Formal Specification and Verification

Deductive Verification: An introduction

10.07.2012

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Overview

- **Model checking:**

  Finite transition systems / CTL properties

  States are "entities" (no precise description, except for labelling functions)

  No precise description of actions (only $\longrightarrow$ important)

Overview

- **Model checking:**

  Finite transition systems / CTL properties

  States are "entities" (no precise description, except for labelling functions)

  No precise description of actions (only $\rightarrow$ important)

Extensions in two possible directions:

- More precise description of the actions/events
  - Propositional Dynamic Logic (last time)
  - Hoare logic (not discussed in this lecture)
- More precise description of states (and possibly also of actions)
  - succinct representation: formulae represent a set of states
  - deductive verification (today)

Transition systems (Reminder)

- Model to describe the behaviour of systems
- Digraphs where nodes represent states, and edges model transitions
- **State:** Examples
  - the current colour of a traffic light
  - the current values of all program variables $+$ the program counter
  - the current value of the registers together with the values of the input bits
- **Transition** ("state change"): Examples
  - a switch from one colour to another
  - the execution of a program statement
  - the change of the registers and output bits for a new input

Transition systems

Definition.

A transition system TS is a tuple $(S, Act, \longrightarrow, I, AP, L)$ where:

- S is a set of states
- Act is a set of actions
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ is a set of initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function

S and Act are either finite or countably infinite

Notation: $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \longrightarrow$.

Programs and transition systems

Program graph representation

Program graph representation

Some preliminaries

- typed variables with a valuation that assigns values in a fixed structure to variables
  - e.g., $\beta(x) = 17$ and $\beta(y) = -2$
- Boolean conditions: set of formulae over *Var*
  - propositional logic formulas whose propositions are of the form "$x \in D$"
  - $(-3 < x \leq 5) \wedge (y = green) \wedge (x \leq 2 * x')$
- effect of the actions is formalized by means of a mapping:

$$Effect : Act \times Eval(Var) \rightarrow Eval(Var)$$

  - e.g., $\alpha \equiv x := y + 5$ and evaluation $\beta(x) = 17$ and $\beta(y) = -2$
  - $Effect(\alpha, \beta)(x) = \beta(y) + 5 = 3$,
  - $Effect(\alpha, \beta)(y) = \beta(y) = -2$

Program graph representation

Program graphs

A program graph PG over set Var of typed variables is a tuple

$$(Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

where

- Loc is a set of locations with initial locations $Loc_0 \subseteq Loc$
- Act is a set of actions
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is the effect function
- $\rightarrow \ \subseteq \ Loc \times (\underbrace{Cond(Var)}_{\text{Boolean conditions on } Var} \times Act) \times Loc$, transition relation
- $g_0 \in Cond(Var)$ is the initial condition.

Notation: $l \overset{g:\alpha}{\rightarrow} l'$ denotes $(l, g, \alpha, l') \in \rightarrow$.

From program graphs to transition systems

- Basic strategy: unfolding
  - state = location (current control) l + data valuation β $\qquad (l, \beta)$
  - initial state = initial location + data valuation satisfying the initial condition g_0

- Propositions and labeling
  - propositions: "at l" and "$x \in D$" for $D \subseteq dom(x)$
  - $< l, \beta >$ is labeled with "at l" and all conditions that hold in β.

- $l \xrightarrow{g:\alpha} l'$ and g holds in β then $< l, \beta > \xrightarrow{\alpha} < l', Effect(< l, \beta >) >$

Transition systems for program graphs

The transition system $TS(PG)$ of program graph

$$PG = (Loc, Act, Effect, \rightarrow, Loc_0, g_0)$$

over set Var of variables is the tuple $(S, Act, \rightarrow, I, AP, L)$ where:

- $S = Loc \times Eval(Var)$
- $\rightarrow S \times Act \times S$ is defined by the rule:
  If $l \stackrel{g:\alpha}{\rightarrow} l'$ and $\beta \models g$ then $< l, \beta > \stackrel{\alpha}{\rightarrow} < l', Effect(< l, \beta >) >$
- $I = \{< l, \beta >| \ l \in \mathrm{Loc}_0, \beta \models g_0\}$
- $AP = Loc \cup Cond(Var)$ and
- $L(< l, \beta >) = \{l\} \cup \{g \in Cond(Var) \mid \beta \models g\}$.

Problem

Set of states: $S = Loc \times Eval(Var)$

$Eval(Var)$ can be very large
(some variables can have values in large data domains e.g. integers)

Therefore it is also difficult to concretely represent $\rightarrow$
(the relation usually very large as well)

Solution

Succinct representation of sets of states and of transitions between states

- Set of states: Formula (property of all states in the set)
- Transitions: Formulae (relation between the old values of the variables and the new values of the variables)

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

States:

(l, β), where l location and β assignment of values to the variables.

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

States:

(l, β), where l location and β assignment of values to the variables.

Idea: Take into account an additional variable pc (program counter), having as domain the set of locations.

State: assignment of values to the variables and to pc

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

States:

(l, β), where l location and β assignment of values to the variables.

Idea: Take into account an additional variable pc (program counter), having as domain the set of locations.

State: assignment of values to the variables and to pc

Set of states: Logical formula

Example:

$y \geq z$: The set of all states (l, β) for which $\beta(y) \geq \beta(z)$ (i.e. $\beta \models y \geq z$)

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
       x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Transition relation: $(l, \beta) \longrightarrow (l', \beta')$

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Transition relation: $(l, \beta) \longrightarrow (l', \beta')$

Expressed by logical formulae: Formula containing primed and unprimed variables.

Example:

- $\rho_1 = (move(l_1, l_2) \wedge y \geq z \wedge skip(x, y, z))$
- $\rho_2 = (move(l_2, l_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z))$
- $\rho_3 = (move(l_2, l_3) \wedge x \geq y \wedge skip(x, y, z))$
- $\rho_4 = (move(l_3, l_4) \wedge x \geq z \wedge skip(x, y, z))$
- $\rho_5 = (move(l_3; \; l_5) \wedge x + 1 \leq z \wedge skip(x, y, z))$

Abbreviations:

$move(l, l') := (pc = l \wedge pc' = l')$

$skip(v_1, \dots, v_n) := (v_1' = v_1 \wedge \dots \wedge v_n' = v_n)$

Programs as transition systems

Verification problem: Program + Description of the “bad” states

Succinct representation:

$$P = (Var, pc, Init, \mathcal{R}) \qquad\qquad \phi_{err}$$

- V - finite (ordered) set of program variables
- pc - program counter variable (pc included in V)
- $Init$ - initiation condition given by formula over V
- $\mathcal{R}$ - a finite set of transition relations

  Every transition relation $\rho \in \mathcal{R}$ is given by a formula over the variables V and their primed versions V'

- ϕ_{err} - an error condition given by a formula over V

States, sets and relations

- Each program variable x is assigned a domain of values D_x.
- Program state $=$ function that assigns each program variable a value from its respective domain
- $S =$ set of program states
- Formula with free variables in $V =$ set of program states
- Formula with free variables in V and $V' =$ binary relation over program states
  - First component of each pair refers to values of the variables V
  - Second component of the pair refers to values of the variables V' (typically the new variables of the variables in V after an instruction was executed)

States, sets and relations

- We identify formulas with the sets and relations that they represent
- We identify the entailment relation between formulas $\models$ with set inclusion
- We identify the satisfaction relation $\models$ between valuations and formulas, with the membership relation.

States, sets and relations

- We identify formulas with the sets and relations that they represent
- We identify the entailment relation between formulas $\models$ with set inclusion
- We identify the satisfaction relation $\models$ between valuations and formulas, with the membership relation.

Example:

- Formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- Formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1

States, sets and relations

- We identify formulas with the sets and relations that they represent
- We identify the entailment relation between formulas $\models$ with set inclusion
- We identify the satisfaction relation $\models$ between valuations and formulas, with the membership relation.

Example:

- Formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- Formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1
- If program state s assigns 1, 3, 2, and l_1 to program variables x, y, z, and pc, respectively, then $s \models y \geq z$

States, sets and relations

- We identify formulas with the sets and relations that they represent
- We identify the entailment relation between formulas $\models$ with set inclusion
- We identify the satisfaction relation $\models$ between valuations and formulas, with the membership relation.

Example:

- Formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- Formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1
- If program state s assigns 1, 3, 2, and l_1 to program variables x, y, z, and pc, respectively, then $s \models y \geq z$
- Logical consequence: $y \geq z \models y + 1 \models z$

Example Program

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
       x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Example program

- Program variables $V = (pc, x, y, z)$
- Program counter pc
- Program variables x, y, and z range over integers: $D_x = D_y = D_z = \mathrm{Int}$
  Program counter pc ranges over control locations: $D_{pc} = L$
- Set of control locations $L = \{l_1, l_2, l_3, l_4, l_5\}$
- Initiation condition $Init := (pc = l_1)$
- Error condition $\phi_{\mathrm{err}} := (pc = l_5)$
- Program transitions $\mathcal{R} = \{\rho_1, \ldots, \rho_5\}$, where:

$$\begin{aligned}
\rho_1 &= (move(l_1, l_2) \land y \geq z \land skip(x, y, z)) \\
\rho_2 &= (move(l_2, l_2) \land x + 1 \leq y \land x' = x + 1 \land skip(y, z)) \\
\rho_3 &= (move(l_2, l_3) \land x \geq y \land skip(x, y, z)) \\
\rho_4 &= (move(l_3, l_4) \land x \geq z \land skip(x, y, z)) \\
\rho_5 &= (move(l_3; \; l_5) \land x + 1 \leq z \land skip(x, y, z))
\end{aligned}$$

Initial state, error state, transition relation

- Each state that satisfies the initiation condition *Init* is called an initial state
- Each state that satisfies the error condition *err* is called an error state
- Program transition relation $\rho_{\mathcal{R}}$ is the union of the single-statement transition relations (formula representation: disjunction) i.e.,

$$\rho_{\mathcal{R}} = \bigvee_{\rho \in \mathcal{R}} \rho$$

- The state s has a transition to the state s' if the pair of states (s, s') lies in the program transition relation $\rho_{\mathcal{R}}$, i.e., if $(s, s') \models \rho_{\mathcal{R}}$:
  – $s : V \to \bigcup_{x \in V} D_x, \quad s(x) \in D_x$ for all $x \in V$
  – $s' : V' \to \bigcup_{x \in V} D_x, \quad s(x') \in D_x$ for all $x \in V$
  – $\beta : V \cup V' \bigcup_{x \in X} D_x$ defined for every $x \in V$ by $\beta(x) = s(x), \beta(x') = s'(x)$ has the property that $\beta \models \rho_{\mathcal{R}}$

Computation

A program computation is a sequence of states $s_1 s_2 \ldots$ such that:

- The first element is an initial state, i.e., $s_1 \models \mathit{Init}$
- Each pair of consecutive states (s_i, s_{i+1}) is connected by a program transition, i.e., $(s_i, s_{i+1}) \models \rho_{\mathcal{R}}$.
- If the sequence is finite then the last element does not have any successors i.e., if the last element is s_n, then there is no state s such that $(s_n, s) \models \rho_{\mathcal{R}}$.

Example Program

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
      x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Example of a computation:

$$(l_1, 1, 3, 2), (l_2, 1, 3, 2), (l_2, 2, 3, 2), (l_2, 3, 3, 2), (l_3, 3, 3, 2), (l_4, 3, 3, 2)$$

- sequence of transitions $\rho_1, \rho_2, \rho_2, \rho_3, \rho_4$
- state $=$ tuple of values of program variables pc, x, y, and z
- last program state does not any successors

Correctness: Safety

- a state is reachable if it occurs in some program computation
- a program is safe if no error state is reachable
- ... if and only if no error state lies in ϕ_{reach},

$$\phi_{\mathsf{err}} \wedge \phi_{\mathsf{reach}} \models \perp$$

  where ϕ_{reach} = set of program states which are reachable from some initial state

- ... if and only if no initial state lies in $\phi_{reach^{-1}}$,

$$\mathit{Init} \wedge \phi_{\mathsf{reach}^{-1}}(\phi_{\mathsf{err}}) \models \perp$$

  where $\phi_{\mathsf{reach}^{-1}}(\phi_{\mathsf{err}})$ = set of program states from which some state in ϕ_{err} is reachable

Example

```
1: if (y >= z) then skip else halt;
2: while (x < y) {
       x++;
   }
3: if (x >= z) then skip else goto 5;
4: exit
5: error
```

Set of reachable states:

$$
\begin{aligned}
\phi_{reach} \quad = \quad & (pc = l_1 \vee \\
& (pc = l_2 \wedge y \geq z) \vee \\
& (pc = l_3 \wedge y \geq z \wedge x \geq y) \vee \\
& (pc = l_4 \wedge y \geq z \wedge x \geq y)
\end{aligned}
$$

Post operator

Let ϕ be a formula over V

Let ρ be a formula over V and V'

Define a post-condition function *post* by:

$$post(\phi, \rho) = \exists V'' : \phi[V''/V] \wedge \rho[V''/V][V/V']$$

An application $post(\phi, \rho)$ computes the image of the set ϕ under the relation ρ.

Post operator

Let ϕ be a formula over V

Let ρ be a formula over V and V'

Define a post-condition function *post* by:

$$post(\phi, \rho) = \exists V'' : \phi[V''/V] \wedge \rho[V''/V][V/V']$$

An application $post(\phi, \rho)$ computes the image of the set ϕ under the relation ρ.

post distributes over disjunction wrt. each argument:

- $post(\phi, \rho_1 \vee \rho_2) = post(\phi, \rho_1) \vee post(\phi, \rho_2)$
- $post(\phi_1 \vee \phi_2, \rho) = post(\phi_1, \rho) \vee post(\phi_2, \rho)$

Application of post in example program

Set of states $\phi := (pc = l_2 \wedge y \geq z)$

Transition relation $\rho := \rho_2$

$$\rho_2 = (move(l_2, l_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z))$$

$$\begin{aligned}
post(\phi, \rho) &= \exists V''(pc = l_2 \wedge y \geq x)[V''/V] \wedge \rho_2[V''/V][V/V'] \\
&= \exists V''(pc'' = l_2 \wedge y'' \geq x'') \wedge \\
&\quad (pc'' = l_2 \wedge pc' = l_2 \wedge x'' + 1 \leq y'' \wedge x' = x'' + 1 \wedge y' = y'' \wedge z' = z'')[V/V'] \\
&= \exists V''(pc'' = l_2 \wedge y'' \geq x'') \wedge \\
&\quad (pc'' = l_2 \wedge pc = l_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \wedge y = y'' \wedge z = z'') \\
&= (pc = l_2 \wedge y \leq z \wedge x \leq y)
\end{aligned}$$

Application of post in example program

Set of states $\phi := (pc = l_2 \wedge y \geq z$

Transition relation $\rho := \rho_2$

$$\rho_2 = (move(l_2, l_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z))$$

$$\begin{aligned}
post(\phi, \rho) &= \exists V''(pc = l_2 \wedge y \geq x)[V''/V] \wedge \rho_2[V''/V][V/V'] \\
&= \exists V''(pc'' = l_2 \wedge y'' \geq x'') \wedge \\
&\quad (pc'' = l_2 \wedge pc' = l_2 \wedge x'' + 1 \leq y'' \wedge x' = x'' + 1 \wedge y' = y'' \wedge z' = z'')[V/V' \\
&= \exists V''(pc'' = l_2 \wedge y'' \geq x'') \wedge \\
&\quad (pc'' = l_2 \wedge pc = l_2 \wedge x'' + 1 \leq y'' \wedge x = x'' + 1 \wedge y = y'' \wedge z = z'') \\
&= (pc = l_2 \wedge y \leq z \wedge x \leq y)
\end{aligned}$$

[Renamed] program variables:
$V = (pc, x, y, z)$, $V' = (pc', x', y', z')$, $V'' = (pc'', x'', y'', z'')$

Iteration of post

$post^n(\phi, \rho) = n$-fold application of post to ϕ under ρ

$$post^n(\phi, \rho) = \begin{cases} \phi & \text{if } n = 0 \\ post(post^{n-1}(\phi, \rho)), \rho) & \text{otherwise} \end{cases}$$

Characterize ϕ_{reach} using iterates of post:

$$\begin{aligned} \phi_{\text{reach}} &= \text{Init} \lor post(Init, \rho_{\mathcal{R}}) \lor post(post(Init, \rho_{\mathcal{R}}), \rho_{\mathcal{R}}) \lor \dots \\ &= \bigvee_{i \geq 0} post^i(Init, \rho_{\mathcal{R}}) \end{aligned}$$

disjuncts = iterates for every natural number n ("ω-iteration")

Finite iteration post may suffice

Fixpoint reached in n steps if $\bigvee_{i=1}^{n} post^i(Init, \rho_{\mathcal{R}}) = \bigvee_{i=1}^{n+1} post^i(Init, \rho_{\mathcal{R}})$

Then $\bigvee_{i=1}^{n} post^i(Init, \rho_{\mathcal{R}}) = \bigvee_{i \geq 0} post^i(Init, \rho_{\mathcal{R}})$

Forward reachability analysis

Compute $\bigvee_{i=1}^{n} post^i(Init, \rho_{\mathcal{R}})$, $n \geq 0$.

If there exists $m \in \mathbb{N}$ such that

$$\bigvee_{i=1}^{n} post^i(Init, \rho_{\mathcal{R}}) = \bigvee_{i=1}^{n+1} post^i(Init, \rho_{\mathcal{R}})$$

then fixpoint reached.

Let $\phi_{\text{reach}} := \bigvee_{i=1}^{n} post^i(Init, \rho_{\mathcal{R}})$

If $\phi_{\text{reach}} \cap \phi_{\text{err}} = \varnothing$ then safety is guaranteed.

Backward reachability analysis

Another possibility: Start from a bad state and compute states from which the bad state can be reached.

If the initial states are not among these states then safety is guaranteed.

Pre operator

Let ϕ be a formula over V

Let ρ be a formula over V and V'

Define a pre-condition function *pre* by:

$$\mathit{pre}(\phi, \rho) = \exists V' : \rho \wedge \phi[V'/V]$$

An application $\mathit{pre}(\phi, \rho)$ computes the preimage of the set ϕ under the relation ρ.

Computation of pre^n similar.

Problems

It is not guaranteed that the fixpoint is reached in a finite/bounded number of steps.

Problems

It is not guaranteed that the fixpoint is reached in a finite/bounded number of steps.

Need to analyze alternative solutions

Verification

Modeling/Formalization

System Specification

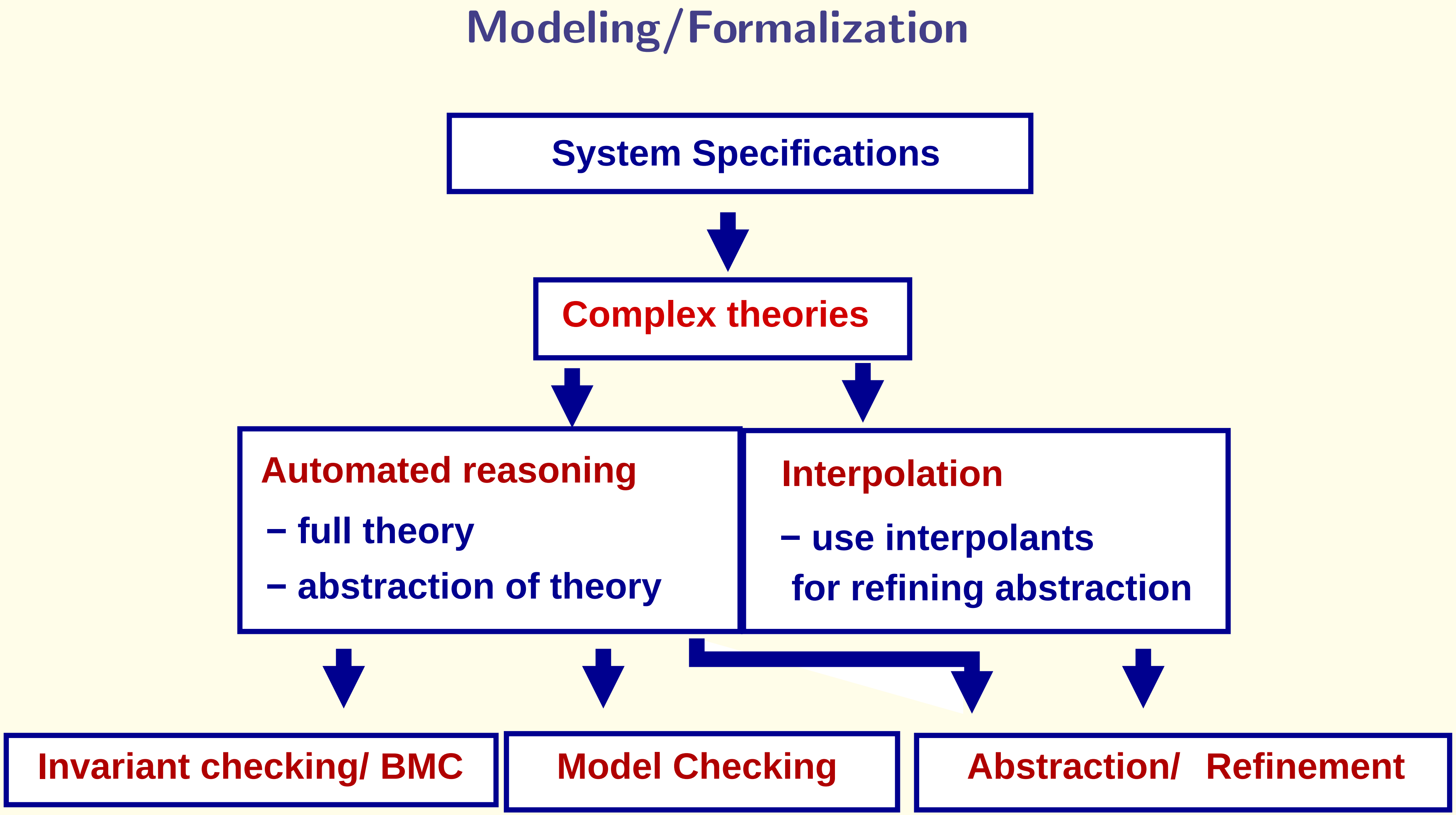
Is the system safe?

Is safety guaranteed on all paths
of length < n which start in an initial state?

Is the safety property an invariant of the system?
Can we generate an invariant which implies safety?

Invariant checking/ BMC	Model Checking	Abstraction/ Refinement

Verification



Invariant checking; Bounded model checking

S specification $\mapsto \Sigma_S$ signature of S; $\mathcal{T}_S$ theory of S; T_S transition system

$\text{Init}(\overline{x})$; $\rho_{\mathcal{R}}(\overline{x}, \overline{x}')$

Given: $\text{Safe}(x)$ formula (e.g. safety property)

- **Invariant checking**

  (1) $\models_{\mathcal{T}_S} \text{Init}(\overline{x}) \rightarrow \text{Safe}(\overline{x})$ (Safe holds in the initial state)

  (2) $\models_{\mathcal{T}_S} \text{Safe}(\overline{x}) \wedge \rho_{\mathcal{R}}(\overline{x}, \overline{x}') \rightarrow \text{Safe}(\overline{x}')$ (Safe holds before $\Rightarrow$ holds after update)

- **Bounded model checking (BMC):**

  Check whether, for a fixed k, unsafe states are reachable in at most k steps, i.e. for all $0 \leq j \leq k$:

$$\text{Init}(x_0) \wedge \rho_{\mathcal{R}}(x_0, x_1) \wedge \cdots \wedge \rho_{\mathcal{R}}(x_{j-1}, x_j) \wedge \neg\text{Safe}(x_j) \models_{\mathcal{T}_S} \perp$$

Reasoning modulo theories

Goal: Devise efficient methods for reasoning modulo theories

Problems

– First order logic is undecidable
– In applications, theories do not occur alone
$\mapsto$ need to consider combinations of theories

+ Fragments of theories occurring in applications are often decidable
+ Often provers for the component theories can be combined efficiently

Probleme

- First order logic is undecidable
- In applications, theories do not occur alone
  $\mapsto$ need to consider combinations of theories

+ Fragments of theories occurring in applications are often decidable
+ Often provers for the component theories can be combined efficiently

Important goals:

- Identify decidable theories which are important in applications (Extensions/Combinations) possibly with low complexity
- Development & Implementation of efficient Decision Procedures

Reasoning modulo theories

Goal: Devise efficient methods for reasoning modulo theories

SAT checking (can reduce entailment to checking satisfiability)

Example:

Check whether conjunctions of constraints in linear arithmetic is satisfiable: classical methods exist, e.g. simplex.

Check whether a conjunction of equalities and disequalities of ground terms is satisfiable: methods exist (e.g. congruence closure)

Challenge: efficient methods for handling arbitrary Boolean combinations of constraints in such theories.

Possible solution: Extend the DPLL method to reasoning modulo theories.

Reminder: The DPLL algorithm

State: $M \mid\mid F$,

where:

- M partial assignment (sequence of literals),

  some literals are annotated (L^d: decision literal)

- F clause set.

A succinct formulation

UnitPropagation

$M \parallel F, C \vee L \Rightarrow M, L \parallel F, C \vee L$ if $M \models \neg C$, and L undef. in M

Decide

$M \parallel F \Rightarrow M, L^d \parallel F$ if L or $\neg L$ occurs in F, L undef. in M

Fail

$M \parallel F, C \Rightarrow$ Fail if $M \models \neg C$, M contains no decision literals

Backjump

$$M, L^d, N \parallel F \Rightarrow M, L' \parallel F \quad \text{if} \begin{cases} \text{there is some clause } C \vee L' \text{ s.t.:} \\ F \models C \vee L', M \models \neg C, \\ L' \text{ undefined in } M \\ L' \text{ or } \neg L' \text{ occurs in } F. \end{cases}$$

SAT Modulo Theories (SMT)

Some problems are more naturally expressed in richer logics than just propositional logic, e.g:

- Software/Hardware verification needs reasoning about equality, arithmetic, data structures, ...

SMT consists of deciding the satisfiability of a ground 1st-order formula with respect to a background theory T

SAT Modulo Theories (SMT)

The "very eager" approach to SMT

Method:

– translate problem into equisatisfiable propositional formula;

– use off-the-shelf SAT solver

- Why "eager"?

  Search uses all theory information from the beginning

- Characteristics:

  + Can use best available SAT solver

  – Sophisticated encodings are needed for each theory

  – Sometimes translation and/or solving too slow

Main Challenge for alternative approaches is to combine:

- DPLL-based techniques for handling the boolean structure

- Efficient theory solvers for conjunctions of $\mathcal{T}$-literals

SAT Modulo Theories (SMT)

"Lazy" approaches to SMT: Idea

Example: consider $\mathcal{T} = UIF$ and the following set of clauses:

$$\underbrace{f(g(a)) \not\approx f(c)}_{\neg P_1} \vee \underbrace{g(a) \approx d}_{P_2}, \quad \underbrace{g(a) \approx c}_{P_3}, \quad \underbrace{c \not\approx d}_{\neg P_4}$$

1. Send $\{\neg P_1 \vee P_2,\ P_3,\ \neg P_4\}$ to SAT solver

   SAT solver returns model $[\neg P_1, P_3, \neg P_4]$
   Theory solver says $\neg P_1 \wedge P_3 \wedge \neg P_4$ is $\mathcal{T}$-inconsistent

2. Send $\{\neg P_1 \vee P_2,\ P_3,\ \neg P_4,\ P_1 \vee \neg P_3 \vee P_4\}$ to SAT solver

   SAT solver returns model $[P_1, P_2, P_3, \neg P_4]$
   Theory solver says $P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4$ is $\mathcal{T}$-inconsistent

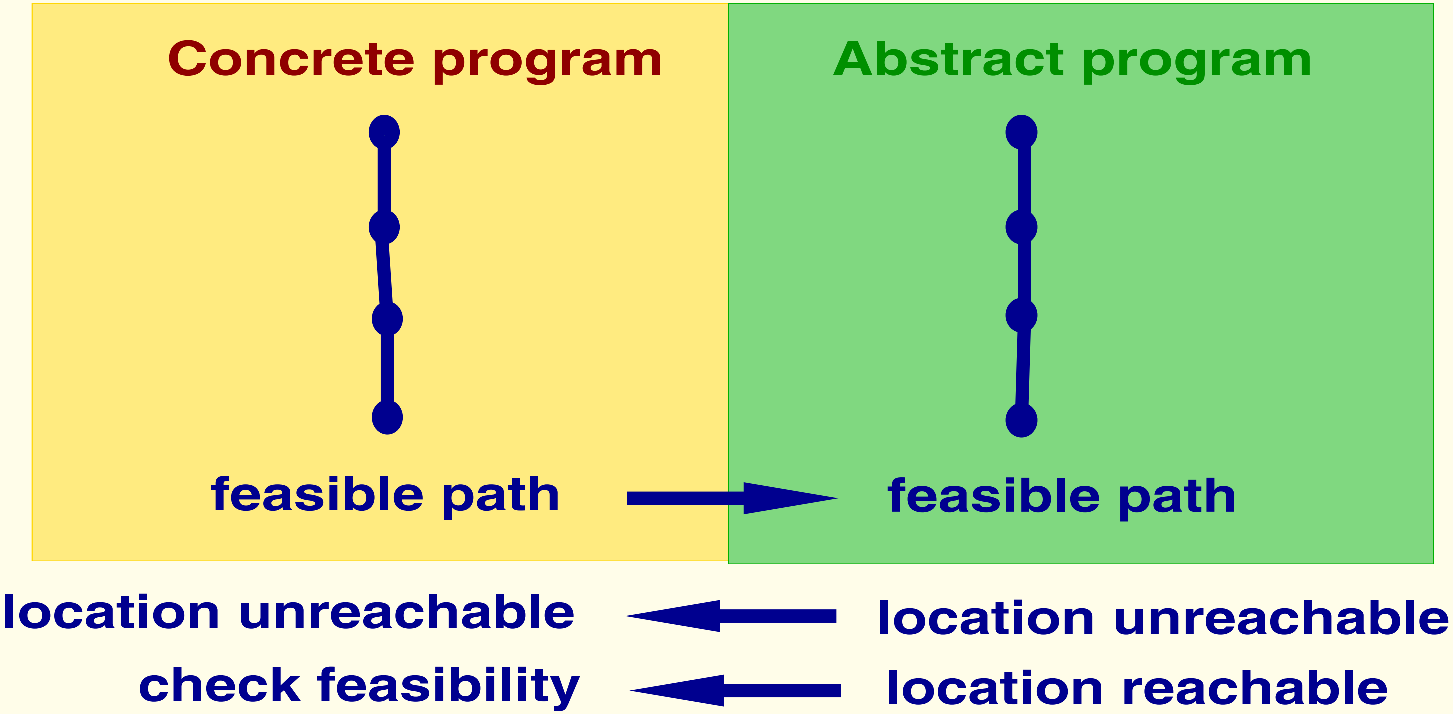
3. Send $\{\neg P_1 \vee P_2, P_3, \neg P_4, P_1 \vee \neg P_3 \vee P_4, \neg P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4\}$ to SAT solver

   SAT solver says UNSAT

Other interesting topics

- Generate invariants
- Verification by abstraction/refinement

Abstraction-based Verification



$\Downarrow$

conjunction of constraints: $Init(x_1) \wedge \rho_{\mathcal{R}}(x_1, x_2) \wedge \cdots \wedge \rho_{\mathcal{R}}(x_{n-1}, x_n) \wedge \neg \mathsf{safe}(x_n)$

- satisfiable: feasible path
- unsatisfiable: refine abstract program s.t. the path is not feasible
  [McMillan 2003-2006] use ‘local causes of inconsistency’
  $\mapsto$ compute interpolants

Overview

- **Basic notions**
  - Propositional logic (Methods for checking validity, satisfiability, entailment: Inference Systems, The Resolution Procedure, Sequent calculi, DPLL, BDDs, OBDDs)
  - First-order logic (Syntax, semantics, Logical theories, Herbrand models, term algebras, free algebras)
- **Specification**
  - Algebraic specification; Transition systems; Program graph representation
- **Verification**
  - LTL, CTL, Model checking
  - Propositional dynamic logic
  - Deductive verification: An introduction