Invariants in verification

ANR Codys

Peter Habermehl IRIF

21 november 2018

Overview

- What is an invariant ?
- What are the major issues concerning invariants in verification ?
  - How to define invariants ?
  - How to verify invariants ?
  - How to obtain invariants ?

What is an invariant ?

- C : a set of possible configurations (of a system)
- R $\subseteq$ C x C: a transition relation
- For S $\subseteq$ C:

  R(S) := {s' : there exists s in S s.t. (s,s') $\in$ R}
- Init $\subseteq$ C: a set of initial configurations
- Reach(Init) := R*(Init) := Init $\cup$ R(Init) $\cup$

  R(R(Init))….
- Inv $\subseteq$ C is an invariant, iff Reach $\subseteq$ Inv

Example

1: x = 3

2: while x > 0 :

3:      x = 2*x – 1

- A program has configurations
- Here: a configuration is a pair (pc,x)
- A (Loop) Invariant : x > 0
- This invariant is not inductive.

Inductive Invariant

- Init $\subseteq$ Inv
- R(Inv) $\subseteq$ Inv

```
x = 3
while x > 0:
    x = 2*x-1
```

- Inductive invariant: x > 1

What is a « best » invariant ?

- A smallest invariant wrt. $\subseteq$
- Reach is an inductive invariant
  - Init $\subseteq$ Reach
  - R(Reach) $\subseteq$ Reach
- Of course Reach might be very « complicated »
- Having an invariant which is sufficiently strong to prove a property is enough.
- We consider safety properties: Nothing bad is ever going to happen, systems always stay save, never violates an assertion,…
- Includes non-termination but not termination

Example

x = 3

while x > 0 :

    x = 2*x – 1

    assert(x is odd)

- Invariant x > 1 is not sufficiently strong to prove that the assertion always holds
- Another Invariant: {x in Nat: x > 2 and x is odd}
- Reach is not needed here.

Major issues

- How to define invariants ?
- How to verify invariants ?
- How to obtain (compute) invariants ?

How to define invariants ?

- Depends on the program (system) on hand
- Programs with
  - Integer variables
  - Floats
  - Arrays
  - Dynamic data structures (Lists, Trees, etc.)
  - Concurrency
  - …
- Suitable logics to describe sets of configurations
  - Hoare-style verification, Variants of predicate logic

Logical formalisms to describe invariants

- For basic data structures : predicate logic (Floyd-Hoare)
- Integer (rational) variables :
  - Intervals (with congruence information)
  - Convex polyhedra
  - Linear inequalities
  - ….
- Dynamic data structures : Separation logic
  - allows to reason about the heap (shape invariants, shape analysis,...)

How to verify invariants ?

- An invariant Inv is typically just a formula in some logic describing a set of configurations
- Inv $\subseteq$ C
- Init $\subseteq$ Inv
- R(Inv) $\subseteq$ Inv
- The logical formalism should be able to express these properties
- Expressing Reach ?
- Decidability of these properties is a bonus
- Tradeoff between expressibility and decidability

How to get invariants

- Ask student(s) to find invariants
- For special cases, compute all invariants
- Abstraction techniques (Abstract interpretation)
- Learning methods

Compute special invariants

- For simple programs
  - Example (related to the Kannan-Lipton orbit problem :

    $\mathbf{x} = \mathbf{x_0}$ **;** while True : $\mathbf{x} = \mathbf{Ax} + \mathbf{b}$ **;** assert ($\mathbf{x} \neq \mathbf{y}$)
- Fix a set Invs of potential invariants
  - Convex polyhedra, linear inequalities, polynomials,...
  - might be given by a scheme
- Does there always exist an invariant from Invs showing the property ?
- Compute the set of invariants I $\subseteq$ Invs
- Compute the best (strongest) invariant(s) in Invs

How to get invariants ? Abstract interpretation

- Fix a « simple » abstract domain
- Compute an abstract fixpoint overapproximating Reach
- Example :

  x=3   [3..3]

  while x > 0 :

  x = x*2 – 1   [3..5] → [3..∞]   (« widening »)

How to get invariants ?

- A wide range of abstract domains
- Abstract interpretation can be combined with CEGAR (Counter-example guided abstraction refinement)
  - Start with an abstraction
  - Compute invariant
  - If too coarse (not strong enough) to prove property, one gets (a) counter-example(s)
  - Refine abstraction and restart

Learning invariants

- The method implemented in DAIKON
- Take a set Invs of potential invariants
- Run the program and throw out successively candidates which are revealed to be not invariants

Learning Invariants (2)

- Use of classical machine learning techniques
- Learning from examples
  - Learner wants to infer a description of a set from a teacher
  - Teacher produces examples (positive or negative) or (stronger) Learner can explicitly ask if some elements are in the set
  - Learner hypothesizes a description of the set
  - Teacher validates or gives counterexample

Learning invariants (3)

- Teacher can produce positive and negative examples easily
- In the context of invariants, what is a counterexample ?
- Hyp is not inductive :
  - x in Hyp, but R(x) not in Hyp
- Has lead to a new framework :

  ICE (Implication counter-examples) learning

Transition invariants

- Generalisation to transition relations
- $R^* := R \cup R \circ R \cup R \circ R \circ R$ ....
- Transition invariant : $R^* \subseteq Tinv \subseteq C \times C$
- Typically more difficult to obtain than an invariant
- Allows for example to prove termination (if Tinv satisfies some properties)

Is an invariant really an invariant ?

- Real life is not a model
- One has to be careful with theoretical invariants which are not correct on a machine