter 5 shows experimental results obtained on a set of real-world C programs, and Chapter 6 presents a so-far experimental extension of our tool for detection of data races. Section 7 then concludes the thesis and discusses possible directions for future work.

Chapter 2

Preliminaries

This chapter presents the theoretical background for the thesis. First, it discusses concurrency errors, particularly deadlocks and data races that are subjects of this thesis. Although it does not use it directly, our analyser works on the top of results computed by *abstract interpretation* and we therefore briefly introduce it. The next section is devoted to a description of the architecture and capabilities of the Frama-C platform. Existing analysers, mainly EVA – Frama-C's value analysis plugin and existing solutions for analysis of multi-threaded programs implemented within the platform, are also mentioned. Finally, existing solutions for deadlock detection are presented, especially RacerX and the analysis implemented in the CPROVER framework, which we are inspired by.

2.1 Concurrency Errors

Concurrency errors can be roughly classified into two categories. The first is caused either by an insufficient or completely missing synchronisation among concurrently running tasks. As a result, some operations can be done in an invalid order, e.g., a file can be written to before opened (*order violation*), or data can become inconsistent as a result of non-atomic manipulation (*data race* and *atomicity violation*). A possible solution is to use some synchronisation mechanism, for example low-level *locks* (also referred to as mutexes) that are frequently used in the C programming language to guarantee mutual exclusion of code parts manipulating shared data or their atomicity.

However, if synchronisation mechanisms are not used correctly, they can give rise to the second category of concurrency errors. Examples are *starvation*, a situation when a process is waiting for an event that is not guaranteed to ever happen, or a *deadlock* that causes a program to freeze forever. This is, in fact, another trickiness of concurrency-related bugs, fixing one may easily introduce another. While starvation, or sometimes even data races on some not so important variables (e.g. computing some statistics), can be in some cases tolerated, a deadlock is usually considered as a serious problem due to its irrecoverability.

In this thesis, we particularly target deadlocks and present an extension for the detection of data races, we therefore provide their exact definitions from [18]:

Data race *A program execution contains a* data race *iff it contains two unsynchronised accesses to a shared variable and at least one of them is a write access.*

```c
int i = 0;

void *thread1 (void *v) {
    i++;
}

void *thread2 (void *v) {
    i++;
}
```

```
thread1: LOAD i
thread2: LOAD i
thread2: INC i
thread2: STORE i ; STORE 1
thread1: INC i
thread1: STORE i ; STORE 1
```

Listing 1: A simple C program with a data race and its execution that will trigger it

A simple example of a data race is given in Listing 1. It may not be obvious at the first sight, but after both threads executes their code, the value of the variable `i` does not have to be necessarily one, but also two, depending on a concrete way how threads are interleaved. The reason is that incrementation is, in fact, not atomic and involves three steps: loading the value of the variable `i`, increasing it, and storing it back. If this sequence is interleaved with another manipulation with the same variable as demonstrated in the second part of the example, inconsistent values of the variable can be created.

Deadlock *A program state contains a set S of deadlocked threads iff each thread in S is blocked and waiting for some event that could unblock it, but such an event could only be generated by a thread from S.*

In the rest of the thesis, we will consider solely deadlocks caused by an incorrect usage of locks. In this particular case, each blocked thread is waiting for a release of a lock that is held by (possibly the same) thread from the set of blocked threads. Such deadlocks can be alternatively defined in terms of the Coffman conditions [6]. The Coffman conditions are sufficient and necessary, but they indicate only the possibility of a deadlock. Whether a concrete execution of the program will lead to a deadlock depends on interleaving of threads. The conditions are as follows:

- **Mutual exclusion:** Shared resources requiring mutual exclusion are used.
- **No preemption:** Resources are always returned only after the thread that owns them finishes their usage (they cannot be removed forcibly).
- **Hold and wait:** When already holding a resource, a thread can ask for another.
- **Circular wait:** There exists a cyclic dependency among waiting threads.

For a typical implementation of locks, e.g., POSIX mutexes, the first two conditions are always satisfied just by their usage. The third one is also usually unavoidable when a thread needs to acquire more than one lock. The possible solution is to use the so-called *locking discipline* that defines an order in which locks are acquired and hence ensures acyclicity and denies the fourth condition. This principle is also often used in tools that try to find deadlocks or prove their absence by showing that the program contains or cannot contain a cyclic dependency among locks, respectively.

```
1   void *thread1 (void *v) {
2       lock(&mutex1);
3       lock(&mutex2);
4       unlock(&mutex1);
5   }
6
7   void *thread2 (void *v) {
8       lock(&mutex2);
9       lock(&mutex1);
10      unlock(&mutex2);
11  }
```

```
1  thread1:    lock(&mutex1)
2  thread2:    lock(&mutex2)
3  thread2:    lock(&mutex2)
4  thread1:    lock(&mutex1)
5
6  // Unreachable code:
7  thread1:    unlock(&mutex1);
8  thread2:    unlock(&mutex2);
```

Listing 2: A simple C program with a deadlock and its execution that will trigger it

An example of a deadlock is given in Listing 2. Two threads try to obtain two locks in a different order and if they are interleaved like in the right part of the example, they remain blocked forever. Despite the fact that such a simple deadlock could be spotted easily, it would be extremely hard to find it by repeated running of the program because both threads are short, and switching the context after the execution of a single line is very unlikely, but still possible.

2.2 Abstract Interpretation

Abstract interpretation is one of the approaches to static program analysis. It was first introduced by Peter Cousot and Radhia Cousot in [9]. The main idea behind abstract interpretation is that the concrete semantics of a program is over-approximated by the abstract semantics such that its properties are decidable. Abstract interpretation is a generic framework, i.e., it can be instantiated to many different concrete analyses. To design an analysis, usually following components need to be defined [17, 21]:

- **The abstract domain** represents possible abstract states that over-approximate concrete states of the program. An example is an *interval domain* that represents a single variable using an interval of its possible values. For higher precision, *relational domains* that take into account relationships among variables can be used: an example is the *convex polyhedra domain* that represents values of n variables as a polyhedron in the n-dimensional space.
- **Abstract transformers** model the effect of program instructions on abstract states.
- **The join operator** ($\circ$) combines abstract states into a single one when several program branches meet.
- **The widening operator** ($\triangledown$) is used to ensure that the computation of a fixpoint on program loops will terminate. When the abstract domain is finite, it can be still used to accelerate the computation.
- **The narrowing operator** ($\triangle$) can be used to refine results after widening. This operator does not have to be necessarily defined for all analyses.

Formally, an abstract interpretation I of a program P with an instruction set `Instr` is defined as a tuple [17]:

$$I = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$$

where

- Q is the abstract domain (set of abstract states),
- $\circ : Q \times Q \to Q$ is the join operator,
- $\sqsubseteq \subseteq Q \times Q$ is an ordering on the abstract domain defined as $x \sqsubseteq y \Leftrightarrow x \circ y = x$, such that $(Q, \sqsubseteq)$ is a complete lattice,
- $\top \in Q$ is the supremum of the abstract domain,
- $\bot \in Q$ is the infimum of the abstract domain,
- $\tau :$ `Instr` $\times Q \to Q$ defines abstract transformers for particular instructions.

2.3 Frama-C

This section is devoted to a description of the Frama-C platform. The description is based on papers [2, 23], various user manuals [4, 22, 8], and the API documentation [5]. The reference version is Frama-C 20.0 (Calcium).

2.3.1 Architecture

Frama-C is an open-source platform for analysis of source codes written in the C programming language. Frama-C has a modular, plugin-based architecture based on a kernel that provides general services for plugins. Plugins can be loaded both statically and dynamically and can communicate in several ways. There are three heavy-weight plugins in the current distribution of Frama-C: Jessie and WP for deductive verification, and EVA over-approximating possible values of variables using abstract interpretation techniques. While the the first two are not relevant in the context of this thesis, EVA is described in its own section because our analyser runs on top of its results.

The Frama-C's kernel is based on a customised version of CIL [20]. CIL (C intermediate language) is a high-level representation of the C code together with tools that can be used for its analysis and transformations. CIL first parses the source code to a normalised form to decrease the number of situations that analysers have to reason about. For example, all loop constructs are normalised to a single form, and expressions are transformed to have no side effects. Besides ANSI-C, CIL also supports some of GNU C and Microsoft C extensions. CIL represents a program as an abstract syntax tree (AST), but it also provides support to work over the control flow graph (CFG) of a program by computing predecessors and successors for all statements.

The kernel also provides general services for plugins including a unified way for handling parsing of command-line parameters, printing, and debugging. More high-level services include a system of so-called *projects* that enables analyses to switch between several ASTs and register properties to be updated when a project is changed (which is useful, e.g., for code transformations), and mechanisms for collaboration of plugins using code annotations [23].

2.3.2 Memory Model

Several memory models exists in Frama-C, but from the point of view of this thesis, only the model implemented in the `Location_Bytes` module [5] and used by EVA is interesting. In this model, addresses are represented as pairs consisting of a base and an offset in bytes with respect to the base. Each variable defines its own base address (for example, arrays and structures define a single base and their members are represented using an offset), and new bases can be also created by dynamic allocation. While there is always a finite number of statically allocated bases, dynamic allocation can produce a possibly unbounded number of bases, and therefore so-called *weak bases* are introduced to represent several bases to ensure termination. EVA relies on the following hypothesis about base separation: *"it is possible to pass from one address to another through the addition of an offset if and only if the two addresses share the same base address"* [4]. This hypothesis does not hold for the C language itself, but it is necessary for the efficiency of the analysis. The user is required to provide a special treatment for programs breaking the hypothesis.

The memory model is untyped and parametrised by the so-called *machine dependency model* that defines low-level details of the target platform, e.g., endiannes or sizes of C types [4].

2.3.3 EVA – Evolved Value Analysis

EVA computes an over-approximation of sets of possible values of variables at each program point. Its results can be used for proving absence of generic errors, undefined behaviours, or assertions written in a specialised assertion language. They can also serve as the input for other plugins. EVA uses abstract interpretation extended by a novel method that enables communication of several abstract domains in a modular way. A detailed description of this method is, however, far beyond the scope of this thesis and can be found in [2].

From the user point of view, EVA provides its results using two modules that can be seen as abstract domains. The module `Cvalue.V` is used to represent possible values of a single variable by using the memory model presented in the previous section, and the module `Cvalue.Model` represents possible values of a set of variables and therefore a state of the program memory.

Some examples of other Frama-C plugins are *Metrics* for automated computation of code metrics, e.g., cyclomatic complexity, *PathCrawler* for automatic test-case generation, or various plugins implementing code transformations – *Slicing*, *Spare code*, or *Semantic constant folding*. Those usually use results computed by EVA to improve their precision. An example of a plugin cooperating with the part of the Frama-C platform for deductive verification is the *Aoraï* plugin that can be used to verify properties written in the linear temporal logic.

2.3.4 Concurrency Analysers in Frama-C

The current version of EVA is limited to analysis of sequential programs only. In the past, there have appeared the following analysers for concurrent code:

- **Simple concurrency**[1] is a light-weight plugin for inspection of interrupt-driven programs. It can be used to identify interrupt service routines, variables shared between

[1] https://bitbucket.org/adelard/simple-concurrency/src/default/

the main function and some routine, and accesses to these variables. It targets programs for embedded devices without an operating system and does not take into account pointer aliasing.

- **Conc2Seq** rather than analysing the original concurrent code transforms it and its annotations into an equivalent sequential simulation [1]. This approach assumes that the input code is sequentially consistent, i.e., it does not contain any concurrency-related errors. It is primarily designed for collaboration with deductive verification plugins.
- **Mthread** is built on the top of EVA and computes an over-approximation of the behaviour of all threads based on detection of threads and the fixpoint algorithm described in [24]. It can be used for detection of run-time errors as well as data races.

Unfortunately, both Simple concurrency and Conc2Seq are no longer under active development, and Mthread is available under a proprietary licence only (and, as far as we know, is not being actively developed either). As a result, Frama-C still lacks a reasonable support for analysis of concurrent programs.

2.4 Deadlock Detection

In this section we present existing solutions for deadlock detection. First, we give examples of dynamic tools and approaches behind them. Apart from this approach, we further describe three static analysers, which use quite different approaches but are in fact based on the common underlying principle of the computation of locksets and building the lock-order graph.

2.4.1 Dynamic Tools

In the introduction, we have mentioned that dynamic analysis tools can have problems to detect some concurrency bugs, especially those that manifest only very rarely. However, that does not mean that those tools cannot be used at all, and in fact, there is a lot of dynamic analysis tools using different approaches to increase their chances to find concurrency errors – in particular, deadlocks.

A classical approach to dynamic deadlock detection is the *Goodlock* algorithm introduced in [15]. It constructs a tree for each thread denoting its locking patterns and searches for lock cycles between two threads. The work also presents a problem caused by the so-called *gatelocks* – a lock cycle is protected by a common lock and the deadlock therefore cannot happen.

A possible way to increase chances of finding deadlocks or other concurrency-related bugs is *noise injection*. This technique is implemented, e.g., in the IBM ConTest [11] and FIT BUT's ANaConDA [14] tools. These tools try to inject various forms of forced context switches or other forms of disturbance of the usual way how threads are scheduled in order to increase chances to spot a rare behaviour and thus rarely occurring bugs (or at least their symptoms).

2.4.2 RacerX

RacerX [12] is explicitly designed to handle large code bases and therefore resigns on soundness and does not perform any pointer analysis to identify locks. It uses a generic lockset

algorithm do detect both deadlocks and data races based on top-down, context- and flow-sensitive, interprocedural depth-first traversal of the CFG. During the traversal, locksets, i.e., sets of locks held at particular program points, are computed and the graph denoting ordering in which locks were acquired is iteratively built. Caching on the level of statements as well as on the level of functions is performed to speed-up the analysis. Various techniques are used to eliminate false positives. For example, an *unlockset analysis*, a backward analysis that computes *unlocksets* analogically to the way how locksets are computed is used to eliminate invalid locksets. These techniques are motivated by analysing huge projects like Linux or FreeBSD and specific situations arising when operating systems are analysed.

2.4.3 Analysis in the CPROVER Framework

In contrast to RacerX, the analysis implemented in the CPROVER framework [16] implements a deadlock analysis for C/PThreads that is sound, i.e., misses no deadlocks. The analysis itself is built on top of a context- and thread-sensitive framework proposed in [16] using abstract interpretation techniques. To speed up the pointer analysis that takes up to 93 % of its running time, the tool uses a *dependency analysis* that helps ignore assignments and functions calls that do not affect parameters of locks, unlocks, and thread-related operations. To reduce the number of false positives, a non-concurrency analysis based on a graph search is used.

2.4.4 L2D2

A completely different method than above is implemented in the L2D2 (Low-Level Deadlock Detector) [19] plugin of Facebook Infer. Following the philosophy of Infer, it uses a compositional approach that analyses each function without any calling context. While this approach promises to be more scalable since every function is analysed exactly once, it can also produce more false alarms as well as true negatives since it does not use any pointer analysis.

Chapter 3

Design of a Deadlock Detector

We base the design of our analyser on the lockset algorithm used in RacerX [12]. However, since it does not use any pointer analysis and we want to use capabilities of the EVA plugin of Frama-C to get may-points-to relations to help us to reason precisely about locks, we need to extend it in several ways. We also use concurrency checking inspired by CPROVER [16] but in a less precise way since our lockset analysis is not thread-sensitive, i.e., we analyse each thread separately without information which other thread created it.

The analyser runs in three logical phases. In the first phase, we need to deal with a lack of support for a concurrent analysis by EVA. Our approach to this problem is based on a computation of initial states of threads, which are later analyse as sequential programs without considering any interleavings, and it is presented in Section 3.1. In the second phase, described in Section 3.2, we use the computed information and analyse each thread separately. We perform a lockset analysis that computes sets of locks held at particular program points and builds a lock-order graph (further referred to simply as a lockgraph). Section 3.3 describes the last phase that detects cycles in the lockgraph, denoting possible deadlocks, and filters out those that very likely cannot happen in a concrete execution of the analysed program due to non-concurrency (caused, e.g., by create-join relations between threads). Finally, we use a ranking to choose deadlocks that are reported to the user. Section 3.4 presents a simple heuristic that avoids the described use of EVA, which makes the analysis more scalable for the price of precision.

All examples in this chapter use the Pthreads API with simplified names of locking functions and simplified signatures of thread-related functions. We assume that all used locks are properly initialised. Parts of the tool design described below were presented in our paper [10].

3.1 Thread Analysis

As was mentioned in Section 2.3.4, support for analysis of concurrent programs is currently limited in Frama-C, and, as far as we know, there is currently no solution that would enable one to run EVA or other analysers within Frama-C on multi-threaded code in a scalable way. For example, if one tries to analyse a multi-threaded program using EVA, only the main thread will be analysed because, without knowledge of the multi-threaded execution model, other threads will be marked as unreachable and therefore not further analysed. In the rest of the thesis, we use the term *thread* as an abstraction representing all threads (instances that could be created during an execution of a program) with the same entry

Algorithm 1: Computation of initial states of threads

```
   Input: create_stmts ... statements where threads can be created
          main ... representation of the main thread with already computed initial state
 1 function build_graph(threads)
 2     G = empty_graph()
 3     foreach thread ∈ threads, stmt ∈ create_stmts do
 4         set_active_thread(thread)
 5         if is_reachable_by_thread(stmt, thread) then
 6             children = get_threads(stmt)
 7             foreach child ∈ children do
 8                 G.add_edge(thread, stmt, 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
```

point (and hence the same control). The reason is minimisation of the number of threads to be analysed by EVA (which is usually the most demanding part of the whole analysis).

Our approach is based on computing 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 employs a nested fixpoint computation that iterates over so-far known threads, analyses them through EVA, and propagates 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. The approach is similar to the one used in Mthread that also discovers threads and computes a fixpoint, but it considers all possible values of variables (taking into account possible interleavings of threads and communication among

```
1  int i = 0; int j = 0;
2
3  void *worker (void *v) {
4      lock(&mutex);
5      if (i < THREAD_MAX) {
6          i++;
7          unlock(&mutex);
8          create(worker);
9      }
10     else {
11         unlock(&mutex);
12     }
13     return NULL;
14 }
15
16 int main(int argc, char **argv) {
17     create(worker);
18     j++;
19 }
```

Listing 3: A program consisting of a main thread and a worker thread. The worker thread is created by the main thread and repeatedly creates itself until a certain limit (unknown at the compilation time) is reached.

threads). However, only a sketch of its fixpoint algorithm is given in [24], so a more detailed comparison, concerning, for example, scalability, is not possible.

Our method of computing a thread graph and 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 part of a wrapper over EVA (described in Section 4.1) and used to set its context according to 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 implementation of the fixpoint computation is more described in Section 4.2. 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 3. In this example, the main thread creates a worker thread that creates itself repeatedly until a certain limit, unknown at

the compilation time, is reached. The algorithm starts with the set *create_stmt* containing *stmt*8 and *stmt*17. Only the first one is reachable by the main and the initial thread-graph therefore contains a single edge $main \xrightarrow{stmt17} worker$. The fixpoint computation over this graph is trivial – the state of *main* at *stmt*17 ($\{i \mapsto \{0\}, j \mapsto \{0\}\}$) is propagated as the initial state of *worker*. Afterwards, we build a new graph based on new initial states. We find that the edge $worker \xrightarrow{stmt8} worker$ was added. Now the computation of the inner fixpoint is not trivial anymore since there is a self-loop in the graph. It would eventually terminate because the underlying CVALUE domain is finite, but the computation would have to perform a huge number of steps, each of them containing possibly expensive re-analysis of the *worker* thread by EVA. Hence, after a small number of steps, a widening operator is used to accelerate the computation by using all positive integers up to *maxint* according to a size of integer defined in the Frama-C's machine dependency model. Since there is no other thread and the graph remains unchanged, we return the initial state of *worker* computed as follows (thread arguments are ignored):

$$worker : \{i \mapsto \{0, 1, ..., maxint\}, j \mapsto \{0\}\}$$

Note that the incrementation of the variable `j` on line 18 is not reflected in the initial state of *worker* because it is done after the thread creation.

3.2 Lockset Analysis

The 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, and it does not join locksets coming from different paths to improve precision. The path-insensitivity was originally used in RacerX that has no information about possible values of variables. Despite the fact that we have this information computed by EVA, we use a path-insensitive approach to reduce the under-approximation that we have introduced in the thread analysis phase.

The depth-first traversal of the CFG of a thread is started with the empty lockset, which is modified in each step by applying a transfer function that models the effect of the encountered statement on the lockset. The high-level schema of the traversal is given in Algorithm 2. The function `traverse_function` first checks whether fn has already been analysed with *entry_lockset* and, if so, it returns the matching result from the cache. Otherwise it starts analysing the first statement of fn. The analysis of a statement first computes its effect to *entry_lockset* by applying a transfer function, and calls analysis recursively on its successors. During the recursive call, the analysis is forked for each pair consisting of a successor and a possible lockset. The transfer function, mapping a statement and a lockset to the set of possible locksets after an execution of the statement, is defined as follows ($[\![p]\!]$ denotes the set of possible values of the variable p):

$$t(stmt, ls) = \begin{cases} \{ls \cup \{lock\} \mid lock \in [\![p]\!]\} & \text{if } stmt \text{ is lock}(p) \wedge |[\![p]\!]| > 0 \\ \{ls \setminus \{lock\} \mid lock \in [\![p]\!]\} & \text{if } stmt \text{ is unlock}(p) \wedge |[\![p]\!]| > 0 \\ \{ls \setminus \{lock\} \mid lock \in ls\} \cup \{ls\} & \text{if } stmt \text{ is unlock}(p) \wedge |[\![p]\!]| = 0 \\ \{ls\} & \text{otherwise} \end{cases}$$

Algorithm 2: Schema of the lockset analysis

```
1  rec function analyse_stmt(stmt, entry_lockset)
2      if stmt is locking operation then
3          exit_locksets = update_on_lock(entry_lockset)
4      else if stmt is unlocking operation then
5          exit_locksets = update_on_unlock(entry_lockset)
6      else if stmt is call of fn then
7          exit_locksets = analyse_fn(fn, entry_lockset)
8      else
9          exit_locksets = {entry_lockset}
10     end
11
12     if stmt is end of path then
13         return exit_locksets
14     else
15         acc = ∅
16         foreach (succ, ls) ∈ stmt.succs × exit_locksets do
17             acc = acc ∪ analyse_stmt(succ, ls)
18         end
19         return acc
20     end
21
22 function analyse_fn(fn, entry_lockset)
23     if (fn, entry_lockset) ∈ cache then
24         return cache_find(fn, entry_lockset)
25     else
26         stmt = fn.stmts[0]
27         exit_locksets = analyse_stmt(stmt, entry_lockset)
28         cache_add(fn, entry_lockset, exit_locksets)
29         return exit_locksets
30     end
```

```
1  void f() {                    // entry_lockset = {}
2      lock(mutex_ptr);          // [[mutex_ptr]] = {mutex1, mutex2}
3      unlock(mutex_ptr);        // [[mutex_ptr]] = {mutex1, mutex2}
4  }                             // exit_locksets = {{mutex1}, {mutex2}}
5
6  void g() {
7      f();
8      f();
9  }
```

Listing 4: An example of usage of the transfer function when multiple locks can be locked at a single statement

In the most simple case, when the analysed statement is neither lock nor unlock, the transfer function returns a singleton set containing the current lockset. When the statement is `lock(p)`, it returns the set of locksets that arises by adding one of possible locks from $[\![p]\!]$ to the current lockset. The case of unlocking is almost the same with one exception. As mentioned later, sometimes the set of may-points-to locks can be empty. In this case, we will simply ignore any effect of locking (the fourth case). A sound analysis would use a special *indeterminate lock* and replace it by all possible locks in the resulting lockgraph, but this would lead in most of the cases to creating a complete lockgraph and hence a huge number of reported deadlocks. We rather report such statements as a source of imprecision at the end of the analysis. However, since we want to use the lockset analysis to compute also must-locksets (for detection of gatelocks), we have to ensure that indeterminate unlocking will not result in false must-locksets. Hence, the situation is handled such that any lock from the current lockset can be removed, or no lock is removed at all (the third case). After applying the transfer function, the analysis is forked for each pair consisting of a successor statement and a possible lockset.

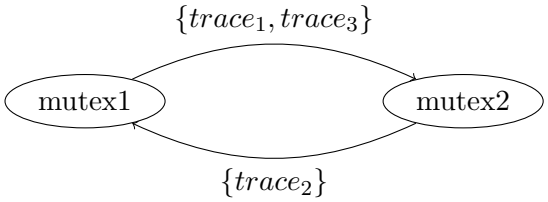
Locks are represented as pairs consisting of a variable and an offset with respect to the base of the variable. The may-point-to information is computed using queries to results of EVA, which are simplified into the mentioned representation. During the simplification, logical bases that do not correspond to any variable, e.g., NULL, are removed. The result can be an empty set when a lock pointer is evaluated either to top (any possible lock) or bottom (the analysed statement is unreachable). The second case can rise when (1) due to path-insensitivity, an unreachable path is analysed, (2) the under-approximation introduced during thread analysis made the analysed statement unreachable, or (3) after some serious error, possibly caused by an insufficient parametrisation, had been reported, EVA stopped the analysis at some point and did not compute any information for the given statement. Hence, we cannot assume that the statement is truly unreachable.

Due to the way the transfer function is defined, it works with may-locksets by analysing every possible combination of locking and unlocking. After the analysis is finished, may- and must-locksets of statements may be obtained using the domain of the statement cache that contains all evaluations of the transfer function:

$$may\_ls(stmt) = \bigcup_{(stmt,ls)\in Dom(cache)} ls \qquad must\_ls(stmt) = \bigcap_{(stmt,ls)\in Dom(cache)} ls$$

```
1  void f() {
2      lock(&mutex1);
3      lock(&mutex2);
4      unlock(&mutex2);
5      unlock(&mutex1);
6  }
7
8  void g() {
9      lock(&mutex2);
10     lock(&mutex1);
11 }
```

```
12 void *thread(void *v) {
13     f();
14 }
15
16 int main (int argc, char **argv) {
17     create(thread);
18     f();
19     g();
20 }
```



$$trace_1 = ([main, 18 : f(), 2 : lock(\text{mutex1})], [main, 18 : f(), 3 : lock(\text{mutex2})])$$
$$trace_2 = ([main, 19 : g(), 9 : lock(\text{mutex2})], [main, 19 : g(), 10 : lock(\text{mutex1})])$$
$$trace_3 = ([thread, 13 : f(), 2 : lock(\text{mutex1})], [thread, 13 : f(), 3 : lock(\text{mutex2})])$$

Listing 5: An example of a program and a lockgraph computed by our lockset analysis (function calls and lock acquisitions are prefixed by numbers of corresponding lines)

In other words, the must-lockset of a statement contains locks that were locked on all paths through the statement, while a single path suffices for the may-lockset. Listing 4 demonstrates the approximation together with the sensitivity of the lockset analysis to errors. Applying the transfer function on the first statement of the function f results into the set of locksets $\{\{mutex1\}, \{mutex2\}\}$. The rest of the function is then analysed separately for $\{mutex1\}$ and $\{mutex2\}$. In both cases, after applying the transfer function on the second statement, we assume that both $\{mutex1\}$ as well as $\{mutex2\}$ 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 $\{\{mutex1\}, \{mutex2\}\}$ instead of the expected empty set. The locksets created by f will probably stay until the exit point of the whole program (since the appropriate unlock has already happened) and possibly create a huge number of false dependencies. In such a case, just another call of f suffices to create a cycle in the lockgraph.

To reduce the number of false deadlock warnings created by such situations, we use a heuristic that assumes that locks and unlocks that happen within a single function and that work with (syntactically) identical expressions do indeed lock/unlock the same lock.

3.2.1 Lockgraph Construction

Lockgraphs are used to represent the orders in which locks are acquired. An edge $a \to b$ denotes a situation when a thread holds the lock a and tries to acquire the lock b. According to this definition, whenever a lock l is added to a nonempty lockset ls, the set of edges computed as $ls \times \{l\}$ is added to the lockgraph. To allow us to track how an edge was created, each edge is labelled by a set of traces, which lead to its creation at different program points. For each lock, we store information about the call stack of the analysis at the time of its acquisition. The call stack can be described by the regular expression $\langle thread \rangle \langle fn - call \rangle^* \langle lock - acquisition \rangle$. The label of an edge is a set of pairs of such call stacks (a single call stack cannot describe the exact trace of an edge – after acquiring the first lock, the program may return from some calls, call other functions and then try to acquire the second lock). Listing 5 shows an example of a program and its lockgraph.

At the end of the lockset analysis, all smallest cycles of the obtained lockgraph are computed as an input of the deadlock analysis phase. The 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.

3.2.2 Function Summaries

Function summaries are an efficient way to speed up interprocedural analysis by avoiding repeated analysis runs on the same function for the same parameters. In our analysis, function summaries are represented by a mapping from pairs (*function, entry lockset*) to pairs (*exit locksets, created edges*). The interpretation is the following: for the case when *function* is called with *entry lockset*, *exit locksets* is the union of the sets of locksets at each of its exit points, and *created edges* is the set of edges emitted during its analysis.

The structure of summaries is inspired by RacerX, but we also have to track the edges that are created and add them to the lockgraph when a summary is used instead of a function analysis. Otherwise, we could not filter out non-concurrent traces in the next phase. In the example in Listing 5, the edge $mutex_1 \xrightarrow{trace_1} mutex_2$ is created during the analysis of the function `f` called from `main`, and its analysis will produce the following summary:

$$(f, \emptyset) \mapsto (\{\emptyset\}, \{mutex_1 \xrightarrow{trace_1} mutex_2\}).$$

Later, when `thread` is analysed, the call of `f` at line 13 meets the precondition of the above summary, and it can be therefore used instead of a new analysis of the function. However, without creating the new trace $trace_3$ and adding it to the set of traces of the edge $mutex_1 \to mutex_2$, the only possible deadlock represented by traces $trace_1$ and $trace_2$ would be filtered out as non-concurrent in the next phase, and the real deadlock between `main` and `thread` would be missed. The new trace is created by replacing its prefix up to the call of a cached function (including the call itself) of the old trace by the current call stack of the analysis.

3.2.3 Context Sensitivity

So-far we have been considering a context-insensitive evaluation of locking parameters, i.e., we analysed functions without any information about their calling context. This approach enables us to efficiently use function summaries, but it can be a source of imprecision. This

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

Listing 6: An example of a lock wrapper

happens, e.g., when wrappers of locking functions or generally functions taking locks as arguments, are used. This scenario is demonstrated in Listing 6. The context-insensitive evaluation of the variable `mutex` used inside the lock wrapper will merge information from all of its calls. As a result, the analyser will assume that the wrapper can lock any mutex used in the program. Then, on line 7, besides the real dependency `mutex1` → `mutex2`, the dependency `mutex2` → `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 using the algorithm described in Section 4.3. The section also describes how function summaries are used in such cases. 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*.

3.3 Deadlock Analysis

The input of the last phase is a set of cycles detected in the lockgraph. Each of these cycles can have its edges labelled by multiple traces. Each cycle therefore represents a set of possible deadlocks with the same involved locks. For each such a set, we want to choose a single deadlock that will be reported to the user. First, we try to filter out those deadlocks (possibly all of them) that are very likely impossible due to the involved lock actions cannot happen in concurrently running threads. If the resulting set still contains more than one deadlock, we perform a ranking to choose the most likely one. The schema of this phase is illustrated in Algorithm 3. For each cycle, the algorithm first creates a work list as a product of labels of all edges (labels are sets of possible traces for given edges) in the cycle. Then it filters out non-concurrent deadlocks from the worklist and chooses a deadlock with the highest rank from among the remaining ones. Currently, the process is implemented only for deadlocks on two locks, which is, however, the majority of the cases in practice.

3.3.1 Concurrency Checking

Our approach to concurrency checking is inspired by the CPROVER tool. However, unlike CPROVER, our analysis is not thread-sensitive. Instead of representing threads using their identifiers, which represent concrete instances of threads, we use the notion of *ab-*

Algorithm 3: Deadlock analysis

Input : cycles ... a set of cycles detected in the lockgraph, each cycle represents a set of deadlocks with the same involved locks

Output: deadlocks ... a set of deadlocks with different involved locks that will be reported to the user

```
1  deadlocks = ∅
2  foreach cycle ∈ cycles do
3  |  worklist = { ∏ labels(edge) | edge ∈ cycle}  ▷Product of labels of a cycle
4  |  concurrent = ∅
5  |  foreach deadlock ∈ worklist do
6  |  |  if is_concurrent(deadlock) then
7  |  |  |  concurrent = concurrent ∪ {deadlock}
8  |  |  end
9  |  |  deadlocks = deadlocks ∪ {find_max_rank(concurrent)}
10 |  end
11 end
12 return deadlocks
```

stract thread that possibly merges several identifiers. Consequently, we do not have precise information how threads involved in deadlocks were created, and we therefore have to use some approximation, which makes our approach necessarily less precise. We will use the following notation of *graph dominators*, which we assume to be computed for the CFG of the whole program:

A statement s_1 dominates a statement s_2 ($s_1 \gg s_2$) iff all paths from the entry node of the given CFG to s_2 must go through s_1.

To list situations we check, let us consider a simplified representation of two edges $e_1 = (thread_1, stmt_1)$ and $e_2 = (thread_2, stmt_2)$, each of them consisting of a thread entry point and a statement in which the edge was created, and a set $ids(t)$ that denotes identifiers of all instances of the abstract thread t. Moreover, let $create(id)$ and $join(id)$ denote all statements that create and join a thread identified by id, respectively. The two given edges are considered to be non-concurrent if any of the following conditions holds:

1. **Same instances.** Both threads are represented by the same abstract thread ($thread_1 = thread_2$) that is not created multiple times during the execution of the program. A special case is the main thread that is always created once only.

2. **Non-concurrent threads.** Threads $thread_1$ and $thread_2$ can never run concurrently, i.e., $thread_1$ is always joined before $thread_2$ is created or vice versa. This can be approximated as:

$$\forall id_1 \in ids(thread_1)\ \forall id_2 \in ids(thread_2)$$
$$\forall c_2 \in create(id_2)\ \exists j_1 \in joins(id_1) : j_1 \gg c_2$$

3. **Before creation.** The creation of e_1 always precedes the creation of $thread_2$ (analogically for e_2 and $thread_1$). The precedence can be checked using an ordering on statements.

4. **After join.** Analogically to the previous case, the creation of e_1 is always preceded by the join of $thread_2$ (analogically for e_2 and $thread_1$). A possibility that the thread is joined only on some paths or is not joined at all must be taken into account. This can be approximated as follows (for all creations there is a join that dominates $stmt_1$):

$$\forall id \in create\_ids(thread_2)\ \exists j \in join(id) : j \gg stmt_1$$

5. **Gatelock.** Statements $stmt_1$ and $stmt_2$ are protected by a common lock and consequently their mutual exclusion is granted. This is the case when:

$$must\_ls(stmt_1) \cap must\_ls(stmt_2) \neq \emptyset$$

Note that cases (2) and (4) are indeed approximations because (a) there could a path on which a thread was not created at all and therefore it does not need to be joined, (b) a thread can be always joined, but using other statements, (c) we currently identify the identifiers only as strings and therefore some restrictions on their usage are assumed, e.g., they are not manipulated nor reused, and (d) if the thread creation is inside a loop, we assume that if its join is also in a loop, both loops perform the same number of iterations. The approach is also sensible to thread wrappers because string identification will merge together all real identifiers used by the wrapper, and so we currently limit it in such a way that creating and joining a thread must happen directly in the main thread (there are also technical reasons for this requirement that are related to the fact that dominators are computed only on the level of functions by Frama-C).

To sum up, there is a lot of room for the algorithm to be imprecise, but we believe it could be useful for some class of rational programs and possibly improved based on further experiments, e.g., by using EVA to get more precise information about used identifiers.

All situations discussed above are demonstrated in Listing 7. It involves two threads created from the main thread and two functions that acquire locks in different orders. The following are examples of pairs of non-concurrent edges ($edge_N$ denotes an edge created by a function call at line N): (1) Edges $edge_{33}$ and $edge_{34}$ are created in threads with the same entry point that is created only once, (2) $edge_{14}$ is not concurrent with $edge_{34}$ since its creation always precedes creation of `thread2`, (3) $edge_{28}$ is not concurrent with $edge_{34}$ since their threads can never run simultaneously, (4) $edge_{23}$ is not concurrent with $edge_{28}$ since `thread1` is always joined before line 23 is reached, and finally (5) $edge_{18}$ and $edge_{28}$ are both in critical sections protected by a common mutex and therefore not concurrent.

3.3.2 Deadlock Ranking

After non-concurrent deadlocks are removed, there could still be multiple possible deadlocks over the same locks. In order to choose the most likely one, which will be reported to the user, a simple ranking based on the computed must-locksets and the length of traces leading to the deadlocks is performed. For each set of possible deadlocks over the same locks, each consisting of a set of edges in the form of $lock_1 \xrightarrow{stmt} lock_2$, the decision is based on:

1. How many edges of the deadlock were created using locks from must-locksets, i.e., we count edges where $lock_1$ is a member of $must\_ls(stmt)$ for the given edge.

2. If we cannot decide according to point 1 above, we pick a deadlock with the shortest trace to be reported (an example of how a deadlock is reported is given in Section 4.4).

```
1  void lock_pattern1() {
2      lock(&mutex1);
3      lock(&mutex2);
4      unlock(&mutex2);
5      unlock(&mutex1);
6  }

13 int main(int argc, char **argv) {
14     lock_pattern1();
15
16     create(&t1,thread1);
17     lock(&gatelock);
18     lock_pattern2();
19     unlock(&gatelock);
20     join(t1);
21
22     create(&t2,thread2);
23     lock_pattern2();
24     join(t2);
25 }

7  void lock_pattern2() {
8      lock(&mutex2);
9      lock(&mutex1);
10     unlock(&mutex1);
11     unlock(&mutex2);
12 }

26 void *thread1(void *v) {
27     lock(&gatelock);
28     lock_pattern1();
29     unlock(&gatelock);
30 }
31
32 void *thread2(void *v) {
33     lock_pattern1();
34     lock_pattern2();
35 }
```

Listing 7: An example demonstrating all sources of non-concurrency that are checked

3.4 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 method that completely avoids using EVA and uses purely syntactic information to identify locks and threads using only syntactic information provided by CIL API and Frama-C and completely avoiding using EVA. This way, precision of the analysis is traded for its scalability.

For identification of locks and threads, we use a CIL function that extracts variables from an expressions used as parameters of locking and thread-related functions. This approach is equivalent to the original one when only references to global variables are used to represent locks or threads. For locks that are members of structures or arrays, our approach will merge information about some locks because they will be represented only by the variable that corresponds to the base of the given structure or array. On the other hand, aliasing will create more variables representing a single lock. Another complication is linked with locks represented by formal parameters of a function. Recall the example of the lock wrapper from Listing 6, unlike in the case when the value analysis is used, our heuristic will never create any edge, because it will see only a single lock in this case – the formal parameter of the wrapper function. If more functions will manipulate locks, e.g., as a part of structures that are frequently passed among functions, the heuristic will see different locks since they will be represented by different formal parameters. For dealing with formal parameters, the method that is described in Section 3.2.3 can be used.

In the case of identifying threads, an additional complication is that we need to know the name of the entry point of the thread to find it in the source code by its name. If this is not possible, we assume that every function with a POSIX threads signature *void *(fn)(void *)* can be an entry point of the given thread. Note that this can lead to under-approximation, because the standard can be abused by, e.g., a thread defined without parameters. For all other queries to EVA, including those used in the computation of initial states of threads, top values are returned.

Chapter 4

Implementation

This chapter is devoted to implementation of the analyser presented in the previous chapter. Besides implementation details of the analysis phases, it also describes a wrapper over EVA that we designed to ease communication between our analyser and EVA and that can be used for easier implementation of further approaches. This was the case of already implemented heuristic described in Section 3.4.

Our analyser, called *Deadlock*, is implemented as a plugin of Frama-C and available in a public repository on GitHub[1] under the MIT licence. Same as Frama-C, Deadlock is written in the Ocaml language. Together with the tool itself, the repository also contains python scripts for its automatic evaluation and a set of crafted C programs used for its testing. As we have already said, the current implementation targets multi-threaded C programs that use the Pthreads API, but one can also provide custom locking and thread-related functions if they use a similar execution model. Since there is ongoing work on the Frama-Clang plugin[2], future support for C++ is possible.

4.1 EVA Wrapper

The wrapper over EVA is a layer used to hide the sequential character of the underlying value analysis for the rest of the analyser, enabling it to easily switch analysed threads, and implementing queries to results computed by EVA. To avoid multiple repeated analyses of a thread with the same initial context, it uses a cache mapping threads and their initial states to results obtained by EVA. Same as the interface of EVA, it is implemented in an imperative way, keeping an internal state consisting of a currently active thread and the cache.

The wrapper works with a type representing the abstract thread consisting of an entry point function and an initial state – a join of all states the thread was created with. The initial state itself can be seen as a product of two abstract domains, `Cvalue.Model` representing possible values of a set of variables (global variables in our case) and `Cvalue.V` that represents possible values of a single variable (the argument passed to the thread). Related operations on initial states are defined component-wise using operations on both of the domains. Signatures of modules representing an abstract thread and its initial state are given in Listing 8.

[1] https://github.com/TDacik/Deadlock
[2] http://frama-c.com/frama-clang.html

```
module Thread : sig
    module InitialState : sig

        type t = {
            globals: Cvalue.Model.t (* State of global variables *)
            argument: Cvalue.V.t    (* State of thread argument *)
        }

        val equal : t -> t -> bool
        val compare : t -> t -> int
        val join : t -> t -> t
        val widen : t -> t -> t

    end

    type t = {
        entry_point : Cil_types.fundec  (* Definition of entry point *)
        initial_state : InitialState.t  (* Initial state of thread *)
    }
end
```

Listing 8: The signature of the module representing an abstract thread and its sub-module representing an initial state of a thread

The wrapper provides a function to change the context of the currently analysed *thread* by calling the function `set_active_thread` (Listing 9). Depending on whether the *thread* was already analysed with its current initial context, it will either use cached results or instruct EVA to set the initial state of *thread* and perform the value analysis. Setting the context of the analysis is done in three steps. First, the program entry point is set to the entry point of the thread. Then, the state of global variables and the argument of the thread is set. When setting the argument, we have to check whether the thread entry point is really defined with a single argument since some entry points may abuse the Pthreads API and have either none or too many arguments. After the value analysis is computed, its results are added to the cache. Despite the fact a caching is already done inside EVA, our experiments showed that analysis of a thread using an already analysed context still has negative impact on the running time. This could be caused by repeated reporting of alarms and warnings in each analysis.

The heuristic described in Section 3.4 is implemented as a part of the wrapper. It implements the interface of the wrapper in a way that does not require running EVA and also does not change the workflow of the rest of the analyser and therefore makes the implementation easier as it requires to modify code only inside the wrapper. For example, despite the fact that the heuristic does not need to compute any initial states of threads, the thread graph is still computed using the proposed algorithm, but in such a way that it only trivially add threads and sets initial states to the top values.

```
let set_active_thread thread =
    if Cache.mem thread !cache then
        let results = Cache.find thread !cache in
        Eva.Value_results.set_results results

    else begin
        Globals.set_entry_point (Thread.to_string thread)
        Db.Value.globals_set_initial_state thread.initial_state.globals;

        (** Ignore entry points that does not respect Pthreads API *)
        let n_args = List.length entry_point.sformals in
        if n_args = 1 then
            Db.Value.fun_set_args [thread.initial_state.argument]
        else ()

        !Db.Value.compute ();
        let results = Eva.Value_results.get_results () in
        cache := Cache.add thread results !cache
    end
```

Listing 9: A function for switching the currently analysed thread either by using cached results or by starting a new analysis by EVA

4.2 Thread Analysis

The inner fixpoint computation from Algorithm 1 computing initial states based on the current thread graph is implemented using the Ocamlgraph library[3]. The design of Ocamlgraph is based on using functors (modules that are parametrised by other modules, e.g., `Set` from the Ocaml standard library that is parametrised by a module with a signature of `OrderedType`). Their usage allows one to customise graph structures (e.g., by defining a custom compare function for vertices) and is also used to implement graph algorithms independently on their underlying representation [7].

The fixpoint computation uses the module `ChaoticIteration` that implements a chaotic iteration strategy with widening based on a weak topological ordering (WTO) on the underlying graph [3]. The instatiation of the fixpoint computation is demonstrated in Listing 10, and the computation itself is demonstrated in Listing 11. The function `recurse` is called with the graph and the function `initial_data` that tells the algorithm how to get initial data for each node. Additional parameters tell it to choose widening points according the computed WTO, and to perform widening after two steps. We intentionally use a very low value to avoid a possibly expensive re-analysis of a thread which is included in each fixpoint step. The fixpoint step itself takes an edge $parent \xrightarrow{stmt} child$ and *state* of the parent, sets the context according to the *state*, performs the value analysis, and returns the new state at *stmt* – the new initial state of *child*.

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

```
    (** ThreadGraph is a persistent, directed graph *)
    module ThreadGraph = Graph.Persistent.ConcreteBidirectionalLabeleld
        (Thread)
        (CreateEdge)

    (** Propagation of state by edge *)
    let fixpoint_step (parent, stmt, child) state =
        (** Create a new thread to perform analysis with *)
        let thread = Thread.create parent.entry_point state in
        Eva_wrapper.set_active_thread thread;

        (** Return new initial state of the child *)
        Eva_wrapper.stmt_state stmt

    module Fixpoint = ChaoticIteration.Make
        (ThreadGraph)
        (struct
            type t = InitialState.t
            type edge = ThreadGraph.E.t

            let join = InitialState.join
            let equal = InitialState.equal
            let widening = InitialState.widen
            let analyze = fixpoint_step
        end)
```

Listing 10: Initialisation of modules for the thread graph and the fixpoint computation using functors

```
    (** Function telling how to get initial data for a node *)
    let initial_data t = (t.globals, t.argument) in

    let root = ThreadGraph.get_main g in
    let wto = WTO.recursive_scc g root in
    let widening_set = ChaoticIteration.FromWto in
    let delay = 2 in
    let initial_states =
        Fixpoint.recurse g wto initial_data widening_set delay
    in
    (** Apply map to the graph *)
    ThreadGraph.update initial_states g
```

Listing 11: The fixpoint computation of initial states

Algorithm 4: Context-sensitive evaluation of locking parameters

```
Input  : expression ... expression to be evaluate
         callstack ... current call stack of the analysis
         current_fn ... analysed function
Output: actual_parameters ... set of possible actual parameters
1  rec function find_actuals(expression, callstack, fn)
2      variable = extract_lock_variable(expression)
3      if is_formal(variable) then
4          n = position_of_formal(current_function, variable)
5          (stmt, top_fn) = pop(callstack)                    ▷ top call site
6          if is_entry_point(top_fn) then
7              return values_of(top_fn.argument)
8          else
9              expr = nth_param(stmt, n)              ▷ expression from call site
10             return find_actuals(expr, callstack, top_fn)
11         end
12     else
13         return values_of(expression)          ▷ eval. according to the mode
14     end
```

4.3 Lockset Analysis

The lockset analysis is implemented using CIL functions allowing one to perform the CFG traversal and to straightforwardly implement Algorithm 2. In this section we mainly discuss the implementation of context-sensitive evaluation of locking parameters. The method is presented in Algorithm 4. It uses a top-down traversal of the call stack searching for a function call that has passed the actual parameter. In each iteration, it first extracts the variable representing the lock from the expression that we want to evaluate. If it is a formal parameter, it finds the corresponding actual parameter in lower levels of the call stack. In this way, it either finds a variable that is not formal or reaches the entry point and uses its argument computed in the thread analysis phase. This approach ensures that the algorithm use only the part of the call stack that contains exclusively *context-sensitive* functions (otherwise the algorithm could not iterate through their calls). As a result, only context-sensitive functions require special treatment (currently no caching at all) and summaries of other function can be used in the usual way. The algorithm can be used also when the analyser runs without results computed by EVA – the function `values_of` denotes the particular way of parameter evaluation to be used. Value analysis will use the whole expression and evaluate it using EVA, while our heuristic from Section 3.4 will again extract the locking variable syntactically.

4.3.1 Extended Semantics of Locking Operations

So-far we have been considering only locking using `pthread_mutex_lock` functions that, moreover, never fails. Although this function is most frequently used, the Pthreads API defines other functions on different types of locks. Spin-locks implement active waiting for lock acquisition that can be more efficient than switching the context when a critical section protected by a lock is short, e.g., if it only contains an access to a hardware register.

```
1  void f() {
2
3      if (lock(&mutex) != 0)
4          return FAIL;
5
6
7
8
9      if (unlock(&mutex) != 0)
10         return FAIL;
11
12
13
14     return SUCCESS;
15 }
```

```
1  void f() {
2      tmp = lock(&mutex);
3      if (tmp != 0) {
4          __retres = FAIL;
5          goto return_label;
6      }
7
8      tmp_0 = unlock(&mutex);
9      if (tmp_0 != 0) {
10         __retres = FAIL;
11         goto return_label;
12     }
13     __retres = SUCCESS;
14     return_label: return __retres;
15 }
```

Listing 12: An example of a function that checks the return value of a locking operation and its normalised representation by CIL

Read-write locks can be used to implement the readers–writers scenario, when a lock can be obtained by a single writer or by multiple readers. There are also non-blocking functions `*_timedlock` and `*_try_lock` that can be combined with the sooner mentioned types of locks in several ways. Due to their non-blocking nature, such functions cannot cause a deadlock, but they still can acquire the first lock in a lockgraph edge and should be therefore considered in the lockset analysis.

Currently, the analyser supports two types of functions to be provided by parameters – blocking and non-blocking. Read-write locks are currently not supported. An option for considering all functions from the Pthreads API can be also used. Blocking and non-blocking functions are treated in the same way, except the deadlock analysis phase when edges obtained by non-blocking functions are removed.

Programs that we used for testing usually do not check return values of locking operations. However, there are situations in which the `pthread_mutex_lock` function can fail and for non-blocking functions, checking the return value is natural. In such a case, especially when it is followed by returning an error to the caller as demonstrated in Listing 12, our analyser will compute the exit lockset containing `mutex` for path ending at line 4. Obviously, this is incorrect since line 4 can be reached only when locking failed and no lock was therefore acquired.

Rather than directly modelling a fail of locking, we try to filter invalid locks at function exit points in the following way: if a function exit point is reached with a lockset containing a lock that was acquired within the function, we first check whether a return value of its locking was stored to some variable v and, if so, we check possible values of v at the exit point. If the set of possible values does not contain zero, we can safely say that locking on this path was not successful and remove the lock from the lockset (up to some explicit manipulation with v which we do not assume). The implementation of this heuristic benefits from normalisation of the source code by CIL, since it will always create a new variable (`tmp` at line 2) when the result of a function is taken but not explicitly stored. However,

```
1   void f() {
2       lock(&mutex1);
3   }
4
5   void *thread1(void *v) {
6       f();
7       lock(&mutex2);
8   }
```

```
9   void g() {
10      lock(&mutex2);
11      f();
12  }
13
14  void *thread2(void *v) {
15      g();
16  }
```

```
[deadlock] === Lockgraph: ===
[deadlock] lock2 -> lock1  (1 times)
[deadlock] lock1 -> lock2  (1 times)
[deadlock] ==== Results: ====
[deadlock] Deadlock between threads thread1 and thread2:

 Trace of dependency (lock1 -> lock2):
 In thread thread1:
    Call of f (deadlock.c:6)
    Lock of lock1 (deadlock.c:2)

    Lock of lock2 (deadlock.c:7)

 Trace of dependency (lock2 -> lock1):
 In thread thread2:
   Call of g (deadlock.c:15)
    Lock of lock2 (deadlock.c:10)

    Call of f (deadlock.c:11)
    Lock of lock1 (deadlock.c:2)
```

Listing 13: An example of a program containing a deadlock, and how is the issue reported

Frama-C also normalises functions to a single-entry-point form using gotos, which makes the proposed method unusable since all paths are joined. The solution is straightforward, whenever the analyser reaches a goto statement, it will check the first statement after the target label, and if it is a return statement, it will consider the goto statement as the function exit point.

4.4 Reporting

To be useful in practice, the analyser should be able to provide the user with information helping to understand every reported issue and easily inspect whether it is a false alarm or not. Our analyser reports a deadlock by providing a trace for each its edges. Each edge is reported as the common prefix of the call stacks of both of the locks followed by their suffixes. An example of how our tools report deadlocks is given in 13. According to the

level of verbosity given by an input parameter, the output also shows analysis statistics, the computed thread graph, summaries of functions, and may- or -must locksets for statements. If the analyser ignored some locking operation (e.g., because its parameter was evaluated to the top value), this information is displayed as a source of imprecision.

The analyser also provides an option to store computed initial states of threads to files in such a way they can be later viewed in the Frama-C GUI application. This could be used to browse results computed by EVA and fix possible issues.

Chapter 5

Experimental Evaluation

This chapter presents our experiments on a set of real-world programs preprocessed as a set of benchmarks for the CPROVER's deadlock analyser. We compare the achieved results with the L2D2 plugin of Facebook Infer for low-level deadlock analysis [19] and the CPROVER's deadlock analyser [16]. We discuss some specific cases when our analysis tends to fail as well as the impact that the different phases of the analysis have on the total running time. Our preliminary experimental results were presented in [10].

5.1 Experimental Setup

Our experiments were conducted on the benchmark originally used in [16][1]. The benchmark contains 994 programs that are considered to be deadlock-free and 8 programs with deadlocks, which were introduced by the authors of the benchmark. Programs are taken from the Debian GNU/Linux distribution and use the POSIX thread API. A huge fraction of the benchmarks was rejected by Frama-C due to type errors (probably introduced by a preprocessing step done for the CPROVER tool), and we therefore used only a part of the benchmark.

Another complication was related to problems with parametrisation of Frama-C/EVA described in Section 3.4. Since we are aiming at bug detection, we can use our tool even when the results of EVA are partial only. Consequently, some locking actions or even creation of a thread can be missed by our analyser. To minimise such situations, we used a script that tries to run the analysis multiple times with different parameters to suppress some errors that would otherwise force EVA to stop (assuming, e.g., that all memory accesses are valid or that all variables are initialised). For each test case, the script tries to find a combination of parameters of Frama-C/EVA that minimises the number of situations when EVA returns the top value for a parameter of some locking operation. If there is no combination of parameters such that at least some locking was encountered and at least one thread except the main thread was found, we ignore the test case in the comparison. When *Deadlock* runs without the value analysis, no parametrisation or special treatment is required. However, there were cases when no locking or no thread creation was found even in this mode. Nevertheless, for most of such cases results of CPROVER indicate that those programs are in fact not concurrent. Only in a few cases, *Deadlock* missed some locking reachable only via calls of function pointers that cannot be currently resolved in the heuristic mode. Such cases are ignored as well.

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

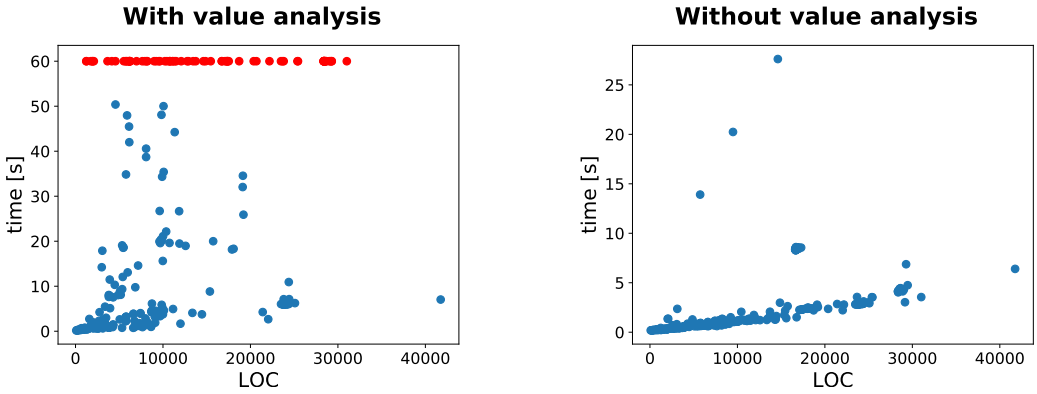


Figure 5.1: The time needed for the analysis (timeouts after 60 seconds are marked by the red colour)

Table 5.1: Experimental results on 293 deadlock-free test cases that Deadlock can handle with value analysis

	correct	false positives	no result
Deadlock	209	4	80
L2D2	273	11	9
CPROVER	92	42	159

Table 5.2: Experimental results on 350 deadlock-free test cases that Deadlock can handle without value analysis

	correct	false positives	no result
Deadlock	347	3	0
L2D2	324	18	8
CPROVER	87	45	218

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 obtain results of CPROVER on a relevant subset of the benchmark, we reproduced them using scripts included in the benchmark, running on a machine with 2.9GHz AMD Opteron 8389 processor and 128 GB RAM, running Debian 9. Time and memory limits were set according to [16]: a timeout of 30 minutes and 24 GB of memory. For L2D2, we use its results from [19].

5.2 Results

Programs with deadlocks. When using value analysis, our tool detected deadlocks in all 8 cases that actually contain a deadlock. Both L2D2 and CPROVER manage to detect them too. Our light-weight version missed one deadlock reachable only by call of a function pointers, which currently cannot be analysed without EVA.

Deadlock-free programs. Tables 5.1 and 5.2 present results that our tool – with and without using EVA, respectively – achieved on relevant deadlock-free programs (programs that Frama-C could handle and *Deadlock* found locking and thread-creating or timeouted) and their comparison with results of CPROVER and L2D2. The different numbers of test cases considered in the two tables are related to the fact that test cases that does not find any locking/thread-creation actions are, as mentioned, ignored. The column *no result* includes cases where (a) our tool hit a timeout, (b) CPROVER timeouted, ran out of memory, or failed to parse the program, and (c) L2D2 hit a compilation error.

Figure 5.1 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. Out of the 80 cases of timeouts, 71 programs hit the timeout in the thread analysis phase (out of them 52 even during the initial analysis of the main thread). When programs were analysed with EVA, the lockset analysis phase took less than 10 % of the total running time in almost all cases. The average memory needed for the analysis was about 350 MB without any significant divergence from this value.

A shortcoming of the benchmark we considered is a small number of programs that actually contain a deadlock. We therefore used a set of crafted programs to successfully verify the basic correctness of the methods presented throughout the thesis. The set contains 47 programs and is available in the project repository[2].

5.3 Discussion of the Results

When compared to our results from [10], we managed to reduce the numbers of false positives in both modes and slightly decrease the number of timeouts. In the heuristic mode, the main factor that contributed to the improvement is implementation of the context-sensitive evaluation of formal parameters. In the mode using EVA, some false alarms were excluded using the heuristic that syntactically pairs locking and unlocking statement as described in Section 3.2. The implementation of caching inside the wrapper over EVA helped to finish the analysis within the given time limit in a few cases too. The remaining false positives seem to be caused by data-dependent locking that we do not handle well due to the path-insensitivity of our lockset analysis.

During the experiments, we have found a reappearing situation in which our method of computing initial states of threads does not work well. The scenario is that a thread is waiting in a loop for a value of some shared variable that can be set only by another thread. Since this value is not computed by our algorithm, EVA will consider the loop non-terminating and the rest of the code unreachable, and therefore we will not derive any information about it. Our further work can focus on detection of such situations and their elimination by adding non-determinism to a condition of the loop that performs the waiting. A possible detection of such situations should not be hard since a waiting is usually implemented using condition variables in programs that use the Pthreads API, namely by the call of the function `pthread_cond_wait`.

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

Chapter 6

Data Race Detection

In this chapter, we present an analyser for data race detection designed as an extension of our deadlock analysis. It is a so-far experimental implementation of a naive approach to race detection, but we use it to demonstrate how the modular design of *Deadlock* can be used to implement new analyses. Moreover, the analyser can already produce reasonable results on smaller programs.

6.1 Design

Our data race detector is designed as a new plugin that heavily uses the API provided by *Deadlock*. Namely, it first computes initial states of threads and performs lockset analysis in the same way as in the case of deadlock detection. Then, it starts a flow-insensitive traversal of the program, computing a set of memory accesses to shared variables using the EVA plugin. After the computation is finished, the analyser checks whether there exists a pair of memory accesses satisfying the definition of a data race. During this step, it uses the concurrency checking implemented in *Deadlock*. Especially the case referred to as a *gatelock* in the context of deadlock analysis is important because it shows whether a pair of accesses is protected or not.

Besides the code that initialises a new plugin, the data race detector implements only the flow-insensitive traversal loop and data types representing accesses and data races since all other necessary functions are already available in *Deadlock*. Some parts of *Deadlock* were extended for this purpose, e.g., cases *before creation* and *after join* from the concurrency checking method from Section 3.3.1 were motivated by situations when shared structures are initialised and destructed in non-concurrent parts of programs. Moreover the lockset analysis was modified to compute both may- and must-locksets in a single run.

6.2 Evaluation and Future Work

For an evaluation of our data race detector, we used a set of 277 student programs with roughly 250 lines of code each. In particular, the programs are from an advance course of operating systems, and the tasks was to implement a ticket algorithm and simple shell using the Pthreads API. For 264 of them, we have no information whether they contain a race or not, but we assume that they are mostly race-free, and 13 of them contain data races that were detected using dynamic analysis and described in [13]. Our tool has detected 8 out of the 13 real data races and reported alarms in 77 out of the rest 264 cases. An example of

an undetected race appears in a program, in which each thread creation initialises a lock and therefore resets its status. Consequently, more threads can enter a critical section that should be protected by this lock. Such a situation is not be detected by our lockset analysis because it does not take into the account initialisation and destruction of locks.

We also tried to run the tool on some programs from the deadlock benchmark. However, our race detector does not scale well on those programs yet. This is caused by a huge number of detected memory accesses, followed by a number of concurrency checks. In the future work, we want to use a more sophisticated algorithm for storing and manipulating memory accesses, and optimise the concurrency checking, e.g., by using caching. Moreover, unlike with deadlocks, some data races may be considered harmless, and therefore a more sophisticated approach is necessary for their ranking.

Chapter 7

Conclusion

In this thesis, we presented a design and implementation of *Deadlock*, a new static analyser focusing on deadlock detection and implemented as a plugin of the Frama-C platform. *Deadlock* implements several heuristics that target common programming patterns used when programming with C/Pthreads (e.g., lock wrappers) and is able to analyse real-world programs with a minimal number of reported false alarms. Since it is sometimes limited by the underlying value analysis computed by the EVA plugin of Frama-C, we also designed a light-weight mode that can be used when the value analysis does not scale well or when its suitable parametrisation is too demanding.

A part of the *Deadlock's* design is a thread analysis that is used to analyse multi-threaded programs using sequential analysers of Frama-C, mainly EVA. This method can be used as a base for new analysers, e.g., a data race detector, which we have already implemented as a prototype tool. Our future work will concentrate on improving it, and since it shares a significant part with our deadlock analysis, we believe that its development can help to improve *Deadlock* as well.

Regarding *Deadlock*, we want to evaluate its precision and scalability on larger code bases – if possible, even on the Linux kernel. This will probably require a careful evaluation and possible adaptations of the techniques we use and probably also design of new techniques to keep the high precision of our analyser but allow it to scale and handle the low-level code of the Linux kernel. Finally, an interesting area of further experiments is collaboration with dynamic analysis tools, namely the ANaConDA framework [14]. For example, information obtained by static analysis can be used to guide noise injection, and ANaConDA could then, hopefully, confirm possible bugs found by our analyser (while not requiring so much resources as without the hints that our analysis can provide).

Bibliography

[1] Blanchard, A., Kosmatov, N., Lemerre, M. and Loulergue, F. 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)*.

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

[3] Bourdoncle, F. Efficient chaotic iteration strategies with widenings. In: *Formal Methods in Programming and Their Applications*. 1993.

[4] Bühler, D., Cuoq, P. and Yakobowski, B. *The Eva plug-in: Release 20.0 (Calcium)* [online] [cit. 2020-5-9]. Available at: http://frama-c.com/download/frama-c-eva-manual.pdf.

[5] CEA LIST. *Frama-C API documentation: Release 20.0 (Calcium)* [online]. [cit. 2020-5-9]. Available at: http://frama-c.com/download/frama-c-20.0-Calcium-api.tar.gz.

[6] Coffman, E. G., Elphick, M. and Shoshani, A. System Deadlocks. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. june 1971, vol. 3, no. 2, p. 67–78. ISSN 0360-0300.

[7] Conchon, S., Filliâtre, J.-C. and Signoles, J. Designing a Generic Graph Library Using ML Functors. In: *Trends in Functional Programming*. 2007.

[8] Correnson, L., Cuoq, P., Kirchner, F., Maroneze, A., Prevosto, V. et al. *Frama-C User Manual: Release 20.0 (Calcium)* [online] [cit. 2020-5-9]. Available at: https://frama-c.com/download/user-manual-20.0-Calcium.pdf.

[9] Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, p. 238–252.

[10] Dacík, T. Static Deadlock Detection in Frama-C. In: *Excel@FIT 2020*. Brno University of Technology, Faculty of Information Technology, 2020.

[11] Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G. et al. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*. Wiley Online Library. 2003, vol. 15, 3-5, p. 485–499.

[12] Engler, D. and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In: *Proc. of SOSP'03*.

[13] Fiedor, J. and Vojnar, T. 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.

[14] Fiedor, J. and Vojnar, T. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In: Qadeer, S. and Tasiran, S., ed. *Runtime Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 35–41. ISBN 978-3-642-35632-2.

[15] Havelund, K. Using Runtime Analysis to Guide Model Checking of Java Programs. In: *Lecture Notes in Computer Science*. February 2001, vol. 1885.

[16] Kroening, D., Poetzl, D., Schrammel, P. and Wachter, B. Sound static deadlock analysis for C/Pthreads. In: *Proc. of ASE'16*.

[17] Lengál, O. and Vojnar, T. *Lecture Notes in Static Analysis and Verification : Abstract Intrepretation*. 2018.

[18] Lourenço, J., Fiedor, J., Křena, B. and Vojnar, T. Discovering concurrency errors. In: Bartocci, E. and Falcone, Y., ed. *Lectures on Runtime Verification*. Springer, 2018, p. 34–60. ISBN 978-3-319-75631-8.

[19] Marcin, V. *Statická analýza v nástroji Facebook Infer zaměřená na detekci uváznutí*. Brno, CZ, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/21920/.

[20] Necula, G. *CIL - Infrastructure for C Program Analysis and Transformation* [online]. [cit. 2020-5-9]. Available at: https://people.eecs.berkeley.edu/~necula/cil/.

[21] Nielson, F., Nielson, H. and Hankin, C. *Principles of Program Analysis*. January 1999. ISBN 978-3540654100.

[22] Signoles, J., Antignac, T., Correnson, L., Lemerre, M. and Prevosto, V. *Plug-in Development Guide: Release 20.0 (Calcium)* [online] [cit. 2020-5-9]. Available at: http://frama-c.com/download/plugin-development-guide-20.0-Calcium.pdf.

[23] Signoles, J., Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V. et al. Frama-c: a Software Analysis Perspective. In: *Formal Aspects of Computing*. October 2012, vol. 27.

[24] Yakobowski, B. and Bonichon, R. *Frama-C's Mthread plug-in manual* [online]. [cit. 2020-5-9]. Available at: https://frama-c.com/download/frama-c-mthread-manual.pdf.

Appendix A

Contents of the Attached Medium

The attached medium contains:

- `/Deadlock/`
  - `/src/` – Source codes of *Deadlock*.
  - `/tests/` – Automated tests.
  - `/include/` – Third-party code (patched files of the Ocamlgraph library)
- `/Racer/`
  - `/src/` – Source codes of *Racer*.
  - `/tests/` – Automated tests.
- `/experiments/` – Results of experiments for both *Deadlock* and *Racer*.
- `/tex/` – Source codes of this thesis.
- `/xdacik00_bp.pdf` – This thesis in PDF.