Model Checking Using SMT and Theory of Lists

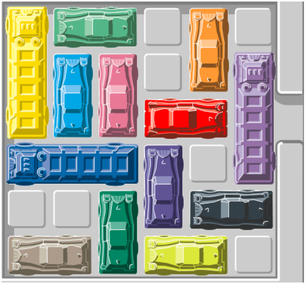
Aleksandar Milicevic [1] Hillel Kugler [2]

[1] Massachusetts Institute of Technology
Cambridge, MA

[2] Microsoft Research
Cambridge, UK

Third NASA Formal Methods Symposium,
April 18, 2011

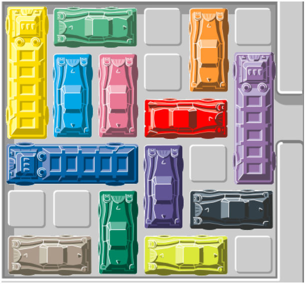
Solving Planning Problems



Rush Hour puzzle

Goal: drive the red car out of the jam

Solving Planning Problems

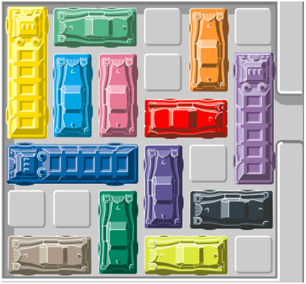


Rush Hour puzzle

Goal: drive the red car out of the jam

- solve using a satisfiability solver

Solving Planning Problems



Rush Hour puzzle

Goal: drive the red car out of the jam

- solve using a satisfiability solver
- problem: number of necessary steps is not known

Software Model Checking without Loop Unrolling

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j];
  a[j] = a[min];
  a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Selection Sort algorithm

Goal: verify for all `int` arrays of size up to N

Software Model Checking without Loop Unrolling

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j];
  a[j] = a[min];
  a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Selection Sort algorithm

Goal: verify for all `int` arrays of size up to N

- verify using model checking with satisfiability solving

Software Model Checking without Loop Unrolling

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j];
  a[j] = a[min];
  a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Selection Sort algorithm

Goal: verify for all `int` arrays of size up to N

- verify using model checking with satisfiability solving
- problem: number of necessary loop unrollings is not known

Software Model Checking without Loop Unrolling

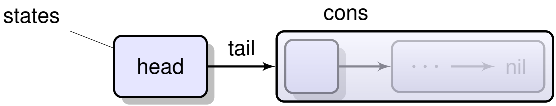
```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j];
  a[j] = a[min];
  a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Selection Sort algorithm

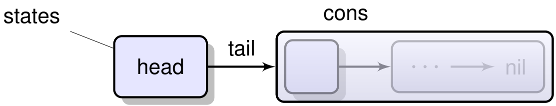
Goal: verify for all `int` arrays of size up to N

- verify using model checking with satisfiability solving
- problem: number of necessary loop unrollings is not known
- moreover, the number of loop unrollings is not independent of N

Use Lists to Model State Transitions

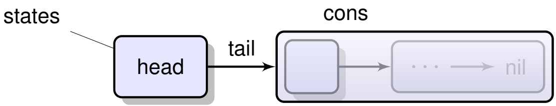


Use Lists to Model State Transitions



The length of the list is not explicitly bounded

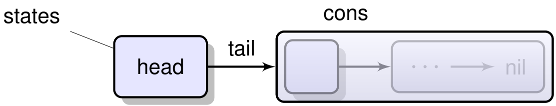
Use Lists to Model State Transitions



The length of the list is not explicitly bounded

Specify what the list should look like, not how long it should be.

Use Lists to Model State Transitions



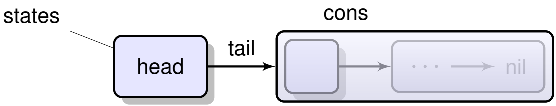
The length of the list is not explicitly bounded

Specify what the list should look like, not how long it should be.

To solve the rush hour puzzle:

- use a list to model a sequence of car movements
- don't have to specify the number of steps

Use Lists to Model State Transitions



The length of the list is not explicitly bounded

Specify what the list should look like, not how long it should be.

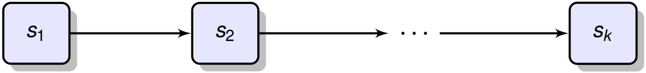
To solve the rush hour puzzle:

- use a list to model a sequence of car movements
- don't have to specify the number of steps

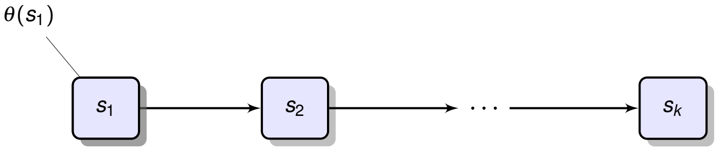
To solve a software model-checking problem:

- use a list to model a program trace
- don't have to specify the number of loop unrollings

Background: Bounded Model Checking



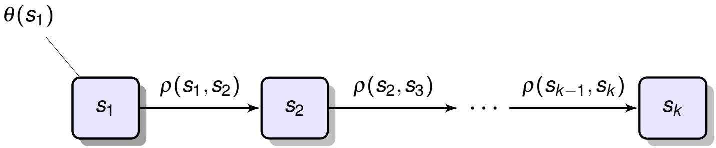
Background: Bounded Model Checking



Initial state constraint:

$$\theta(s_1)$$

Background: Bounded Model Checking



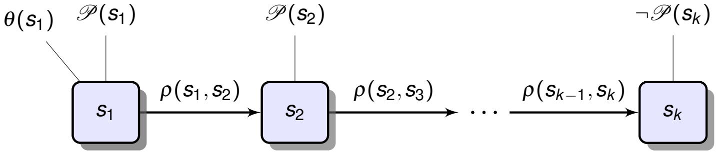
Initial state constraint:

$\theta(s_1)$

Transition constraint:

$\rho(s_1, s_2) \wedge \rho(s_2, s_3) \wedge \cdots \wedge \rho(s_{k-1}, s_k)$

Background: Bounded Model Checking



Initial state constraint:

$$\theta(s_1)$$

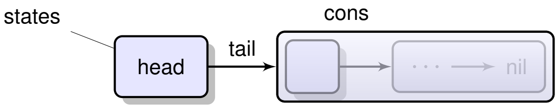
Transition constraint:

$$\rho(s_1, s_2) \wedge \rho(s_2, s_3) \wedge \cdots \wedge \rho(s_{k-1}, s_k)$$

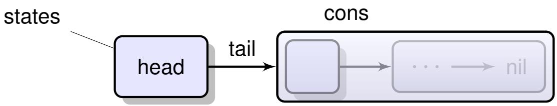
Safety Property constraint:

$$\mathscr{P}(s_1) \wedge \mathscr{P}(s_2) \wedge \cdots \wedge \mathscr{P}(s_{k-1}) \wedge \neg\mathscr{P}(s_k)$$

Translation to SMT

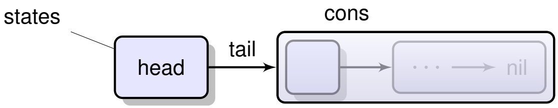


Translation to SMT



Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

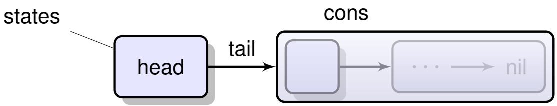
Translation to SMT



Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

Translation to SMT

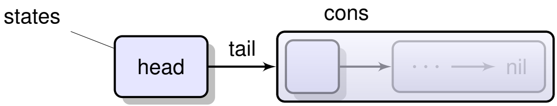


Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

def check_tr: StateList → bool

Translation to SMT



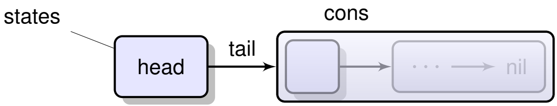
Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

def check_tr: StateList → bool

assert forall lst: StateList
  if (is_cons(lst) **and** is_cons(tail(lst))) **then**
    ρ(head(lst), head(tail(lst))) **and** check_tr(tail(lst)) **and**

Translation to SMT



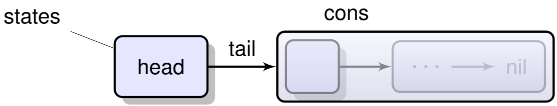
Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

def check_tr: StateList → bool

assert forall lst: StateList
  if (is_cons(lst) **and** is_cons(tail(lst))) **then**
    ρ(head(lst), head(tail(lst))) **and** check_tr(tail(lst)) **and**
    if (**not** $\mathscr{P}$(tail(lst))) **then**
      is_nil(tail(tail(lst)))
    else
      is_cons(tail(tail(lst)))

Translation to SMT



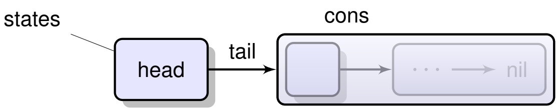
Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

def check_tr: StateList $\rightarrow$ bool

assert forall lst: StateList
  if (is_cons(lst) **and** is_cons(tail(lst))) **then**
    ρ(head(lst), head(tail(lst))) **and** check_tr(tail(lst)) **and**
    if (**not** $\mathscr{P}$(tail(lst))) **then**
      is_nil(tail(tail(lst)))
    else
      is_cons(tail(tail(lst)))
  :pat {check_tr}

Translation to SMT



Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

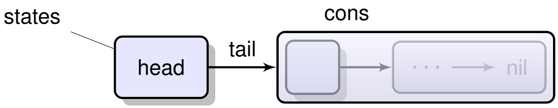
tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

def check_tr: StateList $\rightarrow$ bool

assert forall lst: StateList
  if (is_cons(lst) **and** is_cons(tail(lst))) **then**
    ρ(head(lst), head(tail(lst))) **and** check_tr(tail(lst)) **and**
    if (**not** $\mathscr{P}$(tail(lst))) **then**
      is_nil(tail(tail(lst)))
    else
      is_cons(tail(tail(lst)))
  :**pat** {check_tr}

assert is_cons(states) **and** θ(head(states)) **and** check_tr(states)

Translation to SMT



Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

state declaration

def check_tr: StateList → bool

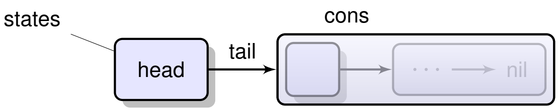
assert forall lst: StateList
 if (is_cons(lst) **and** is_cons(tail(lst))) **then**
  ρ(head(lst), head(tail(lst))) **and** check_tr(tail(lst)) **and**
  if (**not** $\mathscr{P}$(tail(lst))) **then**
   is_nil(tail(tail(lst)))
  else
   is_cons(tail(tail(lst)))
 :pat {check_tr}

state transition and **safety property** enforced with an *uninterpreted function* and an *axiom*

assert is_cons(states) **and** θ(head(states)) **and** check_tr(states)

formula to check

Translation to SMT



Available operations:
- is_nil(lst)
- is_cons(lst)
- head(lst)
- tail(lst)

tupletype State = [v1: INT, v2: INT, ...]
datatype StateList = nil | cons(head: State, tail: StateList)
def states: StateList

state declaration

def check_tr: StateList → bool

assert forall lst: StateList
  if (is_cons(lst) **and** is_cons(tail(lst))) **then**
    ρ**(head(lst), head(tail(lst)))** **and** check_tr(tail(lst)) **and**
    if (**not** $\mathscr{P}$**(tail(lst)))** **then**
      is_nil(tail(tail(lst)))
    else
      is_cons(tail(tail(lst)))
  :pat {check_tr}

state transition and **safety property** enforced with an *uninterpreted function* and an *axiom*

assert is_cons(states) **and** θ**(head(states))** **and** check_tr(states)

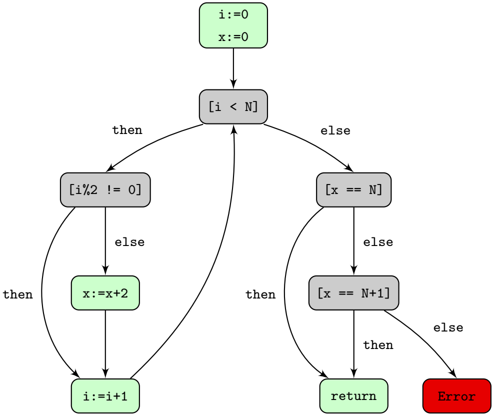
formula to check

Application to Software Model Checking

```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
    x += 2;
  i++;
 }
 assert x == N ||
        x == N + 1;
}
```

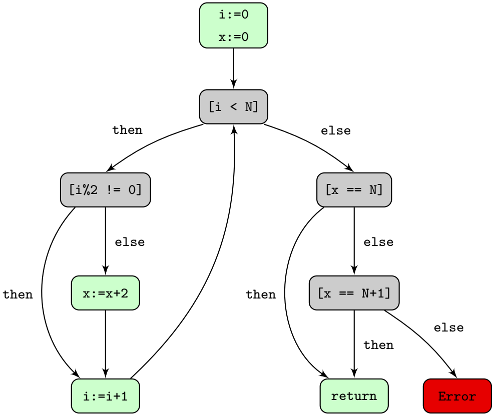
Application to Software Model Checking

```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
    x += 2;
  i++;
 }
 assert x == N ||
        x == N + 1;
}
```



Application to Software Model Checking

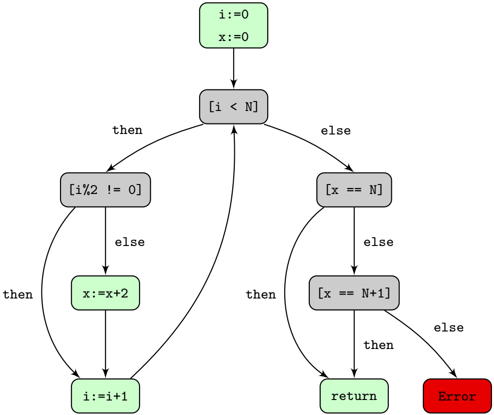
```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
    x += 2;
  i++;
 }
 assert x == N ||
        x == N + 1;
}
```



- goal: find a feasible path from start to an error node

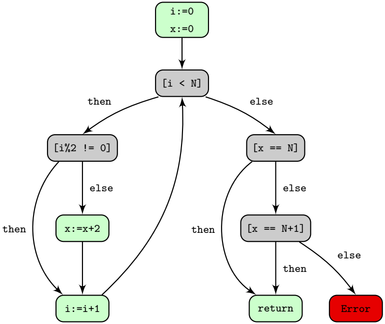
Application to Software Model Checking

```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
   x += 2;
  i++;
 }
 assert x == N ||
        x == N + 1;
}
```



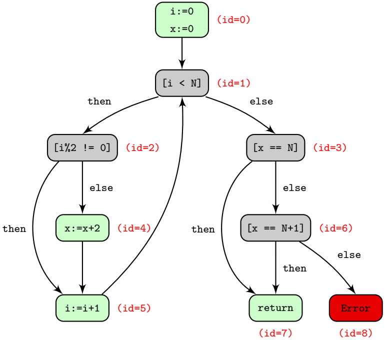
- **goal**: find a feasible path from start to an error node
- **idea**: use a list to represent a path in the graph

From CFG to θ, ρ, $\mathscr{P}$



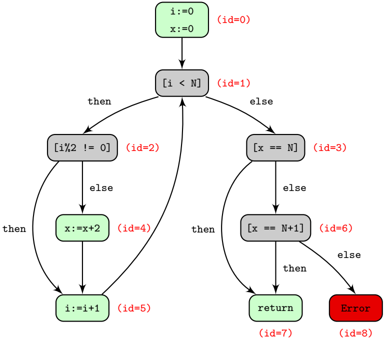
From CFG to θ, ρ, $\mathscr{P}$

1. assign IDs to basic blocks



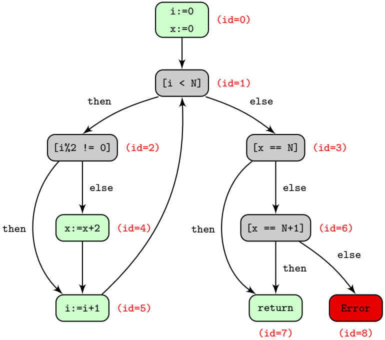
From CFG to θ, ρ, $\mathscr{P}$

1. assign IDs to basic blocks
2. state tuple: [id, i, x]



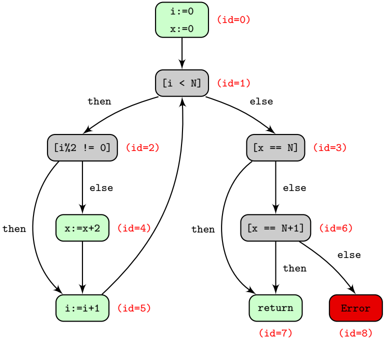
From CFG to θ, ρ, $\mathscr{P}$

1. assign IDs to basic blocks
2. state tuple: [id, i, x]
3. initial state constraint θ:
   head(states).i=0 $\wedge$ head(states).x=0



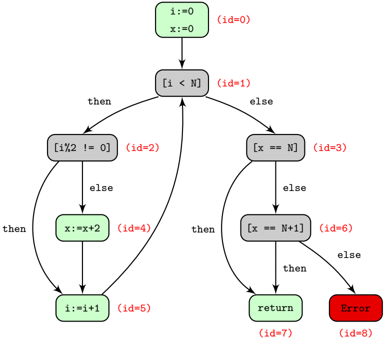
From CFG to θ, ρ, $\mathscr{P}$

1. assign IDs to basic blocks
2. state tuple: [id, i, x]
3. initial state constraint θ:
   head(states).i=0 $\wedge$ head(states).x=0
4. safety constraint $\mathscr{P}(lst)$:
   head(lst).id $\neq$ 8

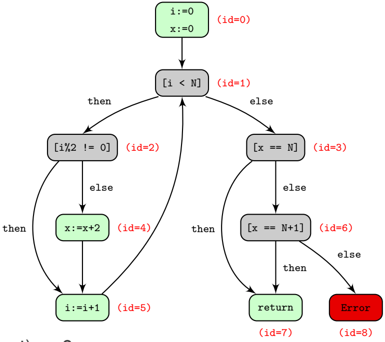


From CFG to θ, ρ, $\mathscr{P}$

1. assign IDs to basic blocks
2. state tuple: [id, i, x]
3. initial state constraint θ:
   head(states).i=0 $\wedge$ head(states).x=0
4. safety constraint $\mathscr{P}(lst)$:
   head(lst).id $\neq$ 8
5. transition constraint $\rho(curr, next)$:
   if head(curr).id=0 **then**
   head(next).id=1 $\wedge$ head(next).i=0 $\wedge$ head(next).x=0
   else if head(curr).id=1 **then**
   if head(curr).i $<$ N **then** head(next).id=2 **else** head(next).id=3
   ...
   else
   false

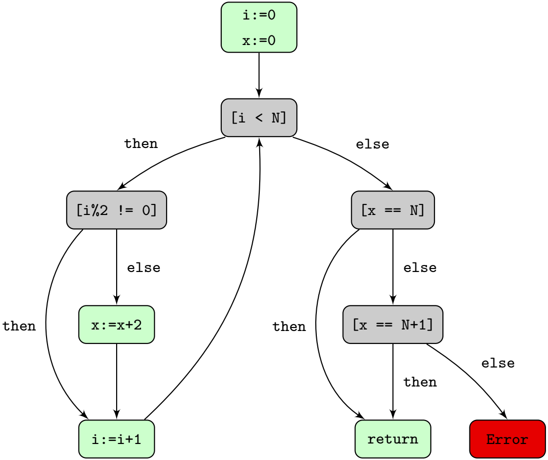


From CFG to θ, ρ, $\mathscr{P}$

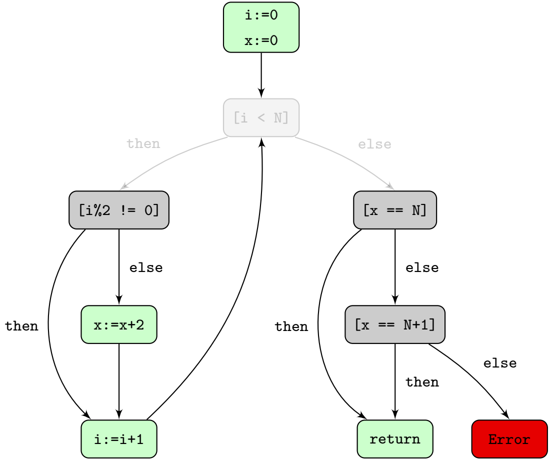


1. assign IDs to basic blocks
2. state tuple: [id, i, x]
3. initial state constraint θ:
   head(states).i=0 $\wedge$ head(states).x=0
4. safety constraint $\mathscr{P}(lst)$:
   head(lst).id $\neq$ 8
5. transition constraint $\rho(curr, next)$:
   if head(curr).id=0 **then**
   head(next).id=1 $\wedge$ head(next).i=0 $\wedge$ head(next).x=0
   else if head(curr).id=1 **then**
   if head(curr).i $<$ N **then** head(next).id=2 **else** head(next).id=3
   ...
   else
   false
6. bounds on some pieces of data: N $>$ 0 $\wedge$ N $<$ 10

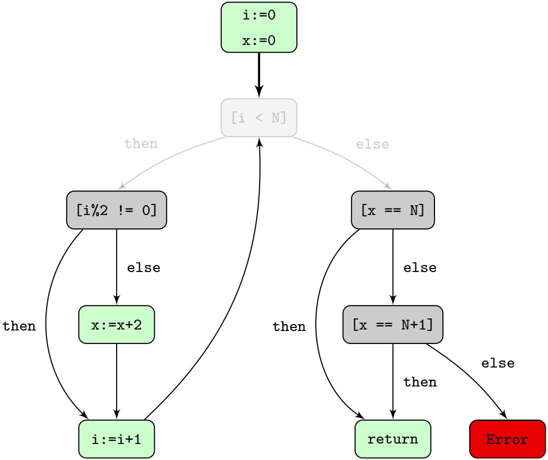
Optimization: Removing Empty Nodes



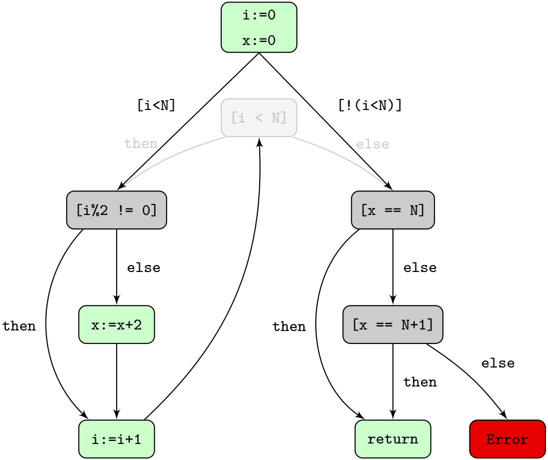
Optimization: Removing Empty Nodes



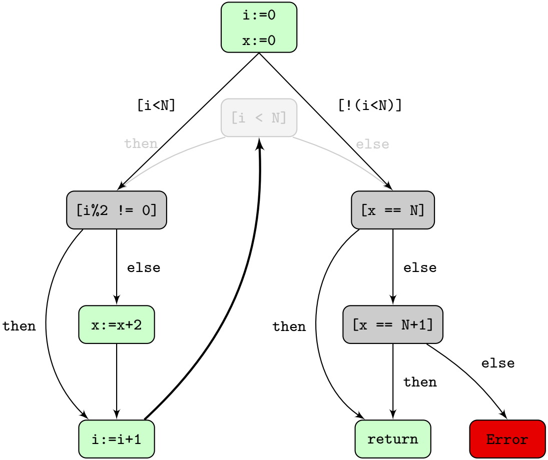
Optimization: Removing Empty Nodes



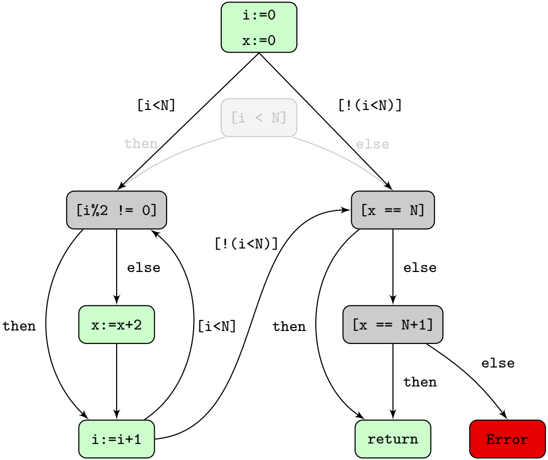
Optimization: Removing Empty Nodes



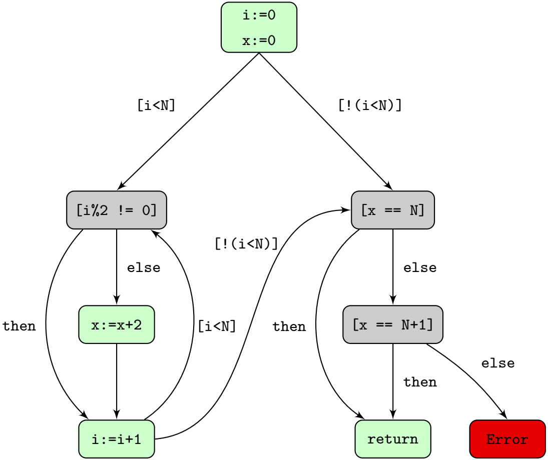
Optimization: Removing Empty Nodes



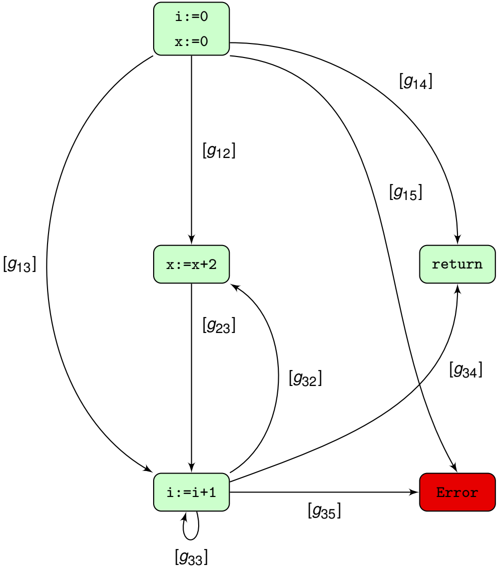
Optimization: Removing Empty Nodes



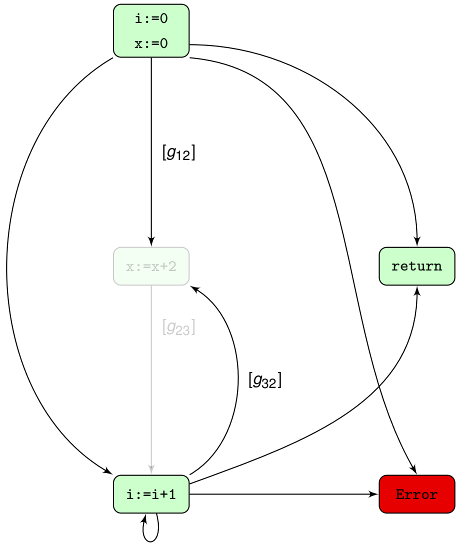
Optimization: Removing Empty Nodes



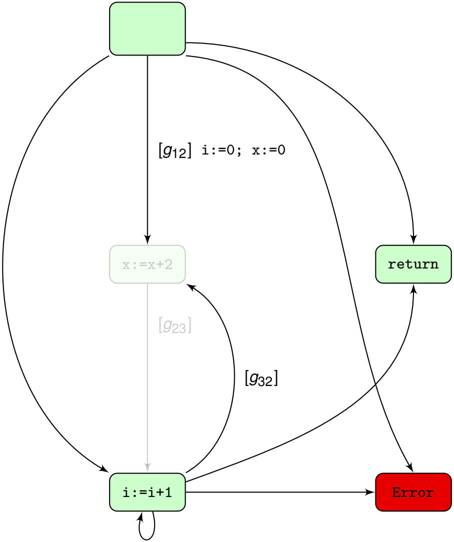
Optimization: Removing Non-Looping Nodes



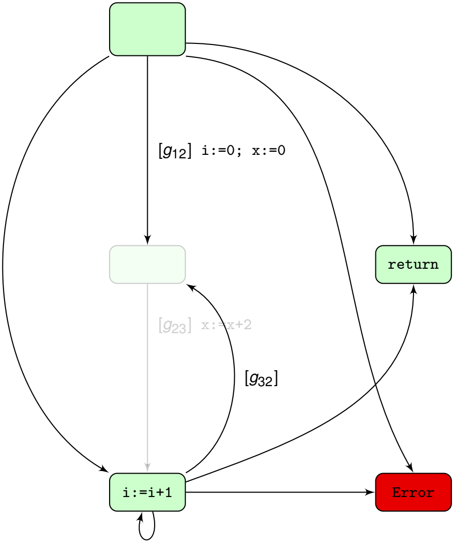
Optimization: Removing Non-Looping Nodes



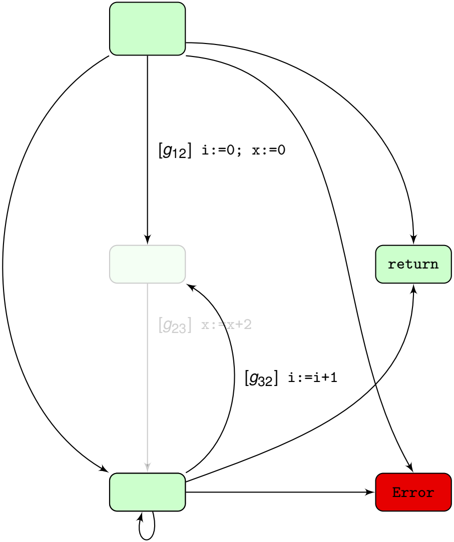
Optimization: Removing Non-Looping Nodes



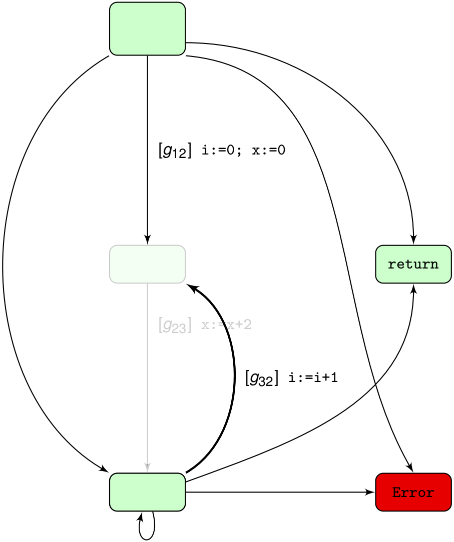
Optimization: Removing Non-Looping Nodes



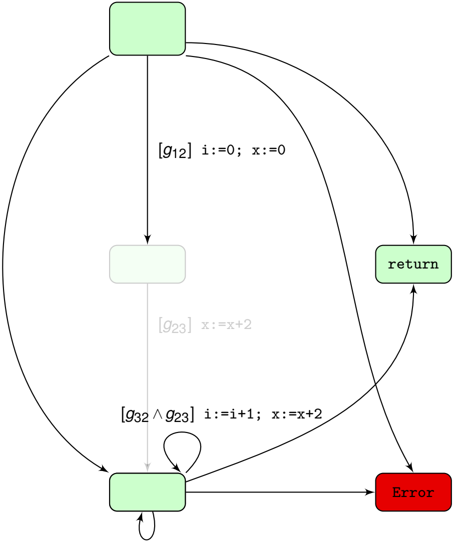
Optimization: Removing Non-Looping Nodes



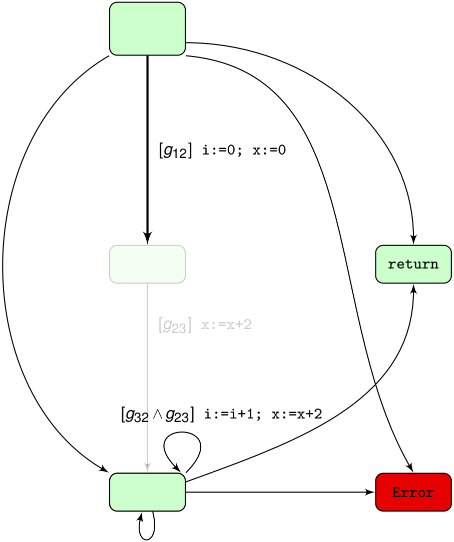
Optimization: Removing Non-Looping Nodes



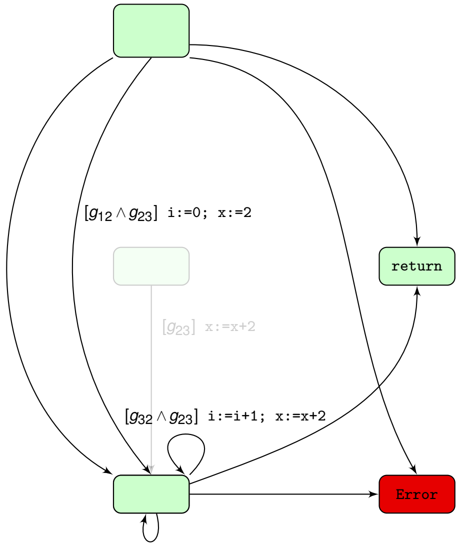
Optimization: Removing Non-Looping Nodes



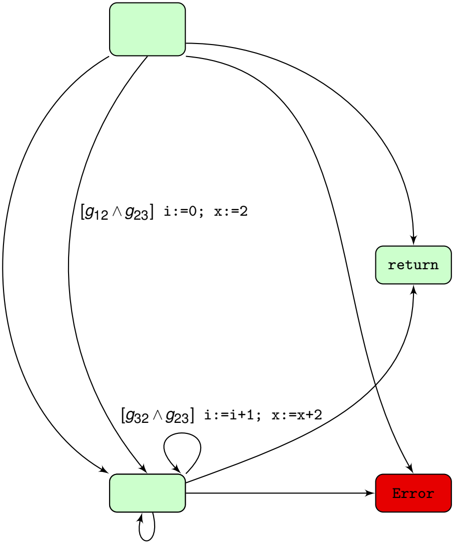
Optimization: Removing Non-Looping Nodes



Optimization: Removing Non-Looping Nodes



Optimization: Removing Non-Looping Nodes



Evaluation: Verifying Simple Algorithms

Simple While Loop

```
void simpleWhile(int N) {
 int x = 0, i = 0;
 while (i < N) {
  if (i % 2 == 0)
    x += 2;
  i++;
 }
 assert x == N ||
        x == N + 1;
}
```

Selection Sort Algorithm

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j]; a[j] = a[min]; a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Integer Square Root Algorithm

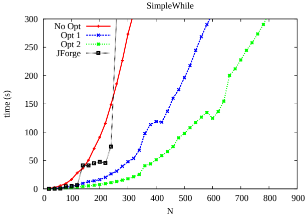
```
int intSqRoot(int N) {
  int r = 1, q = N;
  while (r+1 < q) {
    int p = (r+q) / 2;
    if (N < p*p) q = p;
    else r = p;
  }
  assert r*r <= N &&
         (r+1)*(r+1)>N;
  return r;
}
```

Bubble Sort Algorithm

```
void bubbleSort(int[] a, int N) {
 for (int j=0; j<N-1; j++)
  for (int i=0; i<N-j-1; i++)
   if (a[i] > a[i+1]) {
     int t = a[i];
     a[i] = a[i+1];
     a[i+1] = t;
   }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Evaluation: Verifying Simple Algorithms

Simple While Loop



Selection Sort Algorithm

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j]; a[j] = a[min]; a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Integer Square Root Algorithm

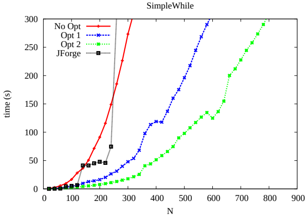
```
int intSqRoot(int N) {
  int r = 1, q = N;
  while (r+1 < q) {
    int p = (r+q) / 2;
    if (N < p*p) q = p;
    else r = p;
  }
  assert r*r <= N &&
         (r+1)*(r+1)>N;
  return r;
}
```

Bubble Sort Algorithm

```
void bubbleSort(int[] a, int N) {
 for (int j=0; j<N-1; j++)
  for (int i=0; i<N-j-1; i++)
   if (a[i] > a[i+1]) {
     int t = a[i];
     a[i] = a[i+1];
     a[i+1] = t;
   }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Evaluation: Verifying Simple Algorithms

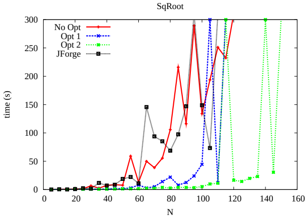
Simple While Loop



Selection Sort Algorithm

```
void selectSort(int[] a, int N) {
 for (int j=0; j<N-1; j++) {
  int min = j;
  for (int i=j+1; i < N; i++)
   if (a[min] > a[i]) min = i;
  int t = a[j]; a[j] = a[min]; a[min] = t;
 }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Integer Square Root Algorithm

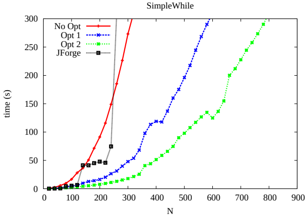


Bubble Sort Algorithm

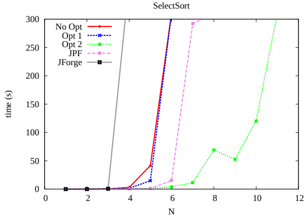
```
void bubbleSort(int[] a, int N) {
 for (int j=0; j<N-1; j++)
  for (int i=0; i<N-j-1; i++)
   if (a[i] > a[i+1]) {
     int t = a[i];
     a[i] = a[i+1];
     a[i+1] = t;
   }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Evaluation: Verifying Simple Algorithms

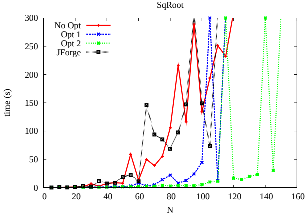
Simple While Loop



Selection Sort Algorithm



Integer Square Root Algorithm

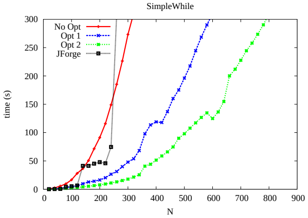


Bubble Sort Algorithm

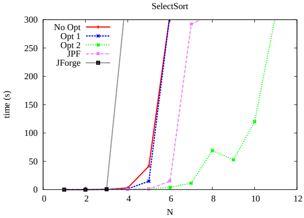
```
void bubbleSort(int[] a, int N) {
 for (int j=0; j<N-1; j++)
  for (int i=0; i<N-j-1; i++)
   if (a[i] > a[i+1]) {
     int t = a[i];
     a[i] = a[i+1];
     a[i+1] = t;
   }
 for (int j=0; j<N-1; j++)
  assert a[j] <= a[j+1];
}
```

Evaluation: Verifying Simple Algorithms

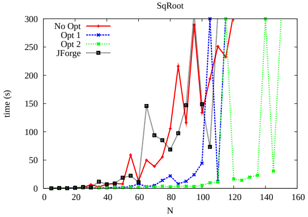
Simple While Loop



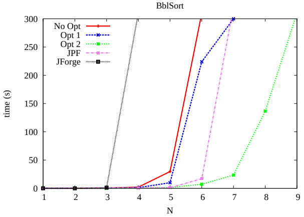
Selection Sort Algorithm



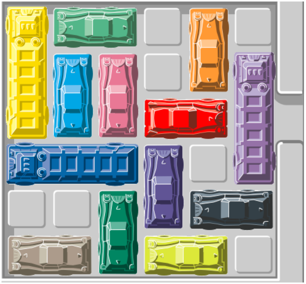
Integer Square Root Algorithm



Bubble Sort Algorithm



Evaluation: Solving the Rush Hour Puzzle



Source:
http://www.puzzles.com/products/rushhour.htm

	Bounded	Using Lists	#steps
Jam 25	1.20s	1.88s	16
Jam 30	1.21s	2.17s	22
Jam 38	4.47s	36.6s	35
Jam 39	1.90s	14.66s	40
Jam 40	6.31s	17.89s	36

bounded: single flat formula, number of steps given up front
using lists: our approach with lists
#steps: min number of steps needed to solve the puzzle

- able to solve all puzzles from • in less than 40 seconds
- limitation: doesn't terminate if puzzle can't be solved
  - possible solution: optimize the solver not to explore same states

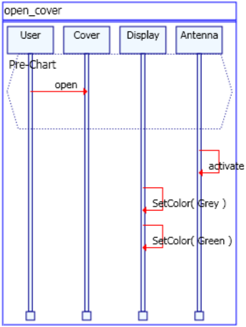
Case Study: Execution of Live Sequence Charts

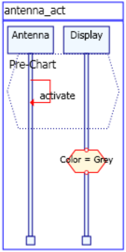
Goal: find valid single and super steps

- single step - a single message that doesn't cause a violation
- super step - a sequence of messages that closes all charts

Approach: formulate as a model-checking problem

- use a list to represent sent messages, and the state after each message
- transition constraint: messages don't cause violations
- safety property: not all charts are closed





Result: incorporated in the *Synthesizing Biological Theories* (SBT) tool [CAV'11]

Summary

- Model-checking technique using SMT Theory of Lists
- Theory of Lists lets you:
  - model unbounded state sequences
  - perform bounded model checking without explicitly bounding the length of counter examples
  - perform software model checking without loop unrolling

Thank You!

Microsoft Research

CSAIL

Related Work

Bounded Model Checking with SMT
(explicit loop unrolling required)

- Armando et al. (STTT 2009)

Unbounded Model Checking with SAT
(multiple invocation of the solver required)

- Kang et al. (DAC 2003), McMillan et al.(CAV 2002)

Bounded Model Checking with SAT
(explicit loop unrolling required)

- CBMC (Clarke04), JForge (Dennis09), Alloy Analyzer (Jackson06)

Explicit State Model Checking

- Java PathFinder (Visser00)

Future Work

Comparison with other tools/approaches

- planning problems: SMT Lists vs Alloy event paradigm
- software model checking: SMT Lists vs unbounded SAT

Optimization of SMT heuristics for theory of lists

- explore implementing fixpoint search inside SMT

Synthesizing Biological Theories

- Try out on more models of biological systems