Trusted Components

Prof. Dr. Bertrand Meyer
October 2006 – February 2007

Model Checking

Lisa Liu
(Original slides from Bernd Schoeller)
22. 01. 2007
We don’t want ...
Did you know?

Microsoft does not like blue screens, too!
On Blue Screens

- The majority of blue screens are caused by 3rd party software
- *Most of this software is device drivers*
  - Complex software (concurrency, race conditions, lock keeping)
  - Running “unprotected” by the OS
  - Written for top performance
  - Written by non-software-engineers
  - Difficult to debug
Overview

- What is Model Checking?
- The SLAM project
- BDD
- SAT solving
Model Checking

Does a program P satisfy a certain property Q?

- Proving is difficult
- Testing is not complete
Model Checking: Let's test every possible input

(this works for hardware!)
But:

We just have too many states (state space explosion)

positive_max (a, b: INTEGER): INTEGER is

  require
  a_positive: a >= 0
  b_positive: b >= 0

do
  if a > b then Result := a else Result := b end

ensure
  result_positive: Result >= 0
end

has got \(2^{64} = 18,446,744,073,709,551,616\) different inputs
Let replace every $x \geq 0$ by $POS_x$

**positive_max** $(POS_a, POS_b : BOOLEAN)$: BOOLEAN is

require

- $a_positive: POS_a$
- $b_positive: POS_b$

do

- if ? then $POS_Result := POS_a$
  else $POS_Result := POS_b$ end

ensure

- $result_positive: POS_Result$

end

How many possible input do we have now?
SLAM

- Model Checker for C device drivers
- Looks for the possible violation of temporal properties
  - Properties describe well-known mistakes in driver development
- Uses Boolean abstraction
- Part of the Windows Driver Foundation
The SLAM process

- **C - Code**
- **Boolean Program Generator C2BP**
- **Boolean Program**
- **Model Checker for Boolean Programs BEBOP**

### Flowchart:

1. **C - Code** → **Predicate Discover NEWTON**
2. **Bug found** → **Check successful?**
   - **No** → **Error Path**
   - **Yes** → **Code correct**


**SLAM specification of property \( \phi \)**

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

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

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

**Formal Specification**

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

def slic_abort() {
    SLIC_ERROR
}

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

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

**Compilation into C code**
void example () {
    do {
        KeAcquireSpinLock();

        nPacketsOld = nPackets;
        req = devExt->WLHV;
        if (req && req->status) {
            devExt->WLHV = req->Next;
            KeReleaseSpinLock();

            irp = req->irp;
            if (req->status > 0) {
                irp->IoS.Status = SUCCESS;
                irp->IoS.Info = req->Status;
            } else {
                irp->IoS.Status = FAIL;
                irp->IoS.Info = req->Status;
            }
            SmartDevFreeBlock(req);
            IoCompleteRequest(irp);
            nPackets++;
        }
    } while (nPackets != nPacketsOld)
    KeReleaseSpinLock();
}
void example () {
    do {
        KeAcquireSpinLock();
        A: KeAcquireSpinLock_return()
        nPacketsOld = nPackets;
        req = devExt->WLHV;
        if (req && req->status) {
            devExt->WLHV = req->Next;
            KeReleaseSpinLock();
        }
        B: KeReleaseSpinLock_return()
        irp = req->irp;
        if (req->status > 0) {
            irp->IoS.Status = SUCCESS;
            irp->IoS.Info = req->Status;
        } else {
            irp ->IoS.Status = FAIL;
            irp->IoS.Info = req->Status;
        }
        SmartDevFreeBlock(req);
        IoCompleteRequest(irp);
        nPackets++;
    }
} while (nPackets != nPacketsOld)
KeReleaseSpinLock();
C: KeReleaseSpinLock_return()
Refinement algorithm

1. Apply C2BP to construct the boolean program BP(P', E_i).
2. Apply BEBOP to check if there is a path p_i in BP(P', E_i) that reaches the SLIC_ERROR label. If BEBOP determines that SLIC_ERROR is not reachable, then the property $\phi$ is valid in P, and the algorithm terminates.
3. If there is such a path p, then we use NEWTON to check if p is feasible in P. There are two outcomes:
   - “yes”: the property $\phi$ has been invalidated in P, and the algorithm terminates with an error path p_i
   - “no”: NEWTON finds a set of predicates F_i that explain the infeasibility of path p_i in P.
4. Let $E_{i+1} = E_i \cup F_i$, and $i := i+1$, and proceed to the next iteration.
void KeAcquireSpinLock_return() {
    if (l)
        slic_abort();
    else
        l, u := T, F;
}

void KeReleaseSpinLock_return() {
    if (u)
        slic_abort();
    else
        l, u := F, T;
}

Let l: state == Locked
Let u: state == Unlocked
E0 = { l, u }
BP(\(P', E_0\))

```c
void example () {
    do {
        skip;
        A: KeAcquireSpinLock_return()
        skip;
        skip;
        if (*) {
            skip;
            skip;
            B: KeReleaseSpinLock_return()
            skip;
            if (*) {
                skip;
                skip;
            } else {
                skip;
                skip;
            }
        } else {
            skip;
            skip;
        }
    } while (*)
    skip;
    C: KeReleaseSpinLock_return()
}
```
Model Checking BP($P'$, $E_0$)

Error Path $p_0 : [A, A, \text{SLIC\_ERROR}]$
Predicate discovery over error path

Does $p_0$ represent a feasible execution path of $P$?

Answer given by NEWTON:
“no”, $(nPackets = nPacketsOId)$
Second iteration

Let $b: n\text{Packets} == n\text{PacketsOld}$
$E_1 := \{l, u, b\}$
void example () {
  do {
    skip;
  
  A: KeAcquireSpinLock_return()
    b := T;
    skip;
    if (*) {
      skip;
      skip;
    
  B: KeReleaseSpinLock_return()
      skip;
      if (*) {
        skip;
        skip;
      } else {
        skip;
        skip;
      }
    }
    skip;
    skip;
    b := choose (F, b);
  }
} while (!b)
skip;

C: KeReleaseSpinLock_return()
Model Checking $BP(P', E_1)$

SLIC_ERROR is unreachable in the program $P$. 
Model Checking (under the hood)

Given a desired property, expressed as a temporal logic formula $p$, and a model $M$ with initial state $s$, check if $M, s \models p$

- BDD based
- SAT based
if-then-else operator:
\[ x \rightarrow y_0, y_1 \]

\[ x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1) \]

Examples:
\[ \neg x = x \rightarrow 0, 1 \]
An If-then-else Normal Form is a Boolean expression built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.
Shannon expansion

t = x -> t[1/x], t[0/x]

Any Boolean expression is equivalent to an expression in INF.
Example

Consider \( t = (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2) \)

\[
\begin{align*}
\hat{t} &= x_1 \to t_1, t_0 \\
\hat{t}_0 &= y_1 \to 0, t_{00} \\
\hat{t}_1 &= y_1 \to t_{11}, 0 \\
\hat{t}_{00} &= x_2 \to t_{001}, t_{000} \\
\hat{t}_{01} &= x_2 \to t_{010}, 0 \\
\hat{t}_{10} &= x_2 \to t_{101}, t_{100} \\
\hat{t}_{000} &= y_2 \to 0, 1 \\
\hat{t}_{001} &= y_2 \to 1, 0 \\
\hat{t}_{100} &= y_2 \to 0, 1 \\
\hat{t}_{101} &= y_2 \to 1, 0
\end{align*}
\]
\( t = x_1 \rightarrow t_1, t_0 \)
\( t_0 = y_1 \rightarrow 0, t_{00} \)
\( t_1 = y_1 \rightarrow t_{11}, 0 \)
\( t_{00} = x_2 \rightarrow t_{001}, t_{000} \)
\( t_{11} = x_2 \rightarrow t_{111}, t_{110} \)
\( t_{000} = y_2 \rightarrow 0, 1 \)
\( t_{001} = y_2 \rightarrow 1, 0 \)
SAT (Boolean satisfiability problem)

Given a Boolean formula, is there an assignment for all variables with TRUE or FALSE that will make the formula true?

Like: ( b or T ) implies ( ( a implies F ) and ( b or a ) )
Theorem 1 (Cook)

SAT is NP-complete.

NP-complete
Problems that are NP-complete can be solved by algorithm that run in exponential time. No polynomial time algorithm are know to exist for any of the NP-complete problems and it is very unlikely that polynomial time algorithm should indeed exist though nobody has yet been able to prove their non-existence.
Zchaff

- SAT solver developed at Princeton University
- One of the fastest prover around
- Problems with millions of variables, with tens of million clauses
Limmat

- Developed by Prof. Biere (now at Linz, Austria)
- [http://fmv.jku.at/software](http://fmv.jku.at/software)
- Won a couple of competitions
- Now replaced by Quantor
Application of SAT solvers

With SAT solvers, we are able to analyze complex boolean properties.