Finding Deadlocks and Data Races with Frama-C

Tomáš Dacík*   Tomáš Vojnar

Brno University of Technology, Faculty of Information Technology

* Supported by Brno Ph.D. Talent scholarship

Introduction

- Two plugins of Frama-C for detection of concurrency bugs developed in my BSc thesis:
    - DeadlockF – deadlock detection
    - RacerF – data race detection

- Both plugins can use the value analysis of EVA to improve their precision (but can also run without it)

- Inspired by the tool RacerX[1]:
    - Quite scalable (successfully evaluated on the Linux kernel)
    - The tool is not available for experiments

[1] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. Symposium on Operating Systems Principles 2003.

Principles

- Lightweight static analysis $\rightsquigarrow$ focus on detection of likely bugs (no soundness/completeness guarantees)
- Both plugins are based on custom CFG traversals algorithm, assuming that branching is always nondeterministic
- Focus on multi-threaded C code with mutexes (binary locks):
  - The main focus is on the *`Pthreads`* library
  - Custom locking/threading functions can be provided by the user

Deadlocks & Data Races

- **Deadlocks** caused by incorrect usage of mutexes:

```
void *thread1(...)          void *thread2(...)
    lock(A);                    lock(B);
    ...                         ...
    lock(B);                    lock(A);
```

- **Data Races** caused by missing (mutex) synchronisation between two memory accesses:

```
void *thread1(...)          void *thread2(...)
    counter++;                  counter--;
```

Common Architecture of Both Plugins

Thread analysis

- Identify all thread entry points; for each, run EVA to compute an (under-approximated) value analysis:
    - Parameters of (un)lock operations
    - Thread-create/join operations

Lockset analysis

- Compute which locks are held at which program points

Concurrency checking

- Determine whether two events may happen in parallel (mostly for data races)

Thread Analysis

- Start with a thread-create graph containing only the ***`main`*** function
- Run EVA for each entry point in the graph with an initial state given as the join of states of its create statement
- If new thread-create statements are found to be reachable, update the thread-create graph
- Repeat until a fixpoint is reached (possibly accelerated using widening) – usually fast since thread-create graphs are usually acyclic

Thread Analysis – Demonstration

```c
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

```c
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```

Thread Analysis – Demonstration

```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```

Thread Analysis – Demonstration

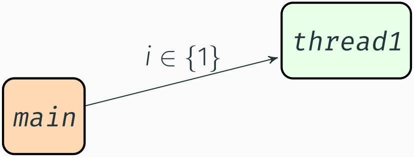
```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```

Thread Analysis – Demonstration

```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

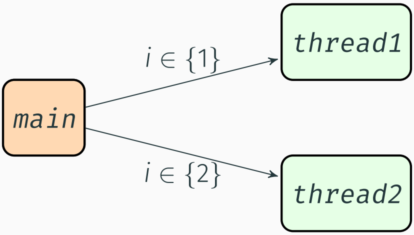
```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```



Thread Analysis – Demonstration

```c
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

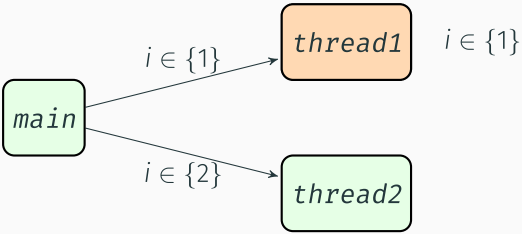
```c
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```



Thread Analysis – Demonstration

```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

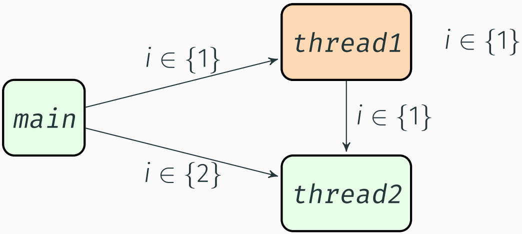
```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```



Thread Analysis – Demonstration

```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

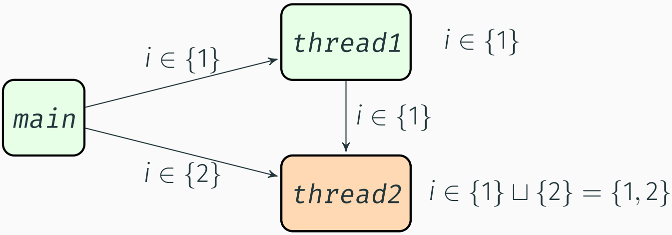
```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```



Thread Analysis – Demonstration

```
int i = 0;

int main(...) {
    i = 1;
    create(thread1);
    i = 2;
    create(thread2);
}
```

```
void thread1(...) {
    i--;
    create(thread1);
}

void thread2(...) {
    i++;
}
```



Lockset Analysis

- Lockset – the set of mutexes locked at the current program point
- For each line, we compute the set of possible locksets
- For each function, we compute its summary as a mapping from input locksets to output sets of locksets

Lockset Analysis

- Lockset – the set of mutexes locked at the current program point
- For each line, we compute the set of possible locksets
- For each function, we compute its summary as a mapping from input locksets to output sets of locksets

```
int f(...) {      ◄                    {∅}
    lock(E);
    ...
    unlock(E);
}
```

Lockset Analysis

- Lockset – the set of mutexes locked at the current program point
- For each line, we compute the set of possible locksets
- For each function, we compute its summary as a mapping from input locksets to output sets of locksets

```
int f(...) {                                        {∅}
    lock(E);      ◄  [[E]] = {A}                    {{A}}
    ...
    unlock(E);
}
```

Lockset Analysis

- Lockset – the set of mutexes locked at the current program point
- For each line, we compute the set of possible locksets
- For each function, we compute its summary as a mapping from input locksets to output sets of locksets

```
int f(...) {                                   {∅}
    lock(E);                                   {{A}}
    ...          ◀                             {{A}}
    unlock(E);
}
```

Lockset Analysis

- Lockset – the set of mutexes locked at the current program point
- For each line, we compute the set of possible locksets
- For each function, we compute its summary as a mapping from input locksets to output sets of locksets

`int f(...) {`		$\{\emptyset\}$
`    lock(E);`		$\{\{A\}\}$
`    ...`		$\{\{A\}\}$
`    unlock(E);`	◄ $[\![E]\!] = \{A, B\}$	$\{\{A\}, \emptyset\}$
`}`		

- Summary: $f : \emptyset \mapsto \{\{A\}, \emptyset\}$

Interpretation of Lockset Results

```
int f(...) {                                              {∅}
    lock(E);                                              {{A}}
    ...                                                   {{A}}
    unlock(E);         ◀  ⟦E⟧ = {A, B}                     {{A}, ∅}
}
```

- The state at the last line can be interpreted in two ways:
  - **May-lockset**: $\{A\}$ (generally union of all locksets)
  - **Must-lockset**: $\emptyset$ (generally intersection of all locksets)

- Duality of deadlock and data-race detection:
  - Must-locksets for conservative deadlock detection
  - May-locksets for conservative data race detection

Deadlock Detection

During the lockset analysis, a lock-dependency graph (lockgraph for short) is created:

- Whenever a lock ℓ is added to a lockset X, an edge $x \to \ell$ is created for each $x \in X$
- Created edges are added as another component of function summaries
- The graph is then checked for cycles representing possibility of deadlocks

Lockset Analysis – Locking Wrappers

- The proposed form of summaries does not work well for locking wrappers (either direct or indirect):

```
void thread(...) {
    wrapper(m1);
    wrapper(m2);
}
```

```
void wrapper(m) {
    lock(m);
    ...
}
```

Lockset Analysis – Locking Wrappers

- The proposed form of summaries does not work well for locking wrappers (either direct or indirect):

```
void thread(...) {
    wrapper(m1);
    wrapper(m2);
}
```

```
void wrapper(m) {
    lock(m);
    ...
}
```

- A heuristic solution: extend summaries with the value of parameters (when they are precisely determined at the call site):

$$\begin{aligned}\texttt{wrapper}:&(\emptyset, m = m_1) \mapsto \{\{m_1\}\},\\ &(\{m_1\}, m = m_2) \mapsto \{\{m_1, m_2\}\}\end{aligned}$$

- Very heuristic, a lot of space for future improvements

Running without EVA

- Running EVA automatically without manually setting its parameters is not always possible
- Locking expressions are often just direct accesses to global variables

$\rightsquigarrow$ A modified version of the algorithm that does not use EVA and relies on syntactic information only

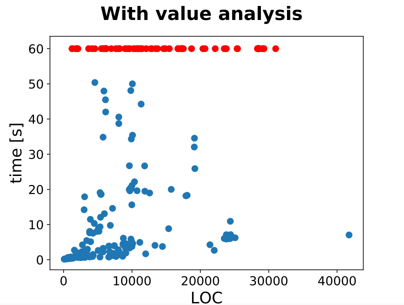
Experimental Evaluation

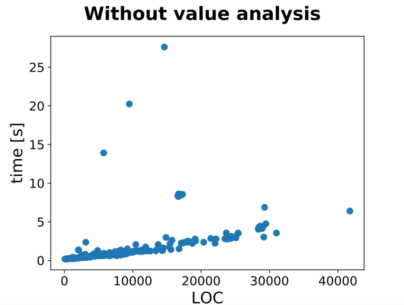
- Benchmark of 997 multi-threaded programs
  - Used for evaluation of a deadlock detector implemented in the CPROVER framework
  - Heavily preprocessed ⇝ not all can be parsed by Frama-C
  - Not all of them contain reachable parallelism (those are ignored in our evaluation)
  - 8 deadlocks manually created by the authors (deadlocks caused solely by locks seem to be hard to find in wild)
    - DEADLOCK can detect all of them with value analysis, and 7 of them without it
- Comparison with:
  - CPROVER deadlock detection (implemented in a fork of CBMC)
  - L2D2 (a plugin of Facebook/Meta INFER, also developed at BUT FIT) – based on a bottom-up lockset analysis

Experimental Results

total: 293	correct	false positive	no result
Deadlock	209	4	80
L2D2	273	11	9
CPROVER	92	42	159

total: 350	correct	false positive	no result
Deadlock	347	3	0
L2D2	324	18	8
CPROVER	87	45	218





The number of LoC is increased by heavy preprocessing done by CPROVER.

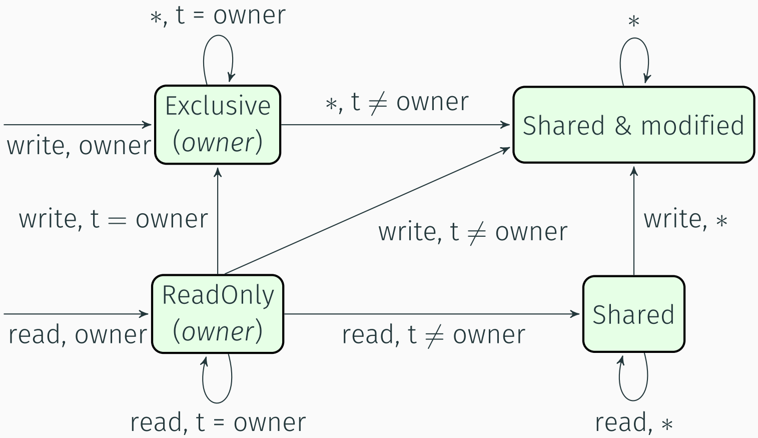
Data Race Detection

- Track memory accesses (in a similar way as locksets) and detect pairs satisfying conditions of a data race:
  - At least one is a write access
  - Can happen in parallel
  - Not-protected (empty intersection of may-locksets)

- Tracking of **all accesses** and checking **all pairs** for races is potentially expensive:
  - ⇝ Track only indistinguishable accesses (related mostly to their traces and quite technical)
  - ⇝ Process accesses more systematically (inspired by dynamic race detectors)

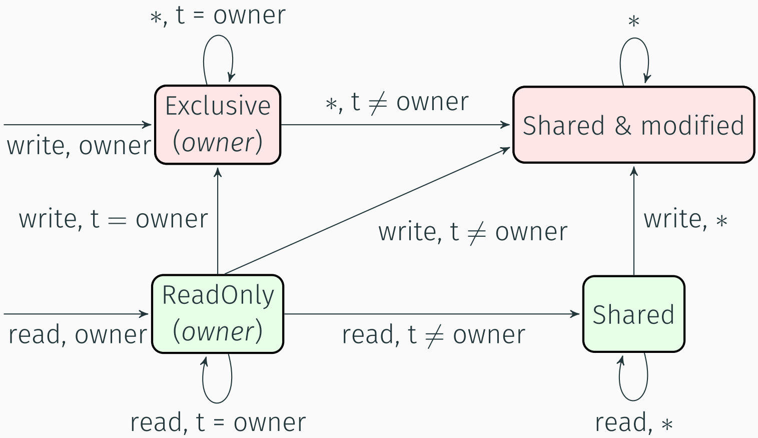
Data Race Detection - Details

Change of a memory location state by access (read/write) of thread t:



Data Race Detection - Details

Only memory locations in Shared & Modified and Exclusive (if the entry point is spawned multiple times) states are searched for races



Concurrency Checking

- Both plugins record traces of the form

$$<\text{entry point}><\text{function call}>^*<\text{event}>,$$

  where $<$event$>$ is either a memory access or creation of a lock dependency
- Traces are useful for reporting (but complicate summaries)

Concurrency Checking

- Both plugins record traces of the form

  <entry point><function call>*<event>,

  where <event> is either a memory access or creation of a lock dependency
- Traces are useful for reporting (but complicate summaries)
- Lightweight checking whether events of two traces cannot happen in parallel:
  - One surely happens before the thread of the other is created (often corresponds to data initialisation)
  - One happens after the thread of the other is surely joined (often corresponds to data postprocessing/deleting)

Experimental Evaluation

- A benchmark of 116 student programs implementing a ticket synchronisation algorithm
  - Smaller programs (200-300 LoC) but heavily concurrent and parametric in the number of threads
  - 23 confirmed data races found by the ANaConDA dynamic analyser

- A comparison with:
  - GOBLINT [2] – over-approximating abstract interpreter
  - O_2 [3] – detection focused on low false positive ratio

[2] Saan, S. et al. Static race detection for device drivers: the Goblint approach. ASE '16.
[3] Bozhen Liu et al. When threads meet events: efficient and precise static race detection with origins. PLDI 2021.

Experimental Results

	Confirmed races (23)		Other (93)	
	detected	missed	race	no race
RACER	20	3	4	89
O_2	12	11	6	87
GOBLINT	21	2	46	47

Experimental Results

	Confirmed races (23)		Other (93)	
	detected	missed	race	no race
RACER	20	3	4	89
O_2	12	11	6	87
GOBLINT	21	2	46	47

- RACER reports false positive races on thread arguments (each thread uses as an argument a different element of an array)
- All tools missed an intricate race caused by re-initialisation of mutexes

Current State

- Plugins are compatible with Frama-C 23.1 (Vanadium)
- DEADLOCK is available as an *opam* package and via github
- Both plugins are available via docker image

DEADLOCK on github

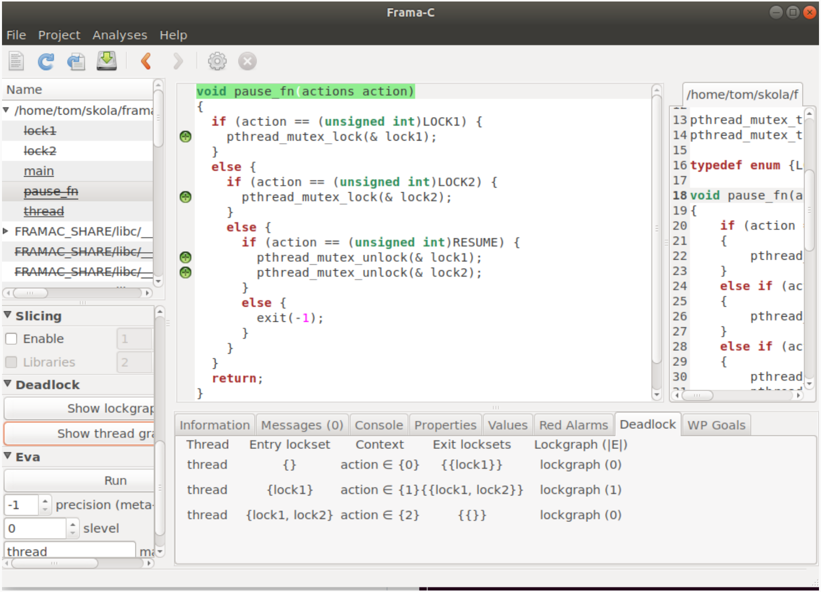
Deadlock & Racer in docker image

Small Demonstration

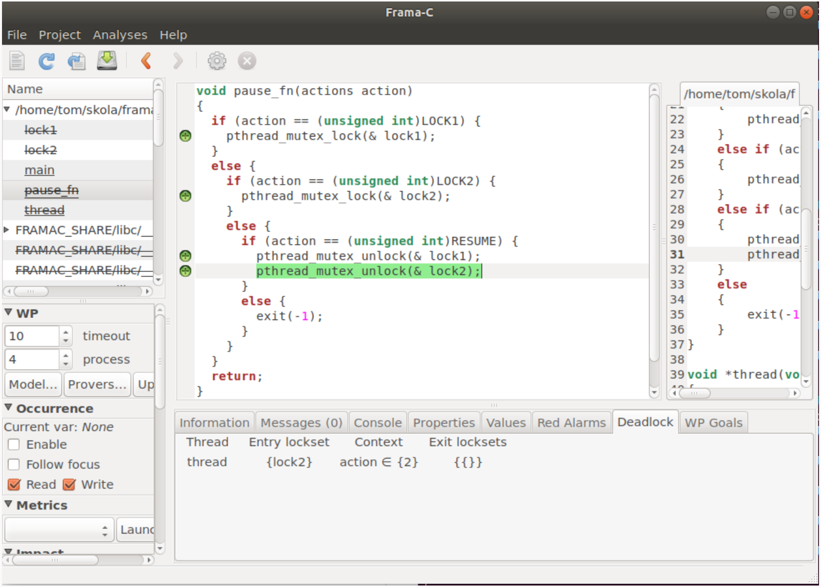
Command-line Output Example

```
[deadlock] === Lockgraph: ===
[deadlock] lock2 -> lock1 (1 times)
[deadlock] lock1 -> lock2 (3 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)
```

GUI example I



GUI example II



Summary

- Frama-C plugins for lightweight detection of deadlocks and data races
- Successfully evaluated on small/medium-size programs (especially nice: low false positive rate)
- Possible future work:
    - Updating to the latest version of Frama-C
    - More systematic implementation of the lockset analysis
    - A focus on data races seems to be a more interesting direction
    - Evaluation on new benchmarks (a new data race category in SV-COMP)
    - Combination with dynamic analysers (e.g., guiding noise insertion)