Static Deadlock Detection in Low-Level C Code

Eurocast'22 — Model-Based System Design, Verification, and Simulation Workshop

Dominik Harmim

Vladimír Marcin, Lucie Svobodová, Tomáš Vojnar

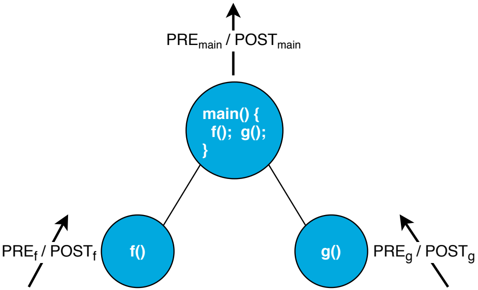
iharmim@fit.vut.cz

Czech Republic, Brno University of Technology, Faculty of Information Technology, VeriFIT

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

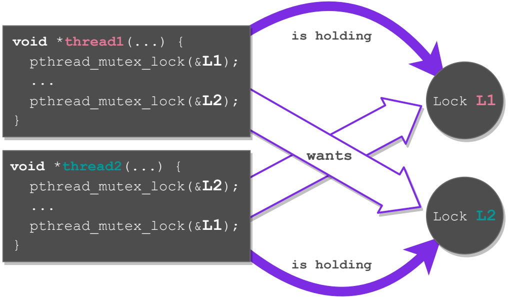
1st March 2022

- Open-source static analysis framework for interprocedural analyses.
    - Based on abstract interpretation.
    - Checks, e.g., for buffer overflows, null-dereferencing, or memory leaks.
- Highly scalable.
    - Compositional and incremental analysis.
    - Computes function summaries bottom-up on call-trees.
- Supports C, C++, Java, Obj-C, C#.



- Deadlocks: among the best-known concurrency errors, caused by a wrong order of locking.
- L2D2: a novel deadlock analyser in INFER.
- For low-level, unstructured, C-style locking.
- Based on computing so-called locksets.
- Lockset analysis:

$\{\,\} \rightarrow$ `lock(L)` $\rightarrow \{\mathbf{L}\} \rightarrow$ `unlock(L)` $\rightarrow \{\,\}$



Related Work

- **Dynamic deadlock analysers and testing:**
  - Test coverage is insufficient (can not be sound).
  - It may be improved by, e.g., systematic testing, noise-based testing, or extrapolation (GoodLock, AirLock — ICSE'20).
  - Requires input data, does not scale enough, and not have to be even complete.

- **Static deadlock analysers:**
  - RacerX — similar to L2D2 but it is context-sensitive (thus not scale so well).
  - Starvation — compositional analyser implemented in Infer (ASE'21).
    - Limited to high-level Java and C++ programs only.
  - Often not sound nor complete in practise (heuristics needed).

- **There is no compositional static deadlock analyser for low-level code.**

Function Summaries — Basic Idea

$$\{ (\mathit{Locked}, \mathit{Unlocked}) \}$$

`foo()`

$$\{ (\mathit{Lockset}, \mathit{Unlockset}, \mathit{Dependencies}) \}$$

- **Pre-Condition:**
  - *Locked*: locks that should be locked before calling the function.
    - The function starts by unlocking the given lock.
  - *Unlocked*: locks that should be unlocked before calling the function.
    - The function starts by locking the given lock.
- **Post-Condition:**
  - *Lockset*: locks that may be locked at exit.
  - *Unlockset*: locks that may be unlocked at exit.
  - *Dependencies*: record that some lock got locked while another lock was still held,
    - i.e., the order of locking.

```
void thread1(...) {
    ...
    pthread_mutex_lock(&L1);
    ...
    pthread_mutex_lock(&L2);
    ...
    pthread_mutex_unlock(&L1);
}
```

```
PRECONDITION:
   Locked       = {}
   Unlocked     = {}

POSTCONDITION:
   Lockset      = {}
   Unlockset    = {}
   Dependencies = {}
```

Function Summaries — An Example

```
void thread1(...) {
    ...
●   pthread_mutex_lock(&L1);
    ...
    pthread_mutex_lock(&L2);
    ...
    pthread_mutex_unlock(&L1);
}
```

```
PRECONDITION:
    Locked       = {}
    Unlocked     = {L1}

POSTCONDITION:
    Lockset      = {L1}
    Unlockset    = {}
    Dependencies = {}
```

```
void thread1(...) {
    ...
    pthread_mutex_lock(&L1);
    ...
    pthread_mutex_lock(&L2);
    ...
    pthread_mutex_unlock(&L1);
}
```

```
PRECONDITION:
  Locked       = {}
  Unlocked     = {L1,L2}

POSTCONDITION:
  Lockset      = {L1,L2}
  Unlockset    = {}
  Dependencies = {L1->L2}
```

```
void thread1(...) {
    ...
    pthread_mutex_lock(&L1);
    ...
    pthread_mutex_lock(&L2);
    ...
    pthread_mutex_unlock(&L1);
}
```

```
PRECONDITION:
   Locked       = {}
   Unlocked     = {L1,L2}

POSTCONDITION:
   Lockset      = {L2}
   Unlockset    = {L1}
   Dependencies = {L1->L2}
```

Function Summaries — An Example

```
void thread1(...) {
    ...
    pthread_mutex_lock(&L1);
    ...
    pthread_mutex_lock(&L2);
    ...
    pthread_mutex_unlock(&L1);
}
```

```
PRECONDITION:
  Locked       = {}
  Unlocked     = {L1,L2}

POSTCONDITION:
  Lockset      = {L2}
  Unlockset    = {L1}
  Dependencies = {L1->L2}
```

1. Locks both acquired and released in a function:
   - Can be deduced from *Unlockset* and *Unlocked*.
   - *Unlockset*/*Unlocked* are erased in some situations.
   - *WereLocked*: all locks at least once locked (never erased).
     - For example: `L` $\in$ *WereLocked* for `f()`.

```
void f()  {
    lock(L);
    ...
    unlock(L);
}
```

2. Locking interleaved with unlocking across functions:
   - Leads to a false dependence.
   - *Order*: records unlock operations preceding lock operations.
     - For example: `L1` $\to$ `L2` $\in$ *Order* for `h()`.

```
void g() {
    lock(L1);
    h();
}
void h() {
    unlock(L1);
    lock(L2);
}
```

Extended Summaries — An Example

```
void g()  {
    lock(L1);
    unlock(L2);
    lock(L3);
    ...
    unlock(L1);
    unlock(L3);
}
void thread2() {
    lock(L2);
    g();
}
```

```
PRECONDITION:
  Locked       = {L2}
  Unlocked     = {L1,L3}

POSTCONDITION:
  Lockset      = {}
  Unlockset    = {L1,L2,L3}
  Dependencies = {L1->L3}
  WereLocked   = {L1,L3}
  Order        = {L2->L3}
```

Extended Summaries — An Example

```
void g()  {
    lock(L1);
    unlock(L2);
    lock(L3);
    ...
    unlock(L1);
    unlock(L3);
}
void thread2() {
    lock(L2);
    g();
}
```

APPLY THE SUMMARY OF g()

```
PRECONDITION:
  Locked       = {L2}
  Unlocked     = {L1,L3}

POSTCONDITION:
  Lockset      = {}
  Unlockset    = {L1,L2,L3}
  Dependencies = {L1->L3}
  WereLocked   = {L1,L3}
  Order        = {L2->L3}
```

```
void g()  {
    lock(L1);
    unlock(L2);
    lock(L3);
    ...
    unlock(L1);
    unlock(L3);
}
void thread2() {
    lock(L2);
    g();
}
```

NEW DEPS:
L2->L1, L2->L3

```
PRECONDITION:
  Locked        = {L2}
  Unlocked      = {L1,L3}

POSTCONDITION:
  Lockset       = {}
  Unlockset     = {L1,L2,L3}
  Dependencies  = {L1->L3}
  WereLocked    = {L1,L3}
  Order         = {L2->L3}
```

```
void g()  {
    lock(L1);
    unlock(L2);
    lock(L3);
    ...
    unlock(L1);
    unlock(L3);
}
void thread2() {
    lock(L2);
    g();
}
```

NEW DEPS:
L2->L1 ✓

```
PRECONDITION:
  Locked        = {L2}
  Unlocked      = {L1,L3}

POSTCONDITION:
  Lockset       = {}
  Unlockset     = {L1,L2,L3}
  Dependencies  = {L1->L3}
  WereLocked    = {L1,L3}
  Order         = {L2->L3}
```

L2 UNLOCKED BEFORE LOCKING L3

Deadlock Detection

- **Pass 1:** summary construction:

$$\{(\mathit{Locked}, \mathit{Unlocked})\}$$

`foo()`

$$\{(\mathit{Lockset}, \mathit{Unlockset}, \mathit{Dependencies}, \mathit{WereLocked}, \mathit{Order})\}$$

- **Pass 2:** compute the transitive closure of *Dependencies* & flag cycles:

```
lock(L1);                lock(L2);
lock(L2);                lock(L1);
```

`L1` $\rightarrow$ **`L2`** $\rightarrow$ **`L1`** $\Rightarrow$ **Deadlock**

Extensions and Heuristics

- Optional reduction of potential false deadlocks by erasing the locksets upon detection of double (un)locking.

- Detecting gate locks and ignoring false deadlocks guarded by them.

- Approximating lock objects using syntactic access paths,
  - i.e., a representation of heap locations via the paths used to access them.

On the benchmarks used for evaluating the CPROVER deadlock checker from:

KROENING, D.; POETZL, D.; SCHRAMMEL, P.; et al.: *Sound Static Deadlock Analysis for C/Pthreads*. ASE 2016.

- 11.4 MLOC from Debian GNU/Linux.
- 100 % deadlock detection rate.
- Roughly 4 % false positives rate.
- Less than 1 % of the time of CPROVER.

	L2D2	CPROVER
Deadlocks	8	8
False Positives	39	114
No Deadlocks	877	292
Failed Cases	80	588
Total	1 002	1 002