# Efficient resolution of logical models

## ENSTA-IA303

Alexandre Chapoutot and Sergio Mover

ENSTA Paris

2020-2021

# Lecture 8: Program verification using SMT

# Main goals for today

In class[1]:

- Understand how SMT solvers can be used to prove/*disprove* program correctness
- Transition systems expressed with FOL formulas
- From (simple) programs to FOL formulas
- Verification of transition systems (bounded model checking, induction)

---

[1] Main references:

- BMC paper, TACAS99
- Software Verification (from Handbook of satisfiability)
- Calculus of Computation, Chapter 5

# Main goals for today

In class[1]:

- Understand how SMT solvers can be used to prove/*disprove* program correctness
- Transition systems expressed with FOL formulas
- From (simple) programs to FOL formulas
- Verification of transition systems (bounded model checking, induction)

In the tutorial:

- Implement BMC for transition systems
- Implement induction for transition systems

---

[1] Main references:

- BMC paper, TACAS99
- Software Verification (from Handbook of satisfiability)
- Calculus of Computation, Chapter 5

1. Program verification using SMT
   - Program verification
   - Infinite-state Transition Systems
   - Bounded Model Checking - Finding a violation
   - Proving safety

# Software Errors

Several examples of software bugs in critical systems:

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer
- Loss of communication with Mars Climate Orbiter, 1998: different modules used different unit of measure (imperial vs. metric system)

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer
- Loss of communication with Mars Climate Orbiter, 1998: different modules used different unit of measure (imperial vs. metric system)
- ...

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer
- Loss of communication with Mars Climate Orbiter, 1998: different modules used different unit of measure (imperial vs. metric system)
- ...
- Toyota car, accelerate unintentionally, 2007: bug in the drive-by-wire throttle system
- ...

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer
- Loss of communication with Mars Climate Orbiter, 1998: different modules used different unit of measure (imperial vs. metric system)
- ...
- Toyota car, accelerate unintentionally, 2007: bug in the drive-by-wire throttle system
- ...
- Boeing 737 Max glitch in MCAS system leading to fatal plane crash, 2018: wrong data from a (single) faulty sensor

# Software Errors

Several examples of software bugs in critical systems:

- Patriot Missile not intercepting enemy's missiles, 1991: bug in the time-step calculation (truncating errors)
- Explosion of the Ariane 5 in 1996: wrong conversion from a 64 bit floating point number to a 16 bit signed integer
- Loss of communication with Mars Climate Orbiter, 1998: different modules used different unit of measure (imperial vs. metric system)
- ...
- Toyota car, accelerate unintentionally, 2007: bug in the drive-by-wire throttle system
- ...
- Boeing 737 Max glitch in MCAS system leading to fatal plane crash, 2018: wrong data from a (single) faulty sensor
- Medtronic pacemakers recalled in 2019: a software bug could cause the device to lose pacing function.

Need for automatic technique to find bugs and certify software correctness

# Software Verification Problem

```
int f(int i, int j) {
  if (i < 0 || j < 0) {
    return 0;
  }
  while (i >= 0) {
    j = j + 1;
    i = i - 1;
  }
  assert (i < 0 && j > i);
  return j;
}
```

Does the assertion $i < 0 \wedge j > 1$ hold, for all possible values of $i$ and $j$?

# Software Verification Problem

```
int f(int i, int j) {
  if (i < 0 || j < 0) {
    return 0;
  }
  while (i >= 0) {
    j = j + 1;
    i = i - 1;
  }
  assert (i < 0 && j > i);
  return j;
}
```

Does the assertion $i < 0 \wedge j > 1$ hold, for all possible values of $i$ and $j$?

- Can we automatically prove the program correct?
  i.e., that the assertion holds **for all** executions of the program

# Software Verification Problem

```
int f(int i, int j) {
  if (i < 0 || j < 0) {
    return 0;
  }
  while (i >= 0) {
    j = j + 1;
    i = i - 1;
  }
  assert (i < 0 && j > i);
  return j;
}
```

Does the assertion $i < 0 \land j > 1$ hold, for all possible values of $i$ and $j$?

- Can we automatically prove the program correct?
  i.e., that the assertion holds **for all** executions of the program
- Can we automatically find bugs?
  i.e., that **there exists** an execution of the program that violatates the assertion

# Different kind of properties

Safety properties (partial correctness):

- Does the program never reach a "bad" state?
  Example: `assert (i < 0 && j > i);`

# Different kind of properties

Safety properties (partial correctness):

- Does the program never reach a "bad" state?
  Example: assert (i < 0 && j > i);

Progress properties (total correctness):

- Does the program eventually reach a set of states?
  Example: The program always terminate.

# Different kind of properties

Safety properties (partial correctness):

- Does the program never reach a "bad" state?
  Example: assert (i < 0 && j > i);

Progress properties (total correctness):

- Does the program eventually reach a set of states?
  Example: The program always terminate.

Today: we focus on safety properties.

# Some of the existing approaches in program verification

- Abstract interpretation

# Some of the existing approaches in program verification

- Abstract interpretation
- Verification using the Control-Flow Automata (e.g., based on SMT)

# Some of the existing approaches in program verification

- Abstract interpretation
- Verification using the Control-Flow Automata (e.g., based on SMT)
- Reduce the safety verification problem to the verification of FOL formulas: based on SMT
  - Verification condition generation
  - Verification of Infinte-state transition systems
  - Verification of Constrained Horn Clauses

# Some of the existing approaches in program verification

- Abstract interpretation
- Verification using the Control-Flow Automata (e.g., based on SMT)
- Reduce the safety verification problem to the verification of FOL formulas: based on SMT
  - Verification condition generation
  - Verification of Infinte-state transition systems
  - Verification of Constrained Horn Clauses

Today:

- Verification using infinite-state transition systems
  Very generic: can be applied to other formalism (e.g., LUSTRE)

# Some of the existing approaches in program verification

- Abstract interpretation
- Verification using the Control-Flow Automata (e.g., based on SMT)
- Reduce the safety verification problem to the verification of FOL formulas: based on SMT
  - Verification condition generation
  - Verification of Infinte-state transition systems
  - Verification of Constrained Horn Clauses

Today:

- Verification using infinite-state transition systems
  Very generic: can be applied to other formalism (e.g., LUSTRE)
- Limitations: no recursion, no dynamic memory allocation (some techniques overcome these limitations)
- Mainly focus on finding violations of safety properties (while not too much emphasis on proving)

# Some of the existing approaches in program verification

- Abstract interpretation
- Verification using the Control-Flow Automata (e.g., based on SMT)
- Reduce the safety verification problem to the verification of FOL formulas: based on SMT
  - Verification condition generation
  - Verification of Infinte-state transition systems
  - Verification of Constrained Horn Clauses

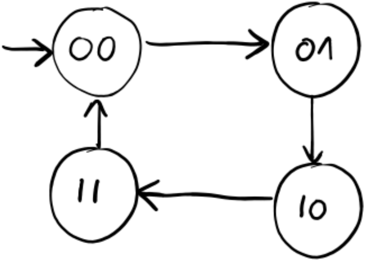
Today:

- Verification using infinite-state transition systems
  Very generic: can be applied to other formalism (e.g., LUSTRE)
- Limitations: no recursion, no dynamic memory allocation (some techniques overcome these limitations)
- Mainly focus on finding violations of safety properties (while not too much emphasis on proving)

**Reminder:** the verification problem is undecidable - the algorithms either approximate the results or may not terminate.

1. Program verification using SMT
    - Program verification
    - Infinite-state Transition Systems
    - Bounded Model Checking - Finding a violation
    - Proving safety

# Symbolic representation of a state machine



We can represent the 2-bit counter with:

- two Boolean variables $V := \{b_0, b_1\}$
- The set of initial state as a Propositional Logic formula: $I(V) := \neg b_0 \wedge \neg b_1$
- The transitions as the formula: $T(V, V') := (b'_0 \leftrightarrow \neg b_0) \wedge (b'_1 \leftrightarrow (b_0 \oplus b_1))$

- A state is an assignment to the variables $V$ $\{b_0 \mapsto \bot, b_1 \mapsto \bot\}$
- Formulas over $V$ represents sets of states
- Formulas over $V', V$ represents a set of transitions ($V'$ is the next state)
- A path $\{b_0 \mapsto \bot, b_1 \mapsto \bot\}; \{b_0 \mapsto \top, b_1 \mapsto \bot\}; \{b_0 \mapsto \top, b_1 \mapsto \top\}; \ldots$

# Finite-state transition system

$S = (V, I(V), T(V, V'))$ is a **finite-state** transition system where:

- $V$ is a set of Boolean variables
- $I(V)$ is a propositional logic formula over the variables $V$
- $T(V', V)$ is a propositional logic formula over the variables $V$

# Finite-state transition system

$S = (V, I(V), T(V, V'))$ is a **finite-state** transition system where:

- $V$ is a set of Boolean variables
- $I(V)$ is a propositional logic formula over the variables $V$
- $T(V', V)$ is a propositional logic formula over the variables $V$

A path $\pi := s_0; s_1; \ldots; s_k$ is a path of $S$ if:

- A state $s_i$ assigns a value to the variables $V$
- $s_0 \models I(V)$
- $s_i, s_{i+1} \models T(V', V)$ for all $0 \leq i < k$

# Infinite-State transition system

$S = (V, I(V), T(V, V'))$ is an **infinite-state** transition system where:

- $V$ is a set of theory variables (i.e., 0-ary functions)
- $I(V)$ is a $\Sigma_{\mathcal{T}}$-formula over the variables $V$
- $T(V', V)$ is a $\Sigma_{\mathcal{T}}$-formula over the variables $V \cup V'$

# Infinite-State transition system

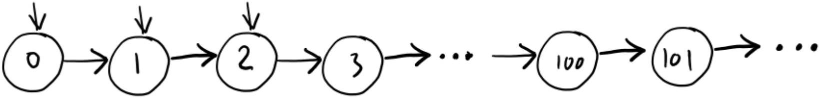
$S = (V, I(V), T(V, V'))$ is an **infinite-state** transition system where:

- $V$ is a set of theory variables (i.e., 0-ary functions)
- $I(V)$ is a $\Sigma_{\mathcal{T}}$-formula over the variables $V$
- $T(V', V)$ is a $\Sigma_{\mathcal{T}}$-formula over the variables $V \cup V'$

A path $\pi := s_0; s_1; \ldots; s_k$ is a path of $S$ if:

- A state $s_i$ assigns a value to the variables $V$
  Since the domain of $V$ is infinite, the system as an infinite number of states.
- $s_0 \models I(V)$
- $s_i, s_{i+1} \models T(V', V)$ for all $0 \leq i < k$

# Infinite state transition system - Example

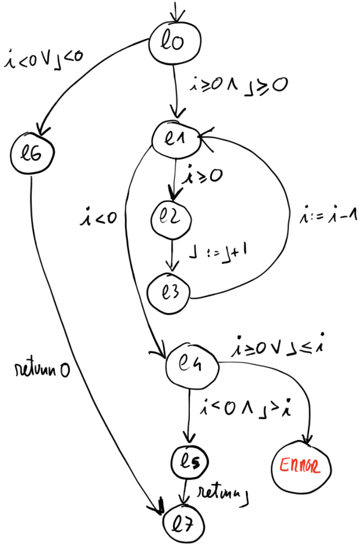


The infinite-state transition system for the counter:

- $V := \{i\}$
- $I(V) := i \leq 2$
- $T(V, V') := i' = i + 1$

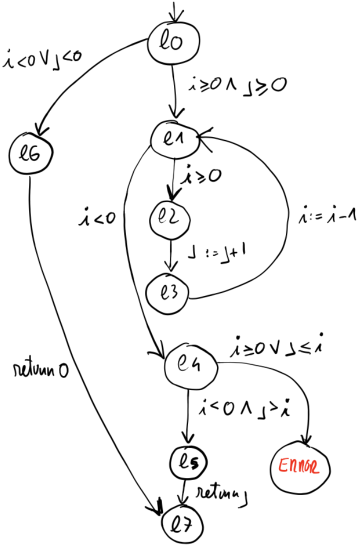
# From programs to TS - Control Flow Automata (CFA)

```
     int f(int i, int j) {
l0:    if (i < 0 || j < 0) {
l6:      return 0;
       }
l1:    while (i >= 0) {
l2:      j = j + 1;
l3:      i = i - 1;
       }
l4:    assert (i < 0 && j > i);
l5:    return j;
l7:
     }
```



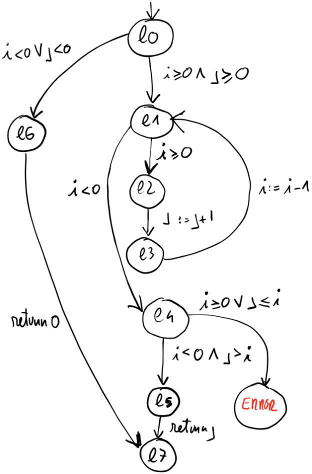
# From programs to TS - Control Flow Automata (CFA)

```
     int f(int i, int j) {
l0:    if (i < 0 || j < 0) {
l6:      return 0;
       }
l1:    while (i >= 0) {
l2:      j = j + 1;
l3:      i = i - 1;
       }
l4:    assert (i < 0 && j > i);
l5:    return j;
l7:
     }
```



CFA are an intermediate representation of the program where the control-flow is explicit

# From Control Flow Automata to Transition System



$$V := \{pc, i\}$$
$$I(V) := pc = l_0$$

$T(V, V') :=$

$(pc = l_0 \wedge i \geq 0 \wedge j \geq 0 \wedge pc' = l_1 \wedge i' = i \wedge j' = j) \vee$

$(pc = l_0 \wedge \neg(i \geq 0 \wedge j \geq 0) \wedge pc' = l_7 \wedge i' = i \wedge j' = j) \vee$

...

...

$(pc = l_4 \wedge i \geq 0 \wedge j \leq i \wedge pc' = error \wedge i' = i \wedge j' = j) \vee$

...

# From programs to TS - Considerations

- Need to represent the intended semantic of the program:
  - Integers in C are not $\mathbb{Z}$
  - Floating point types (e.g., float and double) are not $\mathbb{Q}$
  - In practice:
    - Pick the “right” abstraction, depending on the verification goal
    - Using $\mathbb{Z}$ works assuming there are no overflows (otherwise, need to use bit-vectors)
    - Using $\mathbb{Q}$ works to check algorithm logic (but ignores floating point issues!)

# From programs to TS - Considerations

- Need to represent the intended semantic of the program:
  - Integers in C are not $\mathbb{Z}$
  - Floating point types (e.g., float and double) are not $\mathbb{Q}$
  - In practice:
    - Pick the “right” abstraction, depending on the verification goal
    - Using $\mathbb{Z}$ works assuming there are no overflows (otherwise, need to use bit-vectors)
    - Using $\mathbb{Q}$ works to check algorithm logic (but ignores floating point issues!)
- What do we represent in the TS?
  - The source code?
  - The intermediate representation generated from the compiler? (e.g., optimizations)
  - Another issue: are the transformation from source code correct? (e.g., problem of certified compilers)

In the lab, you will have this translation

1. Program verification using SMT
   - Program verification
   - Infinite-state Transition Systems
   - Bounded Model Checking - Finding a violation
   - Proving safety

# Bounded Model Checking (BMC)

Bounded Model Checking - idea:

- Incomplete verification: can the program reach a violation in k steps?
- Idea: encode all the possible paths of length k that can reach a violation to the property

# Bounded Model Checking (BMC)

Bounded Model Checking - idea:

- Incomplete verification: can the program reach a violation in k steps?
- Idea: encode all the possible paths of length k that can reach a violation to the property

Some history:

- Started for model checking hardware systems (using SAT)
- Applied to software (still using SAT, so finite domains)
- Then extended to use SMT - more expressive (e.g., bit-vectors, integers, reals, . . . )
- "Enabler" of other verification techniques - also to prove safety (e.g., k-induction, interpolant-based verification, IC3, . . . )

# BMC Encoding

Input:

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$

# BMC Encoding

Input:

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$

Encode a path of length $k$ reaching a violation to the property $P$:

$$BMC_k(V) := I(V^0) \wedge \bigwedge_{i=1}^{k} T(V^{i-1}, V^i) \wedge \bigwedge_{i=0}^{k-1} P(V^i) \wedge \neg P(V^k)$$

Some notation:

- $V^i := \{v^i \mid v \in V\}$: copies of the variables $V$ ($k+1$ copies)
- $\phi(V^i)$: substitutes the variables $\phi$ in the formula $\phi(V)$

# BMC Encoding

Input:

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$

Encode a path of length $k$ reaching a violation to the property $P$:

$$BMC_k(V) := I(V^0) \wedge \bigwedge_{i=1}^{k} T(V^{i-1}, V^i) \wedge \bigwedge_{i=0}^{k-1} P(V^i) \wedge \neg P(V^k)$$

Some notation:

- $V^i := \{v^i \mid v \in V\}$: copies of the variables $V$ ($k+1$ copies)
- $\phi(V^i)$: substitutes the variables $\phi$ in the formula $\phi(V)$

We can check the satisfiability of the formula $BMC_k(V)$:

- If $BMC_k(V)$ is satisfiable, then there is a path of length $k-1$ that reach the violates $P$
- What if $BMC_k(V)$ is unsatisfiable?

# BMC Encoding

Input:

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$

Encode a path of length $k$ reaching a violation to the property $P$:

$$BMC_k(V) := I(V^0) \land \bigwedge_{i=1}^{k} T(V^{i-1}, V^i) \land \bigwedge_{i=0}^{k-1} P(V^i) \land \neg P(V^k)$$

Some notation:

- $V^i := \{v^i \mid v \in V\}$: copies of the variables $V$ ($k+1$ copies)
- $\phi(V^i)$: substitutes the variables $\phi$ in the formula $\phi(V)$

We can check the satisfiability of the formula $BMC_k(V)$:

- If $BMC_k(V)$ is satisfiable, then there is a path of length $k-1$ that reach the violates $P$
- What if $BMC_k(V)$ is unsatisfiable? We just know that there no paths of length $k$ can reach $\neg P$

# BMC Encoding - example

Infinite state counter:

- $V := \{i\}$
- $I(V) := i \leq 2$
- $T(V, V') := i' = i + 1$

Property:
$P := i < 5$

BMC encoding for a path of length 3:

$$\begin{aligned} i^0 \leq 2 \wedge \\ i^1 = i^0 + 1 \wedge i^2 = i^1 + 1 \wedge i^3 = i^2 + 1 \wedge \\ i^0 < 5 \wedge i^1 < 5 \wedge i^2 < 5 \wedge \\ \neg i^3 < 5 \end{aligned}$$

The encoding is satisfiable and the counter-example is the assigment:

$$i^0 = 2; i^1 = 3; i^2 = 4; i^3 = 5$$

# BMC - search for a counter-example

We can search for a path violating the property $P$ incrementally:

1. Start with $k = 0$
2. If $BMC_0(V)$ is satisfiable, return the counter-example
3. Otherwise, $k := k + 1$ and iterate.

# BMC - search for a counter-example

We can search for a path violating the property $P$ incrementally:

1. Start with $k = 0$
2. If $BMC_0(V)$ is satisfiable, return the counter-example
3. Otherwise, $k := k + 1$ and iterate.

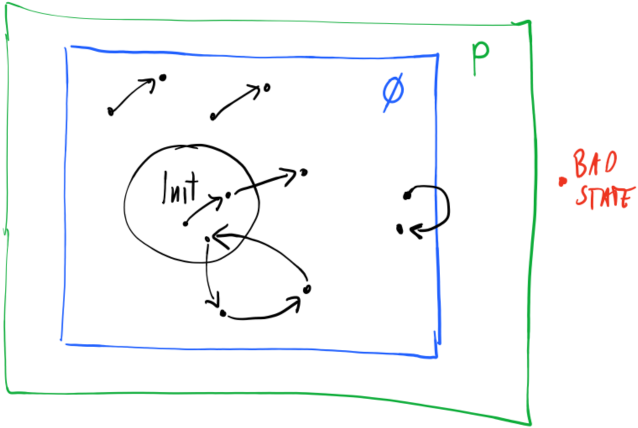
Different strategies are possible:

- Check for the existence of the bug “up to” length $k$
- Increment $k$ by different values (not just 1 every time)
- Use the SMT solver incrementality: most of the formula does not change from $k$ to $k + 1$ (i.e., all the $< k$ are still asserted).

1. Program verification using SMT
   - Program verification
   - Infinite-state Transition Systems
   - Bounded Model Checking - Finding a violation
   - Proving safety

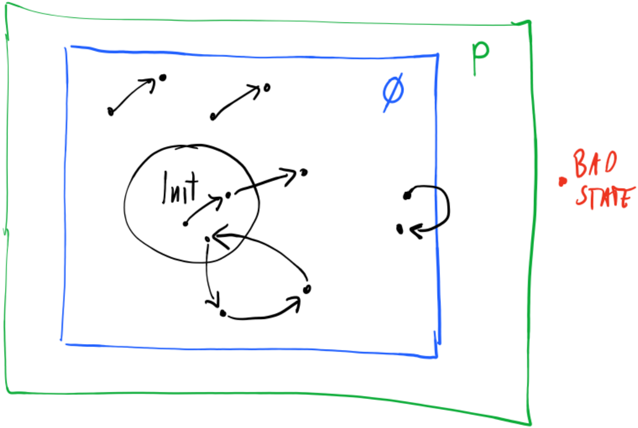
# Inductive invariant - intuition

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$



# Inductive invariant - intuition

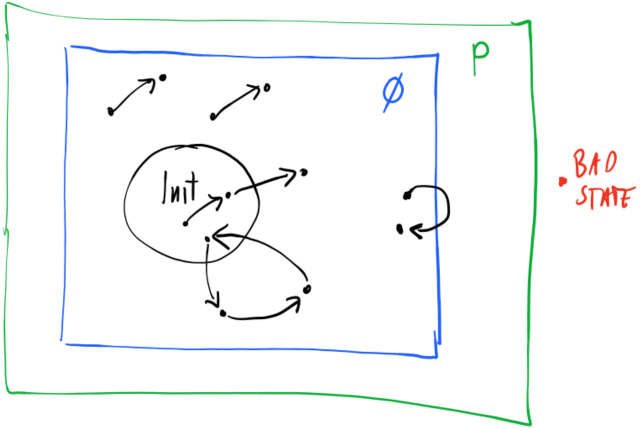
transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$



The set $\phi$ an Inductive invariant.

# Inductive invariant - intuition

transition system $S := (V, I(V), T(V, V'))$ safety property $P(V)$



The set $\phi$ an Inductive invariant.

How can we check if a set of states is an inductive invariant?

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \wedge T(V, V') \models P(V')$

- iff $I(V) \rightarrow \phi(V)$ is valid
- iff $\phi(V) \rightarrow P(V)$ is valid
- iff $(\phi(V) \wedge T(V, V')) \rightarrow P(V')$ is valid

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \wedge T(V, V') \models P(V')$

- iff $I(V) \rightarrow \phi(V)$ is valid
- iff $\phi(V) \rightarrow P(V)$ is valid
- iff $(\phi(V) \wedge T(V, V')) \rightarrow P(V')$ is valid

Check for validity: $\psi$ is valid iff $\neg\psi$ is unsatisfiable

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \wedge T(V, V') \models P(V')$

- iff $I(V) \rightarrow \phi(V)$ is valid
- iff $\phi(V) \rightarrow P(V)$ is valid
- iff $(\phi(V) \wedge T(V, V')) \rightarrow P(V')$ is valid

Check for validity: $\psi$ is valid iff $\neg\psi$ is unsatisfiable

What can we do:

- Check if a formula $\psi$ is an inductive invariant

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \land T(V, V') \models P(V')$

- iff $I(V) \to \phi(V)$ is valid
- iff $\phi(V) \to P(V)$ is valid
- iff $(\phi(V) \land T(V, V')) \to P(V')$ is valid

Check for validity: $\psi$ is valid iff $\neg\psi$ is unsatisfiable

What can we do:

- Check if a formula $\psi$ is an inductive invariant "Easy": satisfiability checks
- Find a formula $\psi$ that is an inductive invariant

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \wedge T(V, V') \models P(V')$

- iff $I(V) \rightarrow \phi(V)$ is valid
- iff $\phi(V) \rightarrow P(V)$ is valid
- iff $(\phi(V) \wedge T(V, V')) \rightarrow P(V')$ is valid

Check for validity: $\psi$ is valid iff $\neg\psi$ is unsatisfiable

What can we do:

- Check if a formula $\psi$ is an inductive invariant "Easy": satisfiability checks
- Find a formula $\psi$ that is an inductive invariant Difficult: need to find $\psi$

# Inductive Invariants with logic

An inductive invariant $\phi(V)$ is a formula such that:

- $I(V) \models \phi(V)$
- $\phi(V) \models P(V)$
- $\phi(V) \wedge T(V, V') \models P(V')$

- iff $I(V) \rightarrow \phi(V)$ is valid
- iff $\phi(V) \rightarrow P(V)$ is valid
- iff $(\phi(V) \wedge T(V, V')) \rightarrow P(V')$ is valid

Check for validity: $\psi$ is valid iff $\neg\psi$ is unsatisfiable

What can we do:

- Check if a formula $\psi$ is an inductive invariant "Easy": satisfiability checks
- Find a formula $\psi$ that is an inductive invariant Difficult: need to find $\psi$

Safety property verification *reduced to* findind an inductive invariant

# Inductive invariant - examples

Infinite state counter:

$$\begin{aligned} V :=& \{i\} \\ I(V) :=& i = 0 \\ T(V, V') :=& ((i < 5 \vee (i > 6 \wedge i \leq 10)) \rightarrow i' = i + 1) \wedge \\ & ((i = 5 \vee i = 6) \rightarrow i' = i) \\ P :=& i \leq 6 \end{aligned}$$

$i \leq 5$ is an inductive invariant:

- $i = 0 \models i \leq 5$
- $i \leq 5 \models i \leq 6$
- $i \leq 5 \wedge T(i, i') \models i' \leq 5$

# When induction fails

Infinite state counter:

$$\begin{aligned}
V :=& \{i\} \\
I(V) :=& i = 0 \\
T(V, V') :=& ((i < 5 \vee (i > 6 \wedge i \leq 10)) \rightarrow i' = i + 1) \wedge \\
& ((i = 5 \vee i = 6) \rightarrow i' = i) \\
P :=& i \leq 10
\end{aligned}$$

$i \leq 10$ is an invariant ($i \leq 5$ is still and inductive invariant), but it is not inductive:

- $i = 0 \models i \leq 10$
- $i \leq 10 \models i \leq 10$
- $i \leq 10 \wedge T(i, i') \models i' \leq 10$

# When induction fails

Infinite state counter:

$$\begin{aligned} V :=& \{i\} \\ I(V) :=& i = 0 \\ T(V, V') :=& ((i < 5 \vee (i > 6 \wedge i \leq 10)) \rightarrow i' = i + 1) \wedge \\ & ((i = 5 \vee i = 6) \rightarrow i' = i) \\ P :=& i \leq 10 \end{aligned}$$

$i \leq 10$ is an invariant ($i \leq 5$ is still and inductive invariant), but it is not inductive:

- $i = 0 \models i \leq 10$
- $i \leq 10 \models i \leq 10$
- $i \leq 10 \wedge T(i, i') \models i' \leq 10$ No!

# To sum up

What did we see today?

- Symbolic representation of infinite-state systems (like software) using FOL
- Find a counter-example to a safety property using BMC
- Prove that a property holds, using inductive invariants
- How can we represent programs as infinite-state transition systems (intuition)

# References I