CS 267: Automated Verification

Lectures 14: Predicate Abstraction, Counter-Example Guided Abstraction Refinement, Abstract Interpretation

Instructor: Tevfik Bultan
Model Checking Programs Using Abstraction

- Program model checking tools generally rely on automated abstraction techniques to reduce the state space of the system such as:
  - Abstract interpretation
  - Predicate abstraction
- If the abstraction is conservative, then, if there is no error in the abstracted program, we can conclude that there is no error in the original program
- In general the problem is to construct a finite state model from the program such that the errors or absence of errors can be demonstrated on the finite state model
  - Model extraction problem
Model Checking Programs via Abstraction

• Bandera
  – A tool for extracting finite state models from programs
  – Uses various abstract domains to map the state space of the program to a finite set of states via abstraction

• SLAM project at Microsoft Research
  – Symbolic model checking for C programs
  – Can handle unbounded recursion but does not handle concurrency
  – Uses predicate abstraction, counter-example guided abstraction refinement and BDDs
Abstraction (A simplified view)

• Abstraction is an effective tool in verification

• Given a transition system, we want to generate an abstract transition system which is easier to analyze

• However, we want to make sure that
  – If a property holds in the abstract transition system, it also holds in the original (concrete) transition system
Abstraction (A simplified view)

• How do we generate an abstract transition system?

• Merge states in the concrete transition system (based on some criteria)
  – This reduces the number of states, so it should be easier to do verification

• Do not eliminate transitions
  – This will make sure that the paths in the abstract transition system subsume the paths in the concrete transition system
Abstraction (A simplified view)

• For every path in the concrete transition system, there is an equivalent path in the abstract transition system
  – If no path in the abstract transition system violate a property, then no path in the concrete system can violate the property

• Using this reasoning we can verify ACTL, LTL and ACTL* properties in the abstract transition system
  – If the property holds on the abstract transition system, we are sure that the property holds in the concrete transition system
  – If the property does not hold in the abstract transition system, then we are not sure if the property holds or not in the concrete transition system
Abstraction (A simplified view)

- If the property does not hold in the abstract transition system, what can we do?
  
- We can *refine* the abstract transition system (split some states that we merged)
  
- We have to make sure that the refined transition system is still an abstraction of the concrete transition system
  
- Then, we can recheck the property again on the refined transition system
  - If the property does not hold again, we can refine again
Abstraction and Simulation

Given two transition systems
• $T_1 = (S_1, I_1, R_1)$ with labeling function $L_1$ and atomic proposition set $AP_1$
• $T_2 = (S_2, I_2, R_2)$ with labeling function $L_2$ and atomic proposition set $AP_2$

We call $H \subseteq (S_1, S_2)$ a simulation relation if,
for any $(s_1, s_2) \in H$
- $L(s_1) \cap AP_2 = L(s_2)$
- For every state $s_1'$ such that $(s_1, s_1') \in R_1$ there exists a state $s_2'$ such that $(s_2, s_2') \in R_2$ and $(s_1', s_2') \in H$

We say that $T_2$ simulates $T_1$ if there exists a simulation relation $H$ such that for each $s_1 \in I_1$, there exists a $s_2 \in I_2$ such that $(s_1, s_2) \in H$. 
Abstraction and Simulation

• If $T_2$ simulates $T_1$ then for every ACTL* formula $f$, $T_2 \models f$ implies $T_1 \models f$

• We can define simulation relations between abstract and concrete transition systems such that
  – the abstract system simulates the concrete system

• Hence when we verify a property in the abstract transition system we know that it also holds for the concrete transition system
Predicate Abstraction

- An automated abstraction technique which can be used to reduce the state space of a program.
- The basic idea in predicate abstraction is to remove some variables from the program by just keeping information about a set of predicates about them.
- For example, a predicate such as $x = y$ maybe the only information necessary about variables $x$ and $y$ to determine the behavior of the program.
  - In that case, we can just store a boolean variable which corresponds to the predicate $x = y$ and remove variables $x$ and $y$ from the program.
  - Predicate abstraction is a technique for doing such abstractions automatically.
Predicate Abstraction

- Given a program and a set of predicates, predicate abstraction abstracts the program so that only the information about the given predicates are preserved.

- The abstracted program adds nondeterminism since in some cases it may not be possible to figure out what the next value of a predicate will be based on the predicates in the given set.

- One needs an automated theorem prover to compute the abstraction.
Predicate Abstraction, A Very Simple Example

- Assume that we have two integer variables $x, y$

- We want to abstract the program using a single predicate “$x=y$”

- We will divide the states of the program to two:
  1. The states where “$x=y$” is true
  2. The states where “$x=y$” is false, i.e., “$x \neq y$”

- We will then merge all the states in the same set
  - This is an abstraction
  - Basically, we forget everything except the value of the predicate “$x=y$”
Predicate Abstraction, A Very Simple Example

• We will represent the predicate “x=y” as the boolean variable B in the abstract program
  – “B=true” will mean “x=y” and
  – “B=false” will mean “x≠y”

• Assume that we want to abstract the following program which contains only one statement:

  y := y+1
Predicate Abstraction, Step 1

- Calculate preconditions based on the predicate

\[
\{x = y + 1\} \quad y := y + 1 \quad \{x = y\}
\]

precondition for \(B\) being true after executing the statement \(y := y + 1\)

Using our temporal logic notation we can say something like:
\[
\{x=y+1\} \equiv AX\{x=y\}
\]

\[
\{x \neq y + 1\} \quad y := y + 1 \quad \{x \neq y\}
\]

precondition for \(B\) being false after executing the statement \(y := y + 1\)

Again, using our temporal logic notation:
\[
\{x \neq y + 1\} \equiv AX\{x \neq y\}
\]
Predicate Abstraction, Step 2

- Use decision procedures to determine if the predicates used for abstraction imply any of the preconditions

\[ x = y \rightarrow x = y + 1 \] ? No

\[ x \neq y \rightarrow x = y + 1 \] ? No

\[ x = y \rightarrow x \neq y + 1 \] ? Yes

\[ x \neq y \rightarrow x \neq y + 1 \] ? No
Predicate Abstraction, Step 3

- Generate abstract code

Predicate abstraction wrt the predicate “x=y”

1) Compute preconditions

{\(x = y + 1\)} \(y := y + 1\) {\(x = y\)}

{\(x \neq y + 1\)} \(y := y + 1\) {\(x \neq y\)}

2) Check implications

3) Generate abstract code

IF B THEN B := false
ELSE B := true | false

\(x = y \rightarrow x = y + 1\) ? No
\(x \neq y \rightarrow x = y + 1\) ? No
\(x = y \rightarrow x \neq y + 1\) ? Yes
\(x \neq y \rightarrow x \neq y + 1\) ? No
Model Checking Push-down Automata

A class of infinite state systems for which model checking is decidable

• Push-down automata: Finite state control + one stack
• LTL model checking for push-down automata is decidable
• This may sound like a theoretical result but it is the basis of the approach used in SLAM toolkit for model checking C programs
  – A program with finite data domains which uses recursion can be modeled as a pushdown automaton
  – A Boolean program generated by predicate abstraction can be represented as a pushdown automaton
Predicate Abstraction + Model Checking Push Down Automata

• Predicate abstraction combined with results on model checking pushdown automata led to some promising tools
  – SLAM project at Microsoft Research for verification of C programs
  – This tool is being used to verify device drivers at Microsoft
• The main idea:
  – Use predicate abstraction to obtain finite state abstractions of a program
  – A program with finite data domains and recursion can be modeled as a pushdown automaton
  – Use results on model checking push-down automata to verify the abstracted (recursive) program
SLAM Toolkit

- SLAM toolkit was developed to find errors in windows device drivers
  - Examples in my slides are from the following paper:
    - “The SLAM Toolkit”, Thomas Ball and Sriram K. Rajamani, CAV 2001
- Windows device drivers are required to interact with the windows kernel according to certain interface rules
- SLAM toolkit has an interface specification language called SLIC (Specification Language for Interface Checking) which is used for writing these interface rules
- The SLAM toolkit instruments the driver code with assertions based on these interface rules
A SLIC Specification for a Lock

SLIC specification:

```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;  
}
```

- This specification states that `KeAcquireSpinLock` has to be called before `KeReleaseSpinLock` is called,
- and `KeAcquireSpinLock` cannot be called back to back before a `KeReleaseSpinLock` is called, and vice versa
A SLIC Specification for a Lock

SLIC specification:

```
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;
}
```

Generated C Code:

```
enum { Unlocked=0, Locked=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;
    }
```
An Example

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();
}
Boolean Programs

• After instrumenting the code, the SLAM toolkit converts the instrumented C program to a Boolean program using predicate abstraction

• The Boolean program consists of only Boolean variables
  – The Boolean variables in the Boolean program are the predicates that are used during predicate abstraction

• The Boolean program can have unbounded recursion
**Boolean Programs**

C Code:

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

Boolean Program:

```plaintext
decl {state==Locked},
    {state==Unlocked} := F,T;

void slic_abort() begin
    SLIC_ERROR: skip;
end

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

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

void example() {
    begin
    do
        skip;
        A: KeAcquireSpinLock_return();
        skip;
        if (*) then
            skip;
        B: KeReleaseSpinLock_return();
        skip;
        if (*) then
            skip;
        else
            skip;
        fi
        skip;
        fi
        while (*);
        skip;
    C: KeReleaseSpinLock_return();
end

C Code:

Boolean Program:

void example()
begin
    do
        skip;
        A: KeAcquireSpinLock_return();
        skip;
        if (*) then
            skip;
        B: KeReleaseSpinLock_return();
        skip;
        if (*) then
            skip;
        else
            skip;
        fi
        skip;
        fi
        while (*);
        skip;
    C: KeReleaseSpinLock_return();
end
Abstraction Preserves Correctness

- The Boolean program that is generated with predicate abstraction is non-deterministic.
  - Non-determinism is used to handle the cases where the predicates used during predicate abstraction are not sufficient enough to determine which branch will be taken.
- If we find no error in the generated abstract Boolean program then we are sure that there are no errors in the original program.
  - The abstract Boolean program allows more behaviors than the original program due to non-determinism.
  - Hence, if the abstract Boolean program is correct then the original program is also correct.
Counter-Example Guided Abstraction Refinement

• However, if we find an error in the abstract Boolean program this does not mean that the original program is incorrect.
  – The erroneous behavior in the abstract Boolean program could be an infeasible execution path that is caused by the non-determinism introduced during abstraction.

• Counter-example guided abstraction refinement is a technique used to iteratively refine the abstract program in order to remove the spurious counter-example traces.
Counter-Example Guided Abstraction Refinement

The basic idea in counter-example guided abstraction refinement is the following:

• First look for an error in the abstract program (if there are no errors, we can terminate since we know that the original program is correct)

• If there is an error in the abstract program, generate a counter-example path on the abstract program

• Check if the generated counter-example path is feasible using a theorem prover.

• If the generated path is infeasible add the predicate from the branch condition where an infeasible choice is made to the predicate set and generate a new abstract program.
void example() {
    do {
        KeAcquireSpinLock();
        A: KeAcquireSpinLock_return();
        nPacketsOld = nPackets;
        req = devExt->WLHV;
        if(req && req->status){
            devExt->WLHV = req->Next;
            KeReleaseSpinLock();
        } else {
            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();
    B: KeReleaseSpinLock_return();
    C: KeReleaseSpinLock_return();
}
Counter-Example Guided Abstraction Refinement

Boolean Program:

```plaintext
void example()
begin
do
  skip;
A: KeAcquireSpinLock_return();
skip;
if (*) then
  skip;
else
  skip;
fi
while (*);
skip;
B: KeReleaseSpinLock_return();
skip;
if (*) then
  skip;
else
  skip;
fi
b := b ? F : *;
fi
C: KeReleaseSpinLock_return();
end
```

Refined Boolean Program:
(using the predicate (nPackets = npacketsOld))

```plaintext
void example()
begin
do
  skip;
A: KeAcquireSpinLock_return();
b := T;
if (*) then
  skip;
else
  skip;
fi
b := b ? F : *
fi
while (!b);
skip;
B: KeReleaseSpinLock_return();
skip;
if (*) then
  skip;
else
  skip;
fi
  b := b ? F : *;
fi
C: KeReleaseSpinLock_return();
end
```

the boolean variable b
represents the predicate
(nPackets = npacketsOld)
Counter-Example Guided Abstraction Refinement

• Using counter-example guided abstraction refinement we are iteratively creating more and more refined abstractions.
• This iterative abstraction refinement loop is not guaranteed to converge for infinite domains.
  – This is not surprising since automated verification for infinite domains is undecidable in general.
• The challenge in this approach is automatically choosing the right set of predicates for abstraction refinement.
  – This is similar to finding a loop invariant that is strong enough to prove the property of interest.
Abstract Interpretation

• Abstract interpretation provides a general framework for defining abstractions

• Different abstract domains can be combined using abstract interpretation framework

• Abstract interpretation framework also provides techniques such as widening for computing approximations of fixpoints
Abstract Interpretation Example

• Assume that we have a program with some integer variables
• We want to figure out possible values these variables can take at a certain point in the program
  – The results will be a set of integer values for each variable (i.e., the result for each variable will be a member of $2^\mathbb{Z}$ where $\mathbb{Z}$ is the set of integers)
• An easy answer would be to return $\mathbb{Z}$ for all the variables
  – I.e., say that each variable can possibly take any value
  – This is not a very precise and helpful answer
• The smaller the sets in our answer, the more precise our answer is
  – Of course we are not allowed to give a wrong answer by omitting a value that a variable can take!
Abstract Interpretation Example

• Assume that we have two integer variables $x$ and $y$
• The answer we return should be something like
  – $x \in \{1, 2, 3, 4\}$
  – $y \in \{n \mid n > 5\}$
  
  the variables $x$ and $y$ should not take any value outside of these sets for any execution of the program

• Unfortunately if we use $2^\mathbb{Z}$ and develop a static analysis to solve this problem the fixpoint computations will not converge since $2^\mathbb{Z}$ an infinite lattice
  – Use abstraction!
Abstract Interpretation Example

• Define an abstract domain for integers
  – For example: \(2^{\{\text{neg, zero, pos}\}}\)

• Define abstraction and concretization functions between the integer domain and this abstract domain

• Interpret integer expressions in the abstract domain

\[
\begin{align*}
\text{if } (y == 0) \{ & \quad \text{if } (y == \{\text{zero}\}) \{ \\
& \quad x = 2; \quad x = \{\text{pos}\}; \\
& \quad y = x; \quad y = x; \\
& \} \}
\end{align*}
\]

• The abstract domain \(2^{\{\text{neg, zero, pos}\}}\) corresponds to a finite lattice, so the fixpoint computations will converge
Abstract Interpretation

In abstract interpretation framework:

• We define an abstraction function from the concrete domain to the abstract domain
  – $\alpha$: Concrete $\rightarrow$ Abstract

• We define a concretization function from the abstract domain to the concrete domain
  – $\gamma$: Abstract $\rightarrow$ Concrete
Abstract Interpretation Example

- Concrete domain: $2^\mathbb{Z}$ (sets of integers)
- Abstract domain: $2^{\{\text{neg, zero, pos}\}}$
- Abstraction function $\alpha$: $2^\mathbb{Z} \rightarrow 2^{\{\text{neg, zero, pos}\}}$
  - $\alpha(c) = a$ such that $(\exists n \in c, n = 0 \iff \text{zero} \in a) \land (\exists n \in c, n > 0 \iff \text{pos} \in a) \land (\exists n \in c, n < 0 \iff \text{neg} \in a)$
- Concretization function $\gamma$: $2^{\{\text{neg, zero, pos}\}} \rightarrow 2^\mathbb{Z}$
  - $\gamma(a) = c$ such that $(\text{zero} \in a \iff 0 \in c) \land (\text{pos} \in a \iff \{n \mid n>0\} \subseteq c) \land (\text{neg} \in a \iff \{n \mid n<0\} \subseteq c)$
Precision Ordering

• Both for the concrete and abstract domains we can define a partial ordering which denotes their precision

• For both the concrete domain $2^\mathbb{Z}$ and the abstract domain $2\{\text{neg, zero, pos}\}$ the precision ordering is $\subseteq$
  – $a \subseteq b$ means that $a$ is more precise than $b$

• $(\alpha, \gamma)$ is called a Galois connection if and only if
  $$\alpha(a) \subseteq b \iff a \subseteq \gamma(b)$$
Abstract Interpretation

Concrete Domain

Abstract Domain
Predicate Abstraction as Abstract Interpretation

S: set of states of the transition system
• Concrete lattice: $2^S$

$p_1, p_2, \ldots, p_k$ set of predicates used for predicate abstraction
$b_1, b_2, \ldots, b_k$ boolean variables representing the predicates
• Abstract lattice: BF, Boolean formulas over $b_1, b_2, \ldots, b_k$

• Concretization function $\gamma$: BF $\rightarrow$ $2^S$
  $\gamma(F) = F[b_1/p_1, b_2/p_2, \ldots, b_k/p_k]$

• Abstraction function $\alpha$: $2^S \rightarrow$ BF
  $\alpha(Q) = \land \{ F \mid Q \subseteq \gamma(F) \}$