Detecting Buffer Overflow for C like languages using CLP

School of IT, University of Sydney

19th November 2010

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

Engineering Software

- It takes genius to write software
- The quality of software products is
- ....average to poor
- Economic benefits of computerization are immense
- If Sydney Harbour bridge were to be made by Software engineers
- We would somehow scramble and get a hanging bridge
- then, we will test to see if it can take the load
- pass one truck on the bridge ... hurray..... the bridge did not collapse
- pass the second truck...wait....
- ...one of the support beam has bent, quick, get someone to reinforce it
- Software engineers lack proper tools for their trade

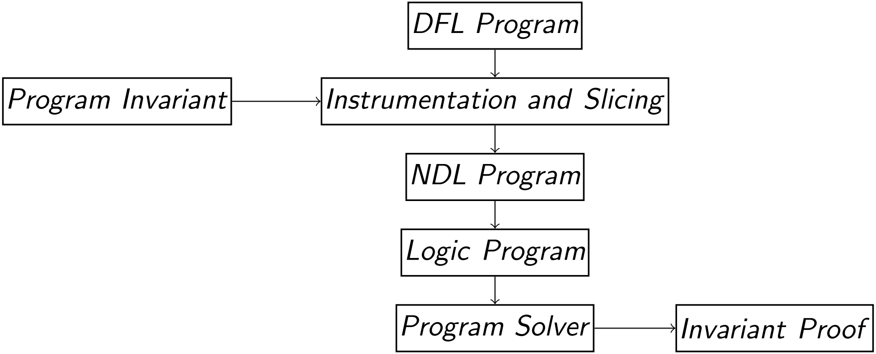
Motivation: Parfait Static Analysis Tool

- Motivated by Parfait, a research project at Oracle Sun Labs
- Layered analyses in time-complexity order
- Passes potential bugs to next layer which is slower but more precise
- Ends with fewer false positives
- Static bug checking framework designed for scalability and precision
- Analyses the OpenSolaris with several million lines of code in less than 30 minutes

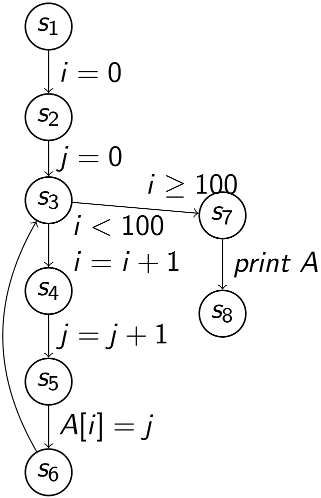
Our technique

- Demand driven
- Applicable to any program property expressed as a program invariant
- Uses standard logic program solvers
- Suitable as final layer in tools like Parfait

Our Technique



Deterministic Flow Graph (DFL)

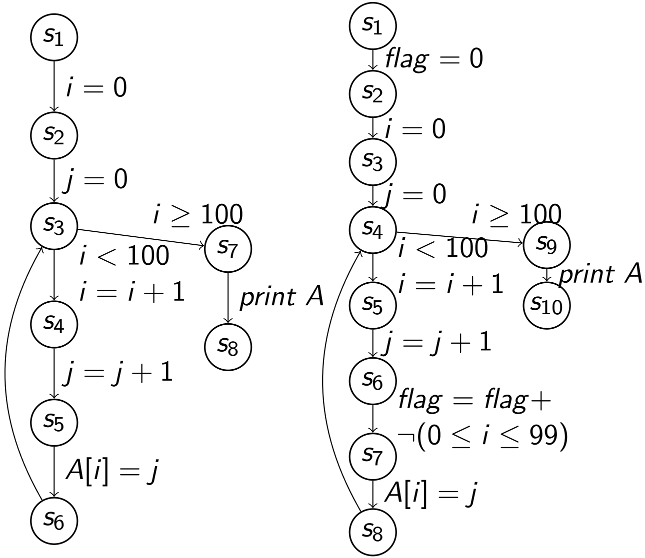


- Nodes are program point
- s_1 is start node
- s_8 is exit node
- Edges are statements
- Edge u with program state σ transforms it to state σ' and passes control to next node u'
- Semantics are defined by state transition function $(u, \sigma) \rightarrow (u', \sigma')$
- A well formed, directed, reachable graph

Non-Deterministic Language

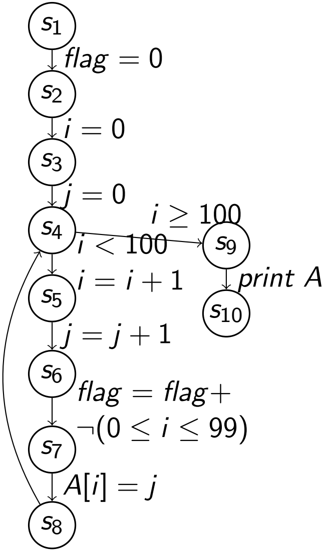
- Process several program states simultaneously
- Program state is not a unique state but a set of states.
- *assign* statement changes the set of program states
- $choose(Stmt_1, Stmt_2)$ produces two output program states for each input state
- $repeat(Stmt)$ produces infinite program states for each input state
- $repeat(Stmt) = choose(Stmt^0, Stmt^1, Stmt^2, \ldots)$
- *choose* and *repeat* increase program states
- *assume(Expr)* prunes program states by filtering them

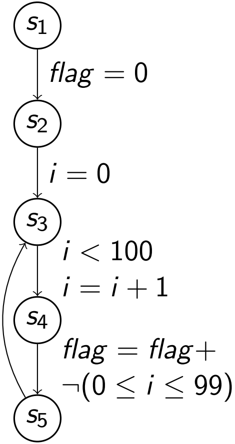
Program Instrumentation



- A new variable called `flag`
- Initially set to zero, indicating property not been identified yet
- Add a non-zero value if property violated
- Value of `flag` at program end proves buffer overflow property

Program Slicing



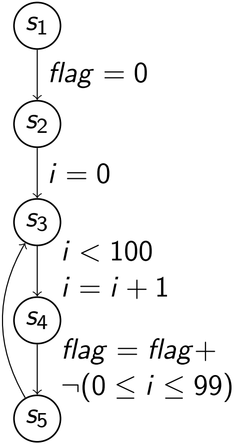


- Program slicing for variable *flag* at node s_3
- Only those statements that affect value of *flag* at node s_3 are retained
- All other statements are removed from the program to obtain program slice
- Slice is smaller but computes same value of *flag* as the original program

Translation to NDL

- With the means of Kildall's monotone dataflow framework and
- Tarjan's path homomorphism
- Imperative programs represented as a flow graph
- A regular expression over graph edges and
- syntactic level rewrite rules
- translate it to non-deterministic language

Regular Path Expression



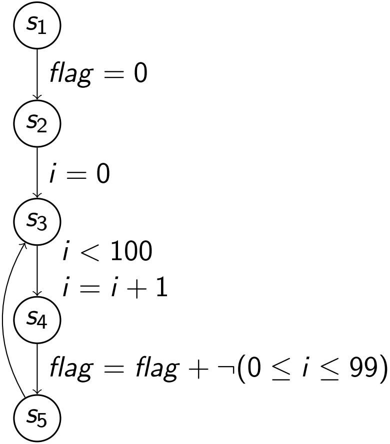
- Paths from start node s_1 to node s_3
- $(s_1, s_2).(s2, s3)$
- $(s_1, s_2).(s_2, s_3).(s_3, s_4).(s_4, s_5).(s_5, s_3)$
- $(s_1, s_2).(s_2, s_3).(s_3, s_4).(s_4, s_5).(s_5, s_3).(s_3, s_4)$ $.(s_4, s_5).(s_5, s_3)$
- Infinite number of paths denoted by $Paths(s_1, s_3)$
- Can be represented by a Regular expression over edges
- $Paths(s_1, s_3)$ is $(s_1, s_2).(s_2, s_3).\left[(s_3, s_4).(s_4, s_5).(s_5, s_3)\right]^*$
- Path expression is used to do translation to NDL

Translation scheme DFL to NDL

- Regular expression consists of alphabets (*edges*), $+$, . and $*$ operators
- Regular expression is rewritten using following rules to create NDL program

$$
\begin{aligned}
K(\epsilon) &= \mathit{skip} \\
K(e) &= \mathit{assume}(\mathit{bp}(e)); \mathit{eff}(e) \\
K(p_1 + p_2) &= \mathit{choose}(K(p_1), K(p_2)) \\
K(p_1 \cdot p_2) &= K(p_1); K(p_2) \\
K(p^*) &= \mathit{repeat}(K(p))
\end{aligned}
$$

Example




```
flag:=0;
i:=0;
repeat(
   assume(i < 100);
   i := i + 1;
   choose(
   assume(¬0 ≤ i ≤ 99);skip,
   assume(0 ≤ i ≤ 99);flag:=flag+1
   )
assume (flag <> 0)
```

$Paths(s_1, s_3)$ is

$(s_1, s_2).(s_2, s_3).\,[(s_3, s_4).(s_4, s_5).(s_5, s_3)]^*$

Translation Rules NDL to Prolog

- Each NDL statement is rewritten to Prolog statement using syntactic rules
- Program variables become a set of parameters to Prolog procedures

Translation Rules NDL to Prolog

skip

$q_{\iota(skip)}(\bar{X}, \bar{Y})\text{:- } Y_1 \text{ is } X_1, \ldots, Y_m \text{ is } X_m.$

assume(Expr)

$q_{\iota(assume(Expr))}(\bar{X}, \bar{Y}) \text{ :- } Y_1 \text{ is } X_1, \ldots, Y_m \text{ is } X_m, Expr \neq 0.$

x_i:=Expr

$q_{\iota(x_i:=Expr)}(\bar{X}, \bar{Y})\text{:- } Y_1 \text{ is } X_1, \ldots, Y_{i-1} \text{ is } X_{i-1}, Y_i \text{ is } Expr,$
$Y_{i+1} \text{ is } X_{i+1}, \ldots, Y_m \text{ is } X_m.$

Stmt1;Stmt2

$q_{\iota(Stmt1;Stmt2)}(\bar{X}, \bar{Y})\text{:- } q_{\iota(Stmt1)}(\bar{X}, \bar{X}'), q_{\iota(Stmt2)}(\bar{X}', \bar{Y}).$

choose($Stmt_1$;$Stmt_2$)

$q_{\iota(choose(Stmt_1;Stmt_2))}(\bar{X}, \bar{Y})\text{:- } q_{\iota(Stmt1)}(\bar{X}, \bar{Y}).$
$q_{\iota(choose(Stmt_1;Stmt_2))}(\bar{X}, \bar{Y})\text{:- } q_{\iota(Stmt2)}(\bar{X}, \bar{Y}).$

repeat(Stmt)

$q_{\iota(repeat(Stmt))}(\bar{X}, \bar{Y})\text{:- } q_{\iota(Stmt)}(\bar{X}, \bar{X}'), q_{\iota(repeat(Stmt))}(\bar{X}', \bar{Y}).$
$q_{\iota(repeat(Stmt))}(\bar{X}, \bar{Y})\text{:- } Y_1 \text{ is } X_1, \ldots, Y_m \text{ is } X_m.$

Example

```
flag:=0;
i:=0;
repeat(
  assume(i < 100);
  i := i + 1;
  choose(
  assume(¬0 ≤ i ≤ 99);
  skip,
  assume(0 ≤ i ≤ 99);
  flag:=flag+1
  )
assume (flag <> 0)
```

```
q(I,FLAG,IO,FLAGO) :- FLAG1 is 0,I1=0,
  q1(I1,FLAG1,I2,FLAG2), (FLAG2=0),
  IO is I2,FLAGO is FLAG2.
q1(I,FLAG,IO,FLAGO) :- q2(I,FLAG,I1,FLAG1),
  q1(I1,FLAG1,IO,FLAGO).
q1(I,FLAG,IO,FLAGO) :- IO is I,FLAGO is FLAG.
q2(I,FLAG,IO,FLAGO) :- (I < 100),I1 is I+1,
  q3(I1,FLAG,I2,FLAG1),IO is I2,FLAGO is FLAG1.
q3(I,FLAG,IO,FLAGO) :- q4(I,FLAG,IO,FLAGO).
q3(I,FLAG,IO,FLAGO) :- q5(I,FLAG,IO,FLAGO).
q4(I,FLAG,IO,FLAGO) :- (0=<I,I=<99),
  IO is I,FLAGO is FLAG1.
q5(I,FLAG,IO,FLAGO) :-(not(0<I);(not(I=<99))),
  FLAG1=FLAG+1,IO is I,FLAGO is FLAG1.
?  q(I,FLAG,IO,FLAG).
```

Prolog Solution

- Solve the Prolog program using a standard logic solver
- For our example program, $i = 0$ is a solution
- This proves that if C program is executed with input value of $i = 0$
- The program will have a buffer overflow error

Summary

- Convert the program flow chart to a flow graph language (DFL)
- Represent program properties (like buffer overflow) as program invariants
- Instrument and slice the DFL program
- Rewrite DFL graph as an NDL program
- Rewrite NDL program as a Prolog program
- Solve Prolog program using standard logic solvers
- A solution disproves the Program property
- Works over finite domains
- Slower than abstract interpretation
- Suitable for a layered analysis tool

Our Contribution

- Semantic preserving translation of a program to
  - a non-deterministic language
  - to a logic program.
- Program invariant property as a solution to the logic program.

?