Automatic Predicate Abstraction of C Programs

Thomas Ball Rupak Majumdar Todd Millstein
Sriram K. Rajamani

Presenter: Petur Olsen

October 12, 2007

Contributions

Predicate abstraction

- First algorithm for industrial strength language
- In this case C programs
- Tool called C2BP
- Integration into the SLAM toolkit

The Article

Published

- Conference on Programming language design and implementation
- ACM SIGPLAN 2001
- Microsoft Research

The Authors

Thomas Ball

- Principal Researcher at Microsoft Research
- Static and dynamic program analysis, model checking and theorem proving techniques
- Developer of SLAM

Rupak Majumdar

- Assistant Professor at University of California, Los Angeles
- Computer-aided verification, software verification and programming languages, logic, and automata theory
- Developer of BLAST

Authors

Todd Millstein

- Assistant Professor at University of California, Los Angeles
- Programming Languages for Sensor Networks

Sriram K. Rajamani

- Senior Researcher at Microsoft Research India
- Leader of the Rigorous Software Engineering Research Group
- Developer of SLAM

Outline

1. C2bp
2. Bebop
3. Tests and Results
4. Conclusion

SLAM

Three Phases

- Abstraction using C2BP
- Model checking using the BEBOP model checker
- Refinement using the NEWTON predicate discoverer

SLAM

Features

- Fully automated
- No spurious error paths
- Automatic refinement
- May not converge
- May terminate with “don’t know”
- Rarely does

C2BP

General

- Translation from C program into Boolean program
- Predicate abstraction
- Only boolean variables with added constructs

C2BP

Boolean language

- Same control-flow structure
- One boolean variable for each predicate
- Boolean expressions over variables in the program
- Represent truth values at statements

```
typedef struct cell {
  int val;
  struct cell* next;
} *list;

list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;
  curr = *l;
  prev = NULL;
  newl = NULL;
  while (curr != NULL) {
    nextCurr = curr->next;
    if (curr->val > v) {
      if (prev != NULL) {
        prev->next = nextCurr;
      }
      if (curr == *l) {
        *l = nextCurr;
      }
      curr->next = newl;
      newl = curr;
    } else {
      prev = curr;
    }
    curr = nextCurr;
  }
  return newl;
}
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};
  {curr==NULL} = unknown();
  {curr->val>v} = unknown();
  {prev==NULL} = true;
  {prev->val>v} = unknown();
  skip;
  while(*) {
    assume(!{curr==NULL});
    skip;
    if (*) {
      assume({curr->val>v});
      if (*) {
        assume(!{prev==NULL});
        skip;
      }
      if (*) {
        skip;
      }
      skip;
      skip;
    } else {
      assume(!{curr->val>v});
      {prev==NULL} = {curr==NULL};
      {prev->val>v} = {curr->val>v};
    }
    {curr==NULL} = unknown();
    {curr->val>v} = unknown();
  }
  assume({curr==NULL});
}
```

```
list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l;

  prev = NULL;

  newl = NULL;
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};

  {curr==NULL} = unknown();
  {curr->val>v} = unknown();

  {prev==NULL} = true;
  {prev->val>v} = unknown();

  skip;
```

```
list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l;

  prev = NULL;

  newl = NULL;
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};

  {curr==NULL} = unknown();
  {curr->val>v} = unknown();

  {prev==NULL} = true;
  {prev->val>v} = unknown();

  skip;
```

```
list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l;

  prev = NULL;

  newl = NULL;
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};

  {curr==NULL} = unknown();
  {curr->val>v} = unknown();

  {prev==NULL} = true;
  {prev->val>v} = unknown();

  skip;
```

```
list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l;

  prev = NULL;

  newl = NULL;
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};

  {curr==NULL} = unknown();
  {curr->val>v} = unknown();

  {prev==NULL} = true;
  {prev->val>v} = unknown();

  skip;
```

```
list partition(list *l, int v) {
  list curr, prev, newl, nextCurr;

  curr = *l;

  prev = NULL;

  newl = NULL;
```

```
void partition() {
  bool {curr==NULL}, {prev==NULL};
  bool {curr->val>v}, {prev->val>v};

  {curr==NULL} = unknown();
  {curr->val>v} = unknown();

  {prev==NULL} = true;
  {prev->val>v} = unknown();

  skip;
```

```
bool unknown() {
  if(*) {
    return true;
  }
  else {
    return false;
  }
}
```

(*) = Non-deterministic choice

```
while (curr != NULL) {
  nextCurr = curr->next;
```

```
while (*) {
  assume(!{curr==NULL});
  skip;
```

```
while (curr != NULL) {
  nextCurr = curr->next;
```

```
while (*) {
  assume(!{curr==NULL});
  skip;
```

```
while (curr != NULL) {
  nextCurr = curr->next;
```

```
while (*) {
  assume(!{curr==NULL});
  skip;
```

Bebop

General

- Set of reachable states for each statement s
- State: Truth values of variables in scope at s
- Set of states: Boolean function over variables in scope at s

```
...
while (curr != null){
  ...
  if(curr->val > v){
    if(prev != NULL)
      prev->next = nextCurr
    ...
    newl = curr;
  } else prev = curr;
  curr = nextCurr;
}
...
```

$$(curr \neq NULL) \bigwedge (curr \rightarrow val > v) \bigwedge$$
$$((prev \rightarrow val \leq v) \bigvee (prev = NULL))$$

Automatic deduction

From this

$(curr \neq NULL) \bigwedge (curr \rightarrow val > v) \bigwedge$
$((prev \rightarrow val \leq v) \bigvee (prev = NULL))$

Bebop deducts

$(prev \neq curr)$

Overview

Two settings

- Tested Windows NT drivers using SLAM
- Checking array bounds and heap invariants

NT Drivers

Overview

- One internally developed driver
- Four drivers from Windows 2000 DDK
- One error in internal driver
- BEBOP finished in under 10 seconds

Results

program	lines	predicates	thm. prover calls	runtime (seconds)
floppy	6500	23	5509	98
ioctl	1250	5	500	13
openclos	544	5	132	6
srdriver	350	30	3034	93
log	236	6	98	5

Overview

Five tests

- Knuth-Morris-Pratt string matcher
- Array implementation of quick sort
- List partitioning
- List search
- Reverse a list twice

Results

program	lines	predicates	thm. prover calls	runtime (seconds)
kmp	75	4	286	7
qsort	45	2	199	5
partition	55	4	263	9
listfind	37	6	4412	172
reverse	73	7	26769	747

Automatic Predicate Abstraction

Advantages

- Used on industrial strength language
- Adaptable to (e.g.) Java
- Fully automatic

Disadvantages

- Limited predicate logic
- Exponential running time

Questions?