Software Verification

Lecture 8: Software Model Checking

Carlo A. Furia
Program Verification: the very idea

\[ P: \text{a program} \]
\[
\text{max (a, b: INTEGER): INTEGER is}
\]
\[
do
\]
\[
\text{if a > b then}
\]
\[
\text{Result := a}
\]
\[
\text{else}
\]
\[
\text{Result := b}
\]
\[
\text{end}
\]
\[
\text{end}
\]

\[ S: \text{a specification} \]
\[
\text{require}
\]
\[
\text{True}
\]
\[
\text{ensure}
\]
\[
\text{Result }\geq a
\]
\[
\text{Result }\geq b
\]

Does \[ P \models S \] hold?

The Program Verification problem:

- \textbf{Given:} a program \( P \) and a specification \( S \)
- \textbf{Determine:} if every execution of \( P \), for any value of input parameters, satisfies \( S \)
Verification of Finite-State Program

P: a program
S: a specification

Does \( P \models S \) hold?

The Program Verification problem is decidable if \( P \) is finite-state

- Model-checking techniques

But real programs are not finite-state.
Software Model-Checking: the Very Idea

The term *Software Model-Checking* denotes an array of techniques to automatically verify real programs based on finite-state models of them.

It is a convergence of verification techniques which started happening during the late 1990's.

The term “software model checker” is probably a misnomer [...] We retain the term solely to reflect historical development.

-- R. Jhala & R. Majumdar: “Software Model Checking”
*ACM CSUR, October 2009*
Abstraction/Refinement Software M.-C.

Software Model-Checking based on CEGAR:
Counterexample-Guided Abstraction/Refinement

- A successful framework for software model-checking

Integrates three fundamental techniques:
- Predicate abstraction of programs
- Detection of spurious counterexamples
- Refinement by predicate discovery
The Big Picture
CouterExample Guided Abstraction Refinement

ABSTRACT PROGRAM

CONCRETE PROGRAM

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM

CONCRETE PROGRAM

PROVE correct

find BUG

(increasing) abstraction
Counterexample Guided Abstraction Refinement

Abstract Program → Refine → Concrete Program

Prove correct

Find bug

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM ➔ MODEL-CHECK ➔ CONCRETE PROGRAM

(increasing) abstraction
CounterExample Guided Abstraction Refinement

verification fails: COUNTEREXAMPLE

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM

(increasing) abstraction
CounterExample Guided Abstraction Refinement

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM

COUNTEREAMPLE not executable

(increasing) abstraction
**CounterExample Guided Abstraction Refinement**

verification fails: COUNTEREXAMPLE is COUNTEREXAMPLE executable?

ABSTRACT PROGRAM

REFINE by ruling out concrete execution

MODEL-CHECK

COUNTEREXAMPLE not executable

CONCRETE PROGRAM

(increasing) abstraction
CounterExample Guided Abstraction Refinement

ABSTRACT PROGRAM

REFINE

CONCRETE PROGRAM

(increasing) abstraction
CouterExample Guided Abstraction Refinement

ABSTRACT PROGRAM CONCRETE PROGRAM

(increasing) abstraction
Outcome: Successful Verification

proof SUCCEEDS: PROGRAM is VERIFIED

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM
Outcome: Real Bug Found

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

ABSTRACT PROGRAM

MODEL-CHECK

CONCRETE PROGRAM

COUNTEREREAMPLE executable: REAL BUG
Outcome: Loop Forever

verification fails: COUNTEREXAMPLE

is COUNTEREXAMPLE executable?

ABSTRACT PROGRAM

REFINE by ruling out concrete execution

MODEL-CHECK

CONCRETE PROGRAM

COUNTEREAMPLE not executable

(increasing) abstraction
CEGAR Software Model-Checking

Integrates three fundamental techniques:

• Predicate abstraction of programs
• Detection of spurious counterexamples
• Refinement by predicate discovery

Let us now present these techniques in some detail.
Predicate Abstraction
Abstraction

Abstraction is a pervasive idea in computer science. It has to do with modeling some crucial (behavioral) aspects while ignoring some other, less relevant, ones.

• **Semantics** of a program \( P \): a set of runs \( \langle P \rangle \)
  - set of all runs of \( P \) for any choice of input arguments
  - a run is completely described by a list of program locations that gets executed in order

• **Abstraction** of a program \( P \): another program \( A_P \)
  - \( A_P \)'s semantics is “similar” to \( P \)'s
    - define some mapping between the runs of \( A_P \) and \( P \)
  - \( A_P \) is more amenable to analysis than \( P \)
Over- and Under-Approximation

Two main kinds of abstraction:

- **over-approximation**: program $AO\_P$
  - $AO\_P$ allows "more runs" than $P$
  - for every $r \in \langle P \rangle$ there exists a $r' \in \langle AO\_P \rangle$
  - intuitively: $\langle P \rangle \subseteq \langle AO\_P \rangle$
  - $AO\_P$ allows some runs that are "spurious" (also "infeasible") for $P$

- **under-approximation**: program $AU\_P$
  - $AU\_P$ allows "fewer runs" than $P$
  - for every $r \in \langle AU\_P \rangle$ there exists a $r' \in \langle P \rangle$
  - intuitively: $\langle AU\_P \rangle \subseteq \langle P \rangle$
  - $AU\_P$ disallows some runs that are "legal" (also "feasible") for $P
Over- and Under-Approximation: Example

max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    else Result := y
  end
end

AO_max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    else Result := y
    end
    if random_Boolean then Result := 3 end
  end

AU_max (x, y: INTEGER): INTEGER
  do
    if x > y
      then Result := x
    else check False
    end
  end
Predicate Abstraction

In predicate abstraction, the abstraction $A_P$ of a program $P$ uses only Boolean variables called “predicates”.

- Each predicate captures a significant fact about the state of $P$.
- The abstraction $A_P$ is constructed parametrically w.r.t. a set $\text{pred}$ of chosen predicates as an over-approximation of the program $P$.
  - the arguments of $A_P$ are the predicates in $\text{pred}$
    - assume arguments are both input and output parameters (this deviates from Eiffel's semantics)
  - each statement $\text{stmt}$ in $P$ is replaced by a (possibly compound) statement $\text{stmt}'$ in $A_P$ such that, for every $p$ in $\text{pred}$:
    - if executing $\text{stmt}$ in $P$ leads to a state where $p$ holds, then $\text{stmt}'$ must possibly set $p$ to True
    - if executing $\text{stmt}$ in $P$ leads to a state where $p$ doesn't hold, then $\text{stmt}'$ must possibly set $p$ to False
  - the details vary from implementation to implementation
Predicate Abstraction: Example

\[ \text{max (x, y: INTEGER): INTEGER do} \]
\[ \quad \text{if } x > y \]
\[ \quad \quad \text{then Result := x} \]
\[ \quad \quad \text{else Result := y} \]
\[ \quad \text{end} \]
\[ \text{ensure Result } \geq x \text{ and Result } \geq y \text{ end} \]

Predicates:

- \( p: x > y \)
- \( q: \text{Result } \geq x \)
- \( r: \text{Result } \geq y \)

and notice that:

- \( p \land q \Rightarrow r \)
- \( \neg p \land r \Rightarrow q \)

\[
\text{Apqr\_max (p, q, r: BOOLEAN) do} \\
\quad \text{if } p \]
\[ \quad \quad \text{then } q := \text{True } ; \ r := \text{True} \]
\[ \quad \quad \text{else } r := \text{True } ; \ q := \text{True} \]
\[ \quad \text{end} \]
\[ \text{ensure } q \text{ and } r \text{ end} \]
Predicate Abstraction and Verification

What does it mean to verify the predicate abstraction $A_P$ of a program $P$?

- $A_P$ is finite state
  - verification is decidable: we can verify $A_P$ automatically
- $A_P$ is an over-approximation of $P$
  - if $A_P$ is correct then so is $P$
    - any run of $P$ is abstracted by some run of $A_P$
  - if $A_P$ is not correct we can't conclude about the correctness of $P$
    - a counterexample run of $A_P$: the abstract counterexample $r$
      - if $r$ is also the abstraction of some run of $P$ then $P$ is also not correct
      - if $r$ is a run which infeasible for $P$ then $r$ is a spurious counterexample
Model-checking a Boolean Program

- For a Boolean program \( P \) over predicates \( \text{pred} = \{ p(1), ..., p(m) \} \)
  - \( P \)'s body: a sequence \( \text{loc} = [L(1), ..., L(n)] \) of instructions or conditional jumps
  - \( P \)'s postcondition: \( \text{post} \)
- Build an FSA \( = [\Sigma, S, I, \rho, F] \) where:
  - \( \Sigma = \text{loc} \)
  - \( S = \{ \text{True, False} \}^n \times (\text{loc} \cup \{ \text{halt} \} ) \)
    - each state in \( S \) denotes a program state:
      - a truth value for every Boolean variable in \( \text{pred} \)
      - a program location which represents the next line to be executed, or \( \text{halt} \) if the execution has terminated
  - \( I = \{ [v(1), ..., v(m), L(1)] \in S \} \)
    - the initial states are for any value of the input Boolean arguments
  - \( L(1) \) is the next instruction to be executed
  - \([v'(1), ..., v'(m), L'] \in \rho ([v(1), ..., v(m), L], L) \) iff
    - \( L \) is a conditional jump and:
      - \([v(1), ..., v(m)] \) satisfies the condition; and
      - \( v'(i) = v(i) \) for all \( 1 \leq i \leq m \); and
      - \( L' \) is the target of the jump when successful.
    - \( L \) is a conditional jump and:
      - \([v(1), ..., v(m)] \) does not satisfy the condition; and
      - \( v'(i) = v(i) \) for all \( 1 \leq i \leq m \); and
      - \( L' \) is the target of the jump when unsuccessful.
    - \( L \) is an instruction and:
      - \([v'(1), ..., v'(m)] \) is the state resulting from executing \( L \) on state \([v(1), ..., v(m)] \); and
      - \( L' \) is the successor of \( L \) (or \( \text{halt} \) if the program halts after executing \( L \))
  - \( F = \{ [v(1), ..., v(m), \text{halt}] \in S \mid \text{post} \) does not hold for \([v(1), ..., v(m)] \} \)
    - error states: halting states where the postcondition doesn’t hold
Predicate Abstraction: Example

Apqr_max (p, q, r: BOOLEAN) do

1: if p
2: then q := True
3: r := True
4: else r := True
5: q := True
end

ensure q and r end

- Error states: including predicates \( \neg q \) or \( \neg r \) without outgoing edges
- There are clearly no accepting (error) runs because the error states are not even connected
- Apqr_max is correct and so is max
Detection of Spurious Counterexamples
Predicate Abstraction and Verification

What does it mean to verify the predicate abstraction $A_P$ of a program $P$?

- $A_P$ is an over-approximation of $P$
  - if $A_P$ is not correct we can't conclude about the correctness of $P$

- a counterexample run of $A_P$: the abstract counterexample $r$
  1. if $r$ is also the abstraction of some run of $P$ then $P$ is also not correct
  2. if $r$ is a run which infeasible for $P$ then $r$ is a spurious counterexample

Let us show an automated technique to detect whether a counterexample is spurious
Abstract Counterexamples

Consider an abstract counterexample (c.e.), i.e. a run of the finite-state predicate abstraction $A_P$

\[
\begin{align*}
&\text{Pred}(0) \quad \text{Abstract initial state} \\
&\text{Stmt}(1) \quad \text{Instruction or test} \\
&\text{Pred}(1) \quad \text{Abstract state} \\
&\text{Stmt}(2) \quad \text{Instruction or test} \\
&\ldots \quad \text{Instruction or test} \\
&\text{Pred}(N) \quad \text{Abstract final state} \\
&\text{Stmt}(N)
\end{align*}
\]

Goal: find whether there exists a concrete run of $P$ which is abstracted by this abstract counterexample
Abstract Counterexamples: Example

\texttt{max (x, y: INTEGER): INTEGER do}

\texttt{if x > y}

\texttt{then Result := x}

\texttt{else Result := y}

\texttt{end}

\texttt{ensure Result \geq x and Result \geq y end}

Predicates:

- \texttt{q: Result \geq x}
- \texttt{r: Result \geq y}

\texttt{Aqr\_max (q, r: BOOLEAN) do}

\texttt{if random\_Boolean}

\texttt{then q := True}

\texttt{else r := True}

\texttt{end}

\texttt{ensure q and r end}
Abstract Counterexamples: Example

\[ A_{qr_{\text{max}}}(q, r: \text{BOOLEAN}) \text{ do} \]

\[ \text{if } \text{random\_Boolean} \]
\[ \text{then } q := \text{True} \]
\[ \text{else } r := \text{True} \]
\[ \text{end} \]
\[ \text{ensure } q \text{ and } r \text{ end} \]

- Error states:
  - including \(-q\) or \(-r\)
  - and without outgoing edges

- An abstract counterexample trace in green
Concrete Run of Abstract C.E.

Because of how $A_P$ has been built, there exists a (possibly compound) statement in $P$ for every statement in $A_P$

Abstract run:  Concrete run:

{ Pred(0) }  Concrete-stmt(1)
 Stmt(1)  
{ Pred(1) }  Concrete-stmt(2)
 Stmt(2)  
...
 Stmt(N)  ...
{ Pred(N) }

Let us check whether the concrete run is feasible or not, according to the semantics of $P$
# Feasibility of a Concrete Run

Compute the **weakest precondition** of `True` over the concrete run (this is doable automatically because there are no loops):

<table>
<thead>
<tr>
<th>Abstract run:</th>
<th>Concrete run:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ Pred(0) }</code></td>
<td><code>{ WP(0) }</code></td>
</tr>
<tr>
<td>Stmt(1)</td>
<td>Concrete-stmt(1)</td>
</tr>
<tr>
<td><code>{ Pred(1) }</code></td>
<td><code>{ WP(1) }</code></td>
</tr>
<tr>
<td>Stmt(2)</td>
<td>Concrete-stmt(2)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Stmt(N)</td>
<td>Concrete-stmt(N)</td>
</tr>
<tr>
<td><code>{ Pred(N) }</code></td>
<td><code>{ True }</code></td>
</tr>
</tbody>
</table>

Every assertion `WP(i)` characterizes the **states of P** leading to a final state where `Pred(N)` holds and hence where the postcondition fails.
Feasibility of a Concrete Run

The concrete run is feasible iff \( WP(i) \) and \( \text{Pred}(i) \) is satisfiable for every \( 1 \leq i \leq N \), otherwise the abstract run does not correspond to any “real” concrete run.

Concrete run:

\[
\begin{align*}
\{ \text{Pred}(0) \text{ and } WP(0) \} \\
\text{Concrete-stmt}(1) \\
\{ \text{Pred}(1) \text{ and } WP(1) \} \\
\text{Concrete-stmt}(2) \\
\ldots \\
\text{Concrete-stmt}(N) \\
\{ \text{Pred}(N) \text{ and } \text{True} \}
\end{align*}
\]
Spurious Counterexamples: Example

Abstract c.e. trace:
\{q, \neg r\}
  \{q, \neg r\}
    \text{random\_Boolean}
    q := \text{True}
  \{q, \neg r\}

Concrete trace:
\{x > y\}
  \{x > y\}
  \{\text{True}\}
  \text{Result} := x
  \{\text{True}\}

The counterexample is \textit{infeasible} because:
\{x > y \text{ and } q \text{ and } \neg r\} \textit{is inconsistent}
because \{x > y \text{ and } q\} \textit{implies} \{r\}
Abstract Counterexamples: Example

neg_pow (x, y: INTEGER): INTEGER do
require x < 0 and y > 0
  from Result := 1
  until y ≤ 0
  loop
    Result := Result * x
    y := y - 1
  end
ensure Result > 0 end

Predicates:

- p: x < 0
- q: y > 0
- r: Result > 0

Apqr_neg_pow (p, q, r: BOOLEAN) do
require p and q
  from r := True
  until ¬q
  loop
    if random_Boolean
      then r := True else r := False
    if random_Boolean and q then q := False end
ensure q and r end
Abstract Counterexamples: Example

Apqr_neg_pow (p, q, r: BOOLEAN) do
require p and q
from r := True
until ¬q
loop
  if random_Boolean
    then r := True else r := False
  if random_Boolean and q then q := False end
ensure q and r end

Predicates:

• p: x < 0
• q: y > 0
• r: Result > 0

Abstract c.e. trace:
{p, q, ¬r}
  r := True
{p, q, r}
  [q]
{p, q, r}
  [random_Boolean]
{p, q, r}
  r := False
{p, q, ¬r}
  [random_Boolean and q]
  q := False
{p, ¬q, ¬r}
  [¬q]
{p, ¬q, ¬r}
Abstract Counterexamples: Example

Abstract c.e. trace:
\{p, q, r\}
  \(r := \text{True}\)
\{p, q, r\}
  \[q\]
\{p, q, r\}
  \[\text{random\_Boolean}\]
\{p, q, r\}
  \(r := \text{False}\)
\{p, q, \neg r\}
  \[\text{random\_Boolean and } q\]
  \(q := \text{False}\)
\{p, \neg q, \neg r\}
  \[\neg q\]
\{p, \neg q, \neg r\}

Concrete trace:
\{y = 1\}
  \(\text{Result} := 1\)
\{y = 1\}
  \[y > 0\]
\{y \leq 1\}

  \(\text{Result} := \text{Result} \ast x\)
\{y \leq 1\}

    \(y := y - 1\)
\{y \leq 0\}

    \[y \leq 0\]
\{\text{True}\}
Abstract Counterexamples: Example

Concrete trace:
\{y = 1\}
  Result := 1
\{y = 1\}
  \[y > 0\]
\{y \leq 1\}

\text{Result} := \text{Result} \ast x
\{y \leq 1\}

\text{y} := \text{y} - 1
\{y \leq 0\}
  \[y \leq 0\]
\{True\}

Predicates:
\begin{itemize}
  \item p: \(x < 0\)
  \item q: \(y > 0\)
  \item r: \text{Result} > 0
\end{itemize}

The counterexample is feasible: we have found a real bug in the concrete program occurring for input \(y = 1\) (and any \(x < 0\)).
Predicate Discovery and Refinement
Predicate Discovery

A spurious counterexample shows that the used abstraction is too coarse.

We build a finer abstraction by adding new predicates to the set $\text{pred}$.

These new predicates must be chosen so that the spurious counterexample is not allowed in the new abstraction.
Syntax-based Predicate Discovery

The simplest way to find new predicates is syntactic:

Concrete run:

\[
\{ \text{Pred}(O) \text{ and WP}(O) \} \quad \{ \text{WP}(O) \setminus \{ \text{Pred}(O) \} \\
\text{Concrete-stmt}(1)
\{ \text{Pred}(1) \text{ and WP}(1) \} \quad \{ \text{WP}(1) \setminus \{ \text{Pred}(1) \} \\
\text{Concrete-stmt}(2)
\]
...

\[
\text{Concrete-stmt}(N) \\
\{ \text{Pred}(N) \text{ and True} \} \quad \{ \text{True} \setminus \{ \text{Pred}(N) \}
\]

Look for predicates that:

- hold in the concrete run
- are not traced by any predicate in the abstract run
- contradict the predicates in the abstract run
Syntax-based Predicate Discovery: Example

Concrete trace:
{x > y} \ {q, ¬r}
[x > y]
{True} \ {q, ¬r}
Result := x
{True} \ {q, ¬r}

Predicates:
• q: Result >= x
• ¬r: Result < y

The predicate from the concrete run that is not traced in the abstract run is:
• p = x > y

Predicate p contradicts {q, ¬r}. It is enough to verify the program with the new abstraction.
Summary, Tools, and Extensions
CEGAR: Summary

- Finite-state **predicate abstraction** of real programs
  - **Static analysis** & abstract interpretation
- Automated **verification** of finite-state programs
  - **Model checking** of reachability properties
- Detection of **spurious counterexamples**
  - Axiomatic semantics & automated theorem proving
- Automated **counterexample-based refinement**
  - **Symbolic model-checking techniques**
Software Model-Checking Tools

- **CEGAR** software model-checkers
  - **SLAM** -- Ball and Rajamani, ~2001
    - first full implementation of CEGAR software m-c
    - used at Microsoft for device driver verification
  - **BLAST** -- Henzinger et al., ~2002
    - does lazy abstraction: partial refinement of abstract program
    - several extensions for arrays, recursive routines, etc.
  - **Magic** -- Clarke et al., ~2003
    - modular verification of concurrent programs
  - **F-Soft** -- Gupta et al., ~2005
    - Combines software model-checking with abstract interpretation techniques
- **Other** software model-checking tools
  - **Verisoft** -- Godefroid et al. ~2001
  - **Java PathFinder** -- Visser et al., ~2000
  - **Bandera** -- Hatcliff, Dwyers, et al., ~2000
Software Model-Checking: Extensions

- Inter-procedural analysis
- Complex data structures
- Concurrent programs
- Recursive routines
- Heap-based languages
- Termination analysis
- Integration with other verification techniques
  - Static analysis
  - Testing
- ...

None of these directions is exclusive domain of software model-checking, of course...