Deriving Invariants by Algorithmic Learning, Decision Procedures, and Predicate Abstraction

Bow-Yaw Wang

ROSAEC Center, Seoul National University
Institute of Information Science, Academia Sinica

September 5, 2009

And the Real Title is...

What We did This Summer

Prof. Yi[♯], Yungbum[♯], Soonho[♯], and Bow-Yaw[♯,♭]

ROSAEC Center, Seoul National University[♯]
Institute of Information Science, Academia Sinica[♭]

September 5, 2009

Invariant Generation

- Consider the following program:

$$\{i = 0\}$$

```
while i < 10 do
    ret := nondet;
    if ret then i := i + 1
end
```

$$\{i = 10 \ \wedge \ ret\}$$

- $i = 0$ is the precondition.
- $i = 10 \wedge ret$ is the postcondition.
- An invariant is a property preserved for each iteration. Moreover, it allows us to prove the postcondition.
  - $(i = 10 \wedge ret) \vee i < 10$
  - $(i = 10 \wedge ret) \vee i < 10 \vee i = 0$

Invariant Generation

- Consider the following program:

$$\{i = 0\}$$
$$\textbf{while } i < 10 \textbf{ do}$$
$$\quad ret := \textbf{nondet};$$
$$\quad \textbf{if } ret \textbf{ then } i := i + 1$$
$$\textbf{end}$$
$$\{i = 10 \ \wedge \ ret\}$$

- $i = 0$ is the precondition.
- $i = 10 \wedge ret$ is the postcondition.
- An invariant is a property preserved for each iteration. Moreover, it allows us to prove the postcondition.
  - $(i = 10 \wedge ret) \vee i < 10$
  - $(i = 10 \wedge ret) \vee i < 10 \vee i = 0$

Invariant Generation

- Consider the following program:

$$
\begin{array}{l}
\{i = 0\} \\
\textbf{while } i < 10 \textbf{ do} \\
\quad ret := \textbf{nondet}; \\
\quad \textbf{if } ret \textbf{ then } i := i + 1 \\
\textbf{end} \\
\{i = 10 \ \wedge \ ret\}
\end{array}
$$

- $i = 0$ is the precondition.
- $i = 10 \wedge ret$ is the postcondition.
- An invariant is a property preserved for each iteration. Moreover, it allows us to prove the postcondition.
  - $(i = 10 \wedge ret) \vee i < 10$
  - $(i = 10 \wedge ret) \vee i < 10 \vee i = 0$

Invariant Generation

- Consider the following program:

```
{ phase = false ∧ success = false ∧ give_up = false ∧ cutoff = 0 ∧ count = 0 }
while ¬(success ∨ give_up) do
   entered_phase := false;
   if ¬phase then
      if cutoff = 0 then cutoff := 1;
      else if cutoff = 1 ∧ maxcost > 1 then cutoff := maxcost;
           else phase := true; entered_phase := true; cutoff := 1000;
      if cutoff = maxcost ∧ ¬search then give_up := true;
   else
      count := count + 1;
      if count > words then give_up := true;
   if entered_phase then count := 1;
   linkages := nondet;
   if linkages > 5000 then linkages := 5000;
   canonical := 0; valid := 0;
   if linkages ≠ 0 then
      valid := nondet; assume 0 ≤ valid ∧ valid ≤ linkages;
      canonical := linkages;
   if valid > 0 then success := true;
end
{ (valid > 0 ∨ count > words ∨ (cutoff = maxcost ∧ ¬search)) ∧ valid ≤ linkages∧
  canonical = linkages ∧ linkages ≤ 5000 }
```

- How much time would you like to invest on it?

Invariant Generation

- Consider the following program:

$\{ \, phase = \text{false} \wedge success = \text{false} \wedge give_up = \text{false} \wedge cutoff = 0 \wedge count = 0 \, \}$

```
while ¬(success ∨ give_up) do
  entered_phase := false;
  if ¬phase then
    if cutoff = 0 then cutoff := 1;
    else if cutoff = 1 ∧ maxcost > 1 then cutoff := maxcost;
         else phase := true; entered_phase := true; cutoff := 1000;
    if cutoff = maxcost ∧ ¬search then give_up := true;
  else
    count := count + 1;
    if count > words then give_up := true;
  if entered_phase then count := 1;
  linkages := nondet;
  if linkages > 5000 then linkages := 5000;
  canonical := 0; valid := 0;
  if linkages ≠ 0 then
    valid := nondet; assume 0 ≤ valid ∧ valid ≤ linkages;
    canonical := linkages;
  if valid > 0 then success := true;
end
```

$\{ \, (valid > 0 \vee count > words \vee (cutoff = maxcost \wedge \neg search)) \wedge valid \leq linkages \wedge canonical = linkages \wedge linkages \leq 5000 \, \}$

- How much time would you like to invest on it?

Motivation

- Generating invariants is hard.
  - However the solution is not unique.
- Machine learning has been applied to assumption generation.
  - It generates contextual assumptions for concurrent systems.
  - Please refer to my talk 3 months ago. (Time does fly fast!)
- Can we apply machine learning in invariant generation?

Outline

1. Overview
2. Learning Boolean Formulae
3. Resolving Queries
4. Experiments
5. Conclusions

What is an Invariant?

- Consider the statement $\{\delta\}$ **while** ρ **do** S **end** $\{\epsilon\}$.
- An invariant ι is a formula such that
  - $\delta \wedge \rho \Rightarrow \iota$ (ι holds when entering the loop);
  - $\iota \wedge \rho \Rightarrow Pre(\iota, S)$ (ι holds at each iteration); and
  - $\iota \wedge \neg\rho \Rightarrow \epsilon$ (ι gives ϵ after leaving the loop).
- Recall the simple example where $\iota : (i = 10 \wedge ret) \vee i < 10$:

$\{i = 0\}$	
while $i < 10$ **do**	$i = 0 \wedge i < 10 \Rightarrow \iota$
$ret :=$ **nondet**;	$\iota \wedge i < 10 \Rightarrow Pre(\iota, S)$
if ret **then** $i := i + 1$	$\iota \wedge \neg(i < 10) \Rightarrow i = 10 \wedge ret$
end	
$\{i = 10 \ \wedge \ ret\}$	

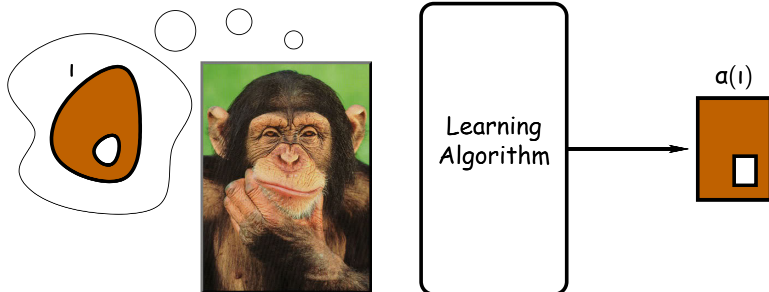
Propositional Formulae as Invariants

- A simple form of invariants is propositional (or quantifier-free) formula.
- Examples of propositional formulae:
  - $(i = 10 \land ret) \lor i < 10$
  - $(m = 0 \lor m > 12) \land 4x + 5y = n - m$
- The basic forms of propositional formulae are called atomic propositions.
- Examples of atomic propositions:
  - $i = 10$, ret, $i < 10$
  - $m = 0$, $m > 12$, $4x + 5y = n - m$

Predicate Abstraction

- If we see atomic propositions as Boolean variables, a propositional formula becomes a Boolean formula
  - $(i = 10 \wedge ret) \vee i < 10$ becomes $(b_{i=10} \wedge b_{ret}) \vee b_{i<10}$
  - $(m = 0 \vee m > 12) \wedge 4x + 5y = n - m$ becomes $(b_{m=0} \vee b_{m>12}) \wedge b_{4x+5y=n-m}$
- This is called predicate abstraction.
- Note that a Boolean formula can be transformed back to a propositional formula as well.

Framework Overview

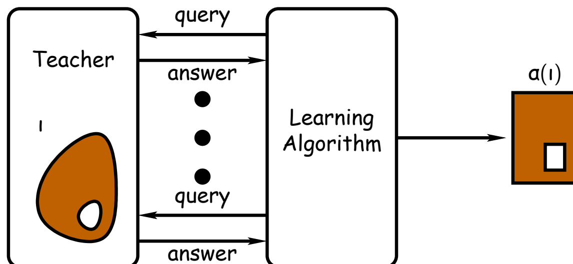


- Suppose ι is an invariant.
- Let α(ι) be the Boolean formula corresponding to ι by predicate abstraction.
- We apply a learning algorithm to learn the Boolean formula α(ι).

The CDNF Algorithm

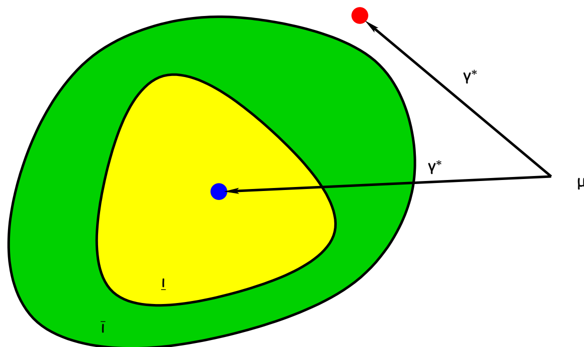
- The CDNF algorithm is an exact learning algorithm for an arbitrary Boolean formula λ.
- The algorithm makes two types of queries.
- A membership query $MEM(\mu)$ asks if the truth assignment μ satisfies λ. If yes, the teacher returns *YES*. Otherwise, the teacher returns *NO*.
  - For instance, let $\lambda = (b_0 \wedge \neg b_1) \vee (\neg b_0 \wedge b_1)$.
  - $MEM(\mu) \to YES$ when $\mu(b_0 b_1) = 01$.
  - $MEM(\mu) \to NO$ when $\mu(b_0 b_1) = 00$.
- An equivalence query $EQ(\beta)$ asks if the Boolean formula β is equivalent to λ. If not, the teacher returns a truth assignment as a counterexample.
  - $EQ((b_0 \vee b_1) \wedge (\neg b_0 \vee \neg b_1)) \to YES$.
  - $EQ(b_0 \vee b_1) \to \mu$ where $\mu(b_0 b_1) = 11$.

Learning Invariants



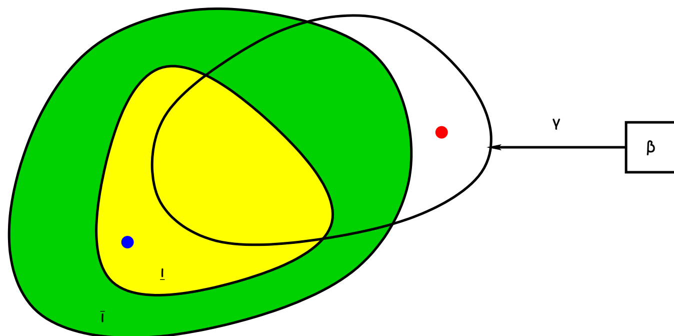
- Suppose ι is an invariant and $\alpha(\iota)$ its corresponding Boolean formula.
- To learn $\alpha(\iota)$, we only need to play the role of a teacher.
- The CDNF algorithm will generate a Boolean formula β equivalent to $\alpha(\iota)$.
- The propositional formula corresponding to β is an invariant.
- It remains to answer queries!

Answering Membership Queries



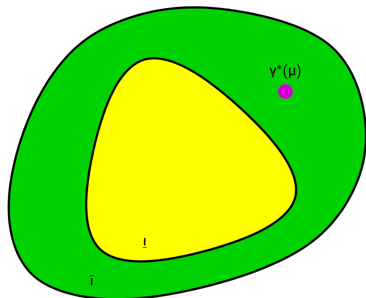
- Invariants are not known, but their approximations can be computed from pre- and post-conditions.
- $\underline{\imath}$ and $\bar{\imath}$ are invariant approximations where $\underline{\imath} \Rightarrow \imath$ and $\imath \Rightarrow \bar{\imath}$.
- In a membership query $MEM(\mu)$, we want to know if the truth assignment μ satisfies $\alpha(\imath)$.

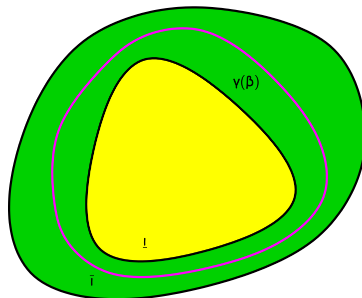
Answering Equivalence Queries



- In an equivalence query $EQ(\beta)$, we want to know if β is equivalent to $\alpha(\iota)$.
- If its corresponding propositional formula $\gamma(\beta)$ is an invariant, return *YES*.
- Otherwise, return a counterexample by comparing $\underline{\iota}$ and $\bar{\iota}$.

Unresolvable Cases





- For unresolvable membership queries, give a random answer.
- For unresolvable equivalence queries, restart.

Coin Tossing?

- Algorithmic learning tries to generate an invariant consistent with query answers.
  - If there is one, the learning algorithm will find it.
- In other words, if there is an invariant consistent with our random answers, we are going to find it eventually!
- This works when there are many invariants.
  - A few random answers cannot rule out all invariants.
- When the algorithm terminates, it always outputs an invariant.
  - On the other hand, it may not terminate if it cannot find an invariant.
- This is called a Las Vegas algorithm.
  - A Monte Carlo algorithm always terminates but may err with a bounded probability.

Experiments

- Consider the following loop from Linux USB driver:

$$
\begin{array}{l}
\{\ \mathit{locked} \wedge i = 0\ \} \\
\textbf{while}\ i < \mathit{entries} \wedge \mathit{status} = 0\ \textbf{do} \\
\quad \mathit{retval} := \textbf{nondet}; \\
\quad \mathit{locked} := \mathrm{false}; \\
\quad \textbf{switch}\ \mathit{retval}\ \textbf{do} \\
\quad\quad \textbf{case ENXIO}:\ \mathit{retval} := 0; \\
\quad\quad \textbf{case EAGAIN}:\ \mathit{retval} := 0; \\
\quad\quad \textbf{case ENOMEM}:\ \mathit{retval} := 0; \\
\quad\quad \textbf{case}\ 0:\ i := i + 1; \\
\quad \textbf{end}; \\
\quad \mathit{locked} := \mathrm{true}; \\
\quad \textbf{if}\ \mathit{retval} \neq 0 \wedge (\mathit{status} = 0 \vee \mathit{status} = \textbf{ECONNRESET})\ \textbf{then} \\
\quad\quad \mathit{status} := \mathit{retval}; \\
\textbf{end} \\
\{\ \mathit{locked} \wedge (i \neq 0 \Rightarrow \mathit{status} = \mathit{retval})\ \}
\end{array}
$$

Experiments

- This is one of the found invariants:

$$
\begin{array}{l}
(\mathit{locked} \wedge i = 0) \vee \\
(\mathit{locked} \wedge \mathit{retval} = 0 \wedge \mathit{status} = \mathit{retval}) \vee \\
(\mathit{locked} \wedge \mathit{status} \neq 0 \wedge \mathit{status} = \mathit{retval})
\end{array}
$$

- For 500 runs of this case, our algorithm generates an invariant in 0.200 seconds on average.
  - 7.0 random answers are needed on average.
  - The algorithm never restarts!

Experiments

- This is extracted from PARSER of SPEC2000.
- How much time would you spend on this program (again)?

```
{ phase = false ∧ success = false ∧ give_up = false ∧ cutoff = 0 ∧ count = 0 }
while ¬(success ∨ give_up) do
   entered_phase := false;
   if ¬phase then
      if cutoff = 0 then cutoff := 1;
      else if cutoff = 1 ∧ maxcost > 1 then cutoff := maxcost;
           else phase := true; entered_phase := true; cutoff := 1000;
      if cutoff = maxcost ∧ ¬search then give_up := true;
   else
      count := count + 1;
      if count > words then give_up := true;
   if entered_phase then count := 1;
   linkages := nondet;
   if linkages > 5000 then linkages := 5000;
   canonical := 0; valid := 0;
   if linkages ≠ 0 then
      valid := nondet; assume 0 ≤ valid ∧ valid ≤ linkages;
      canonical := linkages;
   if valid > 0 then success := true;
end
{ (valid > 0 ∨ count > words ∨ (cutoff = maxcost ∧ ¬search)) ∧ valid ≤ linkages∧
  canonical = linkages ∧ linkages ≤ 5000 }
```

Experiments

- Here is an invariant:

$$\begin{array}{l} \mathit{success} \Rightarrow (\mathit{valid} \neq 0 \wedge \mathit{canonical} \neq 0 \wedge \mathit{valid} \leq \mathit{linkages} \wedge \mathit{linkages} \leq 5000 \wedge \mathit{canonical} = \mathit{linkages}) \bigwedge \\ \mathit{give_up} \Rightarrow (\mathit{valid} \neq 0 \vee \neg \mathit{search} \vee \mathit{count} > \mathit{words}) \bigwedge \\ \mathit{give_up} \Rightarrow (\mathit{valid} \neq 0 \vee \mathit{count} > \mathit{words} \vee \mathit{cutoff} = \mathit{maxcost}) \bigwedge \\ \mathit{give_up} \Rightarrow (\mathit{canonical} \neq 0 \wedge \mathit{valid} \leq \mathit{linkages} \wedge \mathit{linkages} \leq 5000 \wedge \mathit{canonical} = \mathit{linkages}) \vee \\ \qquad (\mathit{valid} = 0 \wedge \mathit{linkages} = 0 \wedge \mathit{canonical} = \mathit{linkages}) \end{array}$$

- On average, it takes 31.196 seconds with 1058.5 random answers to get an invariant.
  - About half a minute is needed on average.
    - About 9 out of 10 times, an invariant can be generated within 2 minutes.

What's Next? Polynomial Invariants

- There is an extact learning algorithm for multivariate polynomials.
  - Assume $f(\overline{x})$ is the unknown polynomial:
    - Membership query $MEM(\overline{a})$ returns $f(\overline{a})$;
    - Equivalence query $EQ(h(\overline{x}))$ returns $\overline{b}$ such that $h(\overline{b}) \neq f(\overline{b})$.
- Problems:
  - How to resolve membership queries?
    - A random value may carry too much incorrect information.
    - Static analysis may help.
  - How to formulate pre- and post-conditions in polynomials?
    - Finding the polynomial invariant is hard (few solutions).
    - Finding a polynomial invariant to prove pre- and post-conditions may be easy (many solutions).
- If you are interested, please contact Prof. Yi.

Conclusions

- Algorithmic learning is useful.
  - All you need to do is to devise a teacher.
- Coin tossing is useful.
  - Algorithm people have observed it for over 20 years.
  - "... we show that at least for min-cut problem, there is an efficient randomized algorithm running in $O(n^2 \log^{O(1)} n)$ time. For dense graphs this is significantly better than the running time $O(mn \log(n^2/m))$ for the best-known max-flow problem."
    pp 289-290, Randomized Algorithms

Thank you!

It is my pleasure to be here.
I wish I had more time to work with you.