Automatic Predicate Abstraction of C Programs

T. Ball R. Majumdar T. Millstein S. K. Rajamani
presented by Jan Mrázek

Masaryk University
Brno, Czech Republic

16th October 2017

Motivation

- model checking has been successful in validating
  - models
  - protocols
  - hardware
- it struggles with
  - arithmetics (finite-state systems)
  - heap manipulation (inifinite-state systems)

Motivation

- model checking has been successful in validating
  - models
  - protocols
  - hardware
- it struggles with
  - arithmetics (finite-state systems)
  - heap manipulation (inifinite-state systems)

Solution: use abstraction

Predicate Abstraction in a Nutshell

- idea:
  - do not keep full states
  - keep only truth values of predicates over data
- abstract function: $\alpha(s) = (p_1(s), p_2(s), ..., p_n(s))$

Predicate Abstraction in a Nutshell

- idea:
  - do not keep full states
  - keep only truth values of predicates over data
- abstract function: $\alpha(s) = (p_1(s), p_2(s), ..., p_n(s))$
- challenge: correctly and efficiently compute transitions

Predicate Abstraction in C2BP

Input:

- C program P
  - no restrictions except concurrency and interprocedural jumps
- set E of predicates
  - pure C boolean expressions
  - no function calls
  - e.g. $\{\texttt{x < y}, \texttt{x > 0}\}$

Output:

- boolean program $\mathcal{B}P(P, E)$
  - C program with several extensions
  - only boolean variables
  - number of variables $= |E|$

Example – List Partition

```
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;
}
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL`, `prev == NULL`, `curr->val > v`, `prev->val > v` $\}$

```
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;
}
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL`, `prev == NULL`, `curr->val > v`, `prev->val > v` $\}$

```
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;
}
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    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;
    }
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    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;
    }
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    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
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    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
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{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;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{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;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{curr==NULL});
        skip();
        if (curr->val > v) {
            if (prev != NULL) prev->next = nextCurr;
            if (curr == *l) *l = nextCurr;
            curr->next = newl; newl = curr;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{curr==NULL});
        skip();
        if (curr->val > v) {
            if (prev != NULL) prev->next = nextCurr;
            if (curr == *l) *l = nextCurr;
            curr->next = newl; newl = curr;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{curr==NULL});
        skip();
        if (choose(2)) {
            assume({cur->val>v});
            if (prev != NULL) prev->next = nextCurr;
            if (curr == *l) *l = nextCurr;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{curr==NULL});
        skip();
        if (choose(2)) {
            assume({cur->val>v});
            if (prev != NULL) prev->next = nextCurr;
            if (curr == *l) *l = nextCurr;
```

Boolean Program Example – List Partition

- $E = \{$ `curr == NULL, prev == NULL, curr->val > v, prev->val > v` $\}$

```
list partition(list *l, int v) {
    bool {curr==NULL}, {prev==NULL};
    bool {curr->val>v}, {prev->val>v};
    {curr==NULL} = unknown();
    {prev==NULL} = true;
    {curr->val>v} = unknown();
    {prev->val>v} = unknown();
    while (choose(2)) {
        assume(!{curr==NULL});
        skip();
        if (choose(2)) {
            assume({cur->val>v});
            if (choose(2)){
                assume(!{(prev==NULL});
```

Properties of Boolean Programs

$\mathcal{BP}(P, E)$:

- has the same control-flow as P
- contains only $|E|$ boolean variables
- is an over-approximation of P

Preprocessing

- control flow is expressed using
  - if-then-else
  - goto
- all expressions are side-effect free
- no short-circuit evaluation
  - `if ( a || b ) { /*body*/ }`
  - `if ( a ) { /*body*/ } else if ( b ) { /*body*/ }`
- no multiple dereferences of a pointer
  - `**p;`
  - `int *x = *p; *x;`
- call only at the top-most level of an expression
  - `z = x + f( y );`
  - `t = f( y ); z = x + t;`

Updating Boolean Variables I

We need to define rules for updating variables' values.

Updating Boolean Variables I

We need to define rules for updating variables' values.

For statement s and a predicate φ:

- denote $WP(s, \varphi)$ as the weakest precondition
- weakest predicate before s entailing truth of φ after s
- example:

$$WP(\texttt{x = x + 1}, x < 5) = (x + 1) < 5 = (x < 4)$$

Updating Boolean Variables I

We need to define rules for updating variables' values.

For statement s and a predicate φ:

- denote $WP(s, \varphi)$ as the weakest precondition
- weakest predicate before s entailing truth of φ after s
- example:

$$WP(\texttt{x = x + 1}, x < 5) = (x + 1) < 5 = (x < 4)$$

- let b be a corresponding boolean variable to φ
- if $\varphi \in E$ and b is true before s, b is true after s

Updating Boolean Variables I

We need to define rules for updating variables' values.

For statement s and a predicate φ:

- denote $WP(s, \varphi)$ as the weakest precondition
- weakest predicate before s entailing truth of φ after s
- example:

$$WP(\texttt{x = x + 1}, x < 5) = (x + 1) < 5 = (x < 4)$$

- let b be a corresponding boolean variable to φ
- if $\varphi \in E$ and b is true before s, b is true after s

- "if `x < 4` is true before `x = x + 1`, then `x < 5` is true afterwards"

Updating Boolean Variables II

- if $\varphi \notin E$ we strenghten φ
  - only over expressions in E
- example:

$$E = \{\texttt{x < 5}, \texttt{x = 2}\}$$
$$WP(\texttt{x = x + 1}, x < 5) = (x < 4)$$
$$\texttt{x = 2} \implies \texttt{x < 4}$$

Updating Boolean Variables II

- if $\varphi \notin E$ we strenghten φ
  - only over expressions in E
- example:

$$E = \{\texttt{x < 5}, \texttt{x = 2}\}$$
$$WP(\texttt{x = x + 1}, x < 5) = (x < 4)$$
$$\texttt{x = 2} \implies \texttt{x < 4}$$

- "if `x = 2` before `x = x + 1`, then `x < 5` is true afterwards"

Strengtening of a Predicate

- denote V as a set of boolean variables $\{b_1, b_2, \cdots, b_n\}$
- *cube* over V is a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i \in \{b_i, \neg b_i\}$
- denote $\mathcal{E}(b_i)$ as a corresponding predicate φ_i to b_i
- extend $\mathcal{E}$ to cubes

Strengtening of a Predicate

- denote V as a set of boolean variables $\{b_1, b_2, \cdots, b_n\}$
- *cube* over V is a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i \in \{b_i, \neg b_i\}$
- denote $\mathcal{E}(b_i)$ as a corresponding predicate φ_i to b_i
- extend $\mathcal{E}$ to cubes

- $\mathcal{F}_v(\varphi)$ – the largest disjuntion of cubes over V such that:
  - $\mathcal{E}(c) \implies \varphi$

Strengtening of a Predicate

- denote V as a set of boolean variables $\{b_1, b_2, \cdots, b_n\}$
- *cube* over V is a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i \in \{b_i, \neg b_i\}$
- denote $\mathcal{E}(b_i)$ as a corresponding predicate φ_i to b_i
- extend $\mathcal{E}$ to cubes

- $\mathcal{F}_v(\varphi)$ – the largest disjuntion of cubes over V such that:
  - $\mathcal{E}(c) \implies \varphi$

- use theorem prover for validating strenghtenings

Strengtening of a Predicate

- denote V as a set of boolean variables $\{b_1, b_2, \cdots, b_n\}$
- *cube* over V is a conjunction $c_1 \wedge c_2 \wedge \cdots \wedge c_k$ where $c_i \in \{b_i, \neg b_i\}$
- denote $\mathcal{E}(b_i)$ as a corresponding predicate φ_i to b_i
- extend $\mathcal{E}$ to cubes

- $\mathcal{F}_v(\varphi)$ – the largest disjuntion of cubes over V such that:
  - $\mathcal{E}(c) \implies \varphi$

- use theorem prover for validating strenghtenings

- define weakening as $\mathcal{G}_v(\varphi) = \neg\mathcal{F}_V(\neg\varphi)$

Handling Aliases

- when pointers are present, $WP(\texttt{x = e}, \varphi) \neq \varphi[e/x]$
  - example $WP(\texttt{x = 3}, \texttt{*p > 5})$ when $\texttt{\&x = p}$

Handling Aliases

- when pointers are present, $WP(\texttt{x = e}, \varphi) \neq \varphi[e/x]$
  - example $WP(\texttt{x = 3}, \texttt{*p > 5})$ when $\texttt{\&x = p}$
- two cases:
  - $\texttt{x}$ and $\texttt{p}$ are aliases
  - $\texttt{x}$ and $\texttt{p}$ are distinct
- $WP(\texttt{x = 3}, \texttt{*p > 5}) = (\&x = p \wedge 3 > 5) \vee (\&x \neq p \wedge *p > 5)$
- for k possible aliases, there will be 2^k disjuncts
- use a pointer may analysis to prune the disjuncts

Handling Assignments

- assignment yields parallel assignment to all boolean variables
- b_i can be after assignment
  - true if $\mathcal{F}_v(WP(\texttt{x = e}, \varphi))$ holds before assignment
  - false if $\mathcal{F}_v(WP(\texttt{x = e}, \neg\varphi))$ holds before assignment
- if neither holds, assign non-deterministic value

Handling Gotos

- no dependency on V
- copy them to $\mathcal{B}P$

Handling Conditionals

- `if (`φ`) { ... } else { ... }`:
  - φ holds in then branch $\rightarrow \mathcal{G}_V(\varphi)$ holds
  - $\neg\varphi$ holds in else branch $\rightarrow \mathcal{G}_V(\neg\varphi)$ holds

Handling Conditionals

- `if (`φ`) { ... } else { ... }`:
  - φ holds in then branch $\rightarrow \mathcal{G}_V(\varphi)$ holds
  - $\neg\varphi$ holds in else branch $\rightarrow \mathcal{G}_V(\neg\varphi)$ holds

```
if ( choose(2) ) {
    assume( G_V( phi ) )
    ...
}
else {
    assume( G_V( !phi ) )
    ...
}
```

Handling Function Calls

- local predicates vs. global predicates
- each function can be transformed independently
  - arguments – values of local predicates referring to formal parameters
  - return value – tuple of updated global and local predicates
- when calling function
  - compute actual value of formal arguments (predicates)
  - call and store return value to a new tuple of variable
  - update local and global predicates (take aliasing into account)

The Enforce Construct

- predicates might be correlated
  - e.g. `x = 1` and `x = 2`
- `enforce(`Φ`)`
  - put `assume(`Φ`)` between every two statements in a procedure
- $\Phi = \mathcal{F}_V(\mathit{false})$

Optimizations

- theorem prover running time is dominating
  - up to exponentially many calls

Optimizations

- theorem prover running time is dominating
  - up to exponentially many calls
- prune cubes
  - consider cubes in increasing length
  - detect cubes implying $\neg\varphi$
- use heuristics to limit predicates
- cone of influence to reduce number of variables in $\mathcal{F}$
- construct $\mathcal{F}$ syntactically
- cache computations

Optimizations

- theorem prover running time is dominating
  - up to exponentially many calls
- prune cubes
  - consider cubes in increasing length
  - detect cubes implying $\neg\varphi$
- use heuristics to limit predicates
- cone of influence to reduce number of variables in $\mathcal{F}$
- construct $\mathcal{F}$ syntactically
- cache computations
- sacrifice precision and limit length of cubes