Towards an Approach for Translation Validation of Thread-Level Parallelizing Transformations using Colored Petri Nets

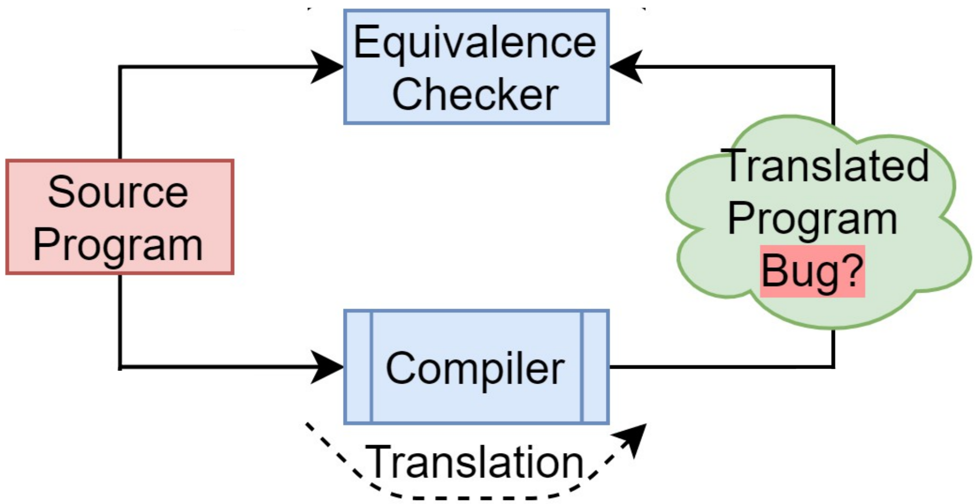
[1,2]Rakshit Mittal, [2]Rochishnu Banerjee, [1,3]Dominique Blouin, [2,3]Soumyadip Bandyopadhyay

[1]Telecom Paris, Institut Polytechnique de Paris, France

[2]Birla Institute of Technology & Science Pilani, Goa, India

[3]Hasso Plattner Institut, Potsdam, Germany

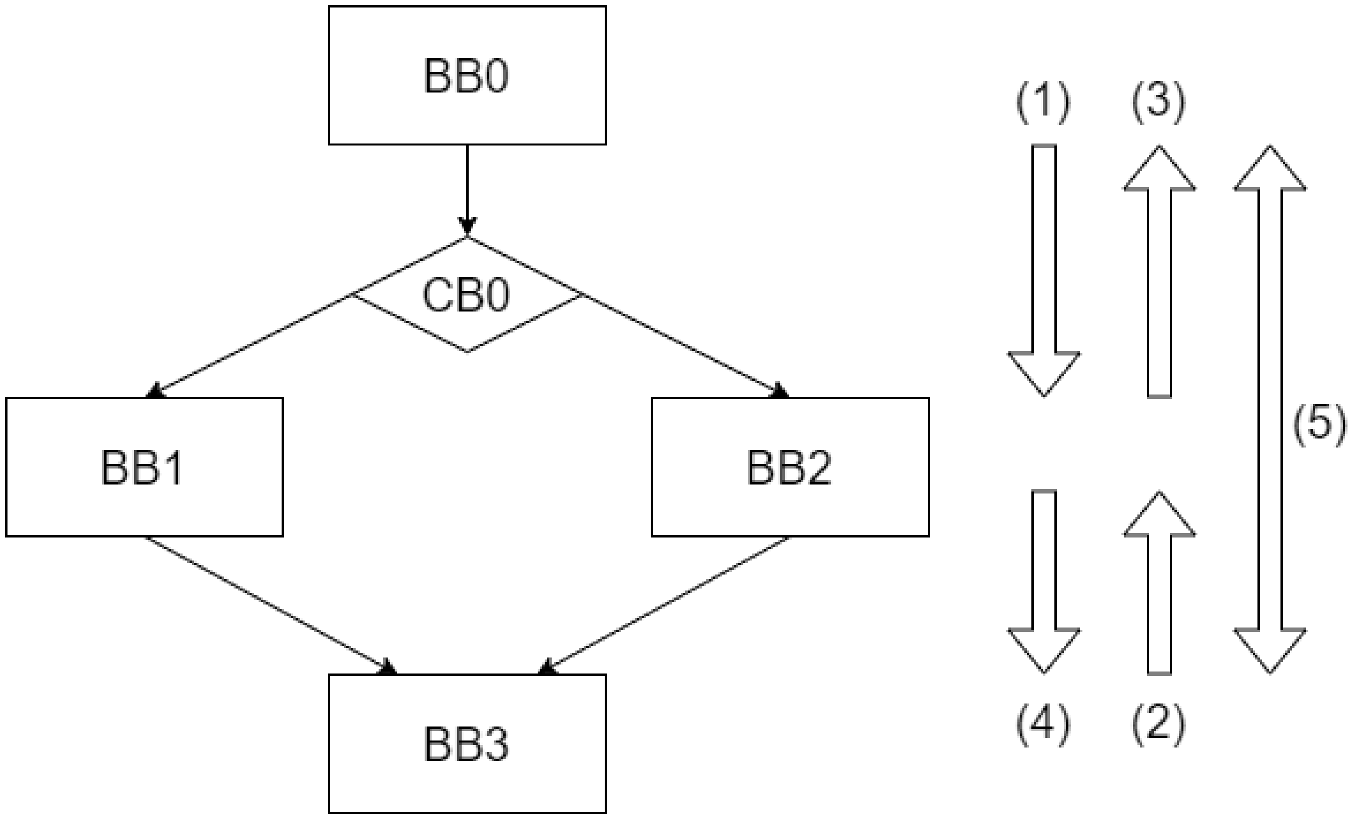
Background



Q. Is the compiler translation correct?
Q. Are there any bugs in the translated program?
Q. Are the source and translated program equivalent?

Background

Optimizing Transformations



1) Duplicating down
2) Duplicating up
3) Boosting up
4) Boosting down
5) Useful Move

Bacon et al, "Compiler Transformations for High-Performance Computing, ACM Computing Survey, 1994

State-of-the-Art

Equivalence Checking

- **Bisimulation-Based Methods**
  - Proposed by Amir Pnueli [TACAS 1998]
  - Enhanced by Necula et al [PLDI 2000]
    and Rinard et al [MIT 2000]
  - Modified by Kundu et al [CAV 2008]

- **Inductive Inference-Based Methods** {Only scalar-handling problems}
  - Matthias et al [ASE 2014]

{Termination not guaranteed}

State-of-the-Art

Equivalence Checking

- **CDFG Path-Based Methods**
  - Karfa et al verify transformations of the SPARK compiler, control structure of program altered [TCAD 2012]
  - Modified by Banerjee et al [TCAD 2014] (value-propagation) and Chouksey et al [TCAD 2019] (extended value-propagation)

- **Petri Net Path-Based Methods**
  - SamaTulyata [ATVA 2017] {Larger model size}
  - SamaTulyata2 [PNSE 2020] {Large model size}

State-of-the-Art

CDFG path-based methods

```
int i = 1, j = 1;
int k;

while (i*7 <= 100)
   {i++;}

while ((j+1)*11 <= 100)
   {j++;}

k = i + j;
```

Source Program

```
int i = 1, j = 1;
int k;

while ((j+1)*11 <= 100)
   {j++;}

while (i*7 <= 100)
   {i++;}

k = i + j;
```

Loop Shifting

```
int i = 1,j;
int k; j = i;

#parbegin scop
while ((j+1)*11 <= 100)
   {j++;}
||
while (i*7 <= 100)
   {i++;}
#parend scop

k = i + j;
```

Parallelizing

[CDFG-based methods fail]

State-of-the-Art

Petri Net path-based methods

```
int i=0,a,b,c,d,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);

while( i < l ) {
  m = m * 10;
  n = n / 10;
  i++;
}

c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b)
e = c / d;
k = m + n + e;
```

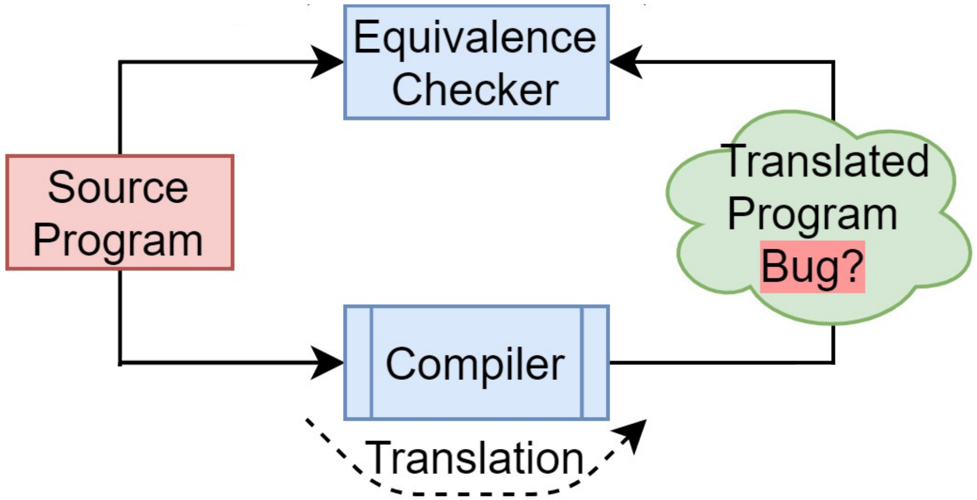
Source Program

```
int i = j = 0,a,b,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);

#parbegin scop
while( i < l ) {
  m = m * 10;
  i++;
}
||
while( j < l ) {
n = n / 10;
j++;}
#parend scop

e = a - b;
k = m + n + e;
```

Arithmetic optimization
+ Parallelizing

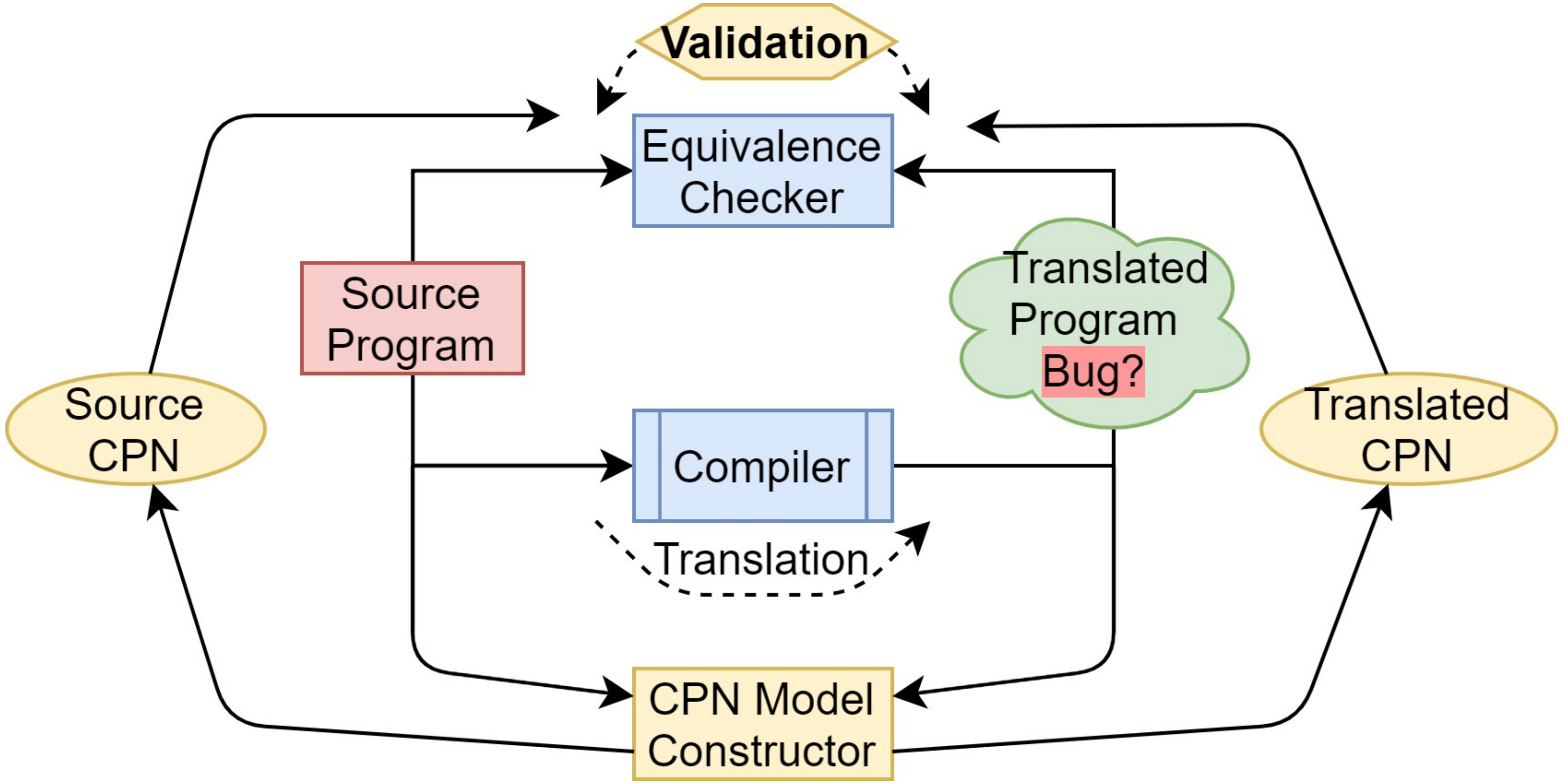
[SamaTulyata, SamaTulyata2 fail]

Background



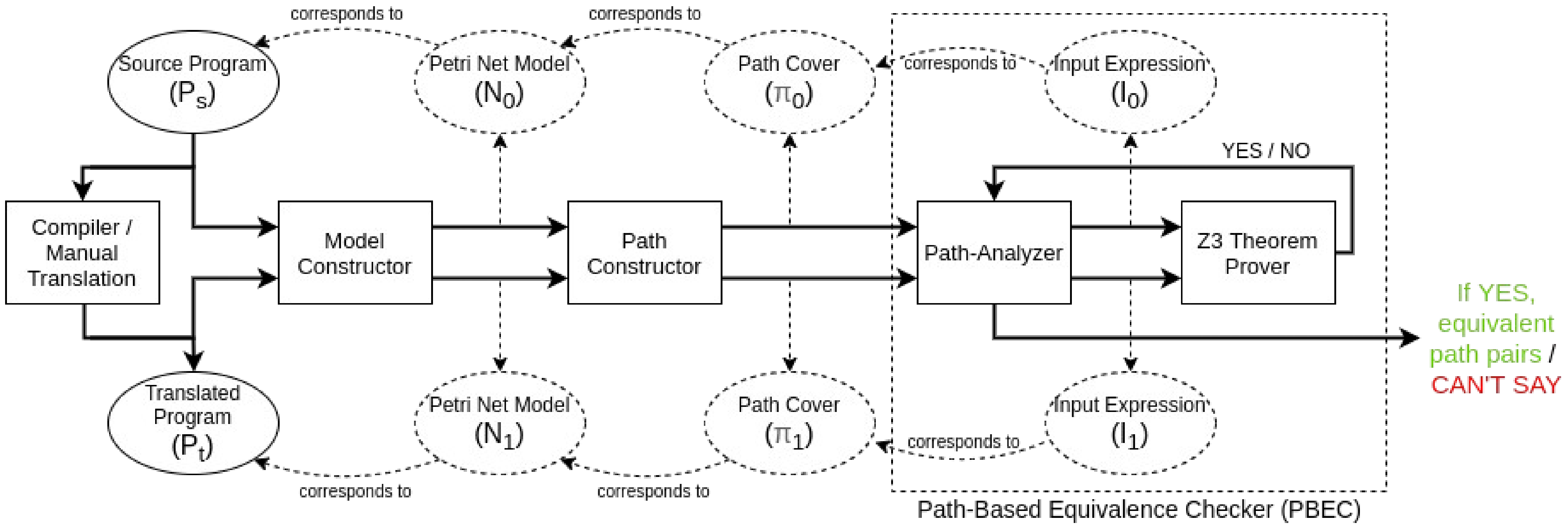
Q. Is the compiler transformation correct?
Q. Are there any bugs in the translated program?
Q. Are the source and translated program equivalent?

Motivation



Q. Are the source and translated CPN Equivalent?

Proposed Toolchain



Cut-point: In-port / Out-port / Place with back-edge

Path Construction: From cut-point to cut-point

Example

```
int i=0,a,b,c,d,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);

while( i < l ) {
  m = m * 10;
  n = n / 10;
  i++;
}

c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b)
e = c / d;
k = m + n + e;
```

Source Program

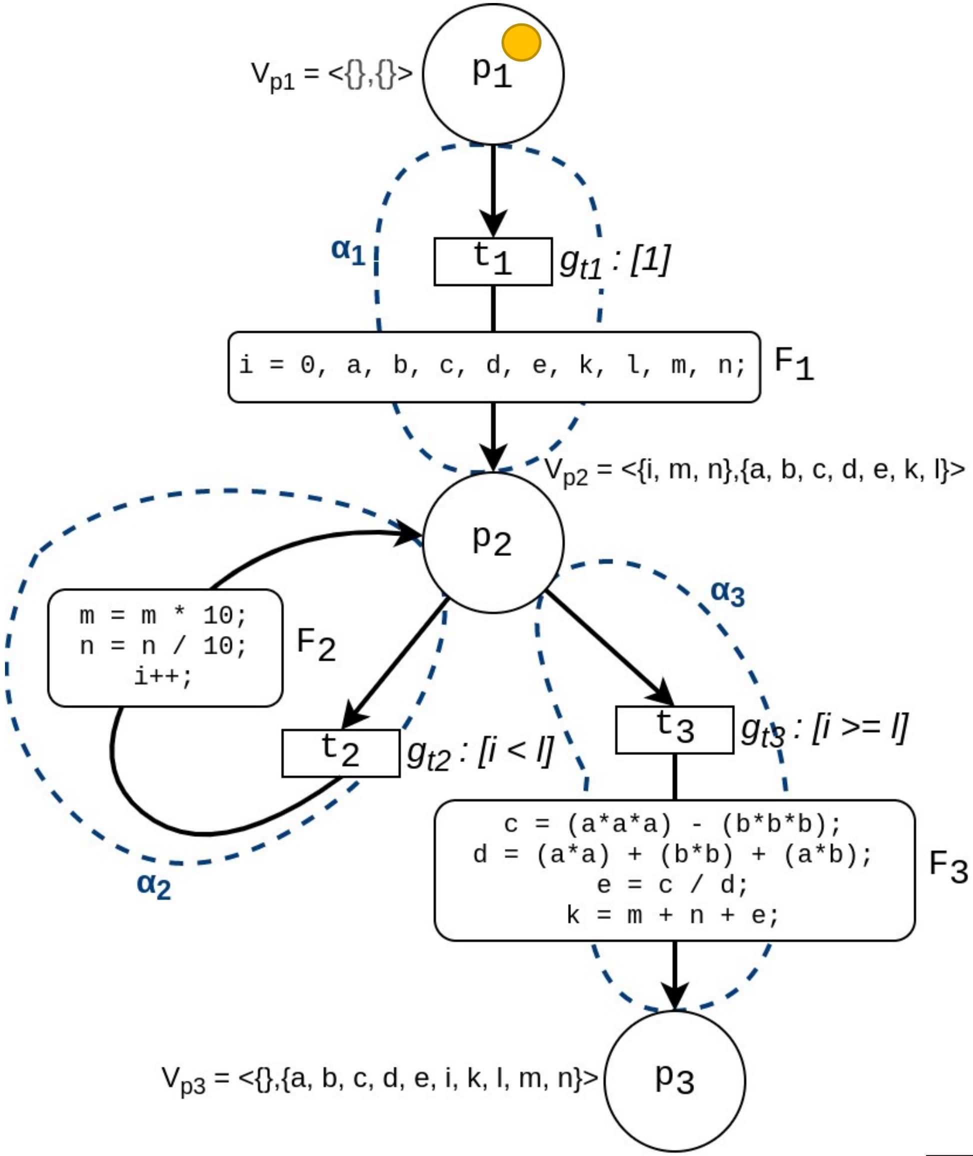
```
int i = j = 0,a,b,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);

#parbegin scop
while( i < l ) {
  m = m * 10;
  i++;
}
||
while( j < l ) {
n = n / 10;
j++;}
#parend scop

e = a - b;
k = m + n + e;
```

Translated Program

Source Petri Net

```
int i=0,a,b,c,d,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);

while( i < l ) {
  m = m * 10;
  n = n / 10;
  i++;
}

c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b)
e = c / d;
k = m + n + e;
```

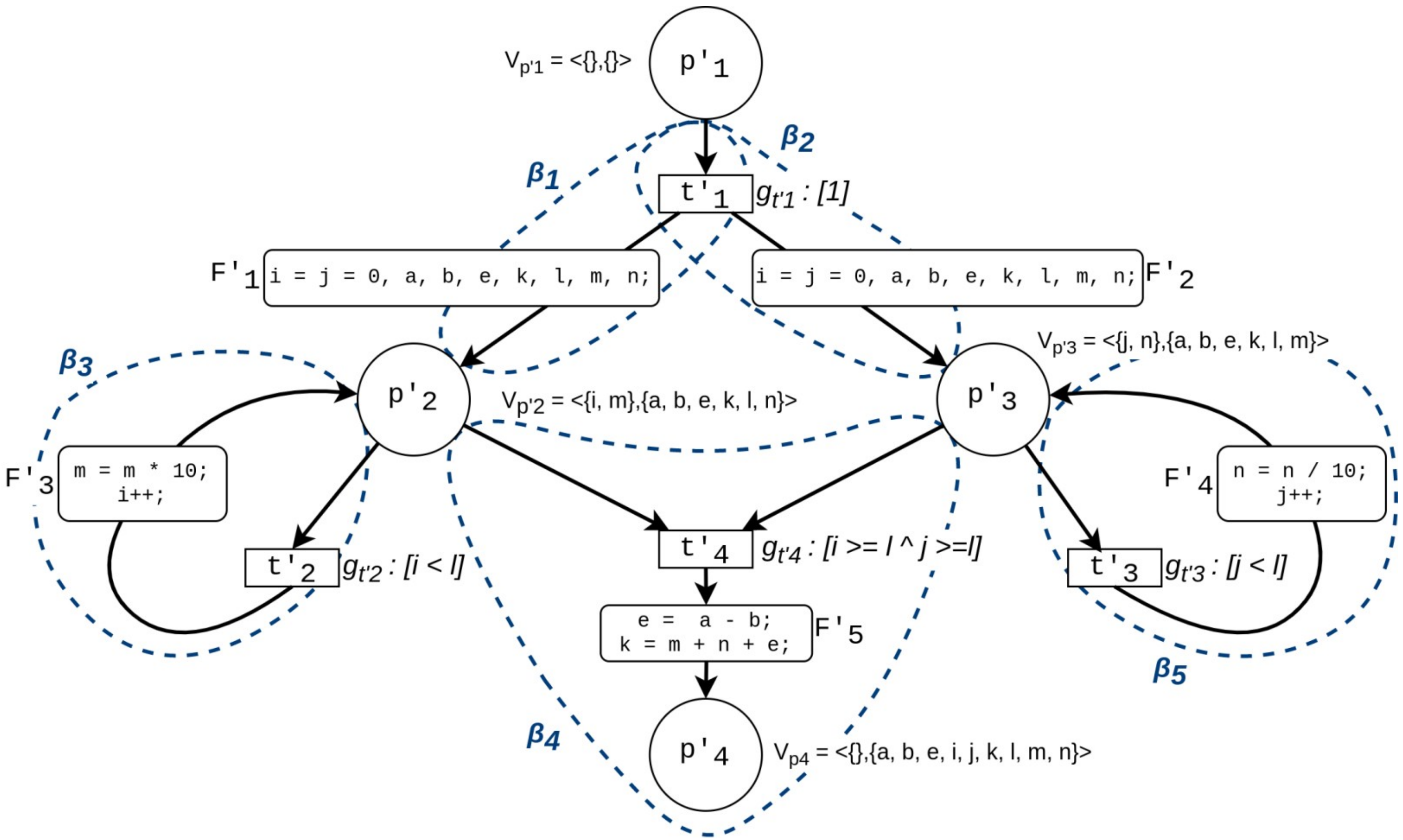
Source Program



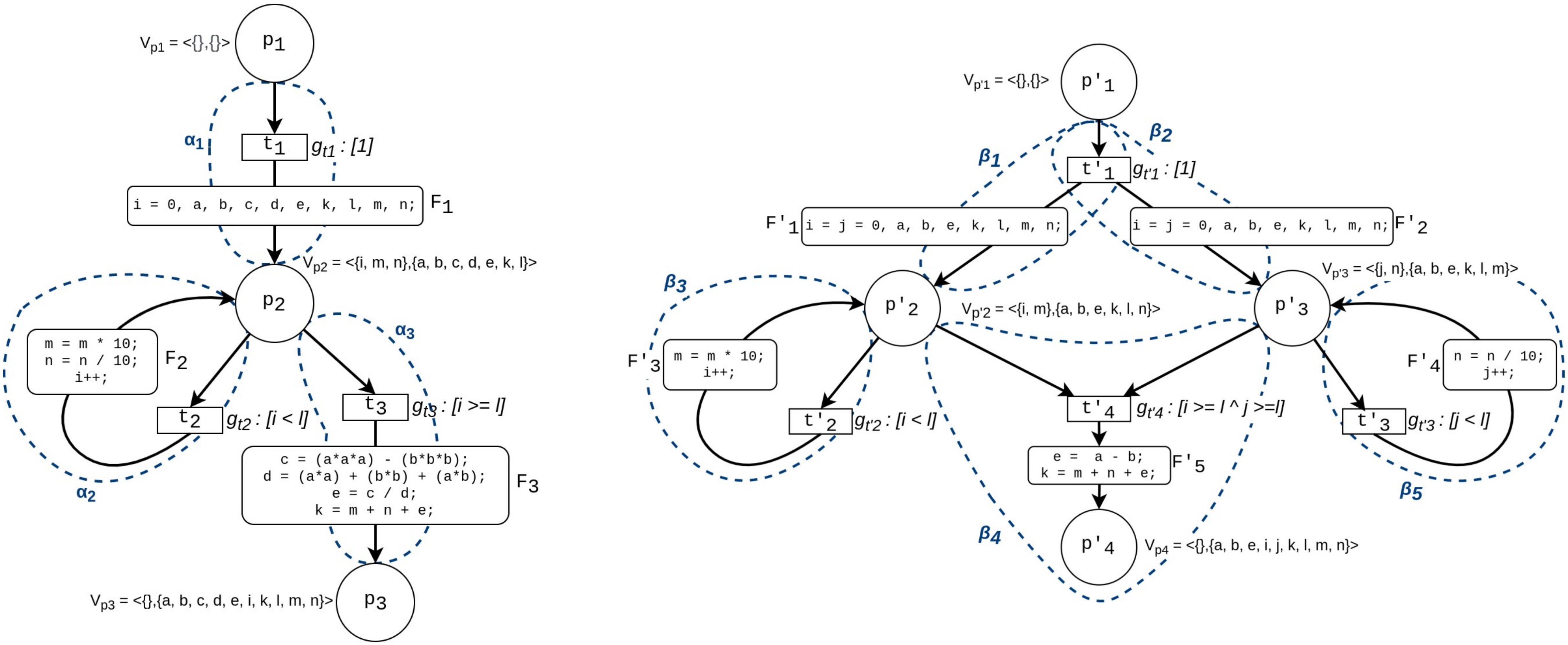
Translated Petri Net

```
int i = j = 0,a,b,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
          &a,&b,&l,&m,&n);

#parbegin scop
while( i < l ) {
  m = m * 10;
  i++;
}
||
while( j < l ) {
n = n / 10;
j++;}
#parend scop

e = a - b;
k = m + n + e;
```

Translated Program

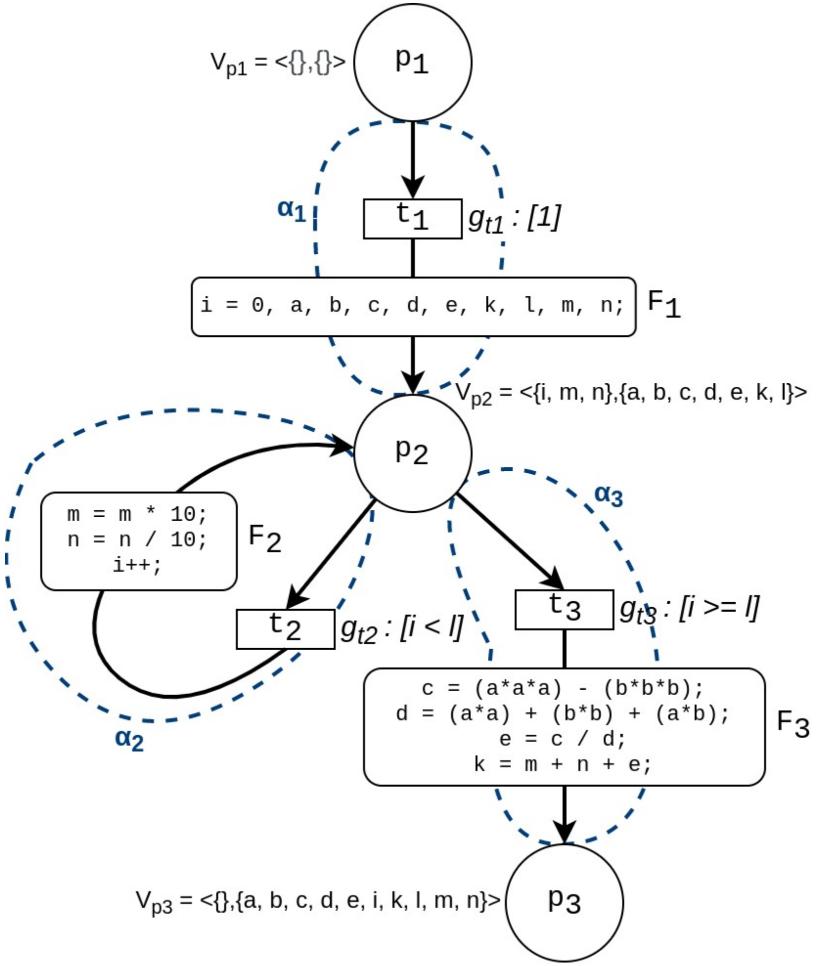


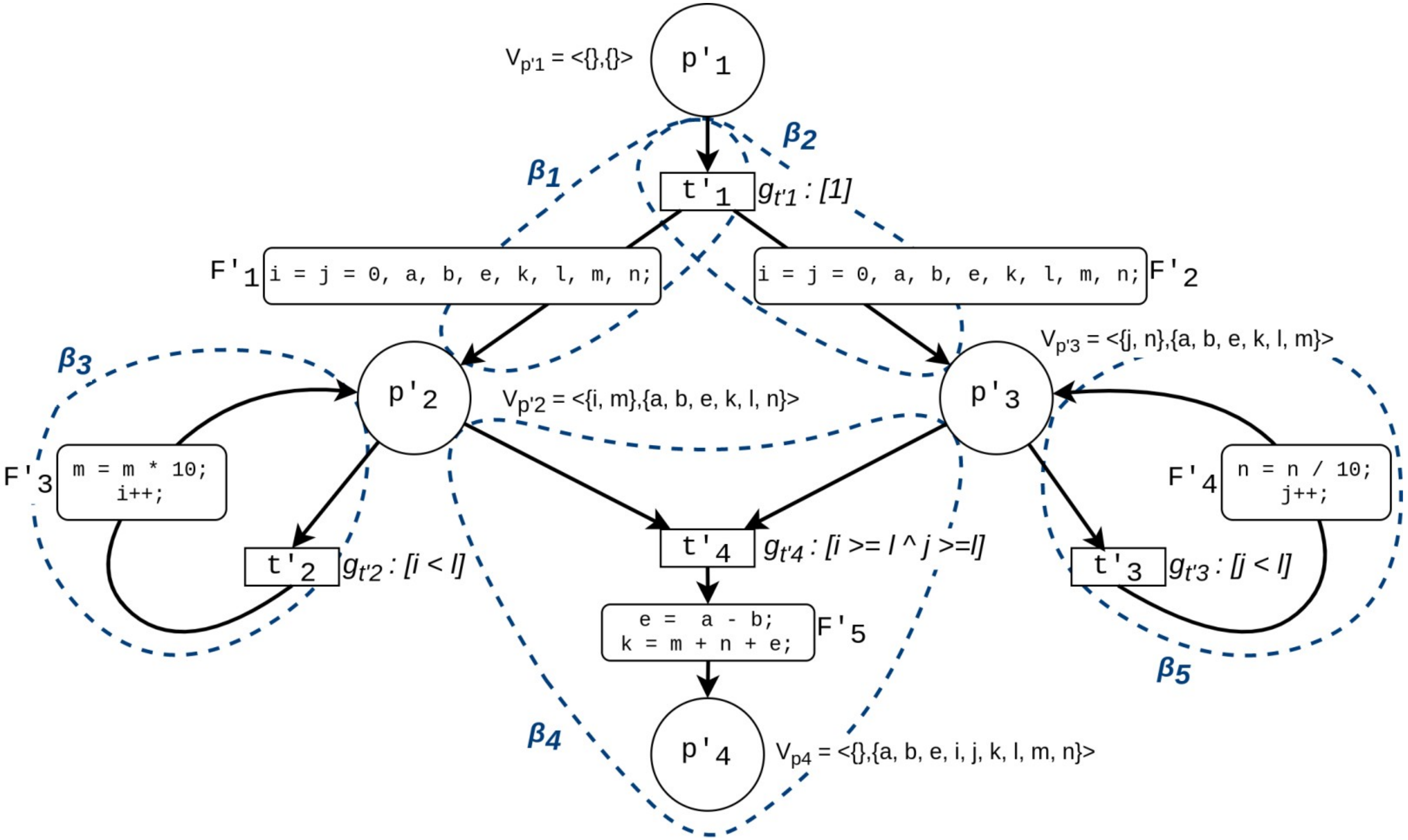
Equivalence Checking Principle



For all paths in N_1 there exists path in N_0 such that the paths are equivalent and vice versa, implies $N_0 \simeq N_1$

Path Analyzer + Z3 Theorem Prover

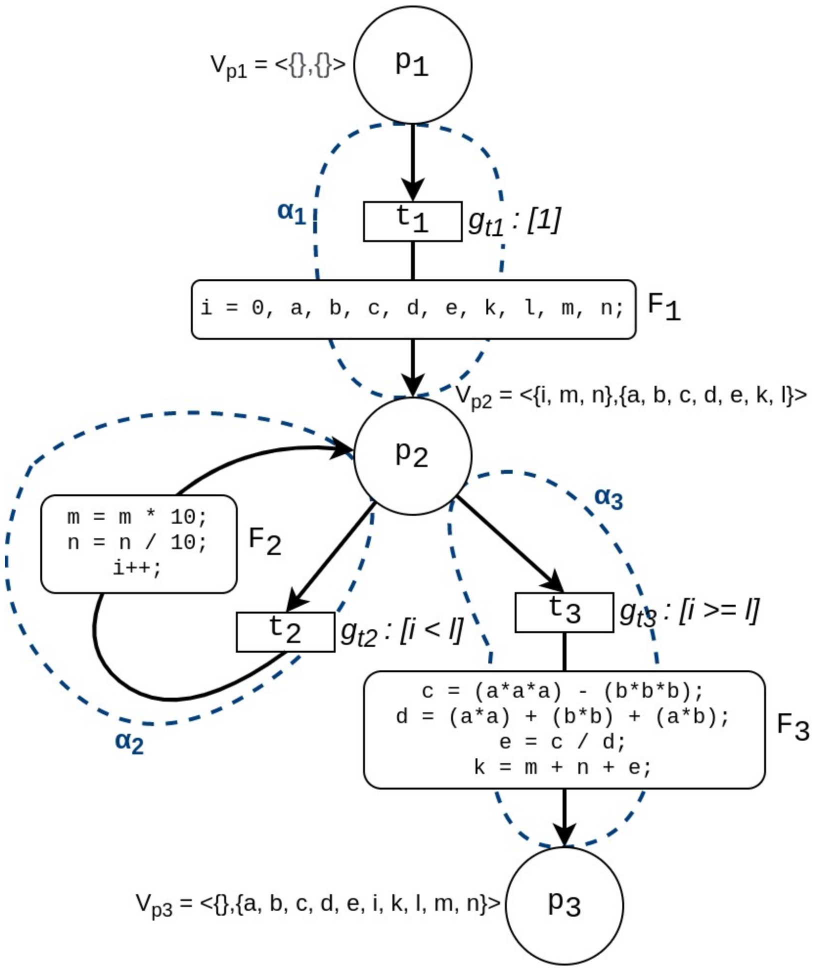


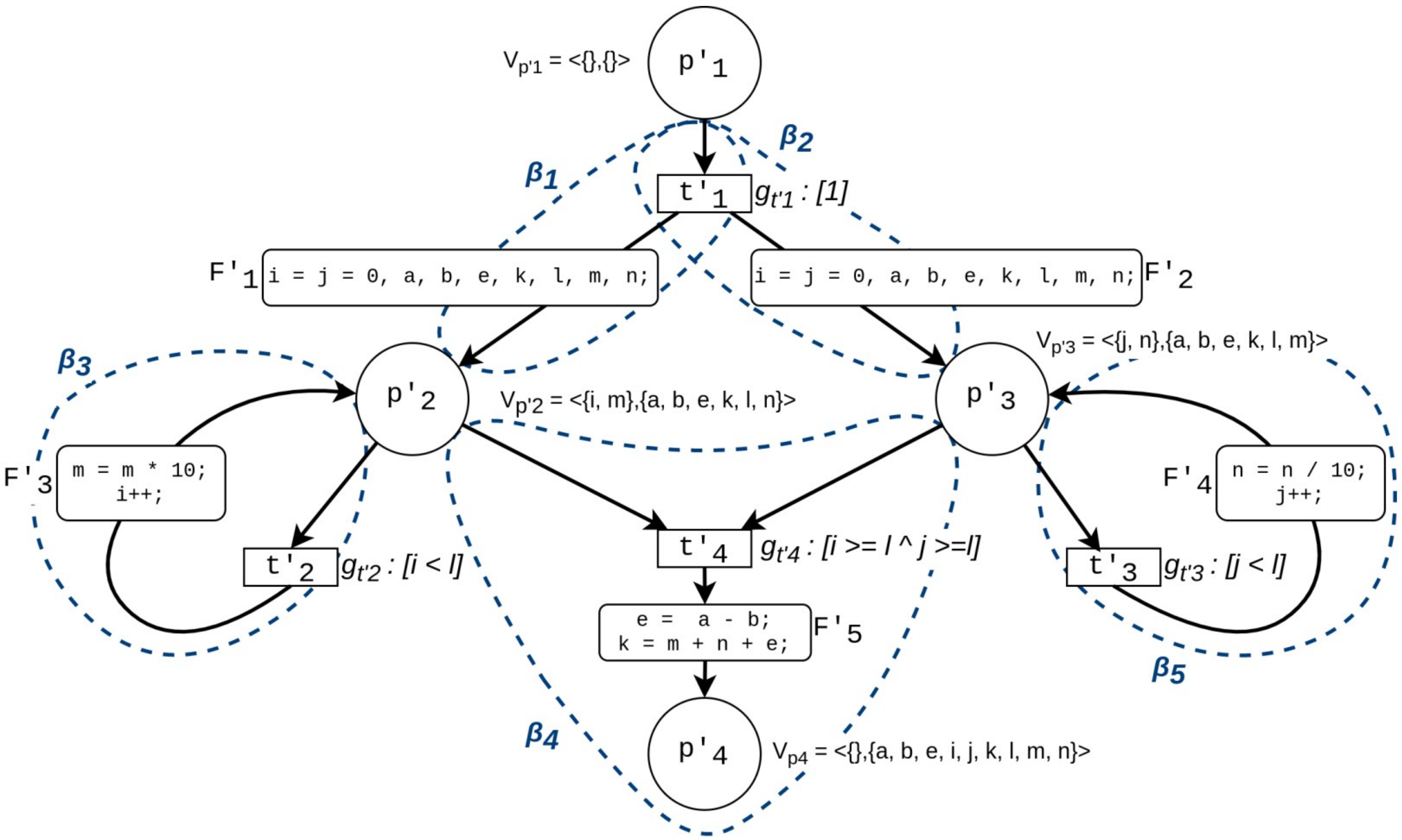


R_{path} : condition of execution

r_{path} : data transformation

Path Analyzer + Z3 Theorem Prover





$$\beta_1 \simeq \alpha_1 \; ; \quad \beta_2 \simeq \alpha_1 \; ; \quad (\beta_3 \,\|\, \beta_5) \simeq \alpha_2 \; ; \quad \beta_4 \simeq \alpha_3$$

Experimentation

Model Size Comparison

Example	ST-1		ST-2		Proposed	
	p	t	p	t	p	t
BCM	34	28	6	6	3	2
MINMAX	31	27	7	7	4	6
PETERSON	11	9	4	2	6	8
DEKKERS	19	14	6	4	6	8
LUP	28	21	6	4	10	16

Equivalence Checking Capability

Example	FSMD-VP	FSMD-EVP	ST-1	ST-2	Proposed
BCM	X	X	X	X	✓
MINMAX	X	X	✓	✓	✓
PETERSON	X	X	X	X	✓
DEKKERS	X	X	X	X	✓
LUP	X	X	✓	✓	✓

Limits and Capabilites

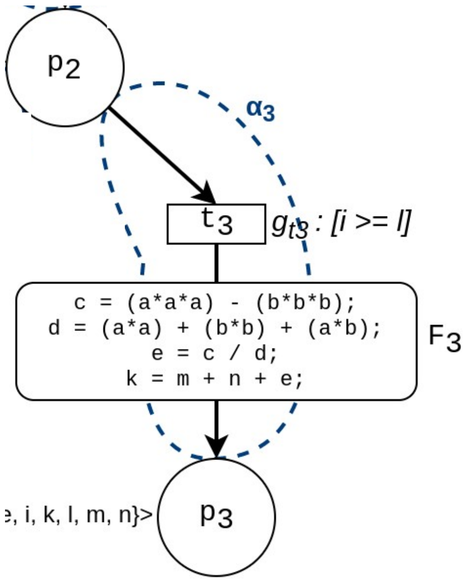
Cannot validate

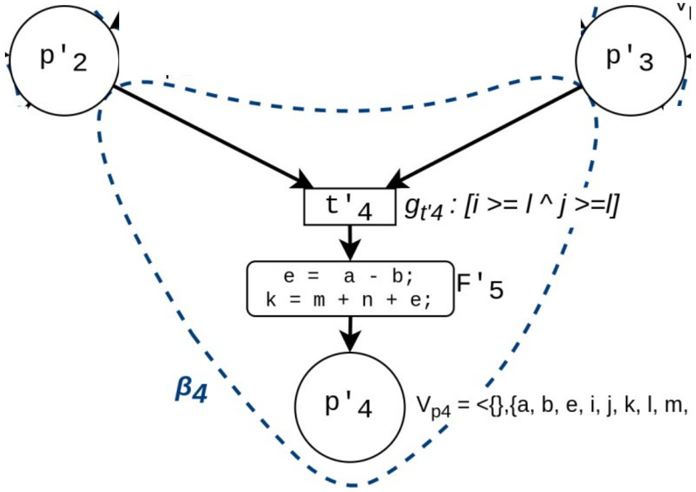
Array-handling programs
Software pipelining transformations
Loop reversal transformations
Invariant assertions based transformations

Can validate

Uniform and non-uniform code transformations
Loop swapping transformation
Thread-level parallelizing transformations
for scalar-handling programs

Z3 SMT Solver






```
(declare-const a_s Int)
(declare-const a_t Int)
(declare-const b_s Int)
(declare-const b_t Int)
(declare-const c_s Int)
(declare-const d_s Int)
(declare-const e_s Int)
(declare-const e_t Int)
(declare-const k_s Int)
(declare-const k_t Int)
(declare-const m_1_s Int)
(declare-const m_1_t Int)
(declare-const n_1_s Int)
(declare-const n_1_t Int)
(assert (= a_s a_t))
(assert (= b_s b_t))
(assert (= m_1_s m_1_t))
(assert (= n_1_s n_1_t))
(assert (= c_s (-(* a_s (* a_s a_s))
        (* b_s (* b_s b_s))))))
(assert (= d_s (+(* a_s a_s) (+(* b_s b_s)
        (* a_s b_s))))))
(assert (= e_s (div c_s d_s)))
(assert (= k_s (+ m_1_s (+ n_1_s e_s))))
(assert (= e_t (+ a_t b_t)))
(assert (= k_t (+ m_1_t (+ n_1_t e_t))))
(assert (not (and (= a_s a_t)
        (and (= b_s b_t) (and (= m_1_s m_1_t)
        (and (= e_s e_t) (and (= n_1_s n_1_t)
        (= k_s k_t))))))))
(check-sat)
```

Listing 4: Checking equivalence of r_{α_3} and r_{β_4}

```
(declare-const g_t3_s Bool)
(declare-const g_t4_t Bool)
(declare-const i_0_s Int)
(declare-const i_0_t Int)
(declare-const j_0_t Int)
(declare-const l_s Int)
(declare-const l_t Int)
(assert (= g_t3_s(>= i_0_s l_s)))
(assert (= g_t4_t(and(>= i_0_t l_t)
                (>= j_0_t l_t))))
(assert (= l_s l_t))
(assert (= i_0_s i_0_t))
(assert (= i_0_t j_0_t))
(assert (not(= g_t3_s g_t4_t)))
(check-sat)
```

Listing 3: Checking equivalence of R_{α_3} and R_{β_4}

Thank You
Questions?

e-mail: rakshit.mittal@telecom-paris.fr (myself)
f2018040@goa.bits-pilani.ac.in (Rochishnu)
dominique.blouin@telecom-paris.fr (Dominique)
soumyadipb@goa.bits-pilani.ac.in (Soumyadip)

or find me and Soumyadip on Slack!