Model Checking II
Today
Today

Last lecture

• Model checking basics
Today

Last lecture

• Model checking basics

Today

• Software model checking with SLAM

Based on lectures by Tom Ball and Sriram K. Rajamani. See the SLAM project webpage for details.
Today

Last lecture
  • Model checking basics

Today
  • Software model checking with SLAM

Reminders
  • **Homework 3** is due on today at 11pm
  • Project demos will be held on Dec 08, 10:30-12:20, in MGH 254

Based on lectures by Tom Ball and Sriram K. Rajamani. See the [SLAM project webpage](#) for details.
Overview of SLAM

Program P

Safety property S

SLAM
Software, programming Languages, Abstraction, and Model checking

A trace of P that violates S
Overview of SLAM

A sequential program (device driver) implemented in C.

Program P

Safety property S

SLAM
Software, programming Languages, Abstraction, and Model checking

A trace of P that violates S
Overview of SLAM

A sequential program (device driver) implemented in C.

Program P

Safety property S

Temporal property (an API usage rule) written in SLIC, such as “a lock should be alternatively acquired and released.”

SLAM
Software, programming Languages, Abstraction, and Model checking

A trace of P that violates S

✓
Overview of SLAM

Most influential PLDI paper award and the 2011 CAV award.

Program P

Safety property S

Ships in Microsoft’s Static Driver Verifier (SDV) tool.

A trace of P that violates S

SLAM
Software, programming Languages, Abstraction, and Model checking
The SLAM process

Program P → Instrumentation → P' → Safety property S
The SLAM process

Program P

Safety property S

Instrumentation

P'

Abstraction

boolean program B
The SLAM process

- Program P
- Instrumentation
  - Safety property S
- P'
- Abstraction
  - boolean program B
- Model checking
The SLAM process

Program P → Instrumentation → P' → Abstraction → boolean program B

Safety property S → Instrumentation → P' → Model checking → ✓
The SLAM process
The SLAM process

Program P

Instrumentation

P'

Abstraction

Model checking

Trace validation

A trace of P that violates S

Safety property S

boolean program B

error trace for B

✓
The SLAM process

- **Program P**
- **Safety property S**
- **Instrumentation**
- **P’**
- **Abstraction**
  - boolean program B
  - error trace for B
- **Model checking**
- **Trace validation**
  - new predicates
- A trace of P that violates S

✓
The SLAM process

Program P → Instrumentation → P' → Bebop → Newton → A trace of P that violates S

Safety property S

C2BP

boolean program B

error trace for B

new predicates

✓
The SLAM process: specifying safety properties
Specification Language for Interface Checking
A finite state language for stating rules for API usage

- Temporal safety properties expressed as safety automata that monitor program’s execution behavior at the level of function calls and returns.

- Familiar C syntax.
**Specification Language for Interface Checking**

A finite state language for stating rules for API usage

- Temporal safety properties expressed as safety automata that monitor program’s execution behavior at the level of function calls and returns.
- Familiar C syntax.

**Suitable for control-dominated properties**

- E.g., ordering of function calls with associated constraints on data values at the API boundary.
A locking protocol in SLIC

- Unlocked
- Locked
- Error

Arrows indicate transition states with labels:
- release
- acquire
A locking protocol in SLIC

The global state structure defines a static set of state variables.

```c
state {
    enum {Locked, Unlocked}
    state = Unlocked;
}
```

The diagram illustrates the states and transitions in the locking protocol:

- **Unlocked**
  - Acquire
  - Release

- **Locked**
  - Acquire
  - Release

- **Error**
  - Acquire
  - Release
A locking protocol in SLIC

Transfer functions define events and event handlers that describe state transitions on events.

```c
enum {Locked, Unlocked} state = Unlocked;

KeAcquireSpinLock.return {
    if (state == Locked)
        abort;
    else
        state = Locked;
}

KeReleaseSpinLock.return {
    if (state == Unlocked)
        abort;
    else
        state = Unlocked;
}
```
The SLAM process: instrumentation

- Program P
- Safety property S
- Instrumentation
- $P'$
- C2BP
- Boolean program B
- Bebop: error trace for B
- Newton: new predicates
- A trace of $P$ that violates $S$
Instrumentation by example: 2 steps

```c
state {
    enum {Locked, Unlocked}
    state = Unlocked;
}

KeAcquireSpinLock.return {
    if (state == Locked)
        abort;
    else
        state = Locked;
}

KeReleaseSpinLock.return {
    if (state == Unlocked)
        abort;
    else
        state = Unlocked;
}
```

```c
void example() {
    do {
        KeAcquireSpinLock();

        nOld = nPackets;

        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            nPackets++;
        }
    } while (nPackets != nOld);

    KeReleaseSpinLock();
}
```

Program P

Simplified code for a PCI device driver.
Step 1: translate the SLIC spec S to C

```c
state {
    enum {Locked, Unlocked}
    state = Unlocked;
}

KeAcquireSpinLock.return {
    if (state == Locked)
        abort;
    else
        state = Locked;
}

KeReleaseSpinLock.return {
    if (state == Unlocked)
        abort;
    else
        state = Unlocked;
}
```

```c
enum {Locked=0, Unlocked=1}
state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
    if (state == Locked)
        slic_abort();
    else
        state = Locked;
}

void KeReleaseSpinLock_return {
    if (state == Unlocked)
        slic_abort();
    else
        state = Unlocked;
}
```

Safety property $S$
Step 2: insert calls to SLIC functions into \( P \)

```c
void example() {
    do {
        KeAcquireSpinLock();
        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            nPackets++;
        }
    } while (nPackets != nOld);
    KeReleaseSpinLock();
}
```

Program \( P' \)

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();
        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);
    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}
```

P satisfies $S$ iff SLIC_ERROR is unreachable in $P'$

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();
        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);
    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}

enum {Locked=0, Unlocked=1}
    state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
    if (state == Locked) {
        slic_abort();
    } else {
        state = Locked;
    }
}

void KeReleaseSpinLock_return {
    if (state == Unlocked) {
        slic_abort();
    } else {
        state = Unlocked;
    }
}
```
The SLAM process: predicate abstraction

Program P → Instrumentation → P' → Bebop → Newton → C2BP

Safety property S → Instrumentation → P' → Bebop → Newton → C2BP

error trace for B → Newton → C2BP

A trace of P that violates S → Newton → C2BP

new predicates

boolean program B
Predicate abstraction of C Programs
Predicate abstraction of C Programs

Given a program $P$ and a finite set $E$ of predicates, C2BP creates a boolean program $B$ that is a sound over-approximation of $P$.

- $B$ has the same control-flow structure as $P$, but only $|E|$ boolean variables.
- For any path $p$ feasible in $P$, there is a corresponding feasible path in $B$. 
Predicate abstraction of C Programs

Given a program P and a finite set E of predicates, C2BP creates a boolean program B that is a sound over-approximation of P.

• B has the same control-flow structure as P, but only |E| boolean variables.

• For any path p feasible in P, there is a corresponding feasible path in B.

Suitable abstraction for checking control-dominated properties (such as SLIC rules).

• Models control flow in P precisely.

• Models only a few predicates about data relevant to each rule being checked (so limits state explosion).
Predicate abstraction by example: 5+ steps

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();

        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);

    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}

enum {Locked=0, Unlocked=1}
    state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
    if (state == Locked)
        slic_abort();
    else
        state = Locked; }

void KeReleaseSpinLock_return {
    if (state == Unlocked)
        slic_abort();
    else
        state = Unlocked; }
```

Program P'
Step 1: extract initial predicates from SLIC rules

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();
        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);
    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}

enum {Locked=0, Unlocked=1}
    state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
    if (state == Locked)
        slic_abort();
    else
        state = Locked; }

void KeReleaseSpinLock_return {
    if (state == Unlocked)
        slic_abort();
    else
        state = Unlocked; }

(state == Locked)
(state == Unlocked)
```

Program P’
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();

        nOld = nPackets;

        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);

    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}

b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
    SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        state = Locked;
}

void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        state = Unlocked; }

(state == Locked)
(state == Unlocked)
Step 3: skip statements with no effect on E

```c
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        skip;
        if (request) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            skip;
        }
    } while (nPackets != nOld);
    skip;
    KeReleaseSpinLock_return();
}
```

```c
b(state==Locked), b(state==Unlocked) := F, T;
```

```c
void slic_abort() {
    SLIC_ERROR: ; }
```

```c
void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        state = Locked;
}
```

```c
void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        state = Unlocked; }
```

```c
(state == Locked)
(state == Unlocked)
```
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        skip;
        if (request) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            skip;
        }  
    } while (nPackets != nOld);
    skip;
    KeReleaseSpinLock_return();
}

b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
    SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := F, T; }
Step 5: use non-determinism for conditions

```c
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        skip;
        if (*) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            skip;
        }
    } while (*);
    skip;
    KeReleaseSpinLock_return();
}
```

```c
b(state==Locked), b(state==Unlocked) := F, T;
```

```c
void slic_abort() {
    SLIC_ERROR: ; }
```

```c
void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := T, F; }
```

```c
void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := F, T; }
```

`(state == Locked)`
`(state == Unlocked)`
Step 5: use non-determinism for conditions

This is a highly simplified example of predicate abstraction. The process is much more complex in reality. For details, see Automatic predicate abstraction of C programs.
The SLAM process: model checking

Program P → Instrumentation → P' → Bebop

- Program P
- Safety property S

- Instrumentation

- P'
- C2BP
- boolean program B
- error trace for B

- Newton
- new predicates

- A trace of P that violates S

✓
Model checking of boolean programs
Model checking of boolean programs

Given a boolean program $B$ and a statement $s$ in $B$, Bebop determines if $s$ is reachable in $B$.

- Produces a shortest trace in $B$ (if any) leading to $S$. 
Model checking of boolean programs

Given a boolean program $B$ and a statement $s$ in $B$, Bebop determines if $s$ is reachable in $B$.

- Produces a shortest trace in $B$ (if any) leading to $S$.

**Performs symbolic reachability analysis using BDDs.**

- Adapts the interprocedural dataflow analysis of Reps, Horwitz and Sagiv (RHS) to decide the reachability of $s$ in $B$.
- Uses BDDs to represent the procedure summaries in RHS, which are binary relations between sets of states.
Model checking of boolean programs

Given a boolean program $B$ and a statement $s$ in $B$, Bebop determines if $s$ is reachable in $B$.

- Produces a shortest trace in $B$ (if any) leading to $S$.

Performs symbolic reachability analysis using BDDs.

- Adapts the interprocedural dataflow analysis of Reps, Horwitz and Sagiv (RHS) to decide the reachability of $s$ in $B$.
- Uses BDDs to represent the procedure summaries in RHS, which are binary relations between sets of states.

For details, see Bebop: A Symbolic Model Checker for Boolean Programs.
Model checking of the example program

```c
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        skip;
        if (*) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            skip;
        }
    } while (*);
    skip;
    KeReleaseSpinLock_return();
}
```

```c
void slic_abort() {
    SLIC_ERROR: ;
}
```

```c
void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := T, F; }
```

```c
void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := F, T; }
```
The SLAM process: trace validation

- Program P
- Instrumentation
- Safety property S
- boolean program B
- error trace for B
- Newton
- new predicates
- A trace of P that violates S

Diagram:

1. Program P
2. Instrumentation
3. Safety property S
4. P
5. boolean program B
6. error trace for B
7. Newton
8. new predicates
9. A trace of P that violates S
10. C2BP
11. Bebop
12. ✓
Error trace validation & abstraction refinement
Error trace validation & abstraction refinement

Given a program $P'$ and a candidate error trace, Newton determines if the trace is feasible.

- Uses verification condition generation for feasibility checking.
- If feasible, the error trace corresponds to a real bug.
- If not, returns a small set of predicates that explain why the path is infeasible. Based on greedy minimal unsatisfiable core computation.
Error trace validation & abstraction refinement

Given a program $P'$ and a candidate error trace, Newton determines if the trace is feasible.

- Uses verification condition generation for feasibility checking.
- If feasible, the error trace corresponds to a real bug.
- If not, returns a small set of predicates that explain why the path is infeasible. Based on greedy minimal unsatisfiable core computation.

For details, see Generating Abstract Explanations of Spurious Counterexamples in C Programs.
Validation & refinement for the example

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();
        nOld = nPackets;
        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);
    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}
```

```c
enum {Locked=0, Unlocked=1}
    state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
    if (state == Locked)
        slic_abort();
    else
        state = Locked;
}

void KeReleaseSpinLock_return {
    if (state == Unlocked)
        slic_abort();
    else
        state = Unlocked;
}

(state == Locked)
(state == Unlocked)
```
Validation & refinement for the example

```c
void example() {
    do {
        KeAcquireSpinLock();
        KeAcquireSpinLock_return();
        nOld = nPackets;

        if (request) {
            request = request->next;
            KeReleaseSpinLock();
            KeReleaseSpinLock_return();
            nPackets++;
        }
    } while (nPackets != nOld);

    KeReleaseSpinLock();
    KeReleaseSpinLock_return();
}
```

```c
enum {Locked=0, Unlocked=1}
    state = Unlocked;

void slic_abort() {
    SLIC_ERROR: ;
}

void KeAcquireSpinLock_return {
    if (state == Locked)
        slic_abort();
    else
        state = Locked;
}

void KeReleaseSpinLock_return {
    if (state == Unlocked)
        slic_abort();
    else
        state = Unlocked;
}
```

\[(nPackets == nOld)\]
\[(state == Locked)\]
\[(state == Unlocked)\]
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        b(nOld==nPackets) := T;
        if (*) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            b(nOld==nPackets) :=
                b(nOld==nPackets) ? F : *;
        }
    } while (!b(nOld==nPackets));

    skip;
    KeReleaseSpinLock_return();
}

b(state==Locked), b(state==Unlocked) := F, T;

void slic_abort() {
    SLIC_ERROR: ; }

void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := T, F; }

void KeReleaseSpinLock_return {
    if b(state==Unlocked)
        slic_abort();
    else
        b(state==Locked),
        b(state==Unlocked) := F, T; }

(nPackets == nOld)
(state == Locked)
(state == Unlocked)
```c
void example() {
    do {
        skip;
        KeAcquireSpinLock_return();
        b(nOld==nPackets) := T;
        if (*) {
            skip;
            skip;
            KeReleaseSpinLock_return();
            b(nOld==nPackets) :=
                b(nOld==nPackets) ? F : *;
        }
    } while (!b(nOld==nPackets));
    skip;
    KeReleaseSpinLock_return();
}
```

```c
b(state==Locked), b(state==Unlocked) := F, T;

```c
void slic_abort() {
    SLIC_ERROR: ;
}

```c
void KeAcquireSpinLock_return {
    if b(state==Locked)
        slic_abort();
    else
        b(state==Locked),
            b(state==Unlocked) := T, F; }
```
Summary

Today

- Software model checking with SLAM
- Predicate abstraction of C programs
- Model checking of boolean programs
- Trace validation and abstraction refinement

Next lecture

- Guest lecture by Zach Tatlock!
- Verifying compiler optimizations with SMT solvers