



# Control-Flow Refinement via Partial Evaluation

**Jesús J. Doménech<sup>1</sup>**

Samir Genaim<sup>1</sup> and John P. Gallagher<sup>2</sup>

<sup>1</sup>Complutense University of Madrid (Spain)

<sup>2</sup>Roskilde University (Denmark)

<sup>2</sup>IMDEA Software Institute (Spain)

**International Workshop on Termination**

Oxford. July, 2018



# CONTROL-FLOW REFINEMENT VIA PARTIAL EVALUATION

---

1. Control-flow Refinement
2. Partial Evaluation
3. Examples
4. Experiments
5. Conclusions and future work



## CONTROL-FLOW REFINEMENT

---

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```



## CONTROL-FLOW REFINEMENT

---

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

- ▶ Two hidden stages.



## CONTROL-FLOW REFINEMENT

---

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.



## CONTROL-FLOW REFINEMENT

---

```
0 // int x, y, z;  
1 while(x > 0)  
2     if (y < z)  
3         y++;  
4     else  
5         x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.
- ▶ Unknown complexity.



## CONTROL-FLOW REFINEMENT

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

```
0 // int x, y, z;  
1 while(x > 0 && y < z)  
2   y++;  
3  
4 while(x > 0 && y >= z)  
5   x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.
- ▶ Unknown complexity.



## CONTROL-FLOW REFINEMENT

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

```
0 // int x, y, z;  
1 while(x > 0 && y < z)  
2   y++;  
3  
4 while(x > 0 && y >= z)  
5   x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.
- ▶ Unknown complexity.
- ▶ Two explicit stages.





## CONTROL-FLOW REFINEMENT

```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

```
0 // int x, y, z;  
1 while(x > 0 && y < z)  
2   y++;  
3  
4 while(x > 0 && y >= z)  
5   x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.
- ▶ Unknown complexity.
- ▶ Two explicit stages.
- ▶ Linear Ranking Function.



## CONTROL-FLOW REFINEMENT

```
0 // int x, y, z;  
1 while(x > 0)  
2   if(y < z)  
3     y++;  
4   else  
5     x--;
```

```
0 // int x, y, z;  
1 while(x > 0 && y < z)  
2   y++;  
3  
4 while(x > 0 && y >= z)  
5   x--;
```

- ▶ Two hidden stages.
- ▶ No Linear Ranking Function.
- ▶ Unknown complexity.
- ▶ Two explicit stages.
- ▶ Linear Ranking Function.
- ▶ Linear complexity.



## CONTROL-FLOW REFINEMENT

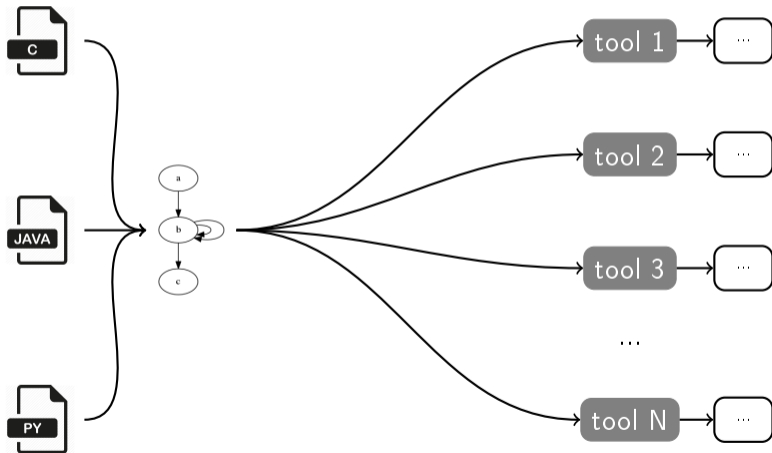
```
0 // int x, y, z;  
1 while(x > 0)  
2   if (y < z)  
3     y++;  
4   else  
5     x--;
```

```
0 // int x, y, z;  
1 while(x > 0 && y < z)  
2   y++;  
3  
4 while(x > 0 && y >= z)  
5   x--;
```

- ▶ S. Gulwani et al. Control-flow refinement and progress invariants for bound analysis.
- ▶ A. Flores-Montoya. Cost Analysis of Programs Based on the Refinement of Cost Relations. Ph.D. thesis

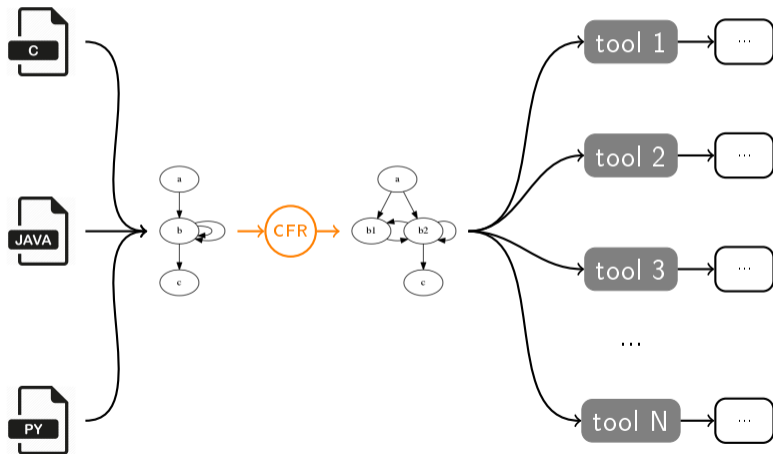


# CONTROL-FLOW REFINEMENT





# CONTROL-FLOW REFINEMENT





## CONTROL-FLOW REFINEMENT

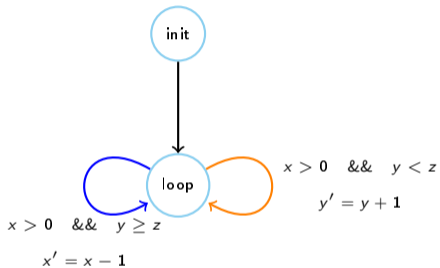
---

```
0 // int x, y, z;  
1 while(x > 0)  
2     if(y < z)  
3         y++;  
4     else  
5         x--;
```



# CONTROL-FLOW REFINEMENT

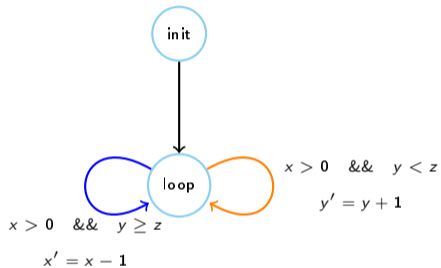
```
0 // int x, y, z;  
1 while(x > 0)  
2   if(y < z)  
3     y++;  
4   else  
5     x--;
```





# CONTROL-FLOW REFINEMENT

$y = 1, z = 3$



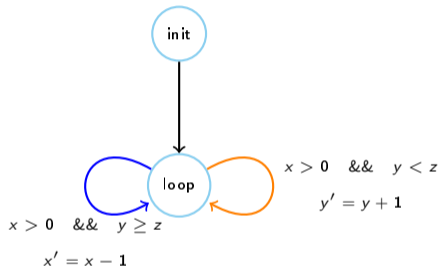




# CONTROL-FLOW REFINEMENT

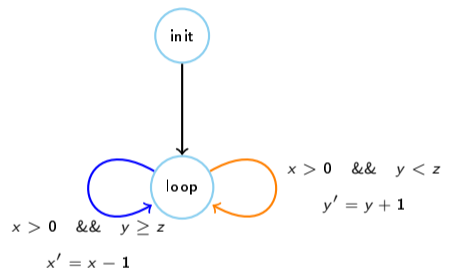
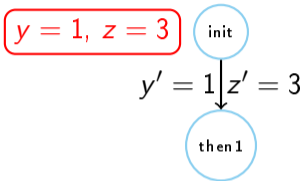
$y = 1, z = 3$

init



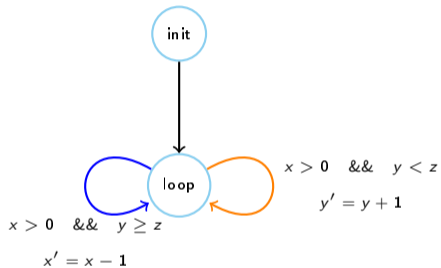
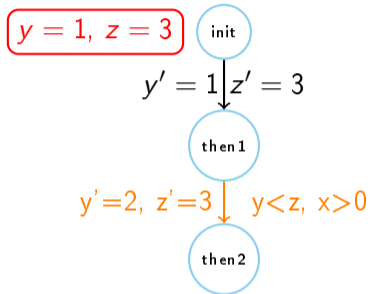


# CONTROL-FLOW REFINEMENT



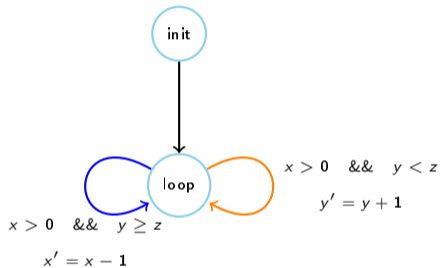
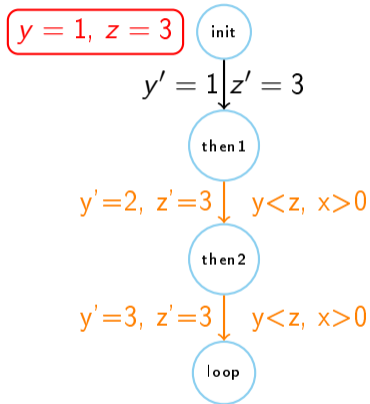


# CONTROL-FLOW REFINEMENT



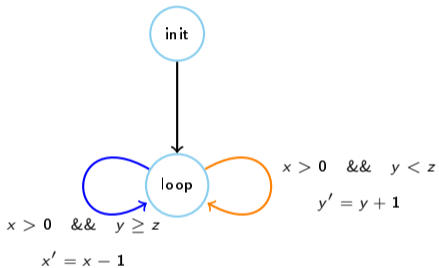
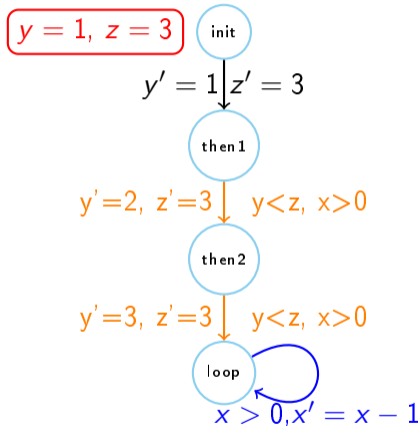


# CONTROL-FLOW REFINEMENT





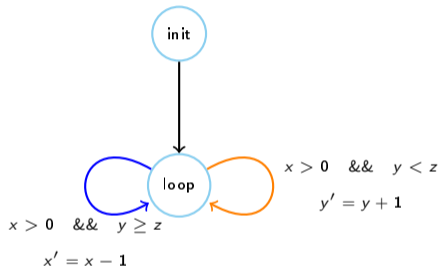
# CONTROL-FLOW REFINEMENT





# CONTROL-FLOW REFINEMENT

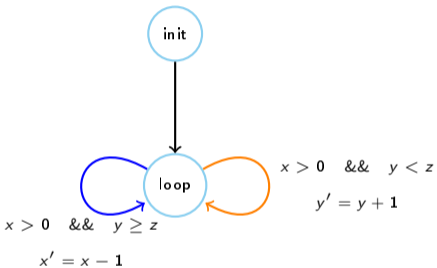
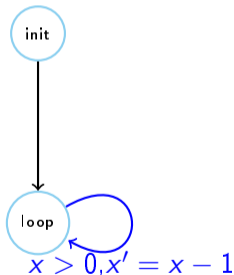
$y = 3, z = 1$





# CONTROL-FLOW REFINEMENT

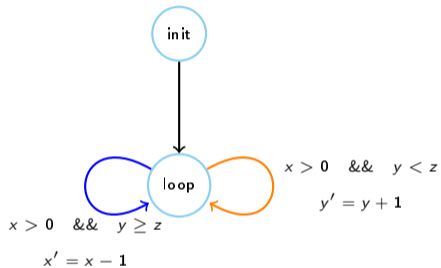
$y = 3, z = 1$





# CONTROL-FLOW REFINEMENT

$y \geq z$

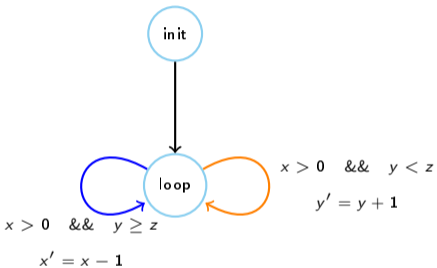
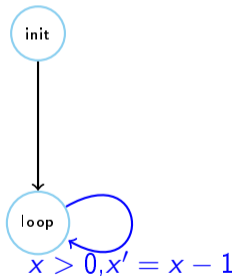






# CONTROL-FLOW REFINEMENT

$y \geq z$





## PARTIAL EVALUATION

---

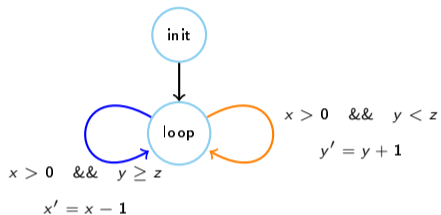
Partial Evaluation (PE) is a program transformation that specializes a program by restricting its meaning in some way.

The PE algorithm used is an instantiation of the described in 1993 by John P. Gallagher. Which has two basic operations:

- ▶ **unFold**: extracts one iteration from a loop.
- ▶ **abstract**: performs a property-based abstraction with a set of properties. So that, each clause is generalized with the subset of properties implied from the clause.

## PARTIAL EVALUATION: EXAMPLE

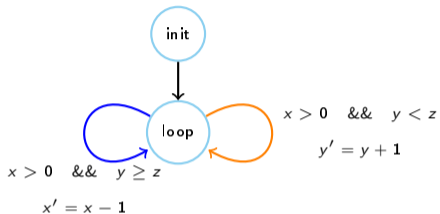
```
0 // int x, y, z;  
1 while(x > 0)  
2   if(y < z)  
3     y++;  
4   else  
5     x--;
```





## PARTIAL EVALUATION: EXAMPLE

```
0 // int x, y, z;  
1 while(x > 0)  
2   if(y < z)  
3     y++;  
4   else  
5     x--;
```

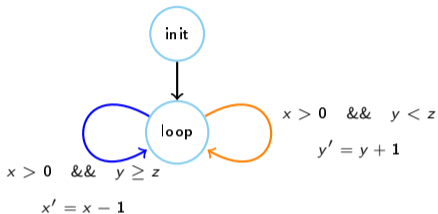


- Properties: any combination of loop conditions, other conditions, body updates...



## PARTIAL EVALUATION: EXAMPLE

```
0 // int x, y, z;  
1 while(x > 0)  
2   if(y < z)  
3     y++;  
4   else  
5     x--;
```



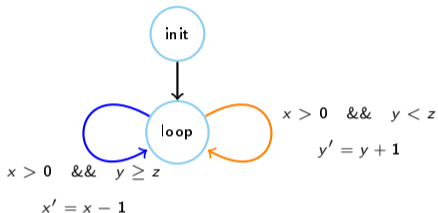
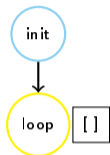
loop:

```
 $\phi_1 = Y \geq Z \ \&\& \ X > 0$   
 $\phi_2 = Y \geq Z$   
 $\phi_3 = X > 0$   
 $\phi_4 = Y \geq Z \ \&\& \ X > -1$   
 $\phi_5 = X > -1$   
 $\phi_6 = Y < Z \ \&\& \ X > 0$   
 $\phi_7 = Y < Z$   
 $\phi_8 = Y + 1 < Z \ \&\& \ X > 0$   
 $\phi_9 = Y + 1 < Z$ 
```

in of loop  
s, body



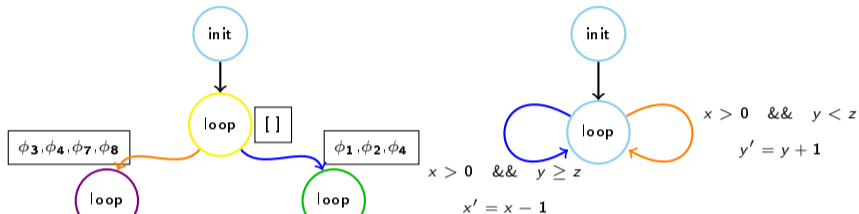
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.



# PARTIAL EVALUATION: EXAMPLE

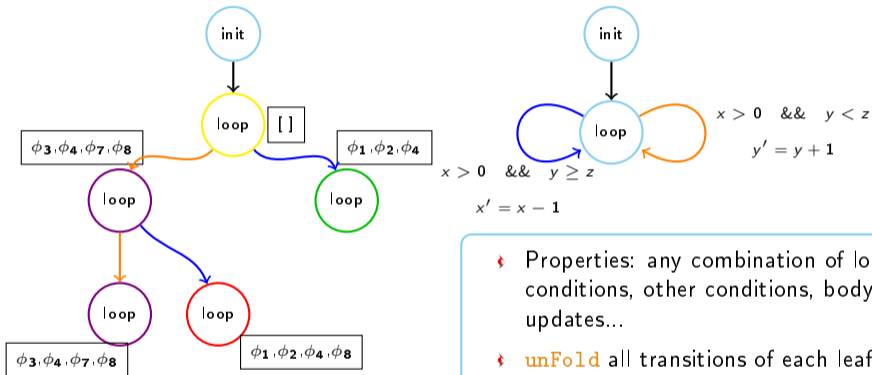


```

loop:
  phi_1 = Y >= Z && X > 0
  phi_2 = Y >= Z
  phi_3 = X > 0
  phi_4 = Y >= Z && X > -1
  phi_5 = X > -1
  phi_6 = Y < Z && X > 0
  phi_7 = Y < Z
  phi_8 = Y + 1 < Z && X > 0
  phi_9 = Y + 1 < Z
  
```



## PARTIAL EVALUATION: EXAMPLE

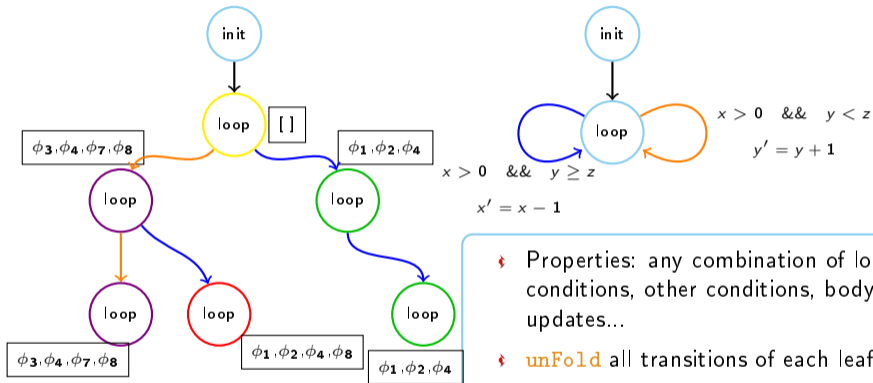


- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.





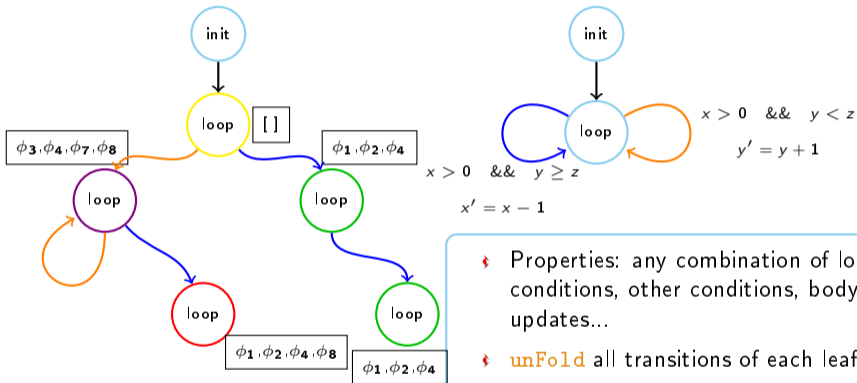
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.



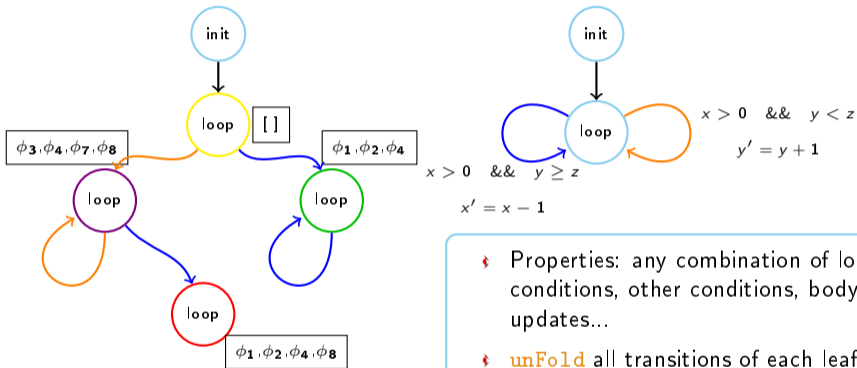
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.



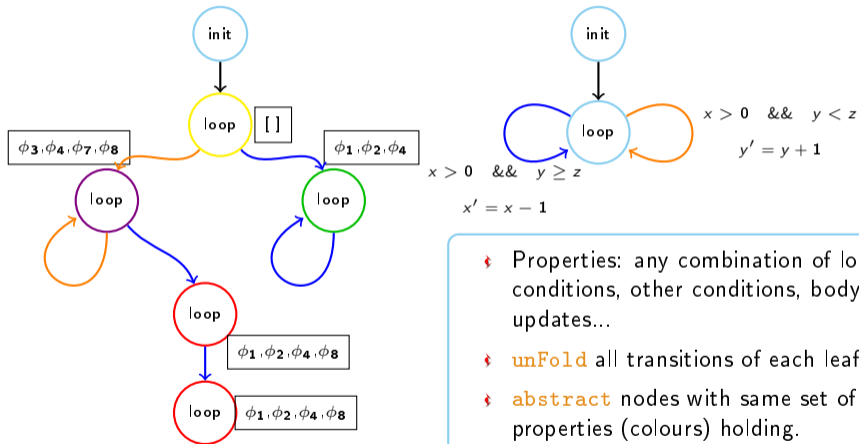
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.



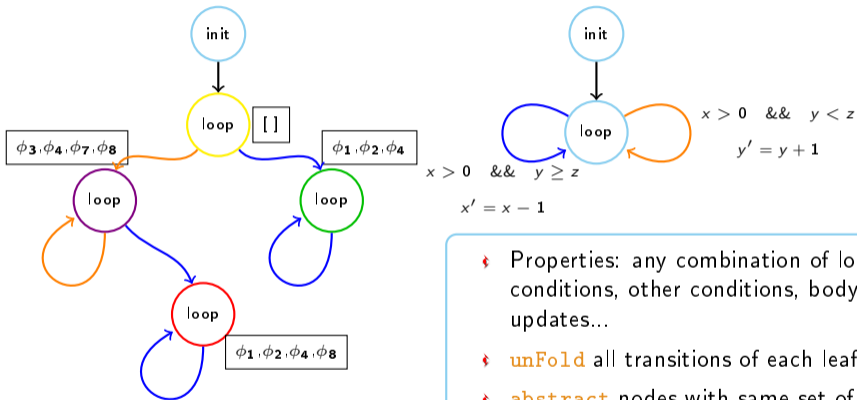
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#\text{properties}}$  possible versions (colours) of each node.



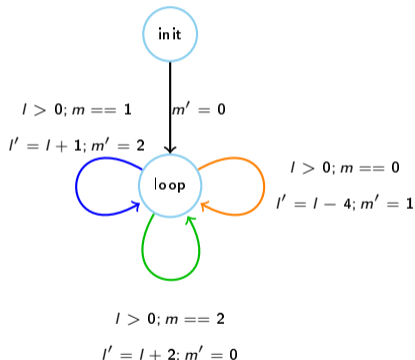
## PARTIAL EVALUATION: EXAMPLE



- ✦ Properties: any combination of loop conditions, other conditions, body updates...
- ✦ **unFold** all transitions of each leaf node.
- ✦ **abstract** nodes with same set of properties (colours) holding.
- ✦  $2^{\#properties}$  possible versions (colours) of each node.

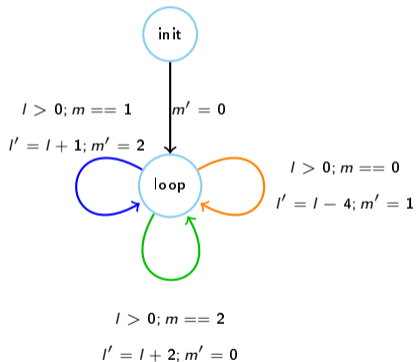
## EXAMPLE: ALTERNATINGGROWREDUCE

```
AlternatingGrowReduce
0 // uint m, l;
1 m=0;
2 while (l > 0){
3   if (m == 0){
4     l = l - 4;
5     m = 1;
6   } else if (m == 1){
7     l = l + 1;
8     m = 2;
9   } else if (m == 2){
10    l = l + 2;
11    m = 0;
12  }
13}
```



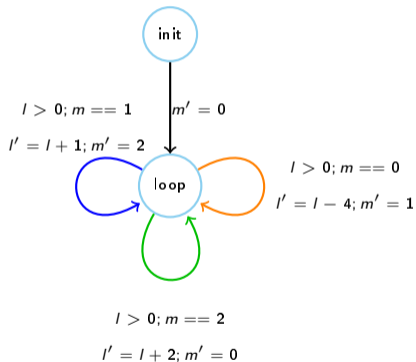
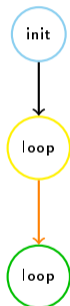


## EXAMPLE: ALTERNATINGGROWREDUCE





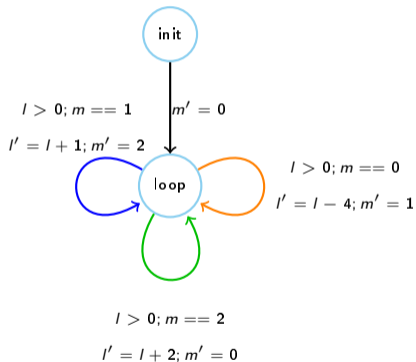
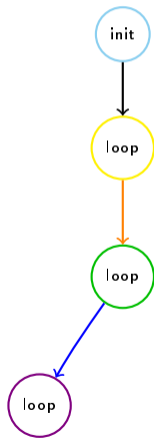
## EXAMPLE: ALTERNATINGGROWREDUCE





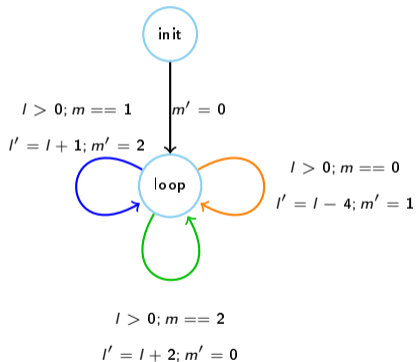
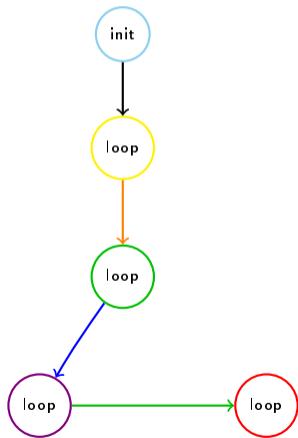


## EXAMPLE: ALTERNATINGGROWREDUCE



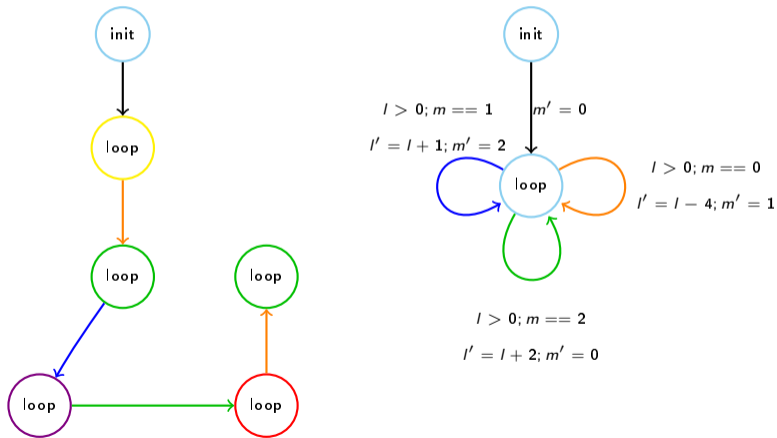


## EXAMPLE: ALTERNATINGGROWREDUCE



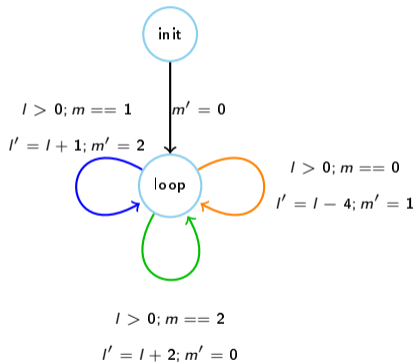
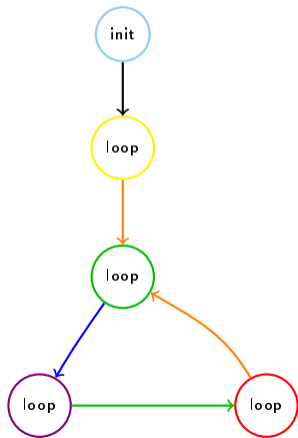


## EXAMPLE: ALTERNATINGGROWREDUCE



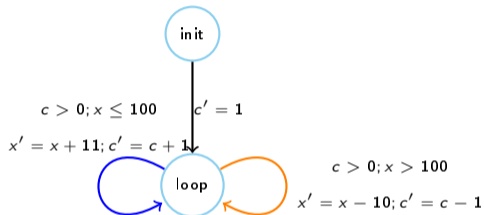


## EXAMPLE: ALTERNATINGGROWREDUCE



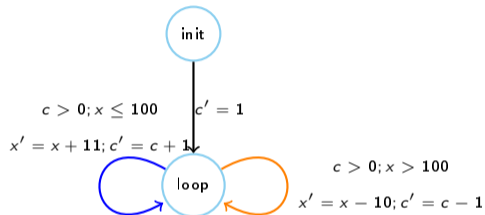
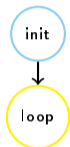
## EXAMPLE: MCCARTHY91

```
McCarthy 91
0 // uint x, c;
1 c = 1;
2 while (c > 0) {
3   if (x > 100) {
4     x = x - 10;
5     c--;
6   } else {
7     x = x + 11;
8     c++;
9   }
10 }
```



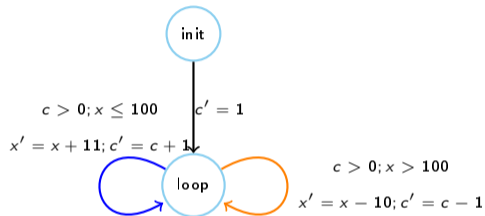
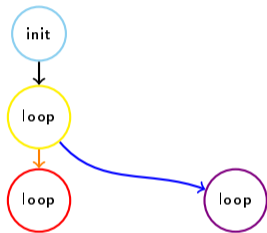


## EXAMPLE: MCCARTHY91



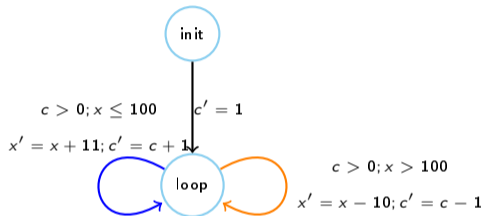
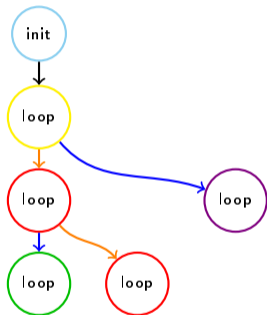


## EXAMPLE: MCCARTHY91





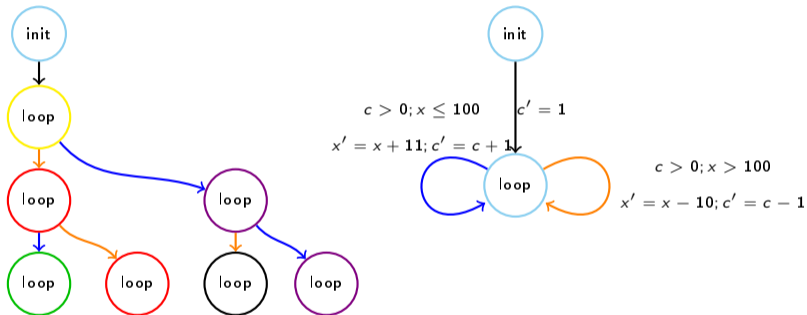
## EXAMPLE: MCCARTHY91





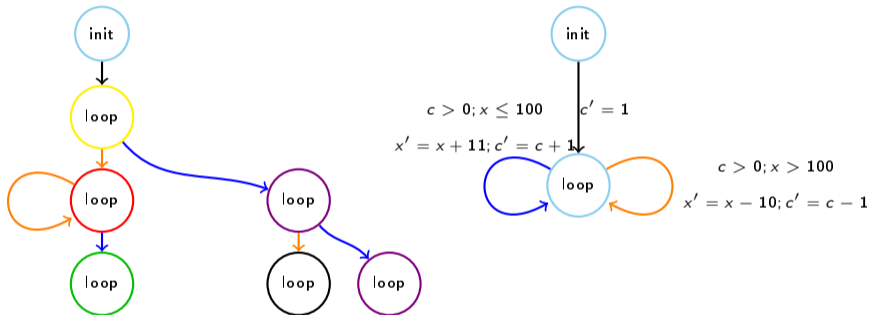


## EXAMPLE: MCCARTHY91



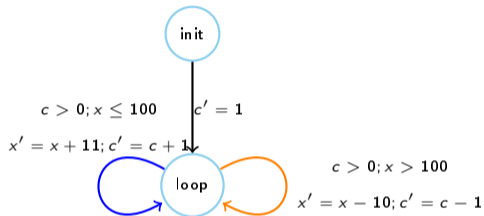
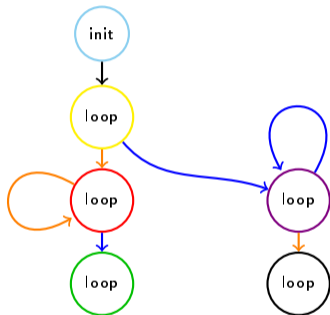


## EXAMPLE: MCCARTHY91



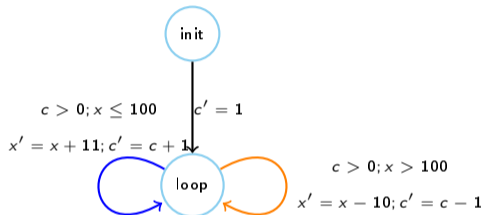
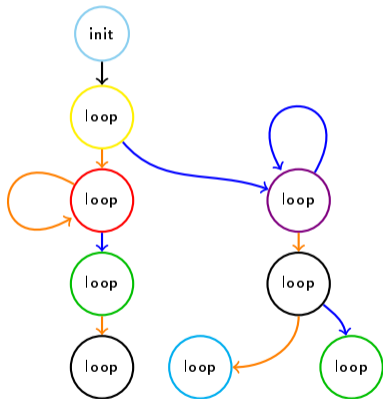


## EXAMPLE: MCCARTHY91



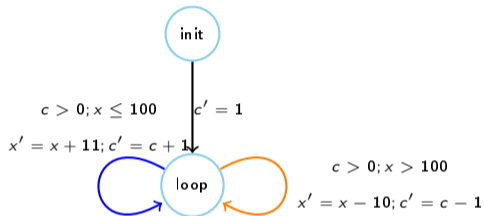
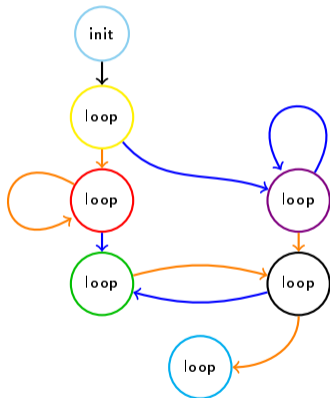


## EXAMPLE: MCCARTHY91



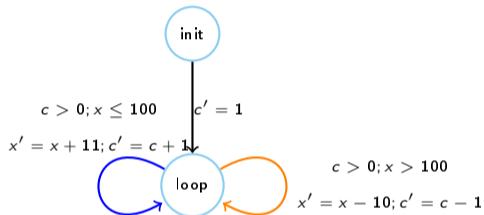
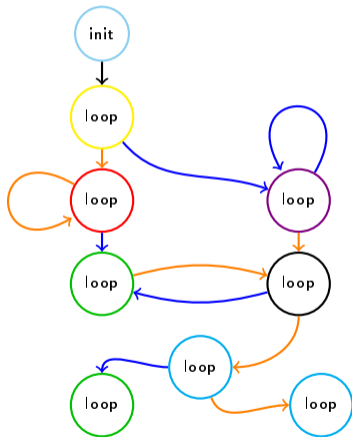


## EXAMPLE: MCCARTHY91



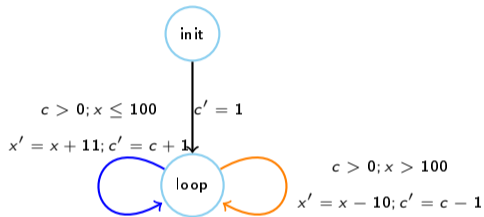
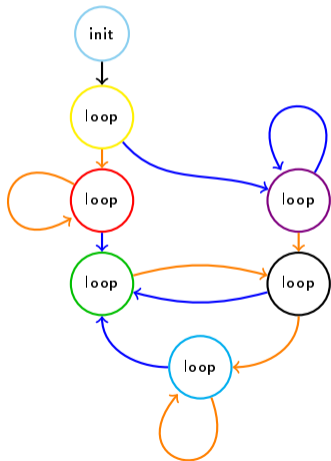


## EXAMPLE: MCCARTHY91





## EXAMPLE: MCCARTHY91





We have compared the result of two tools for different purposes with and without refining the program with Partial Evaluation.

1. Koat: Cost analysis tool for integer transition systems. Marc Brockschmidt et al. *“Analyzing runtime and size complexity of integer programs”*.





We have compared the result of two tools for different purposes with and without refining the program with Partial Evaluation.

1. Koat: Cost analysis tool for integer transition systems. Marc Brockschmidt et al. *“Analyzing runtime and size complexity of integer programs”*.
2. iRankFinder: Our termination analysis tool based on ranking functions as termination arguments.



We have compared the result of two tools for different purposes with and without refining the program with Partial Evaluation.

1. Koat: Cost analysis tool for integer transition systems. Marc Brockschmidt et al. *“Analyzing runtime and size complexity of integer programs”*.
2. iRankFinder: Our termination analysis tool based on ranking functions as termination arguments.

Both tools don't use any control-flow refinement technique.



## EXPERIMENTS: RESULTS

### Termination (iRankFinder)

	Unknown to Termination	Lexicografic to Linear	Timeouts with PE	Linear to Lexicografic
<b>TPDB<sup>a</sup></b>	<b>12</b>	<b>24</b>	<b>9</b>	<b>5</b>
<b>Cofloco<sup>b</sup></b>	<b>8</b>	<b>13</b>	<b>1</b>	<b>2</b>

<sup>a</sup>Termination Problem Database (416 problems)

<sup>b</sup>Cofloco Test Set (188 problems)



## EXPERIMENTS: RESULTS

### Termination (iRankFinder)

	Unknown to Termination	Lexicografic to Linear	Timeouts with PE	Linear to Lexicografic
<b>TPDB<sup>a</sup></b>	<b>12</b>	<b>24</b>	9	5
<b>Cofloco<sup>b</sup></b>	<b>8</b>	<b>13</b>	1	2

### Cost (Koat)

	Unknown to Bounded	Better Complexity Class	Timeouts with PE	Worse Complexity Class
<b>TPDB<sup>a</sup></b>	<b>4</b>	<b>0</b>	2	3
<b>Cofloco<sup>b</sup></b>	<b>10</b>	<b>10</b>	3	3

<sup>a</sup>Termination Problem Database (416 problems)

<sup>b</sup>Cofloco Test Set (188 problems)



## CONCLUSIONS

---

- ▶ We have explored the use of PE as a control-flow refinement technique in the context of termination and cost analysis.
- ▶ Preliminary experiments show that PE improves both analyses.
- ▶ iRankFinder with PE integrated is available at:  
<http://irankfinder.loopkiller.com>



## FUTURE WORK

---

- ▶ To study several cases where the results are worse:
  - ▶ identifying the source for this imprecision.
- ▶ To specialize the PE tool we use to the context of termination and cost analysis.
- ▶ To measure the effect of using PE on performance:
  - ▶ exploring the possibility of applying PE only on parts of the CFG.
- ▶ To measure the effect of using PE on other termination and cost analysis tools.
- ▶ To explore the use of PE for inferring lower-bounds.