Compiler-agnostic Translation Validation

Kunal Banerjee[1]
Chandan Karfa[2]

[1]Intel Labs, Bangalore, India
[2]IIT Guwahati, India

Outline

1. Background
2. Translation validation of code motion transformations
3. Deriving bisimulation relations from path based equivalence checkers
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
6. Other Models of Computation
7. Conclusion and scope of future work

Model Checking

Example Tool: CBMC (Bounded Model Checking over C language)

Example without loop

```
int x, y;
y = 8;
if (x) {
  y--;
} else {
  y++;
}
assert (y == 7 || y == 9); //Always true
```

Model Checking

Example with bounded loop

```
int x, y;
x = 3;
y = 8;
while (x > 0) {
  y--;
  x--;
}
assert (y <= 7 && y >= 0); //true or false?
```

Model Checking

Example with bounded loop

```
int x, y;
x = 3;
y = 8;
if (x > 0) {
  y--; //y == 7
  x--; //x == 2
  if (x > 0) {
    y--; //y == 6
    x--; //x == 1
      if (x > 0) {
      y--; //y == 5
      x--; //x == 0
      assert (!(x > 0)); //unrolled enough
} } }
assert (y <= 7 && y >= 0); //always true
```

Model Checking

Example with (un)bounded loop

```
while (cond) {
  Body;
}
Remainder;
```

```
if (cond) { //first unwind
  Body;
  if (cond) { //second unwind
    Body;
    assert (!cond); //unwinding
assertion
  }
}
Remainder;
```

☞ After k-unwindings, the inner-most "while" loop is either removed or replaced by an unwinding assertion.

☞ If the loop can be executed more than k times, then unwinding assertion will throw an error.

☞ CBMC allows defining different unwind bounds for different loops, allowing the analysis to be more efficient, but requiring more work from the programmer.

Outline

1. **Background**
2. Translation validation of code motion transformations
3. Deriving bisimulation relations from path based equivalence checkers
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. Other Models of Computation
7. Conclusion and scope of future work

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)
Even if we write (syntactically and semantically) correct programs, they may not be "good" programs.

Example of code optimization

```
for(i = 1; i <= N; i++) {
  if( x > 10 )
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

```
if( x > 10 )
  for(i = 1; i <= N; i++)
    a[i] = b[i] + c[i];
else
  for(i = 1; i <= N; i++)
    a[i] = b[i] - c[i];
```

Objectives of code optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)
Even if we write (syntactically and semantically) correct programs, they may not be "good" programs.

Example of code optimization

```
for(i = 1; i <= N; i++) {
  if( x > 10 )
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

```
if( x > 10 )
  for(i = 1; i <= N; i++)
    a[i] = b[i] + c[i];
else
  for(i = 1; i <= N; i++)
    a[i] = b[i] - c[i];
```

Objectives of code optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)
Even if we write (syntactically and semantically) correct programs, they may not be "good" programs.

Example of code optimization

```
for(i = 1; i <= N; i++) {
  if( x > 10 )
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

```
if( x > 10 )
  for(i = 1; i <= N; i++)
    a[i] = b[i] + c[i];
else
  for(i = 1; i <= N; i++)
    a[i] = b[i] - c[i];
```

Objectives of code optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)
Even if we write (syntactically and semantically) correct programs, they may not be "good" programs.

Example of code optimization

```
for(i = 1; i <= N; i++) {
  if( x > 10 )
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

```
if( x > 10 )
  for(i = 1; i <= N; i++)
    a[i] = b[i] + c[i];
else
  for(i = 1; i <= N; i++)
    a[i] = b[i] - c[i];
```

Objectives of code optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: Venit et al., Prelude to Programming: Concepts and Design)
Even if we write (syntactically and semantically) correct programs, they may not be "good" programs.

Example of code optimization

```
for(i = 1; i <= N; i++) {
  if( x > 10 )
    a[i] = b[i] + c[i];
  else
    a[i] = b[i] - c[i];
}
```

```
if( x > 10 )
  for(i = 1; i <= N; i++)
    a[i] = b[i] + c[i];
else
  for(i = 1; i <= N; i++)
    a[i] = b[i] - c[i];
```

Objectives of code optimization:

- To speed-up the computation
- To use less resource, eg. memory, power, etc.

So, we need a **compiler**.

Can you trust your compiler?

- Present day compilers are sophisticated enough to carry out many intricate transformations
- A human expert may also play the role of a compiler
- Compiler transformations, even if theoretically correct, may be buggy because of wrong coding

Erroneous loop interchange

```
sum = 0;
for (j=0; j<N; j++) {
  for (i=0; i<N; i++) {
    sum = sum + a[i][j];
} }
```

```
sum = 0;
for (i=0; i<N; i++) {
  for (j=0; j<=N; j++) {
    sum = sum + a[i][j];
} } // a[i][N] gets accessed
```

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.

A faulty compiler can alter the meaning of a program.

Can you trust your compiler?

- Present day compilers are sophisticated enough to carry out many intricate transformations
- A human expert may also play the role of a compiler
- Compiler transformations, even if theoretically correct, may be buggy because of wrong coding

Erroneous loop interchange

```
sum = 0;
for (j=0; j<N; j++) {
  for (i=0; i<N; i++) {
    sum = sum + a[i][j];
} }
```

```
sum = 0;
for (i=0; i<N; i++) {
  for (j=0; j<=N; j++) {
    sum = sum + a[i][j];
} } // a[i][N] gets accessed
```

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.

A faulty compiler can alter the meaning of a program.

Untrusted compilers is a reality

gcc – Frequently Reported Bugs

There are many reasons why a reported bug doesn't get fixed. It might be difficult to fix, or fixing it might break compatibility. Often, reports get a low priority when there is a simple work-around. In particular, bugs caused by invalid code have a simple work-around: fix the code.
(source: `http://gcc.gnu.org/bugs/#known`)

What is the remedy?

Verified Compiler

All optimized programs will be correct *by construction*
Example: CompCert, INRIA
Limitations

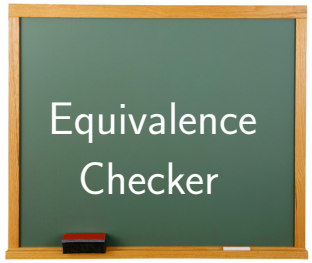
- Very hard to formally verify all passes of a compiler
- Undecidability of the general problem restricts the scope of the input language

Translation Validation

Each individual translation is followed by a validation phase which verifies that the target code produced correctly implements the source code
(This is what we do, i.e., equivalence checking of programs)

Translation Validation

Specification



Implementation

Compiler

Equivalence of finite automata

Definition

Two finite automata (FA) over Σ are equivalent if they accept the same set of strings over Σ.

By corollary, when two FA are not equivalent, with both the FA starting from the initial state, there must be a string w for which one of the two FA reaches a final/accepting state, while the other FA does not.

Finite automata based verification strategy

Limitations

- State based representation not powerful enough to capture execution of programs
- Practical programs have a finite set of states, but that does not permit efficient analysis of programs
- Computed values need to stored
- Separation of control states and data states is needed

So, we need some kind of control and data-flow graph (CDFG) to reason about the programs.

Finite automata based verification strategy

Limitations

- State based representation not powerful enough to capture execution of programs
- Practical programs have a finite set of states, but that does not permit efficient analysis of programs
- Computed values need to stored
- Separation of control states and data states is needed

So, we need some kind of control and data-flow graph (CDFG) to reason about the programs.

How to check equivalence of programs?

The general problem is undecidable.

McCarthy 91 function

```
int M ( int n ) {
if (n > 100)
  return (n - 10);
else
  return M( M (n + 11) );
}
```

```
int M ( int n ) {
if (n > 100)
  return (n - 10);
else
  return 91;
}
```

Comparing two programs in *totality* is impossible – we should break them into *smaller* chunks.

Granularity of the chunks

Instruction level

```
x = a + b;
y = x - a;
z = y + b;
```

```
x = a + b;
y = b;
z = 2 * b;
```

Granularity of the chunks

Instruction level

```
x = a + b;  ✓        x = a + b;  ✓
y = x - a;           y = b;
z = y + b;           z = 2 * b;
```

Granularity of the chunks

Instruction level

```
x = a + b;  ✓          x = a + b;  ✓
y = x - a;  ×          y = b;      ×
z = y + b;             z = 2 * b;
```

So, instruction level checking can be misleading – let's try at basic block level.

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b;
y = x - a;
z = y + b;
do {
  v = v + x;
  w = y * z;
} while( c1 );
```

```
x = a + b;
y = b;
z = 2 * b;
do {
  v = v + x;
} while( c1 );
w = y * z;
```

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b;  ✓
y = x - a;  ✓
z = y + b;  ✓
do {
  v = v + x;
  w = y * z;
} while( c1 );
```

```
x = a + b;  ✓
y = b;      ✓
z = 2 * b;  ✓
do {
  v = v + x;
} while( c1 );
w = y * z;
```

Granularity of the chunks (contd.)

Basic Block level

```
x = a + b;  ✓
y = x - a;  ✓
z = y + b;  ✓
do {
  v = v + x;  ×
  w = y * z;  ×
} while( c1 );
```

```
x = a + b;  ✓
y = b;      ✓
z = 2 * b;  ✓
do {
  v = v + x;  ×
} while( c1 );
w = y * z;
```

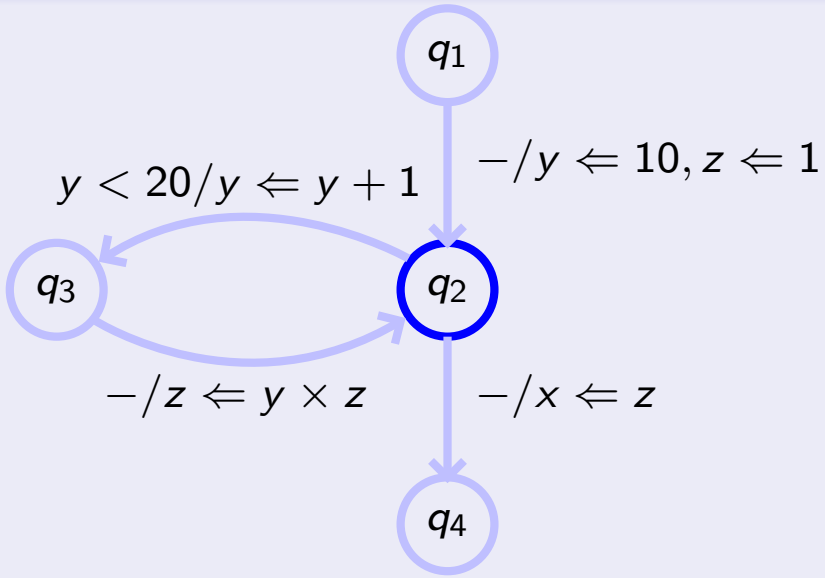
So, checking individual basic blocks is not enough.

Program as a composition of finite paths

A program can have *infinite* number of computations, a single computation can be *indefinitely long* — **cut loops**.

Representing a program using CDFG

```
y := 10;
z := 1;
while ( y < 20 ) {
  y := y + 1;
  z := y × z;
}
x := z;
```



All computations of the program can be viewed as a concatenation of paths.

Example: $p_1.p_3$, $p_1.p_2.p_3$, $p_1.p_2.p_2.p_3$, $p_1.(p_2)^*.p_3$

Outline

1. Background
2. **Translation validation of code motion transformations**
3. Deriving bisimulation relations from path based equivalence checkers
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. Other Models of Computation
7. Conclusion and scope of future work

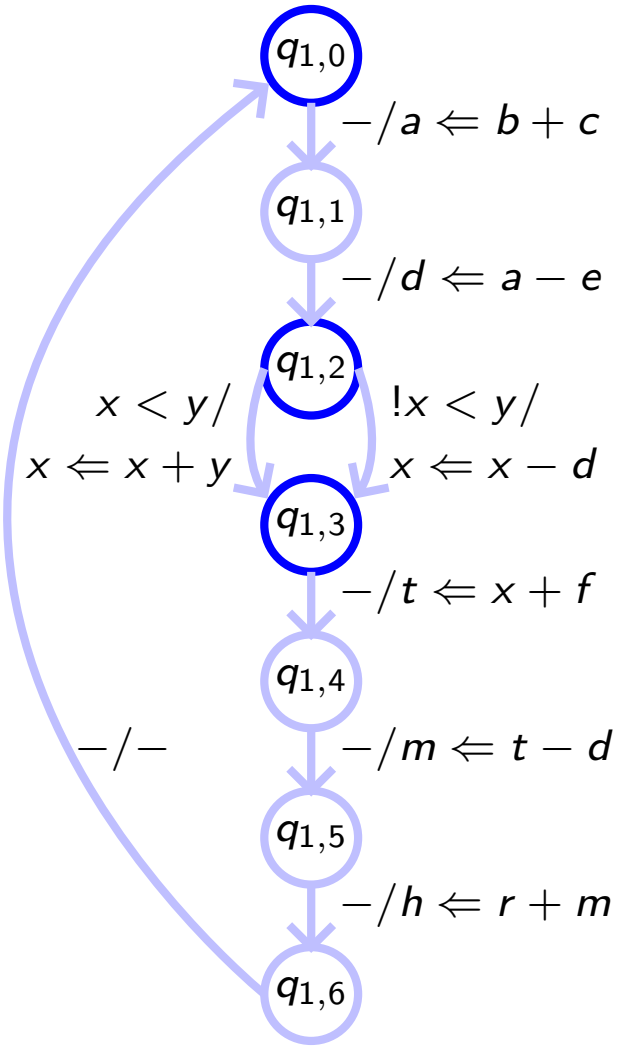
Finite State Machine with Datapath (FSMD)

FSMDs effectively capture both the control flow and the associated data processing of a behaviour.

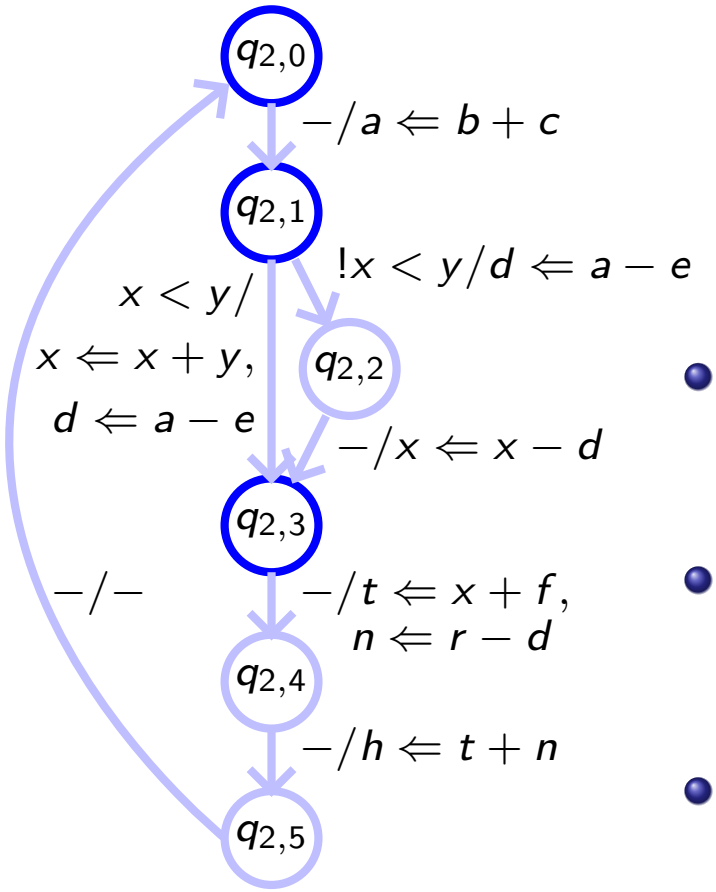
The FSMD model is a seven tuple $F = \langle Q, q_0, I, V, O, f, h \rangle$:

- Q: Finite set of control states
- q_0: Reset state, i.e. $q_0 \in Q$
- I: Set of input variables
- V: Set of storage variables
- O: Set of output variables
- f: State transition function, i.e. $Q \times 2^S \rightarrow Q$
- h: Update function of the output and the storage variables, i.e. $Q \times 2^S \rightarrow U$
  - U represents a set of storage or output assignments
  - S is a set of arithmetic relations between arithmetic expressions

Equivalence checking of FSMDs: A basic example



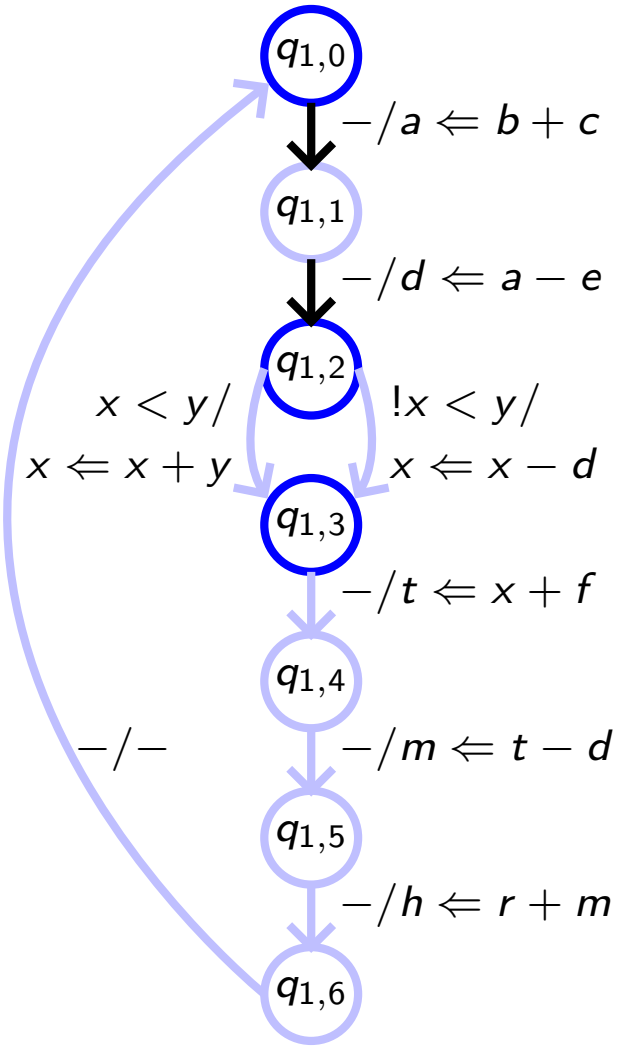
(a) M_1



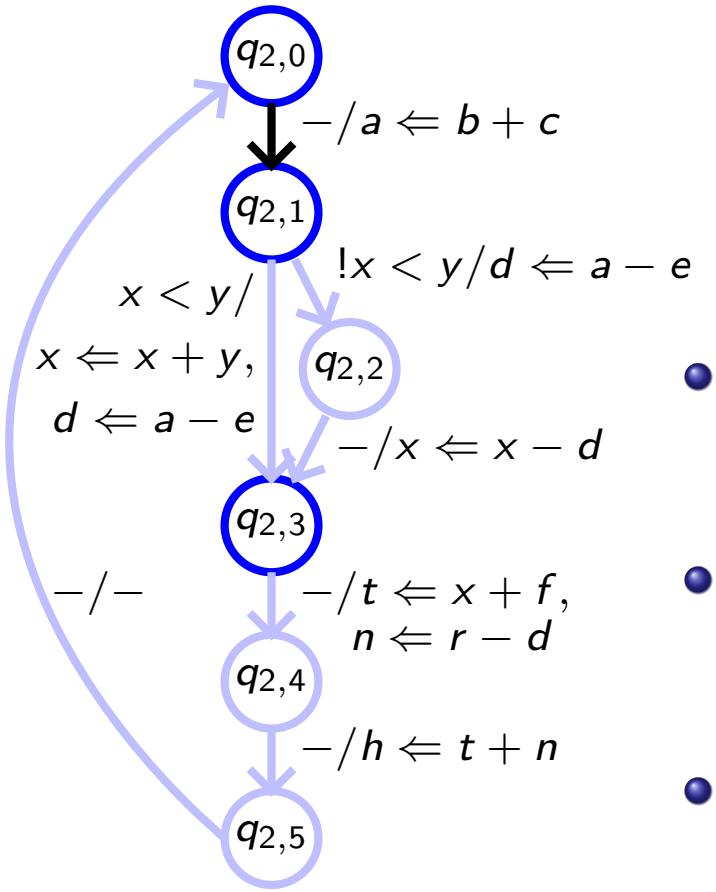
(b) M_2

- Two Finite State Machines with Datapath (FSMDs) M_1 and M_2 are equivalent if for every path in P_1 there is an equivalent path in P_2 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{x<y}{\Longrightarrow} q_{2,3},\ q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{!x<y}{\Longrightarrow} q_{2,3},\ q_{1,3} \Longrightarrow q_{1,0} \simeq q_{2,3} \Longrightarrow q_{2,0}\}$

Equivalence checking of FSMDs: A basic example



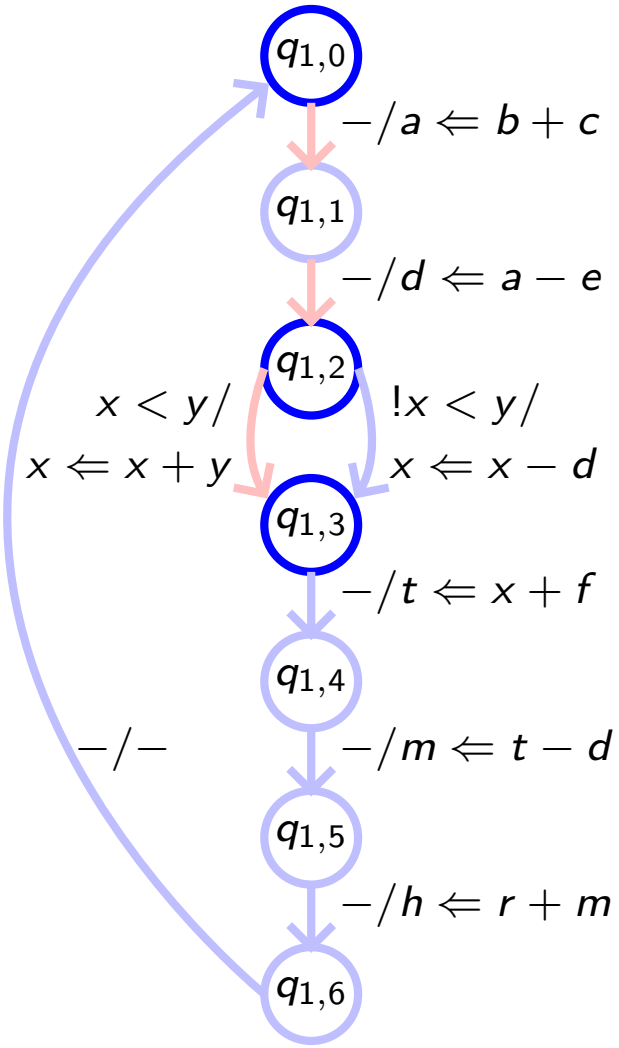
(a) M_1



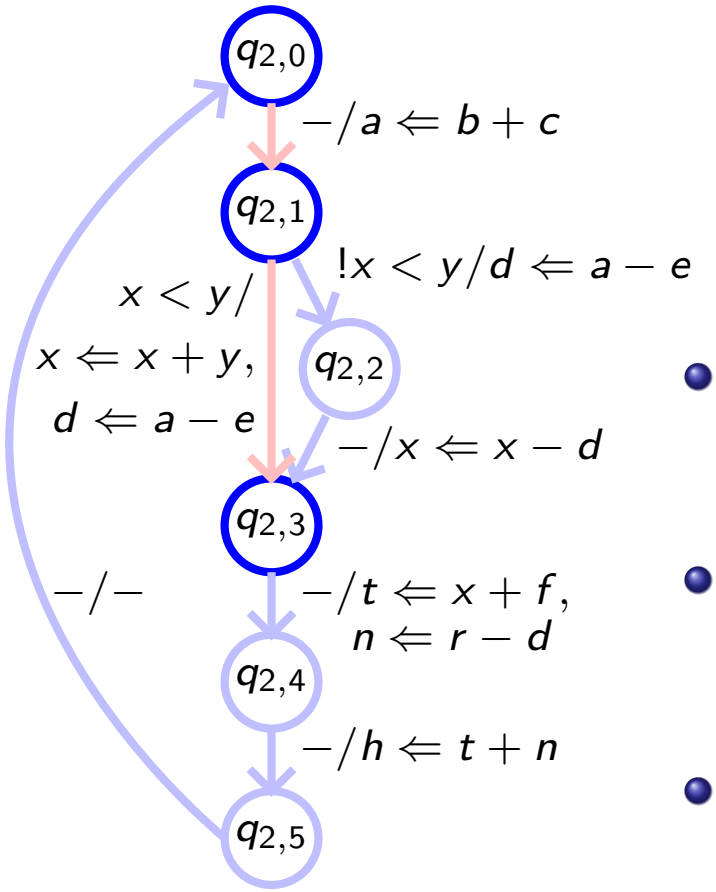
(b) M_2

- Two Finite State Machines with Datapath (FSMDs) M_1 and M_2 are equivalent if for every path in P_1 there is an equivalent path in P_2 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{x<y}{\Longrightarrow} q_{2,3},\ q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{!x<y}{\Longrightarrow} q_{2,3},\ q_{1,3} \Longrightarrow q_{1,0} \simeq q_{2,3} \Longrightarrow q_{2,0}\}$

Equivalence checking of FSMDs: A basic example



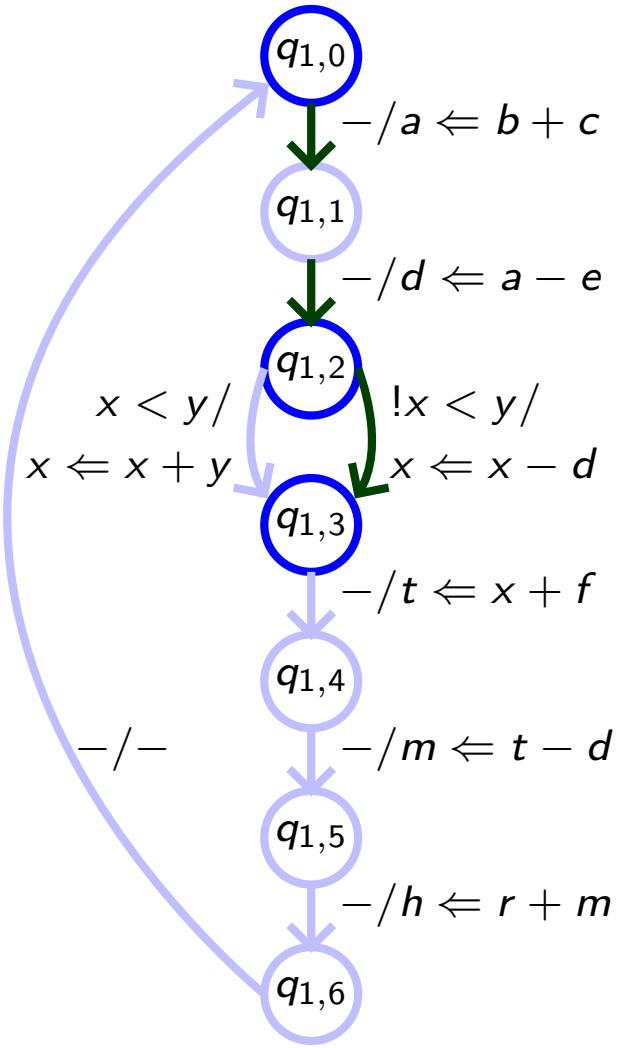
(a) M_1



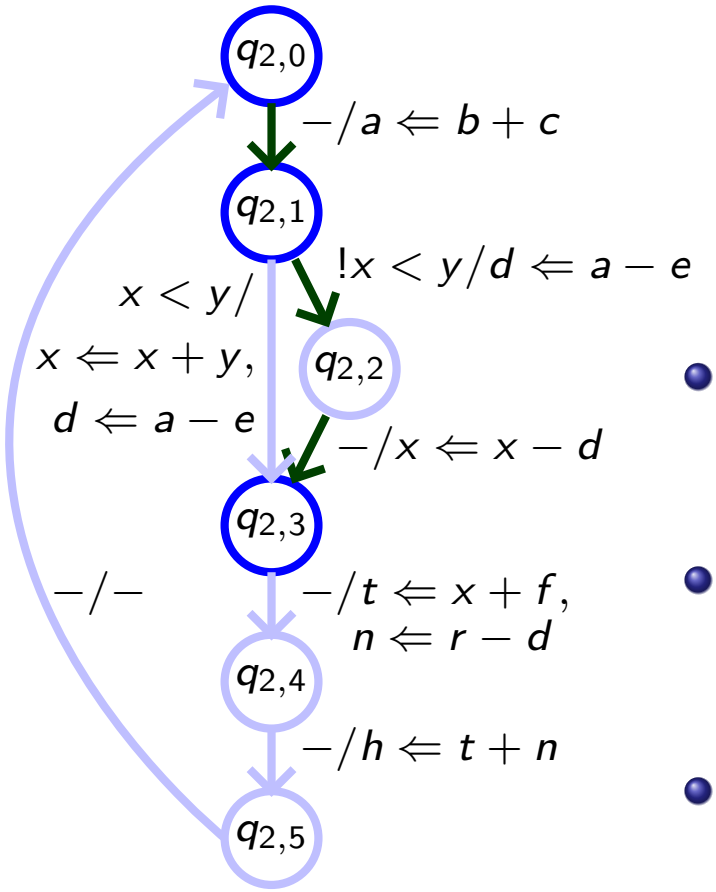
(b) M_2

- Two Finite State Machines with Datapath (FSMDs) M_1 and M_2 are equivalent if for every path in P_1 there is an equivalent path in P_2 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{x<y}{\Longrightarrow} q_{2,3}, q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{!x<y}{\Longrightarrow} q_{2,3}, q_{1,3} \Longrightarrow q_{1,0} \simeq q_{2,3} \Longrightarrow q_{2,0}\}$

Equivalence checking of FSMDs: A basic example



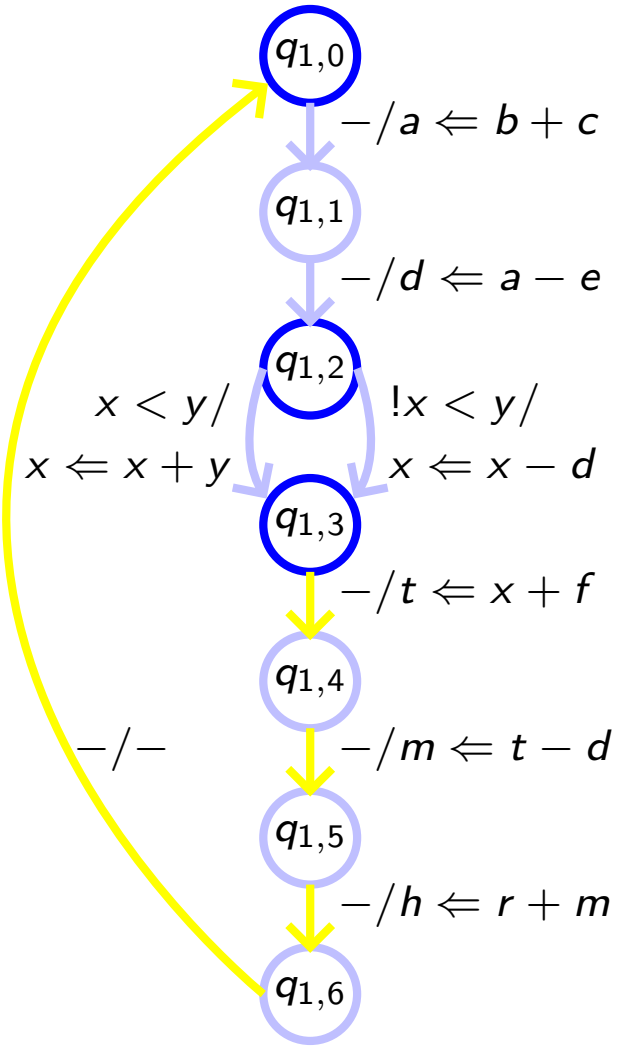
(a) M_1



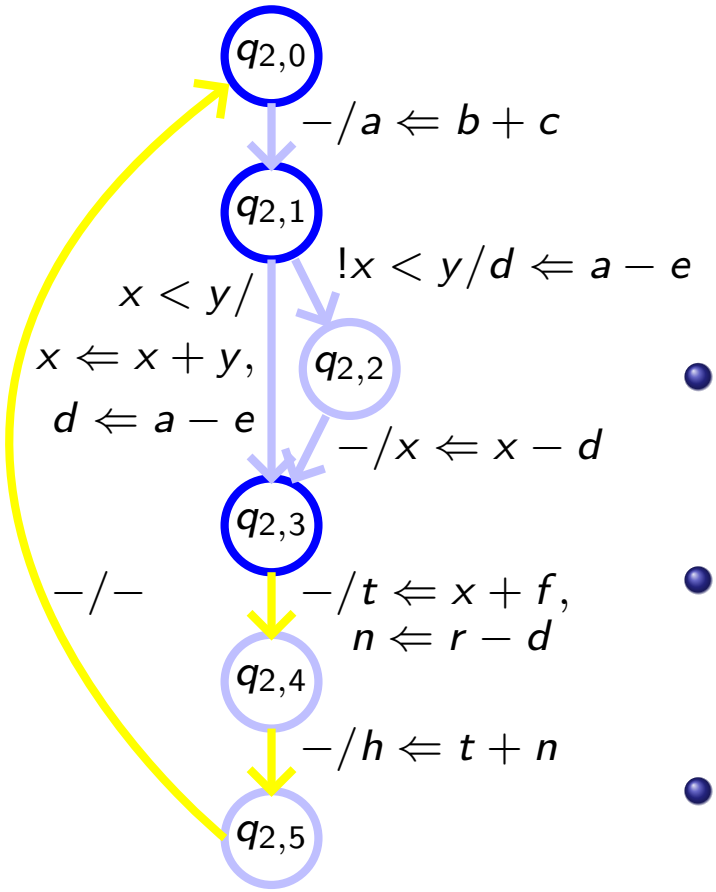
(b) M_2

- Two Finite State Machines with Datapath (FSMDs) M_1 and M_2 are equivalent if for every path in P_1 there is an equivalent path in P_2 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{x<y}{\Longrightarrow} q_{2,3},\ q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{!x<y}{\Longrightarrow} q_{2,3},\ q_{1,3} \Longrightarrow q_{1,0} \simeq q_{2,3} \Longrightarrow q_{2,0}\}$

Equivalence checking of FSMDs: A basic example



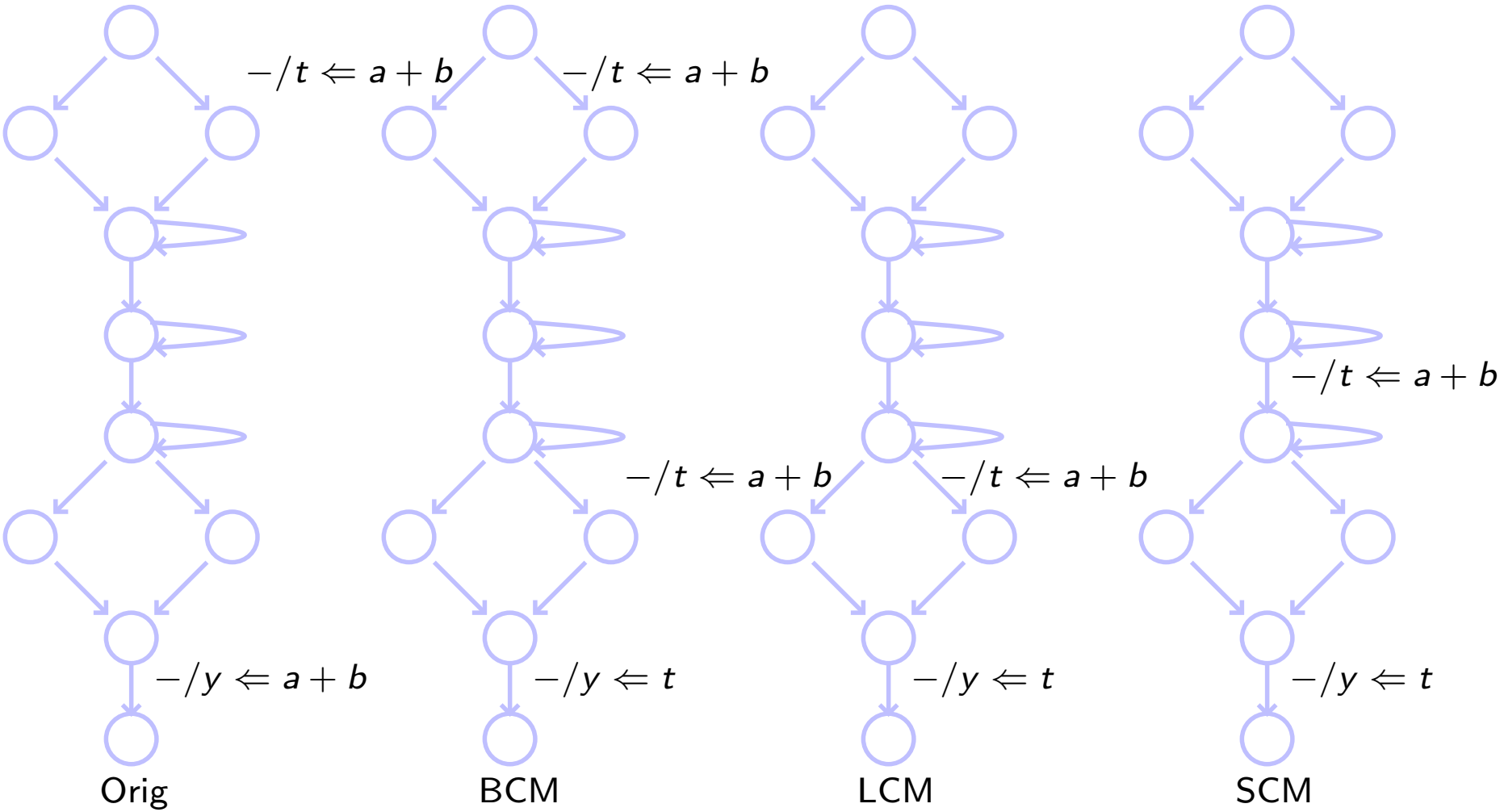
(a) M_1



(b) M_2

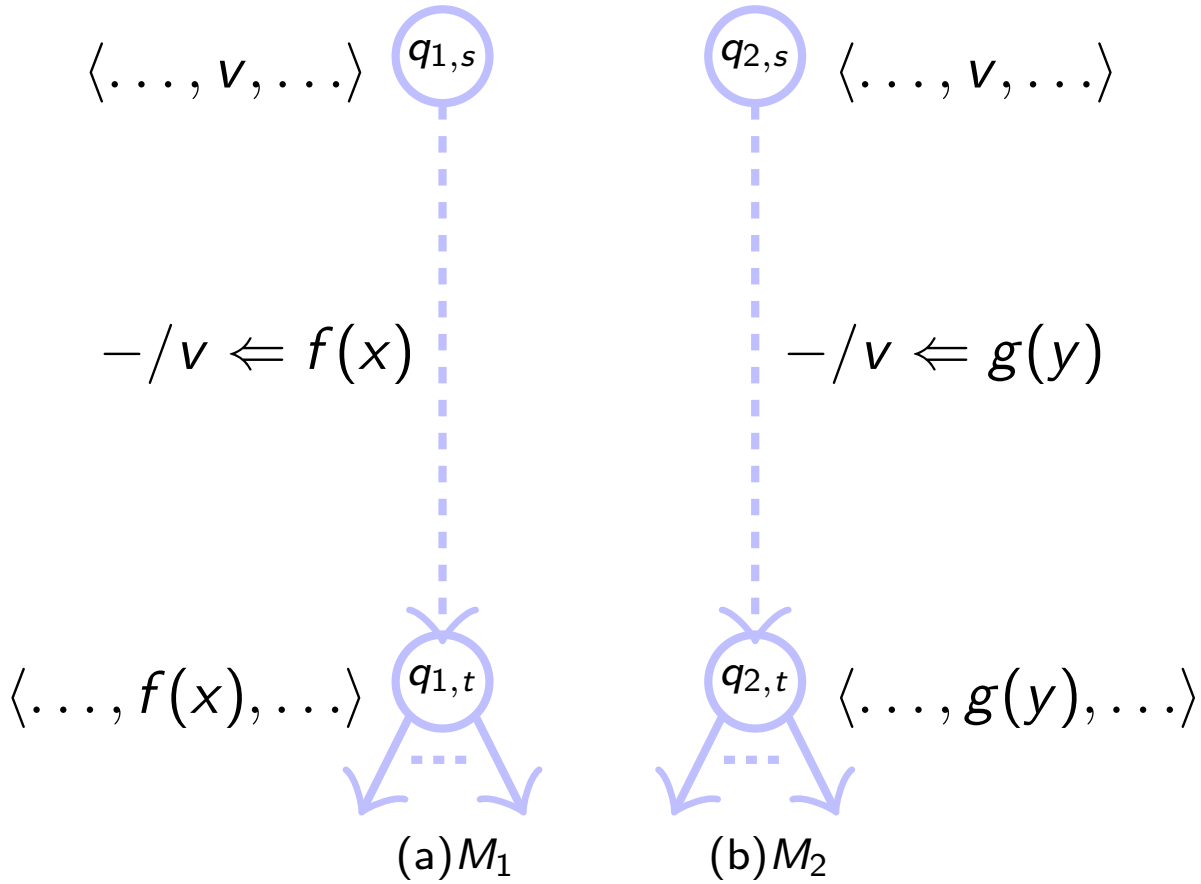
- Two Finite State Machines with Datapath (FSMDs) M_1 and M_2 are equivalent if for every path in P_1 there is an equivalent path in P_2 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{1,0} \overset{x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{x<y}{\Longrightarrow} q_{2,3},\ q_{1,0} \overset{!x<y}{\Longrightarrow} q_{1,3} \simeq q_{2,0} \overset{!x<y}{\Longrightarrow} q_{2,3},\ q_{1,3} \Longrightarrow q_{1,0} \simeq q_{2,3} \Longrightarrow q_{2,0}\}$

A major challenge: Code motions across loops



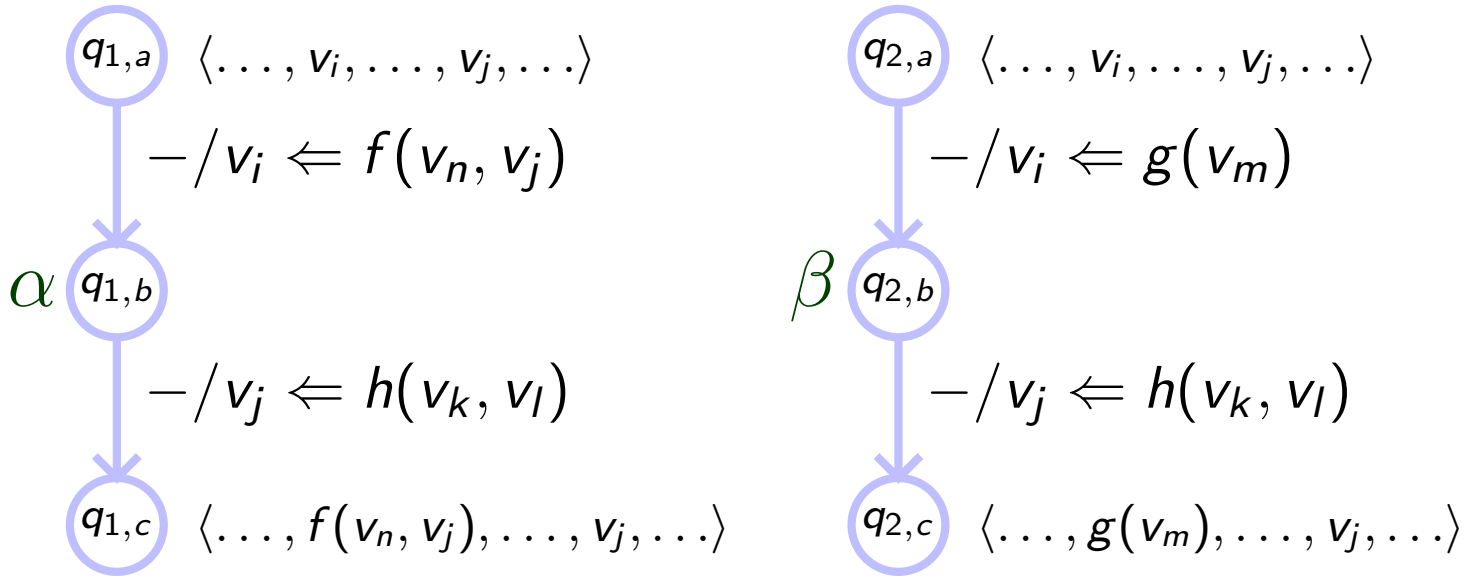
A path, by definition, cannot be extended beyond a loop.

The method of symbolic value propagation



An example of value propagation

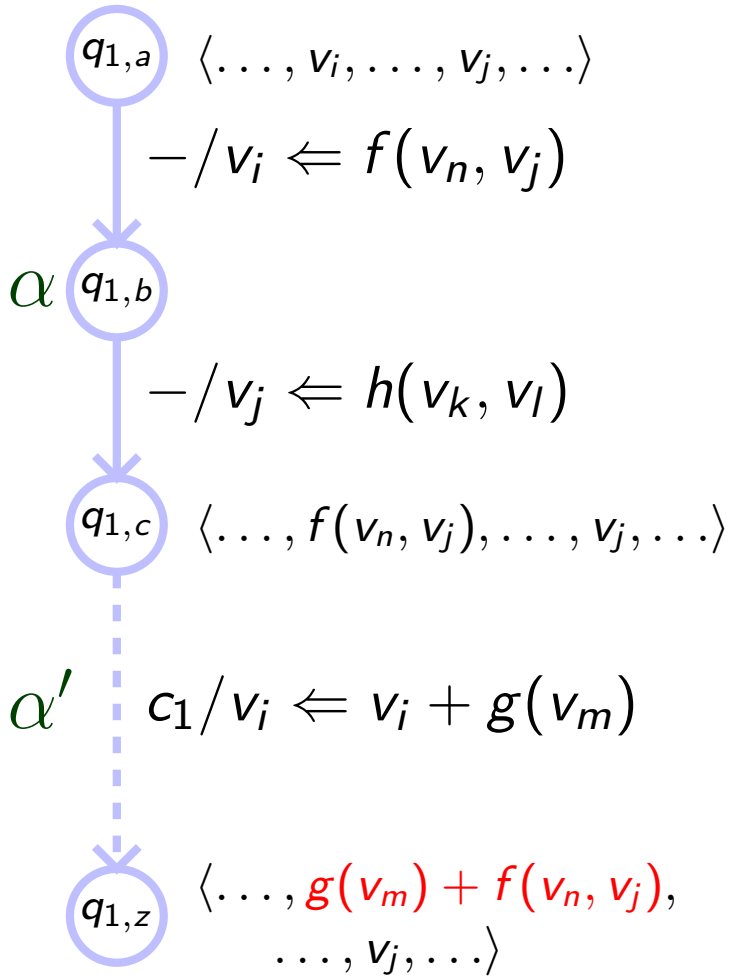
The method of value propagation



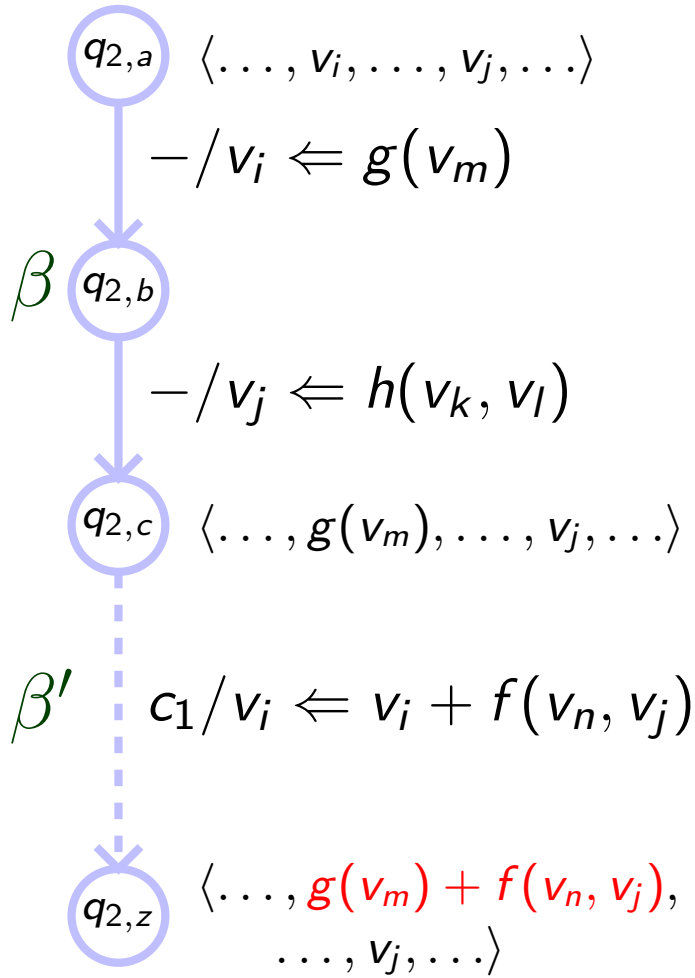
(a) M_1 (b) M_2

An example of value propagation with dependency between propagated values

The method of value propagation



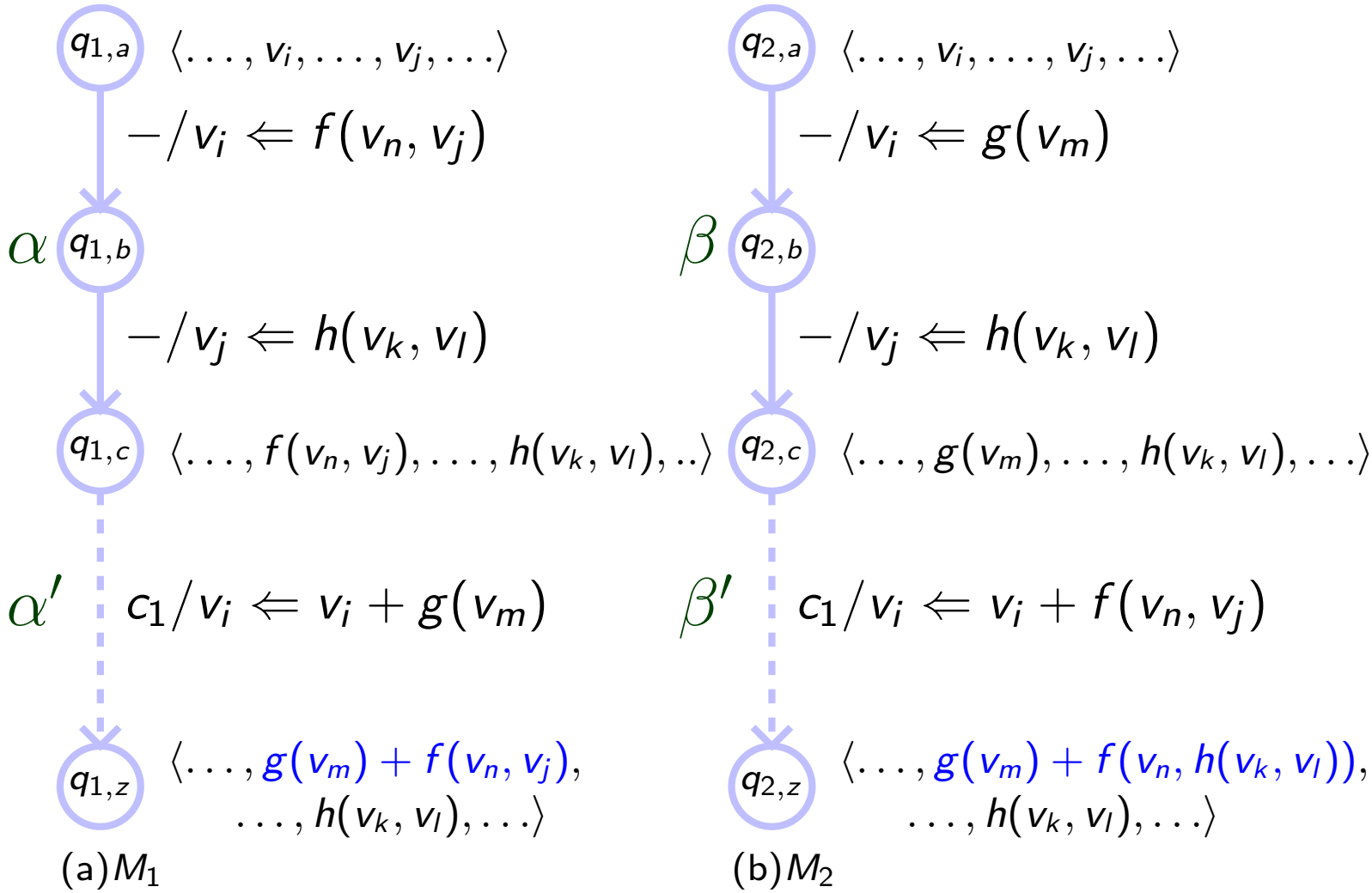
(a) M_1



(b) M_2

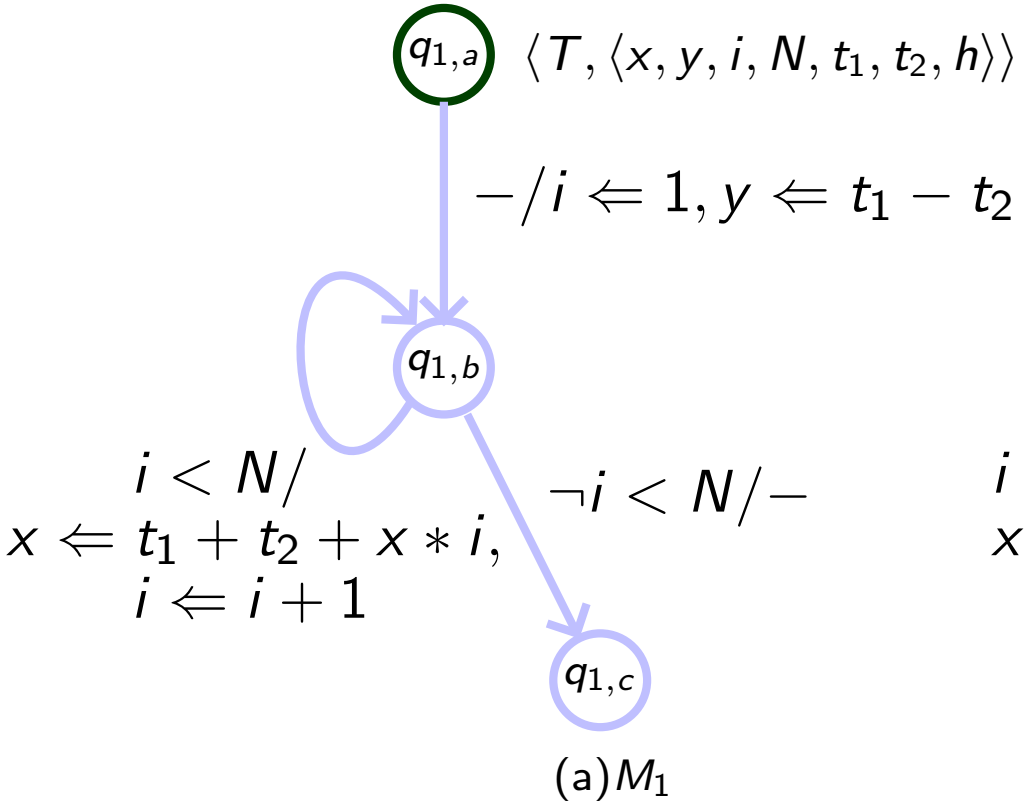
An *erroneous* decision taken

The method of value propagation

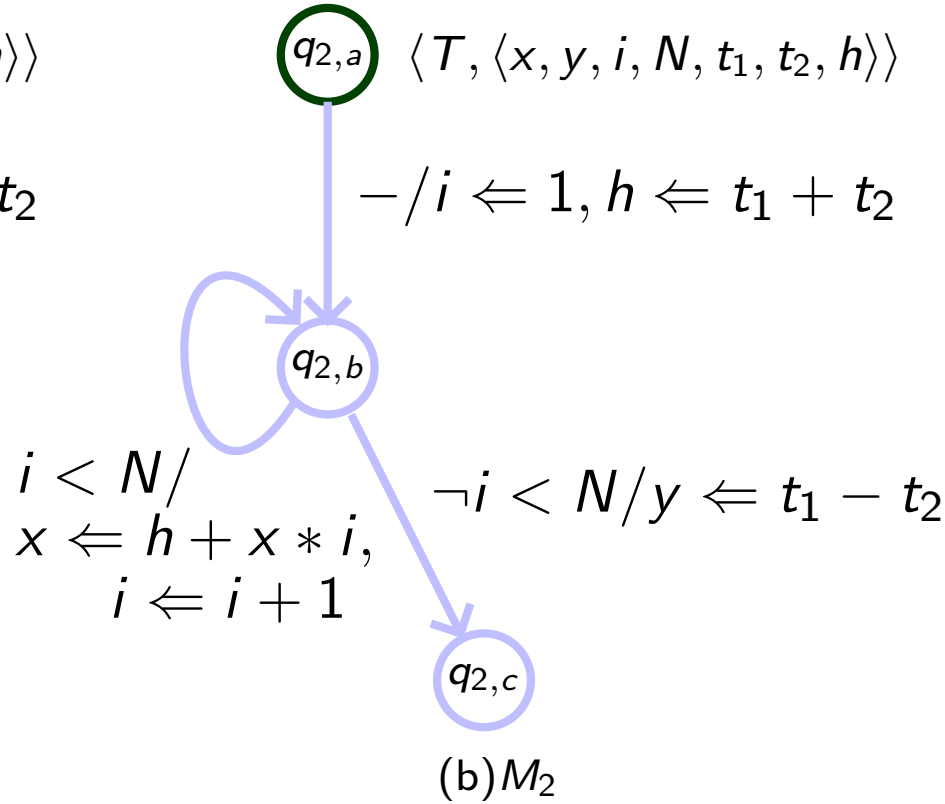


Correct decision taken

Equivalence checking using value propagation



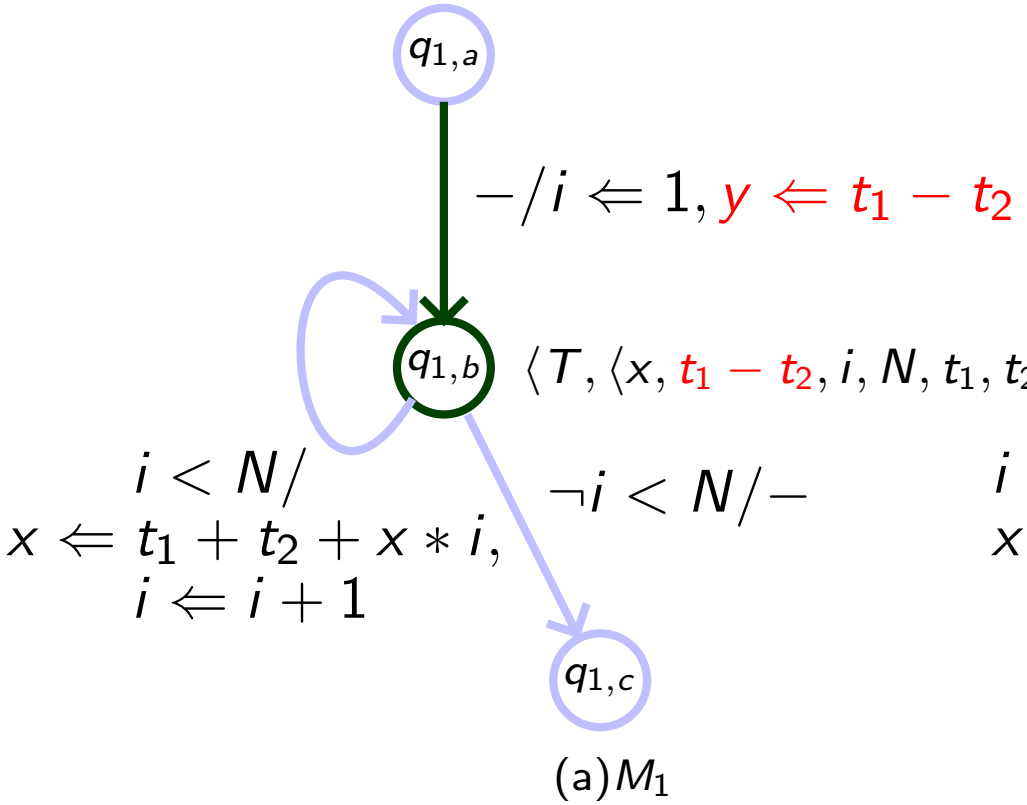
(a) M_1



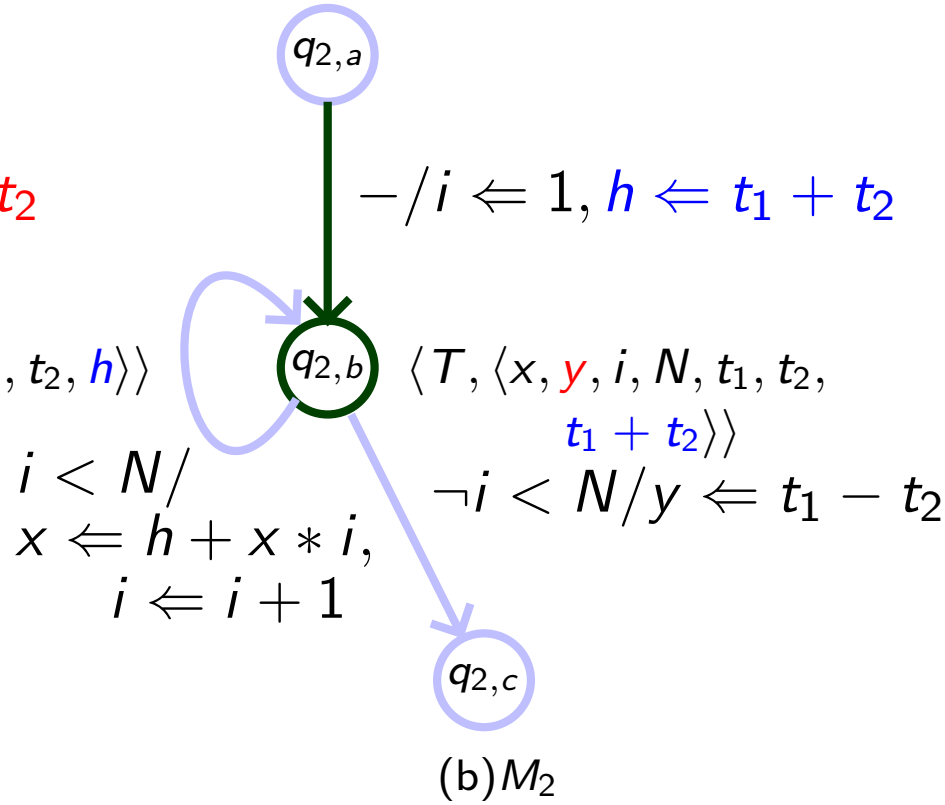
(b) M_2

At the reset states

Equivalence checking using value propagation



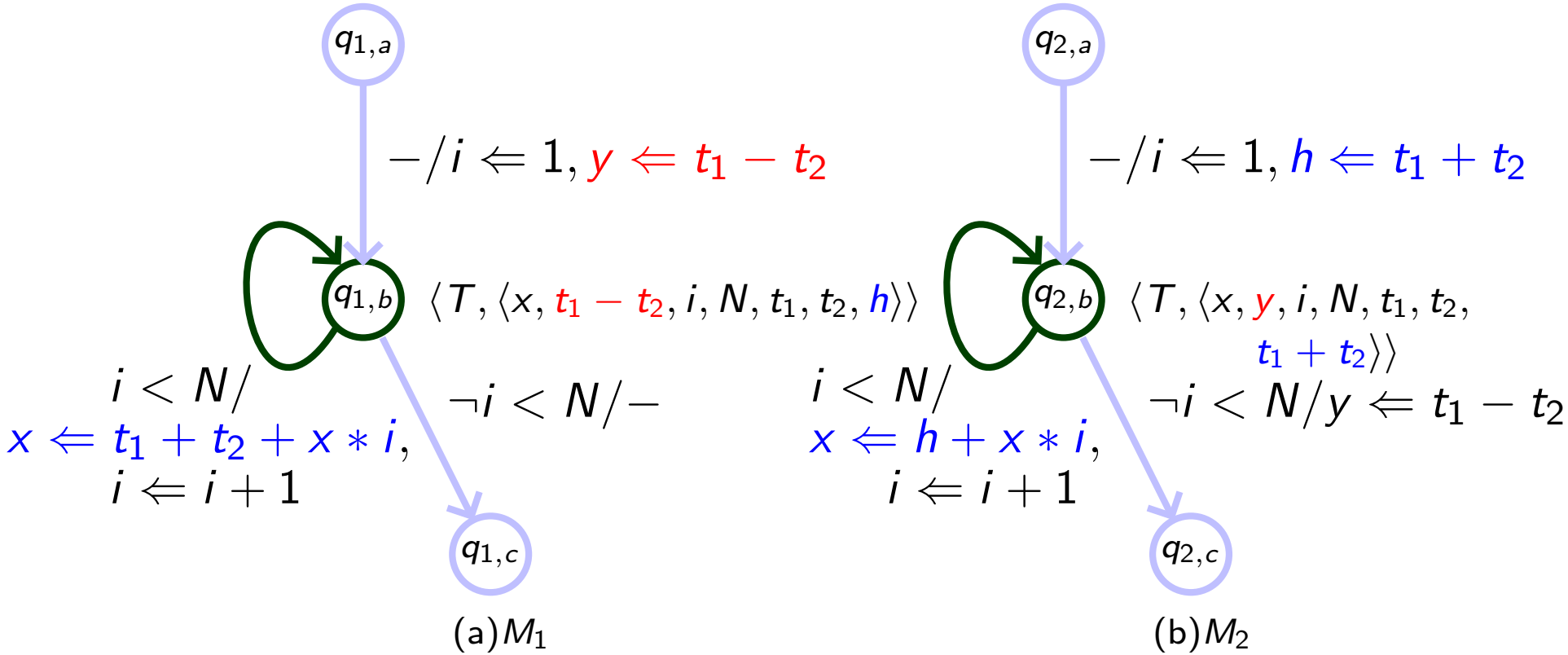
(a) M_1



(b) M_2

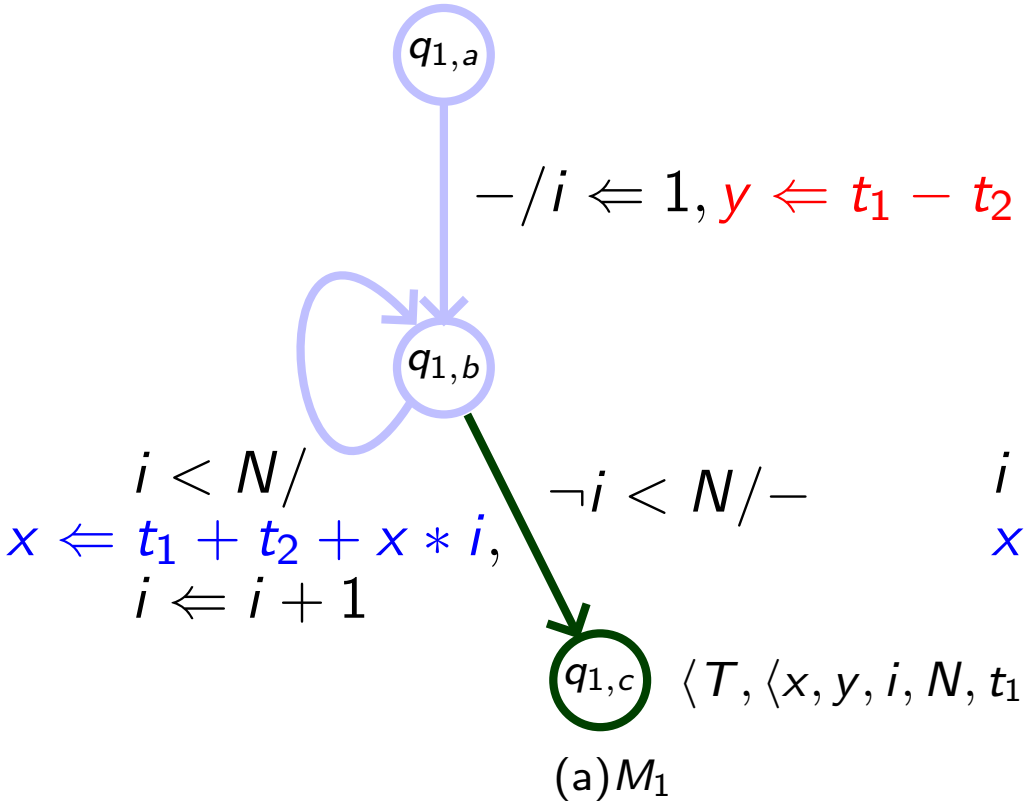
At the beginning of the loops

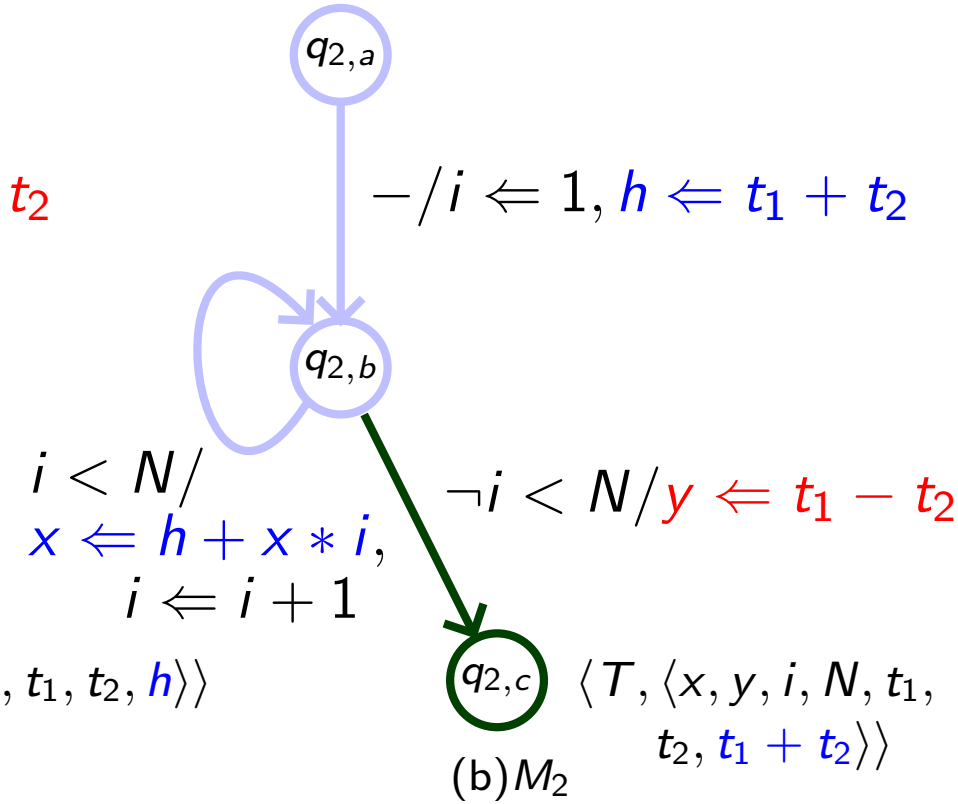
Equivalence checking using value propagation



At the end of the loops

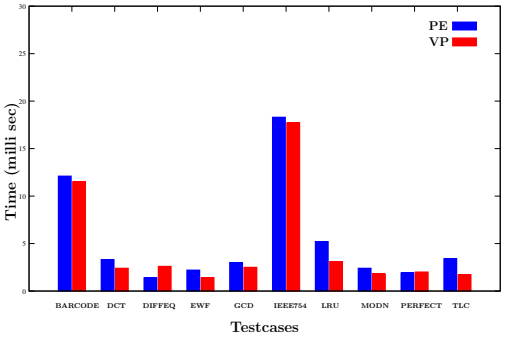
Equivalence checking using value propagation



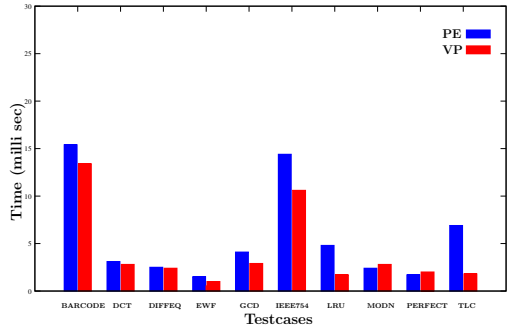


At the end states

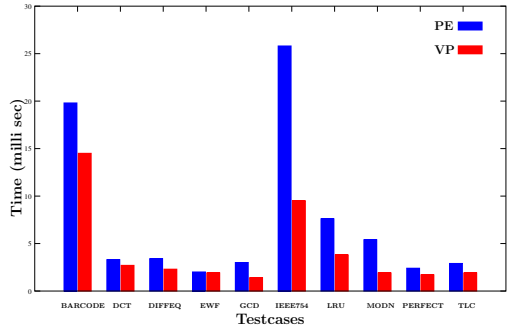
Experimental Results – 1



(a) BB-based



(b) Path-based



(c) SPARK

• C. Mandal, and R. M. Zimmer, "A Genetic Algorithm for the Synthesis of Structured Data Paths," VLSI Design 2000.

• R. Camposano, "Path-based Scheduling for Synthesis," TCAD 1991.

• S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," VLSI Design 2003.

Experimental Results – 1 (contd.)

Benchmarks	Original FSMD		Transformed FSMD		#Variable		#across loops	Maximum mismatch	Time (ms)	
	#state	#path	#state	#path	com	uncom			PE	VP
BARCODE	33	54	25	56	17	0	0	3	20.1	16.2
DCT	16	1	8	1	41	6	0	6	6.3	3.6
DIFFEQ	15	3	9	3	19	3	0	4	5.0	2.6
EWF	34	1	26	1	40	1	0	1	4.2	3.6
MODN	8	9	8	9	10	2	0	3	5.6	2.5
TLC	13	20	7	16	13	1	0	2	9.1	4.1
LCM	8	11	4	8	7	2	1	4	×	2.5
IEEE754	55	59	44	50	32	3	4	3	×	17.7
LRU	33	39	32	38	19	0	2	2	×	4.0
PERFECT	6	7	4	6	8	2	2	2	×	0.9
QRS	53	35	24	35	25	15	3	19	×	15.9

☞ Complexity *identical* to C Karfa et al., TODAES 2012.

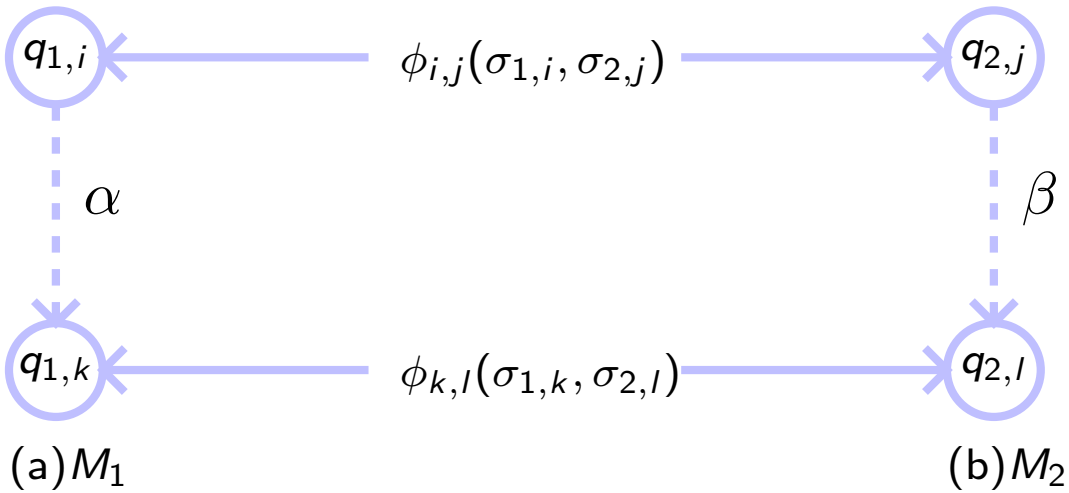
★ K Banerjee et al., "A Value Propagation Based Equivalence Checking Method for Verification of Code Motion Techniques," ISED 2012.

★ K Banerjee et al., "Verification of Code Motion Techniques using Value Propagation," TCAD 2014.

Outline

1. Background
2. Translation validation of code motion transformations
3. **Deriving bisimulation relations from path based equivalence checkers**
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. Other Models of Computation
7. Conclusion and scope of future work

Bisimulation relation based equivalence checking



$$\phi_{i,j}(\sigma_{1,i}, \sigma_{2,j}) \Rightarrow wp(\alpha, wp(\beta, \phi_{k,l}(\sigma_{1,k}, \sigma_{2,l})))$$

- Kundu et al., "Translation Validation of High-Level Synthesis," TCAD 2010.

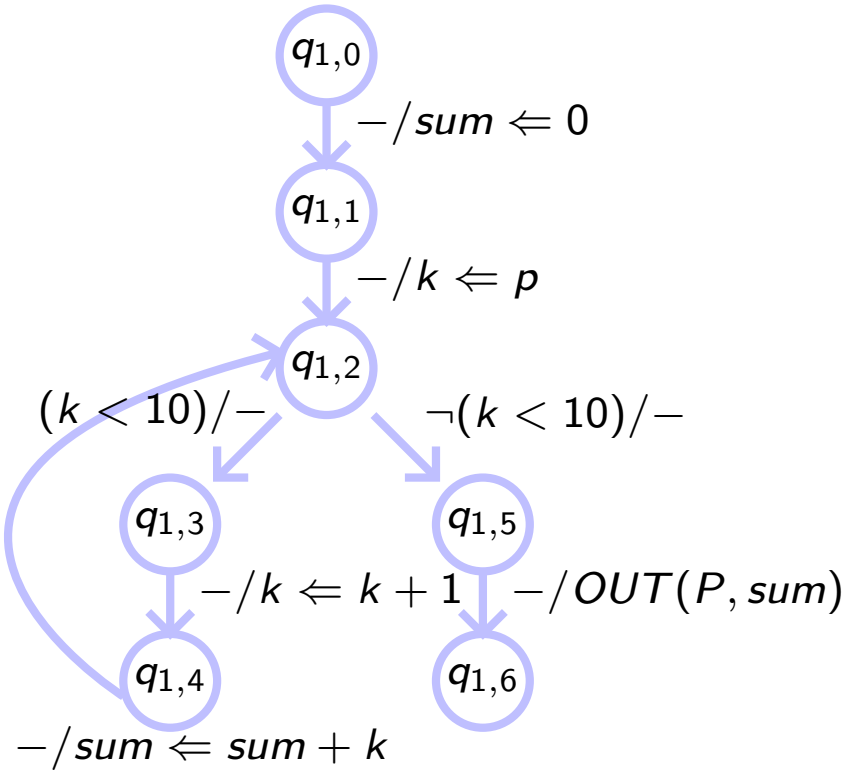
☞ Judicious choice of paired locations is needed to cover multiple edges, if required.

Bisimulation Relation vs Path based Equivalence Checking

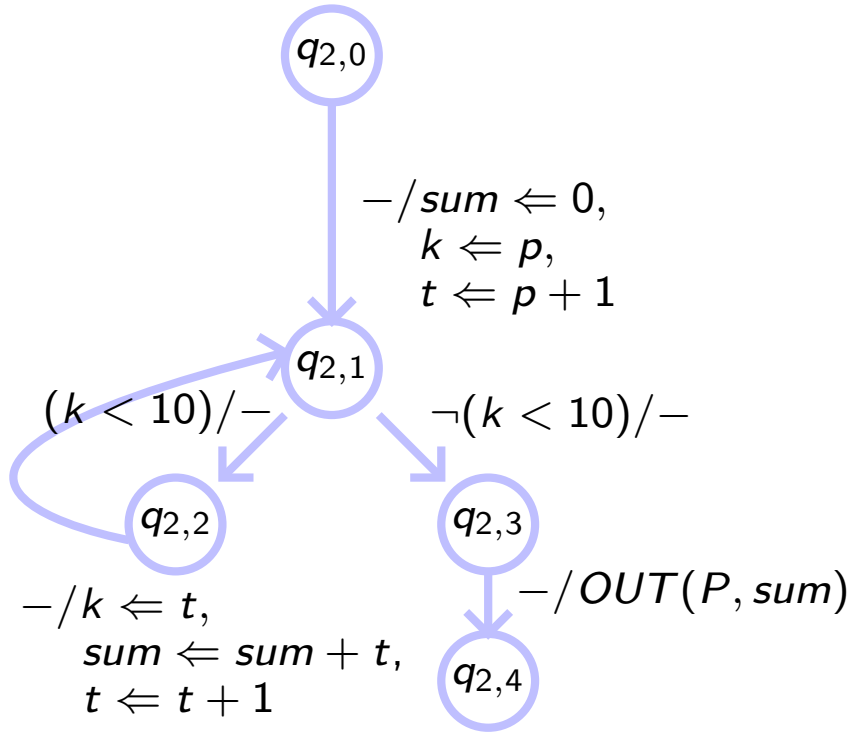
	Bisimulation based verification		Path based equivalence checking
✓	Conventional	×	Unconventional
✓	Can handle loop shifting	×	Cannot handle loop shifting
×	Limited support for non-structure preserving transformations	✓	Adept in handling non-structure preserving transformations
×	Termination not guaranteed	✓	Termination guaranteed

Objective: Derive bisimulation relations from the outputs of path based equivalence checkers

Example of loop shifting (Kundu et al., TCAD 2010)



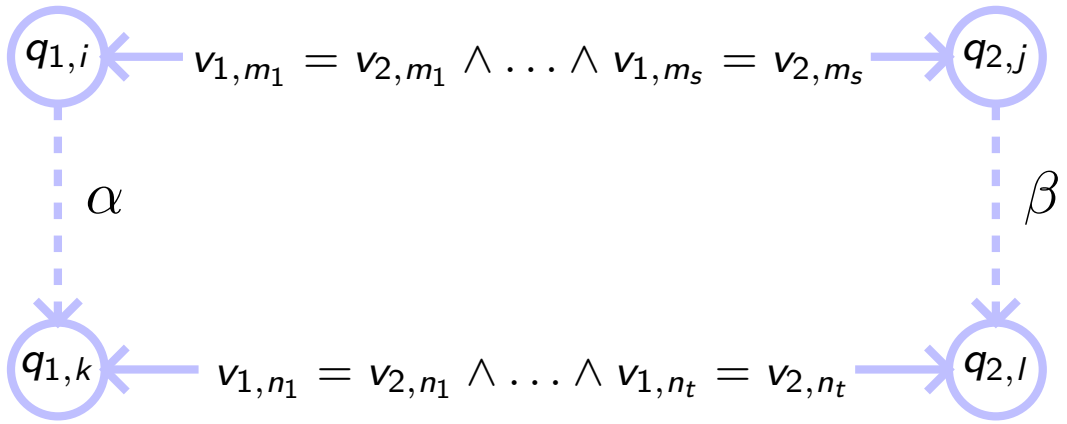
(a) M_1



(b) M_2

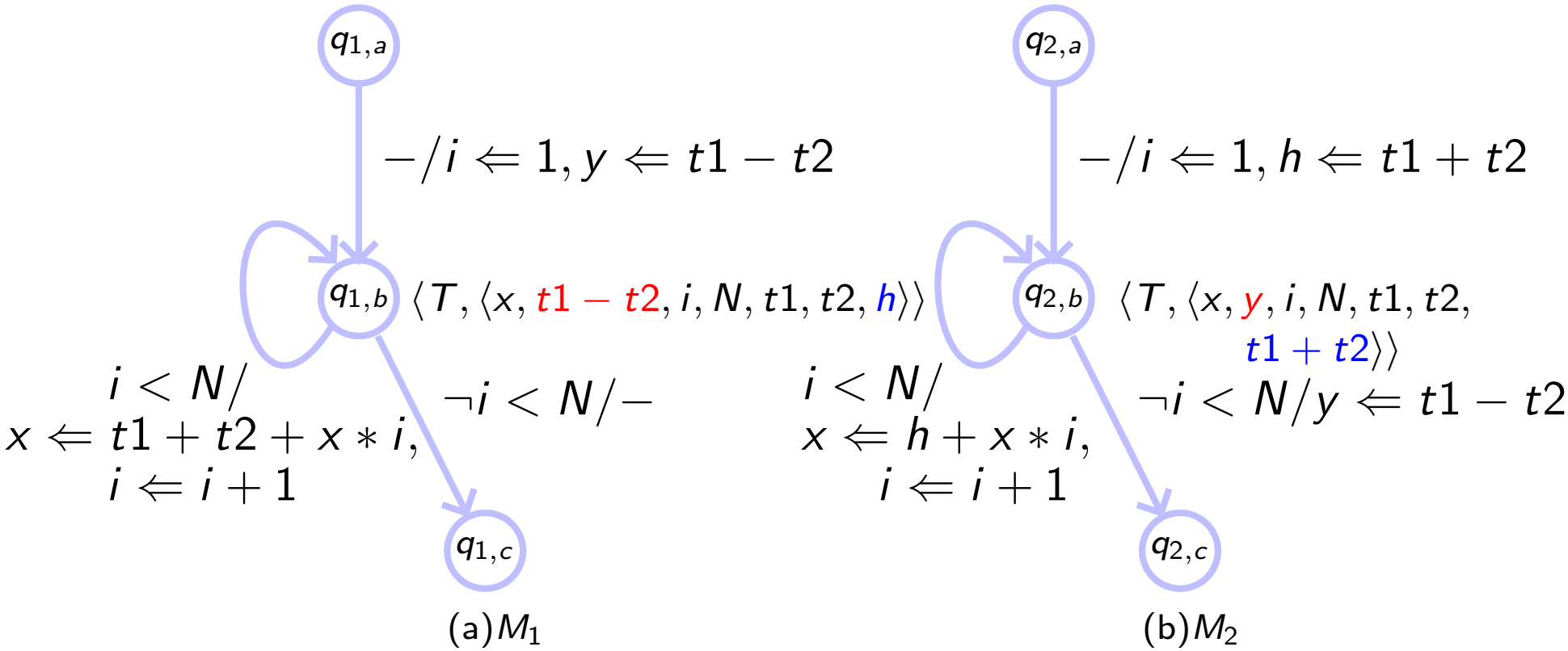
Sl	Loc Pair	1st iter	2nd iter
1	$\langle q_{1,0}, q_{2,0} \rangle$	$p_s = p_i$	$p_s = p_i$
2	$\langle q_{1,2}, q_{2,1} \rangle$	$k_s = k_i$	$k_s = k_i \wedge sum_s = sum_i \wedge (k_s + 1) = t_i$
3	$\langle q_{1,5}, q_{2,3} \rangle$	$sum_s = sum_i$	$sum_s = sum_i$

Bisimulation relation from path based equivalence checker



$v_i, 1 \leq i \leq m_s$, and $v_j, 1 \leq j \leq n_t$, are the live common variables at $(q_{1,i}, q_{2,j})$ and $(q_{1,k}, q_{2,l})$

Bisimulation relation for code motion across loop



$\phi_{m,n} \equiv \exists t1'_1, t2'_1 \exists t1'_2, t2'_2$
$[x_1 = x_2 \wedge i_1 = i_2 \wedge N_1 = N_2 \wedge t1_1 = t1_2 \wedge t2_1 = t2_2 \wedge y_1 = t1'_1 - t2'_1 \wedge h_2 = t1'_2 + t2'_2]$

★ K Banerjee et al., "Deriving Bisimulation Relations from Path Extension Based Equivalence Checkers," IEEE TSE 2017.

★ K Banerjee et al., "Deriving Bisimulation Relations from Path Based Equivalence Checkers," FAOC 2017.

Outline

1. Background
2. Translation validation of code motion transformations
3. Deriving bisimulation relations from path based equivalence checkers
4. **Translation validation of code motion transformations in array-intensive programs**
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. Other Models of Computation
7. Conclusion and scope of future work

Verifying Code Motions of Array-Intensive Programs

The FSMD model does not provide formalism to capture arrays.
So, we proposed the *FSMD having Arrays (FSMDA)* model.
Differences from the FSMD model:

- The sets I, O of input and output variables are divided into disjoint subsets of *scalars* and *arrays*, the set V of storage variables additionally has a disjoint subset for *indices*
- Introduction of McCarthy's *read* and *write* functions (generalized for n-dimensional arrays)

Subsequent changes in equivalence checking procedure:

- New normalization rules
- Computation of characteristic tuple of a path – revised
- Special propagation rules for index variables

McCarthy's Functions

access	change	J. McCarthy. "Towards a Mathematical Science of Computation", IFIP 1962
content(c)	assign(a)	D. M. Kaplan. "Some Completeness Results in the Mathematical Theory of Computation", JACM 1968
select($\downarrow$)	update($\leftarrow$)	P. J. Downey, R. Sethi. "Assignment Commands with Array References", JACM 1978
read	write	A. R. Bradley, Z. Manna. "The Calculus of Computation: Decision Procedures with Applications to Verification," 2007

Statement: $a[i] \Leftarrow b[i] + z$

Representation: $\mathrm{wr}^{(3)}(a, i, \mathrm{rd}^{(2)}(b, i) + z)$

Advantages:

- elegant representation is achieved with each array being depicted as a vector
- sequence of transformations of elements of arrays can be captured
- substitution operations on scalar and array variables can be carried out identically

Normalization Grammar

No canonical representation exists for expressions over integers.

A normalization technique was proposed by *J. C. King, "A Program Verifier," PhD thesis, Carnegie-Mellon University, 1969.*

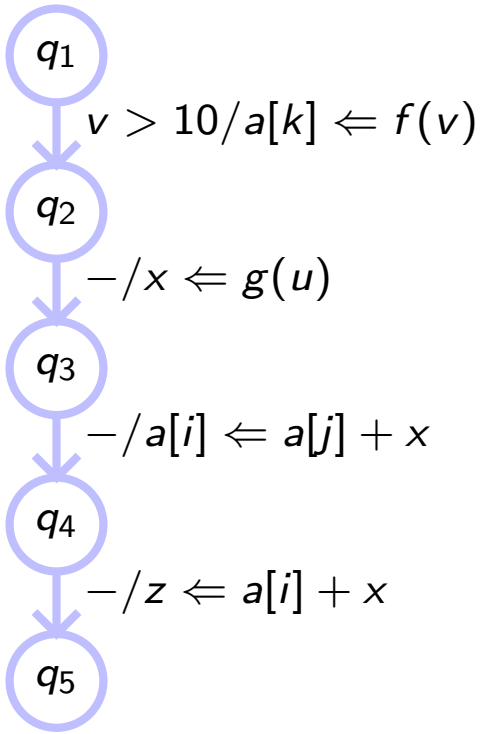
Updated Grammar

1) $S \rightarrow S + T \big| c_s$, where c_s is an integer.
2) $T \rightarrow T * P \big| c_t$, where c_t is an integer.
3) $P \rightarrow \mathrm{abs}(S) \big| (S) \bmod (C_d) \big| S \div C_d \big| v \big| c_m \big| \mathbf{A}$, where $v \in I_s \bigcup V_s \bigcup V_i$, and c_m is an integer.
4) $C_d \rightarrow S \div C_d \big| (S) \bmod (C_d) \big| S$.
5) $\mathbf{A} \rightarrow \mathrm{wr}^{(\mathbf{k+2})}(\mathbf{v'}, \mathbf{S_1}, \ldots, \mathbf{S_k}, \mathbf{S}) \big| \mathrm{rd}^{(\mathbf{k+1})}(\mathbf{v'}, \mathbf{S'_1}, \ldots, \mathbf{S'_k})$, where $v' \in I_a \bigcup V_a$, and $S_1, \ldots, S_k, S'_1, \ldots, S'_k$ are of type S (sum) involving variables $\in V_i$.

Some simplification rules for integers are given in [Karfa et al., TCAD 2008].

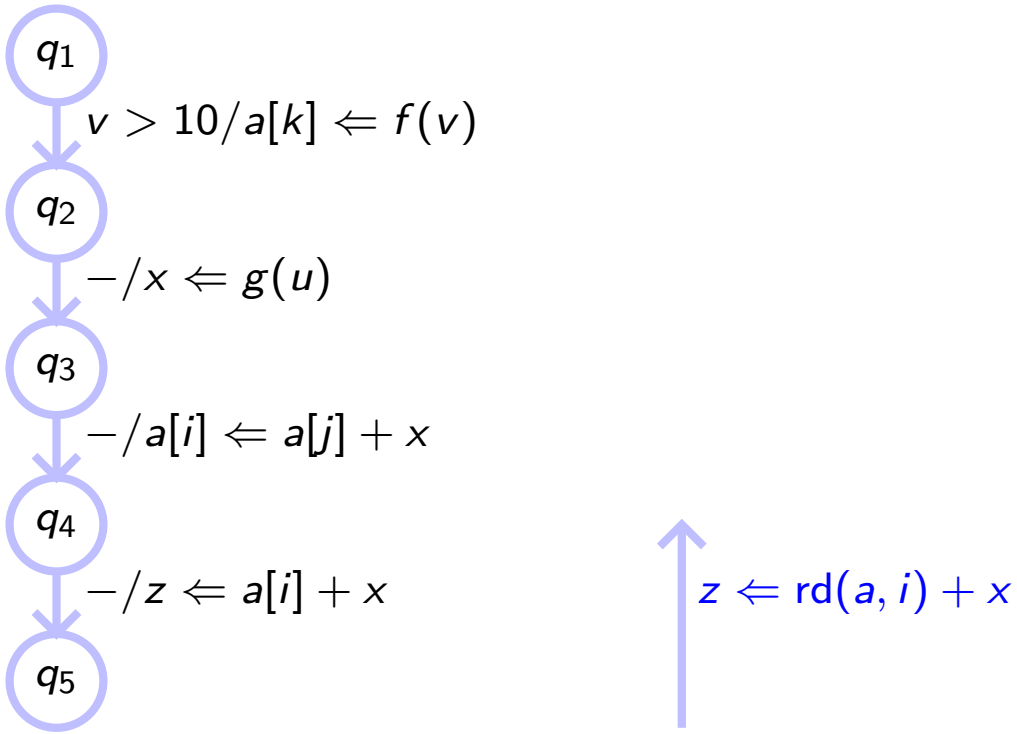
Computing characteristic tuple of a path

Computing path characteristics using backward substitution



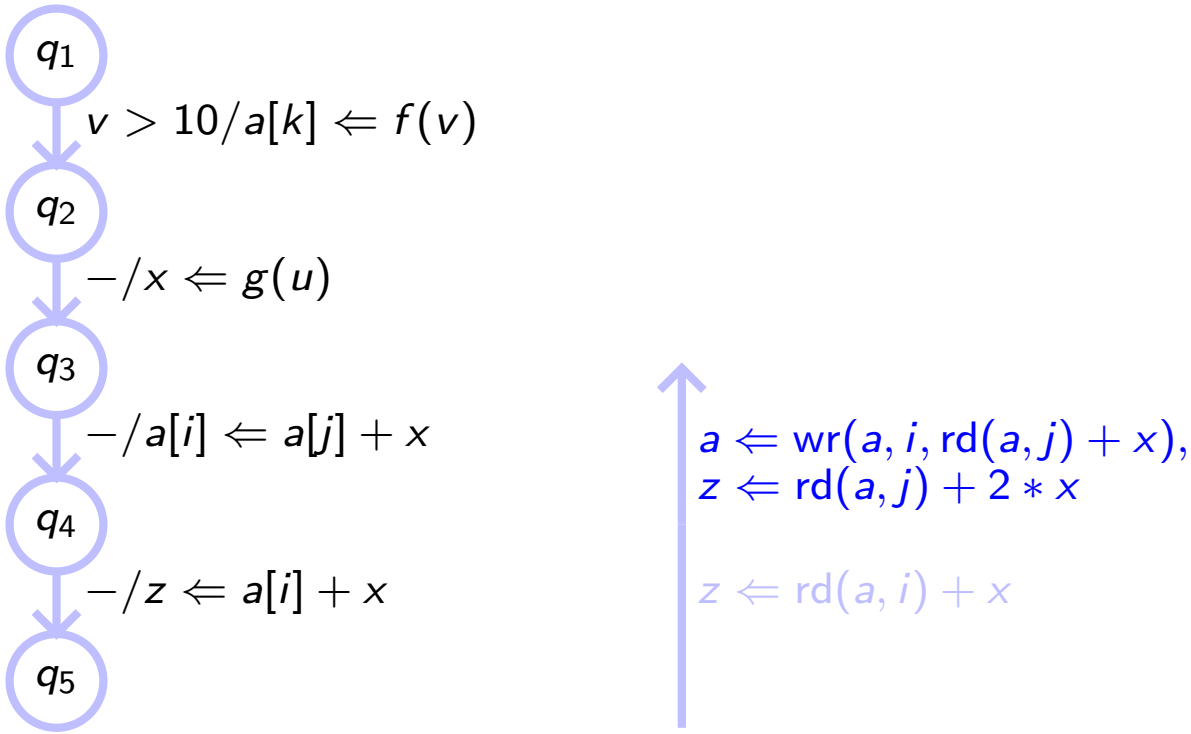
Computing characteristic tuple of a path

Computing path characteristics using backward substitution



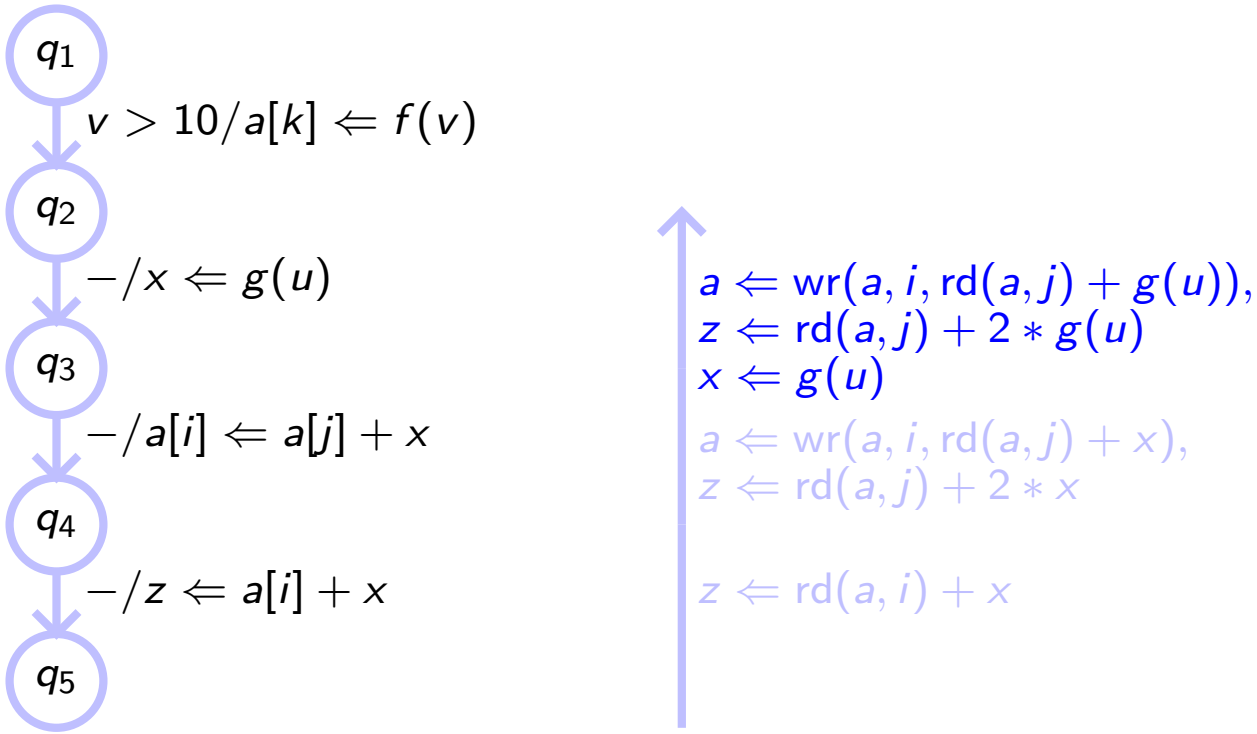
Computing characteristic tuple of a path

Computing path characteristics using backward substitution



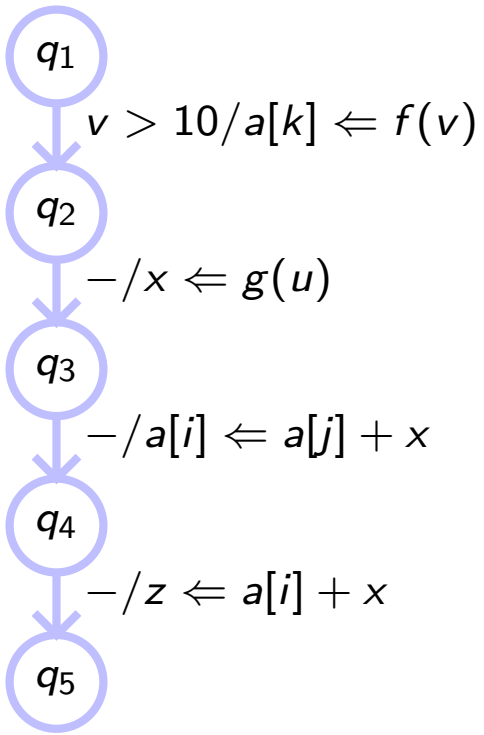
Computing characteristic tuple of a path

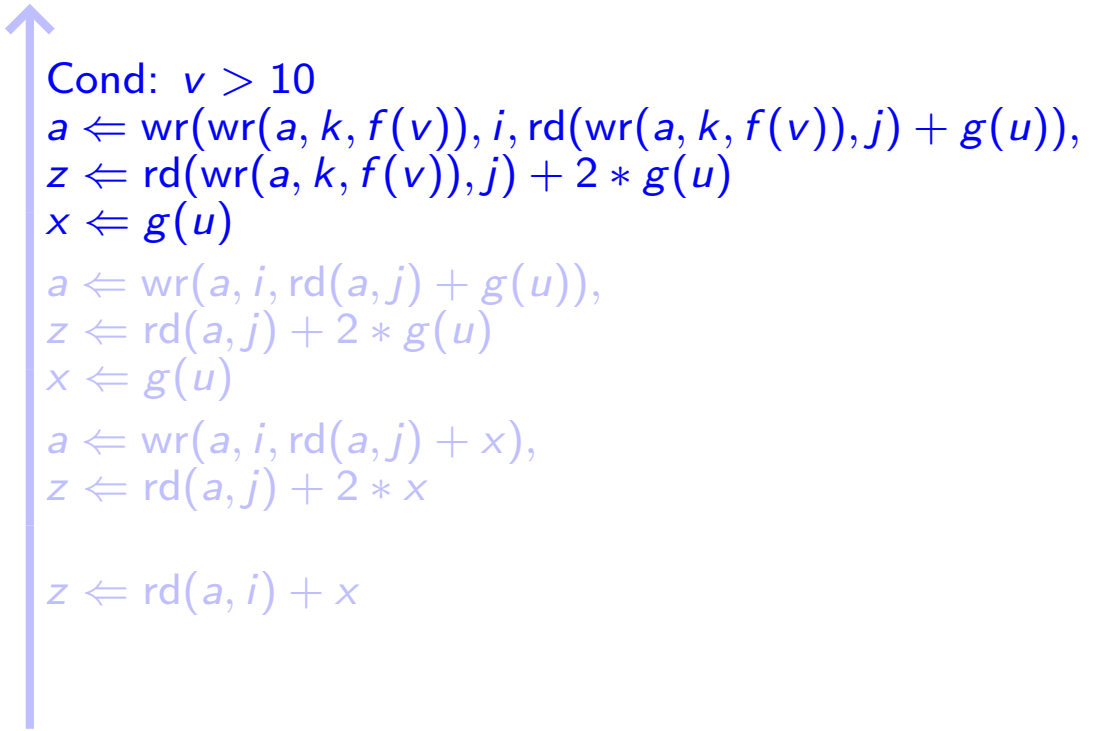
Computing path characteristics using backward substitution



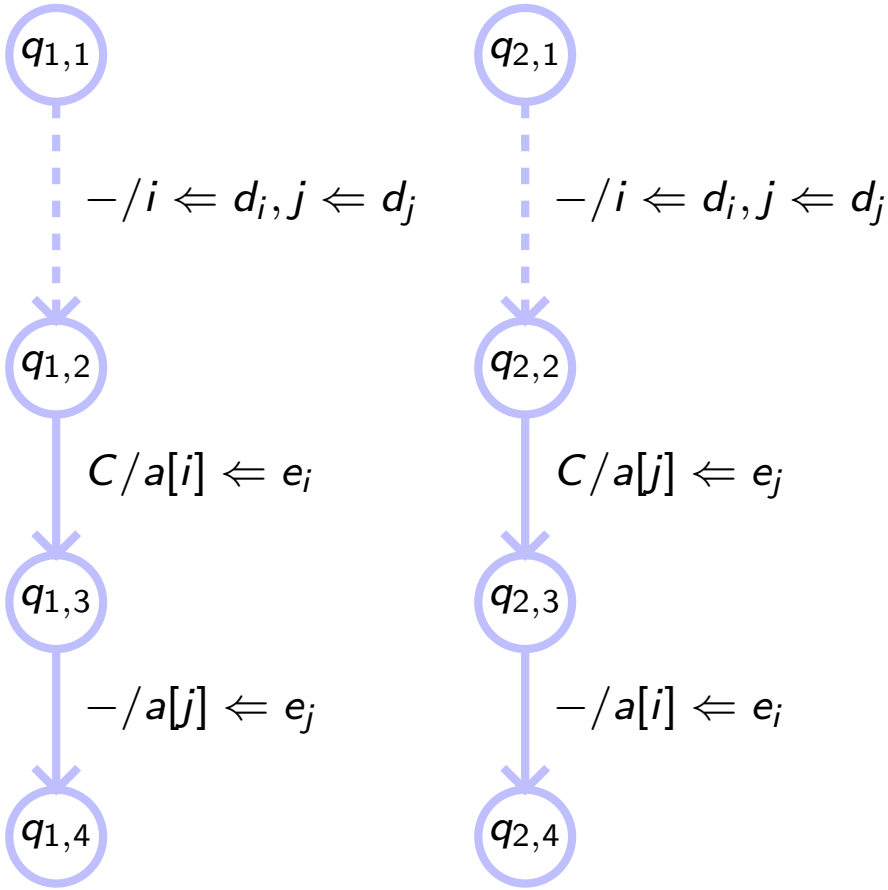
Computing characteristic tuple of a path

Computing path characteristics using backward substitution



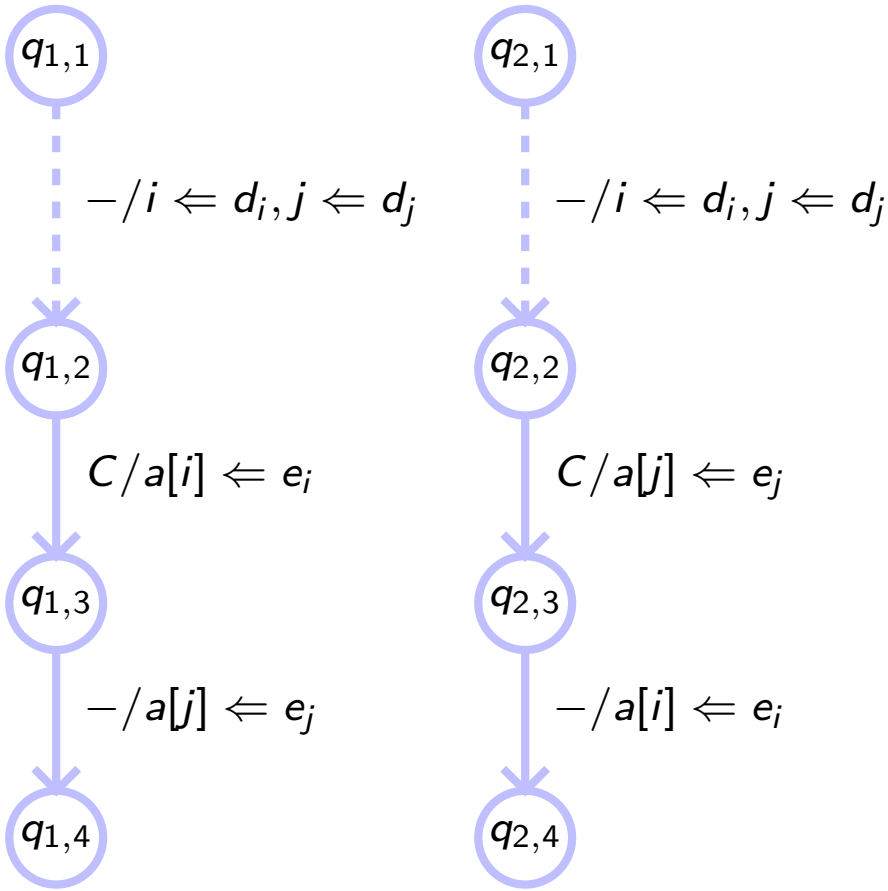


Propagation of index variables' values



(a) M_1 (b) M_2

Propagation of index variables' values

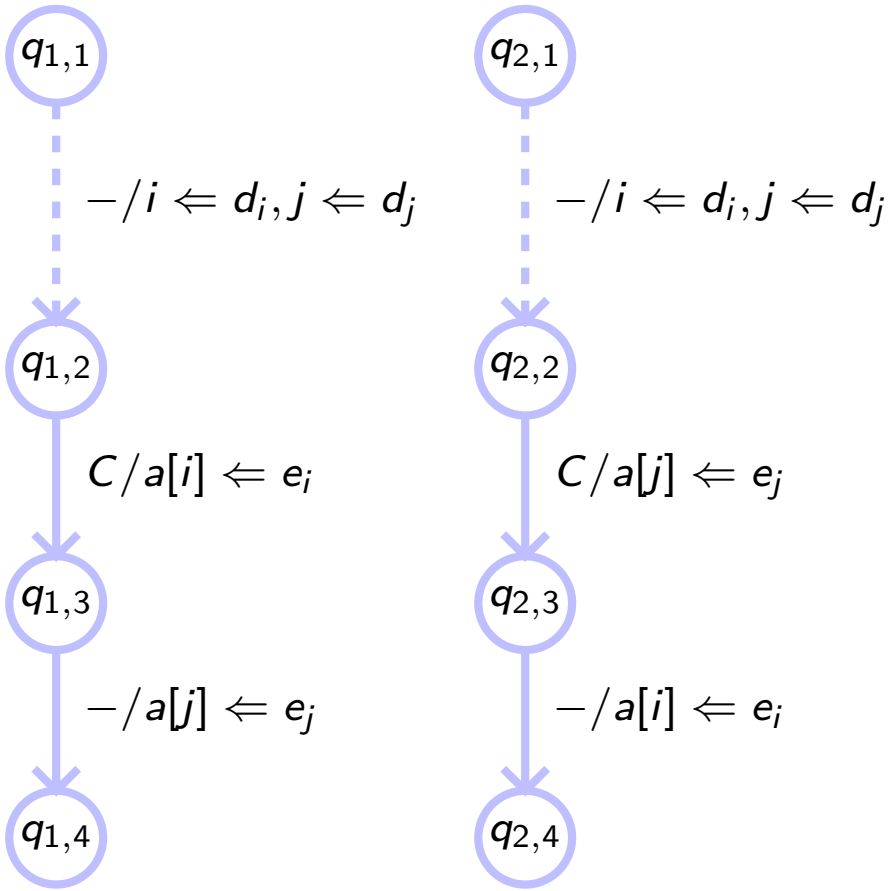


(a) M_1 (b) M_2

$$i \neq j \supset \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), i) = e_1 \ \wedge \ \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), j) = e_2$$

$$i = j \supset \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), i) = e_2 \ \wedge \ \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), j) = e_2$$

Propagation of index variables' values



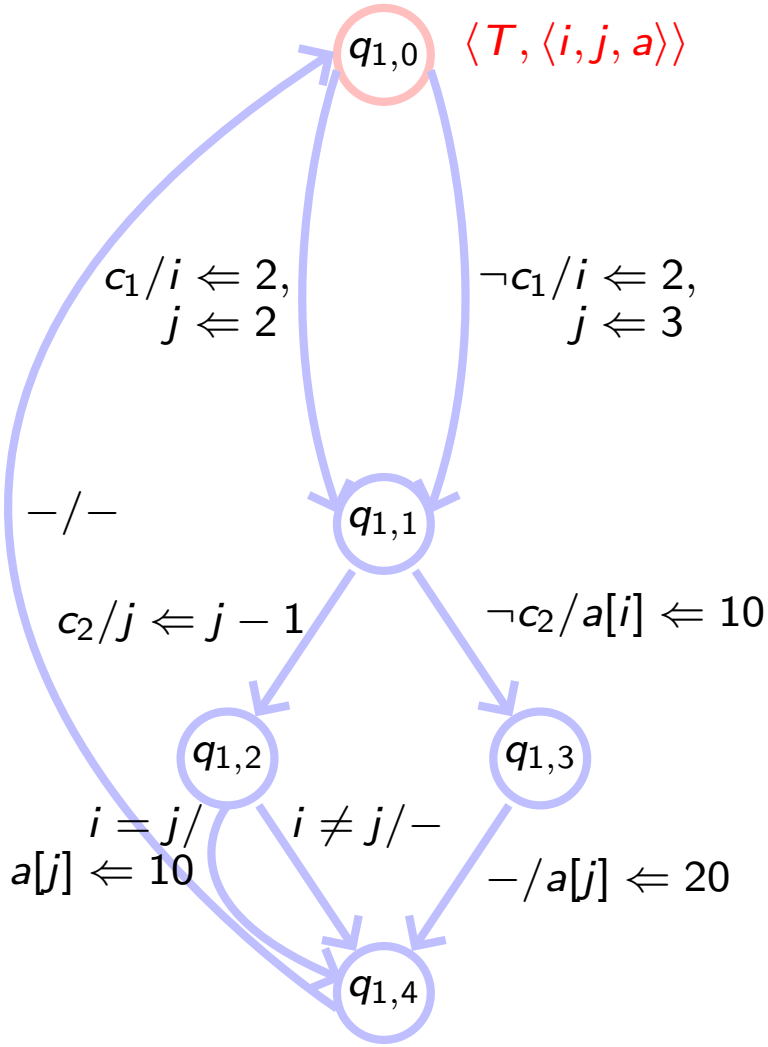
(a) M_1 (b) M_2

$$i \neq j \supset \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), i) = e_1 \ \wedge \ \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), j) = e_2$$

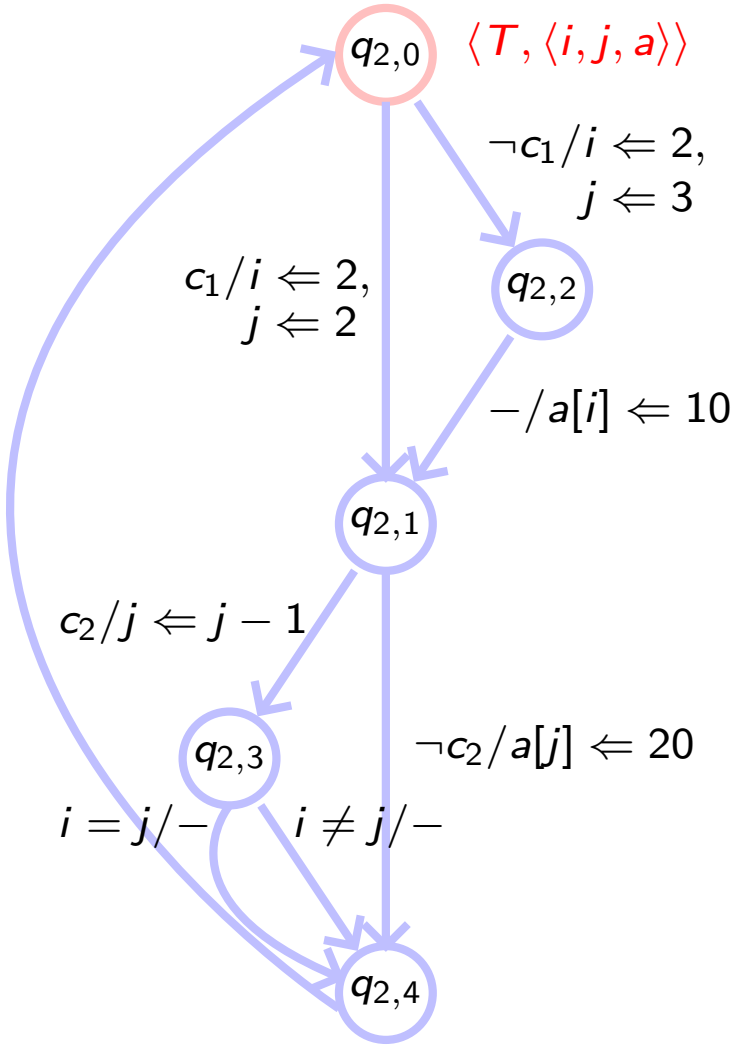
$$i = j \supset \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), i) = e_2 \ \wedge \ \mathrm{rd}(\mathrm{wr}(\mathrm{wr}(a, i, e_1), j, e_2), j) = e_2$$

Propagate index variable values inspite of a match

An illustrative example

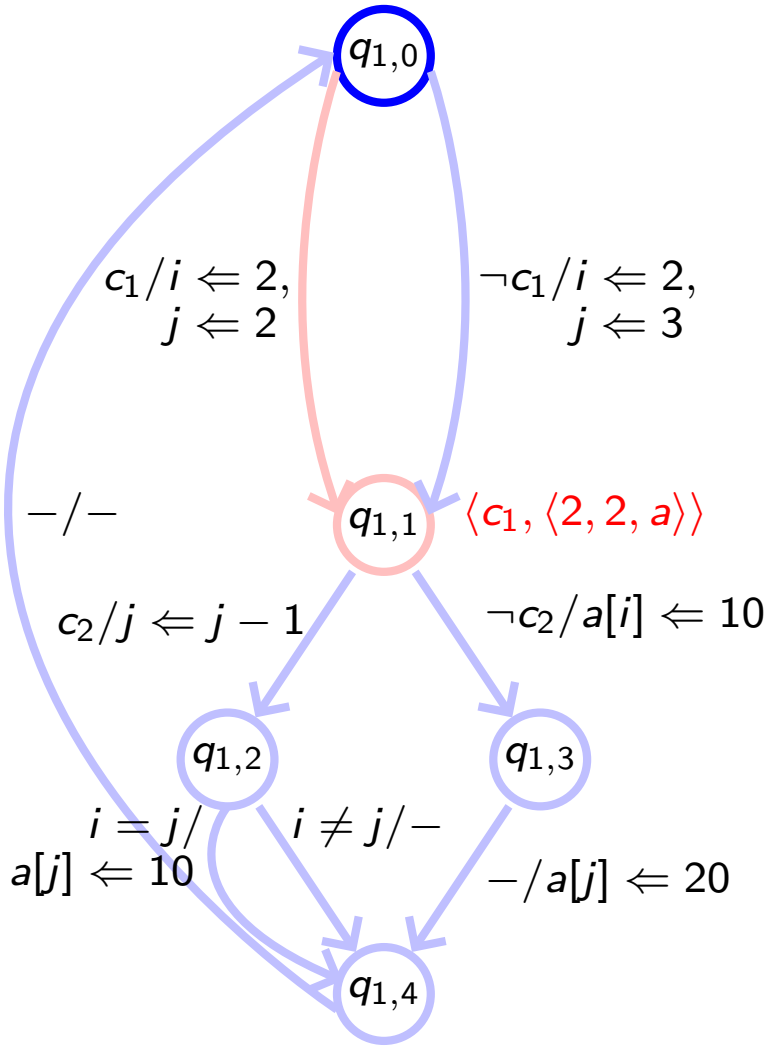


(a) M_1

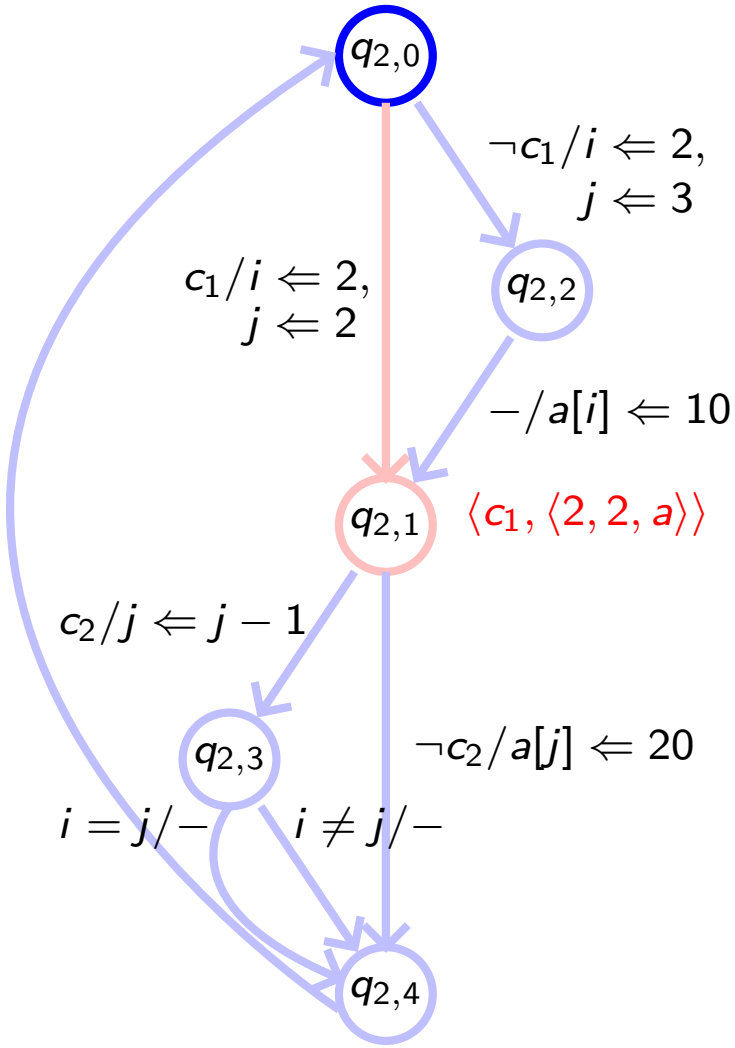


(b) M_2

An illustrative example

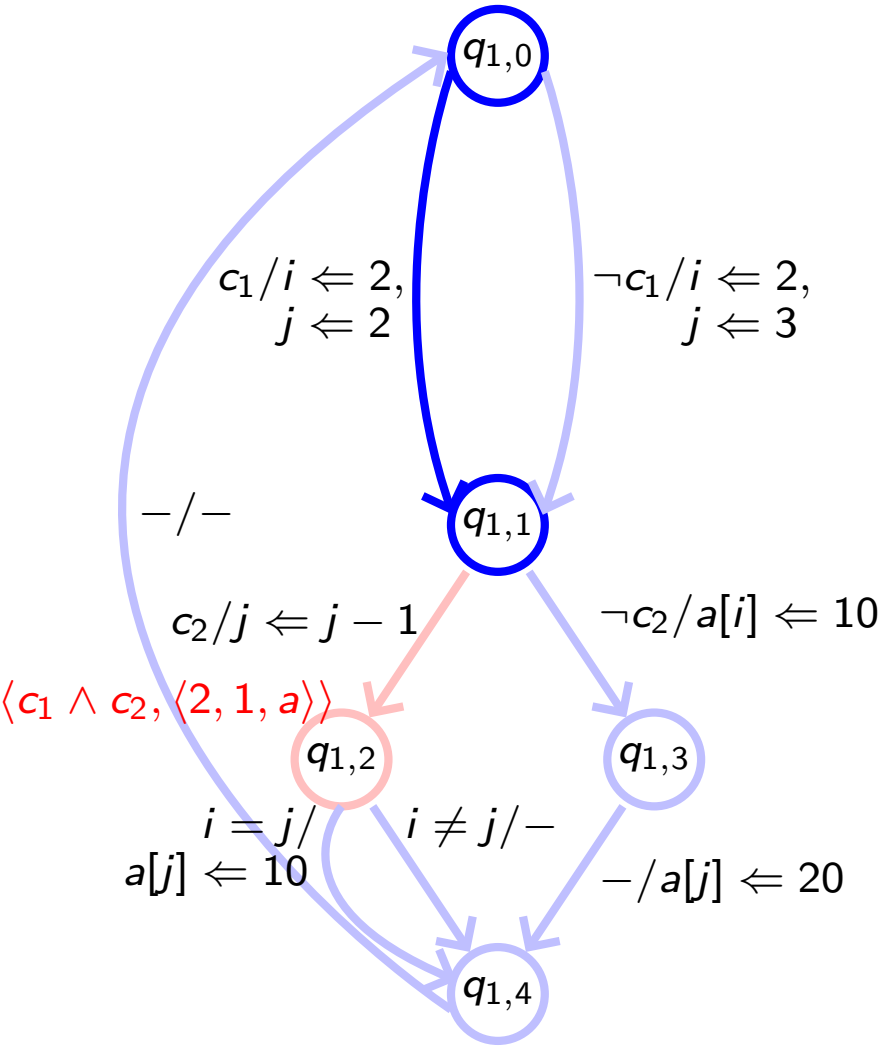


(a) M_1

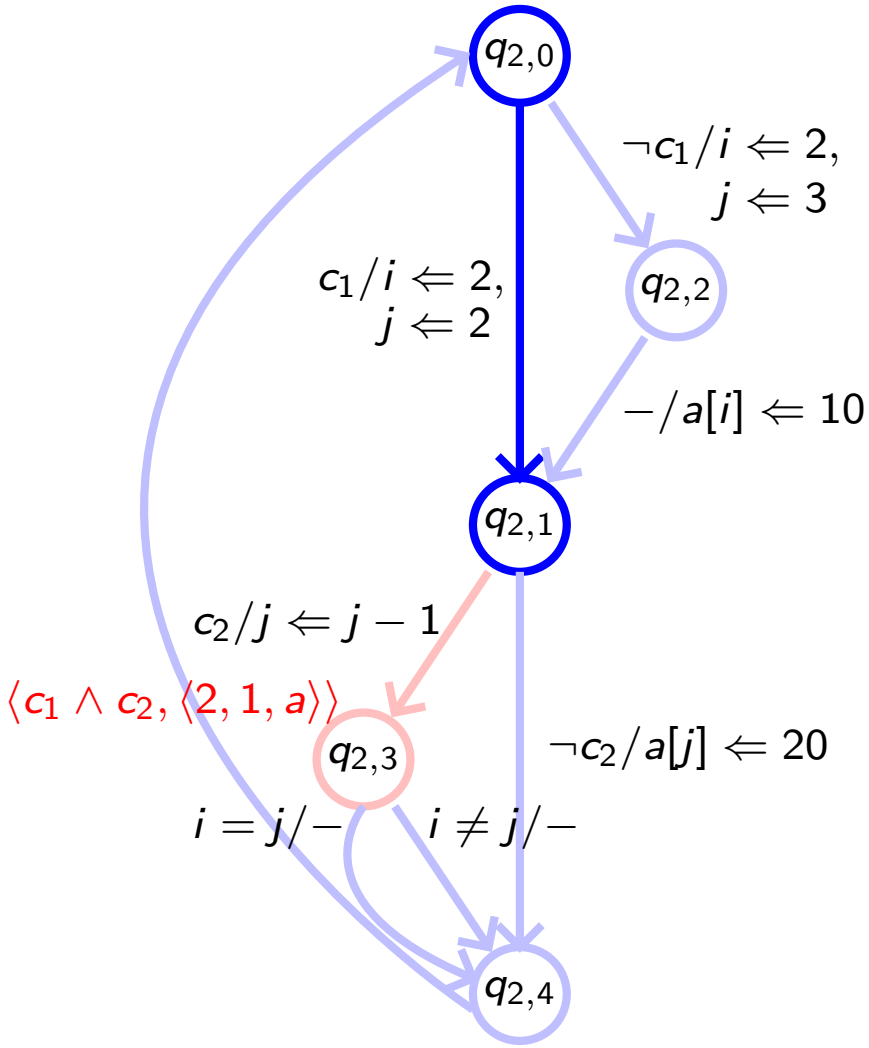


(b) M_2

An illustrative example

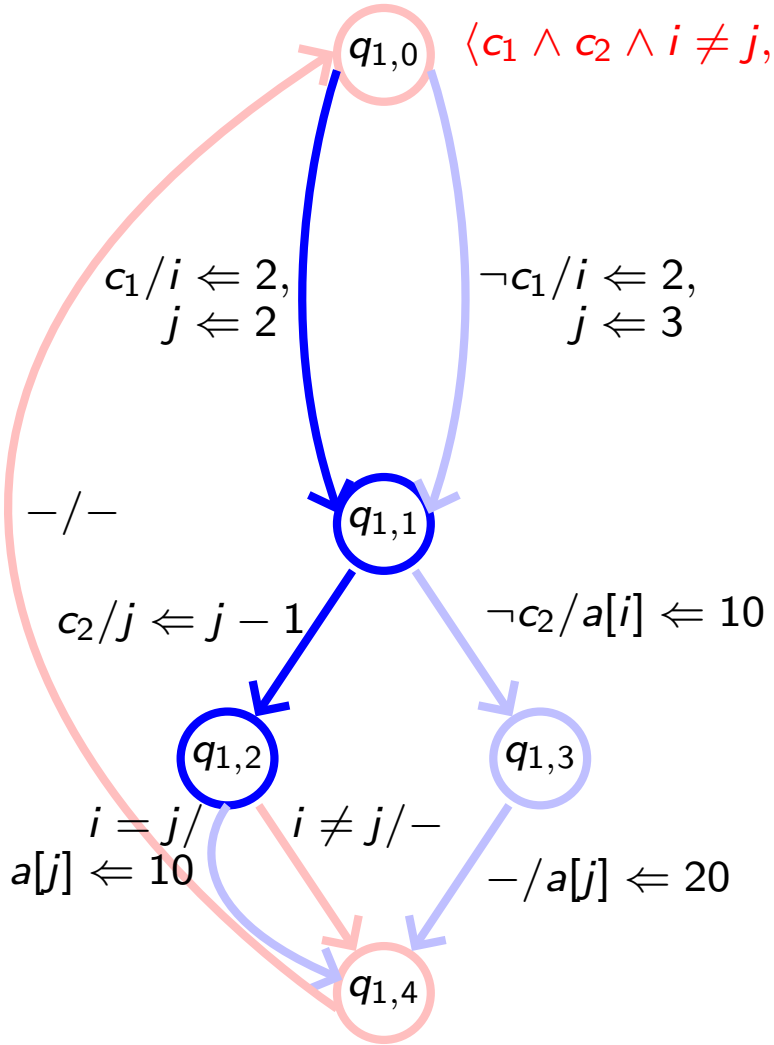


(a) M_1

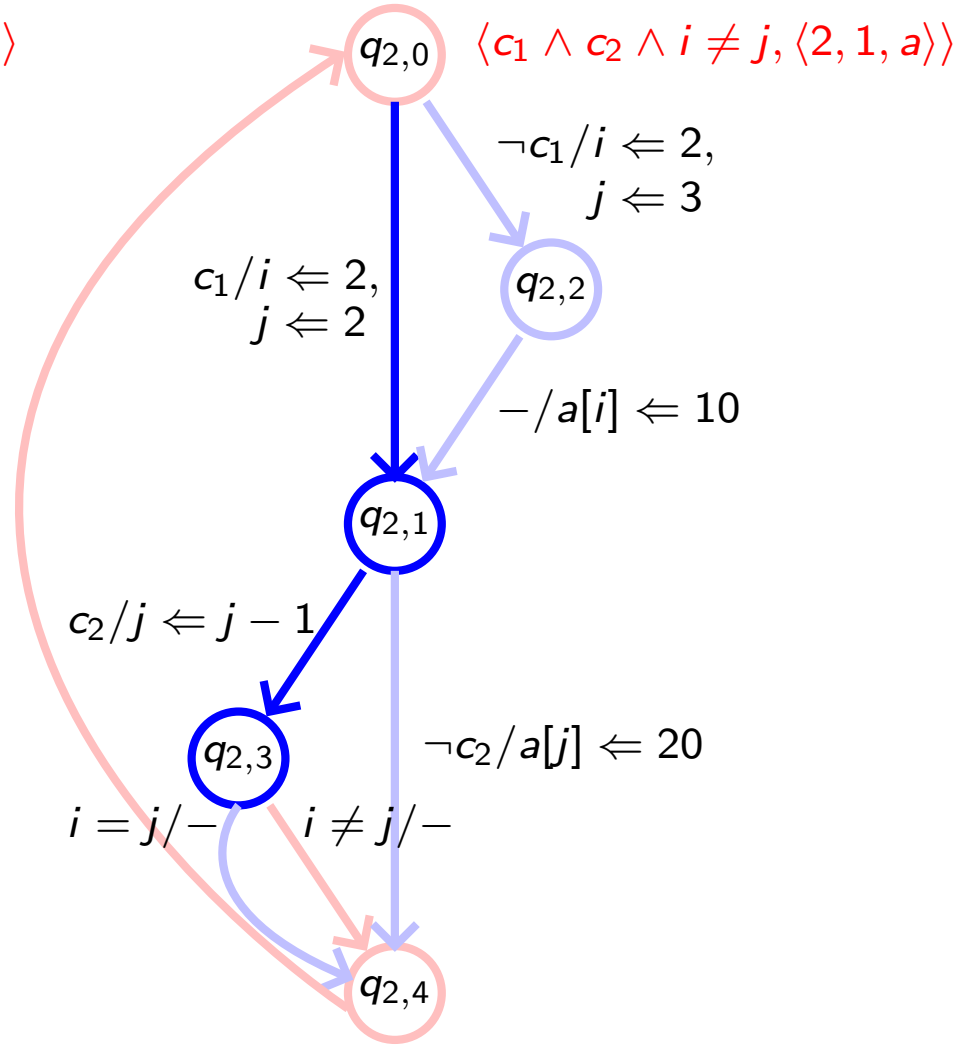


(b) M_2

An illustrative example

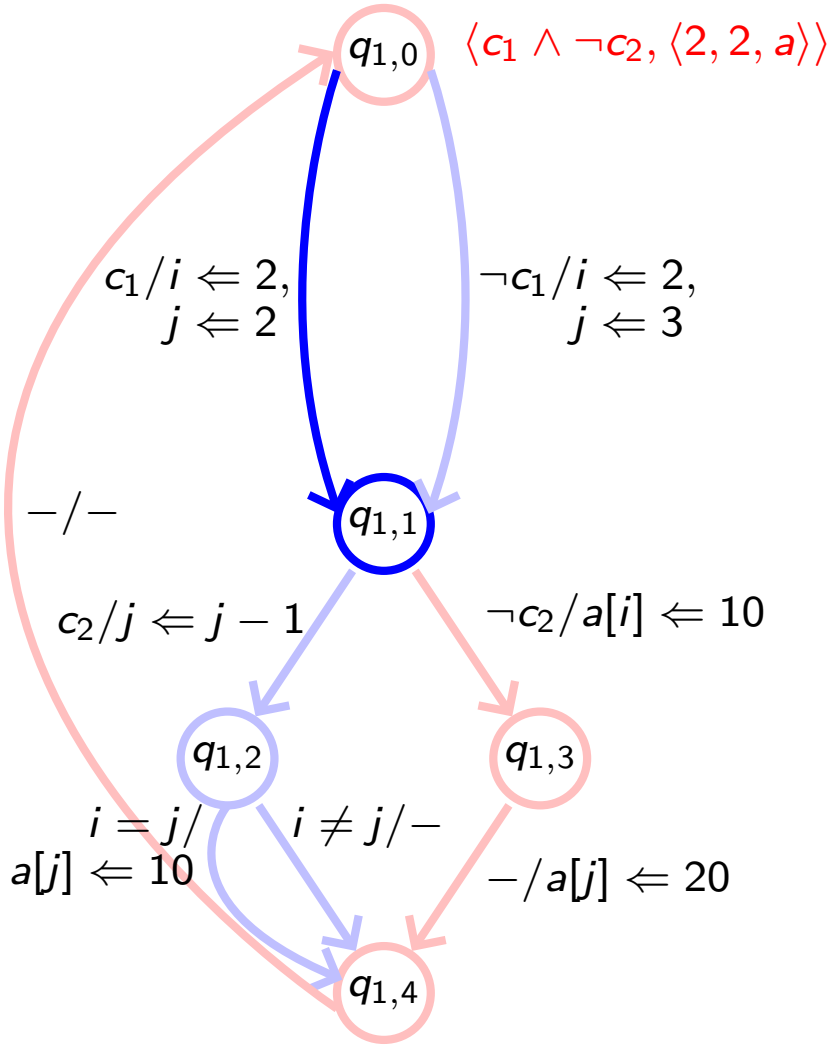


(a) M_1

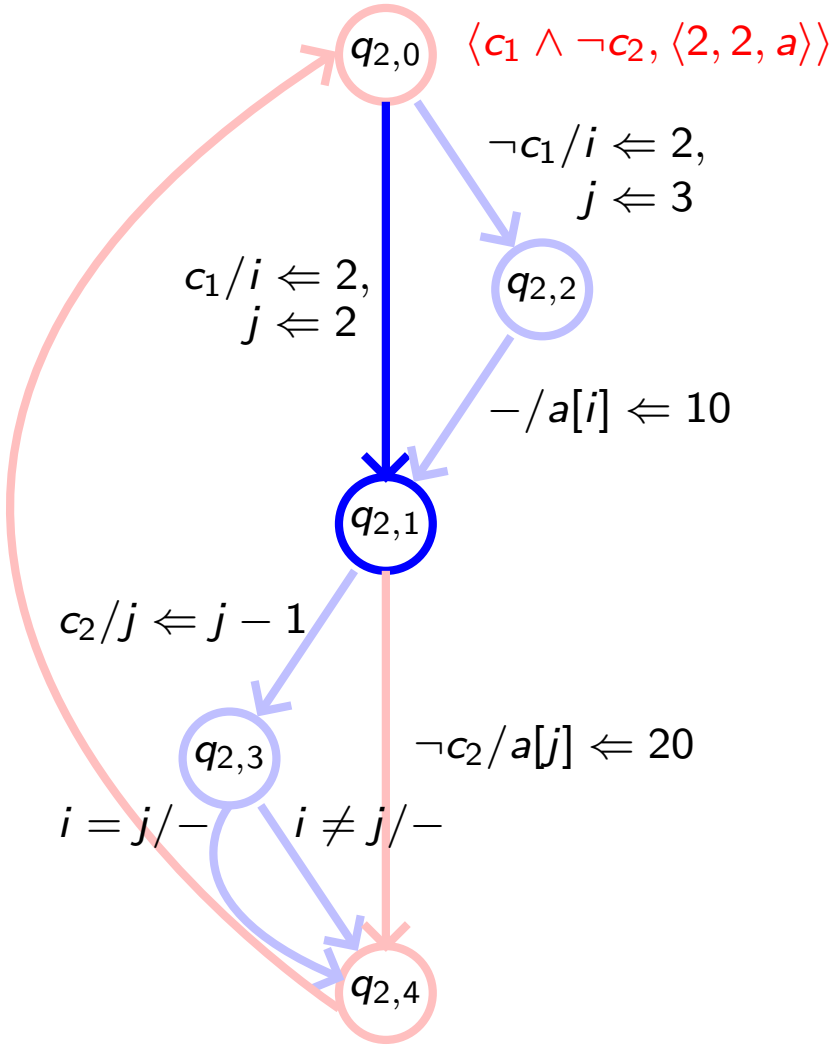


(b) M_2

An illustrative example

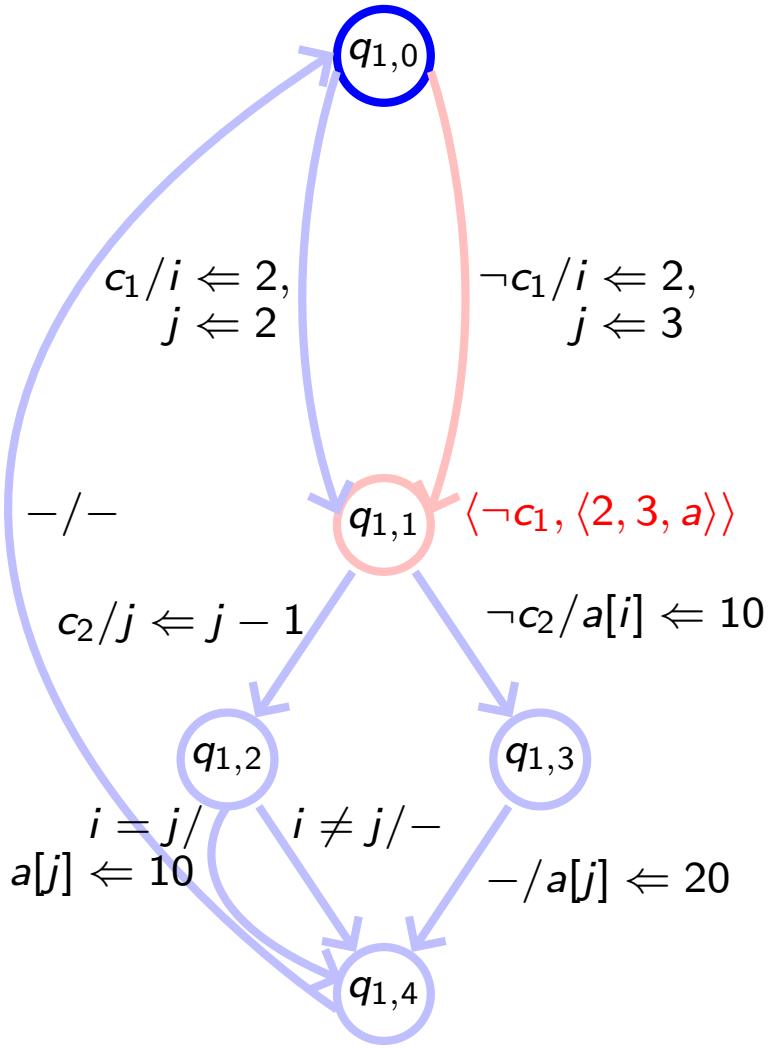


(a) M_1

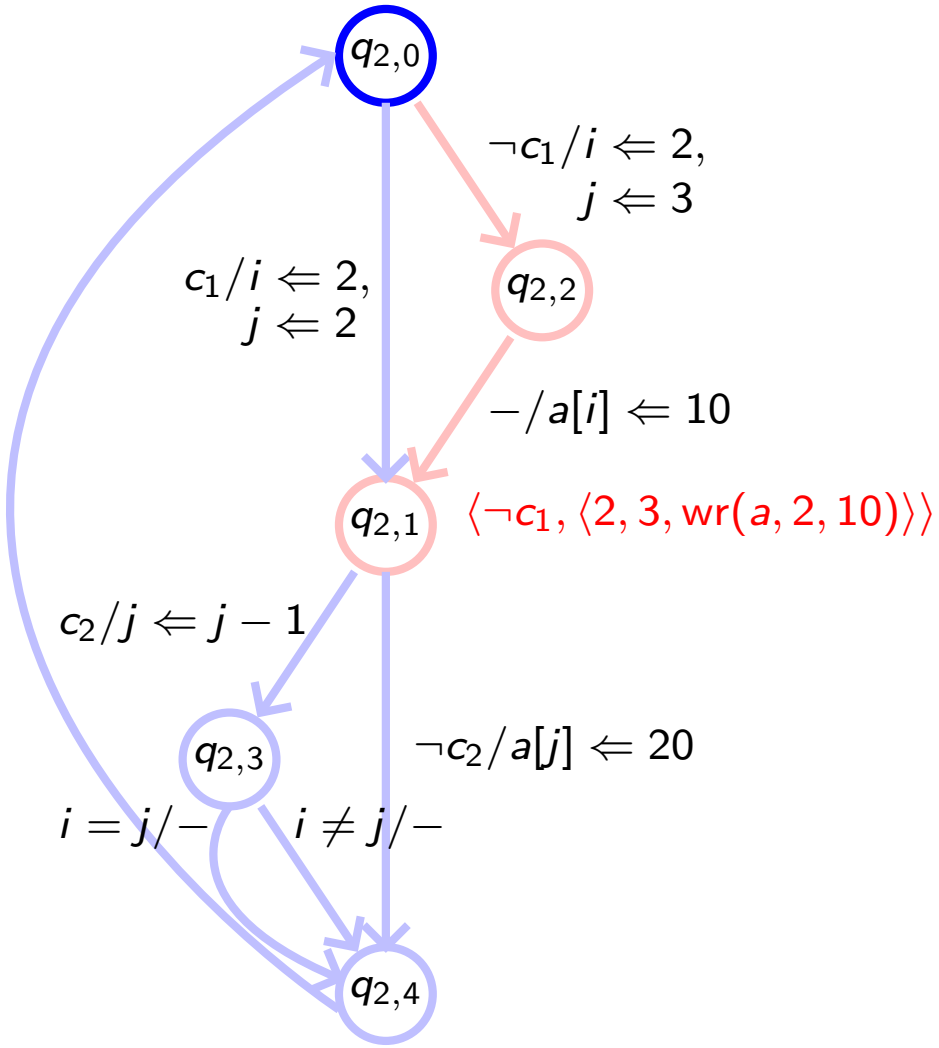


(b) M_2

An illustrative example

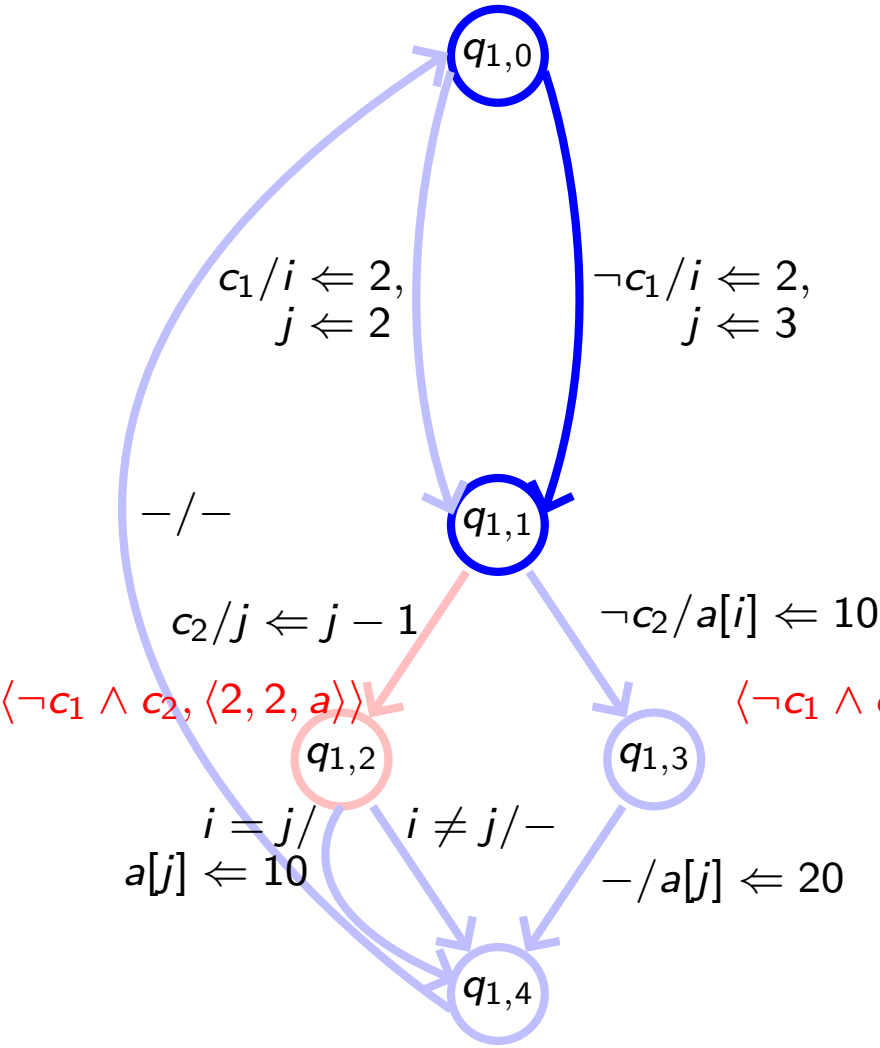


(a) M_1

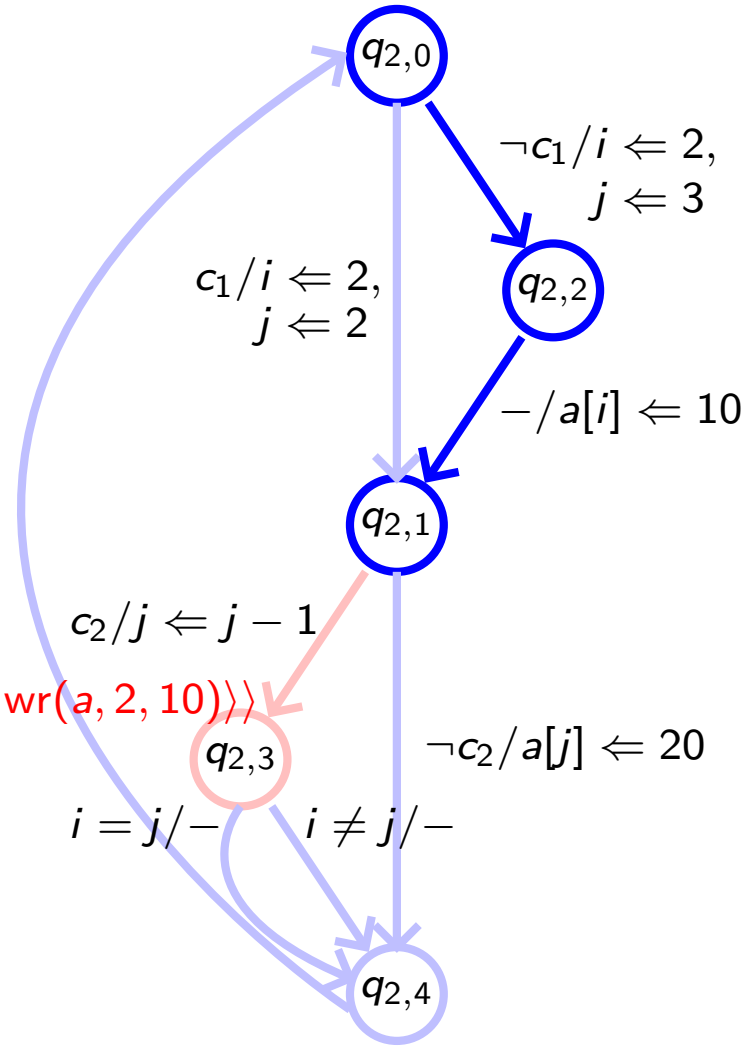


(b) M_2

An illustrative example

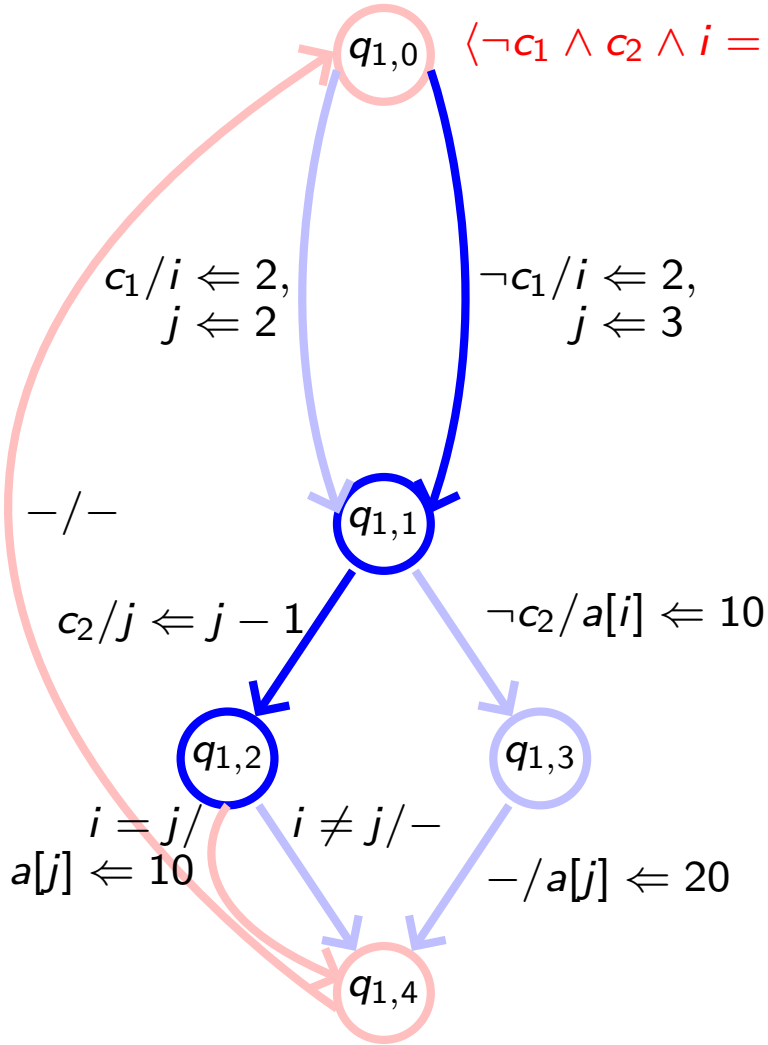


(a) M_1

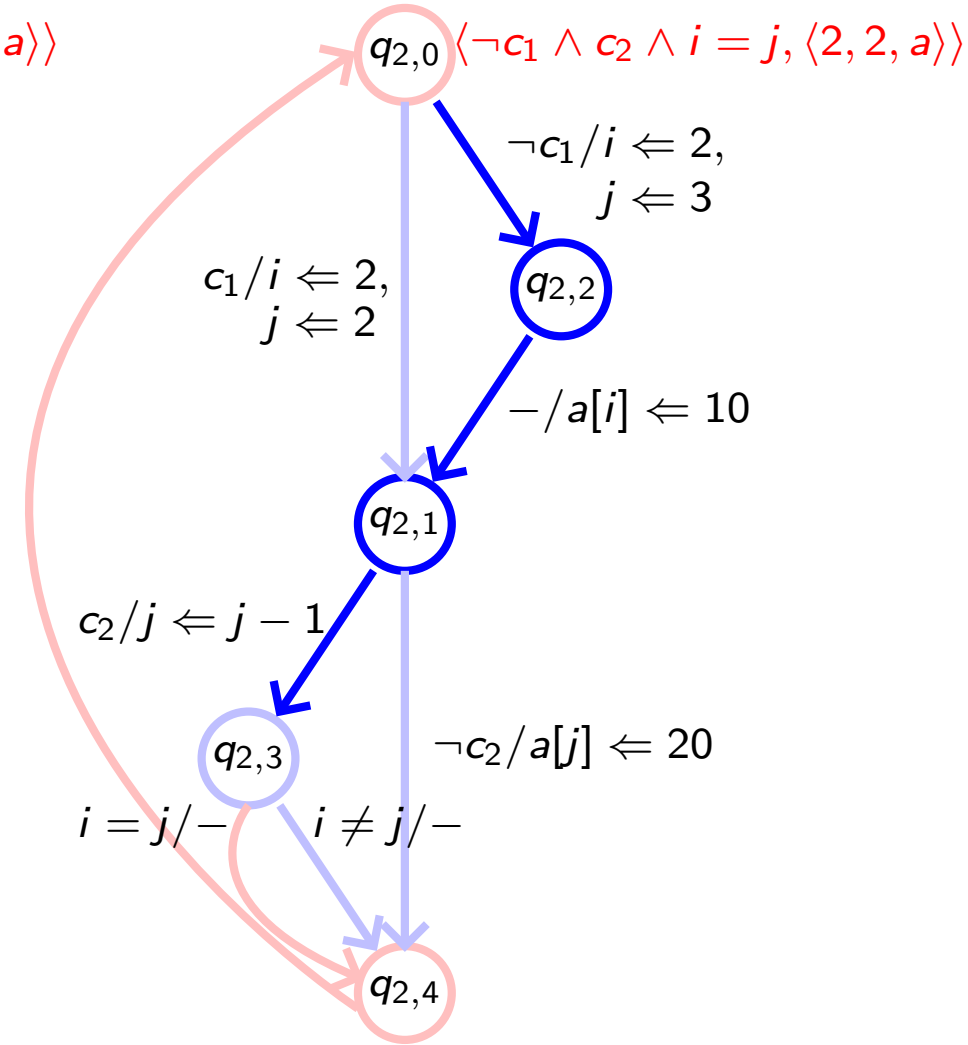


(b) M_2

An illustrative example

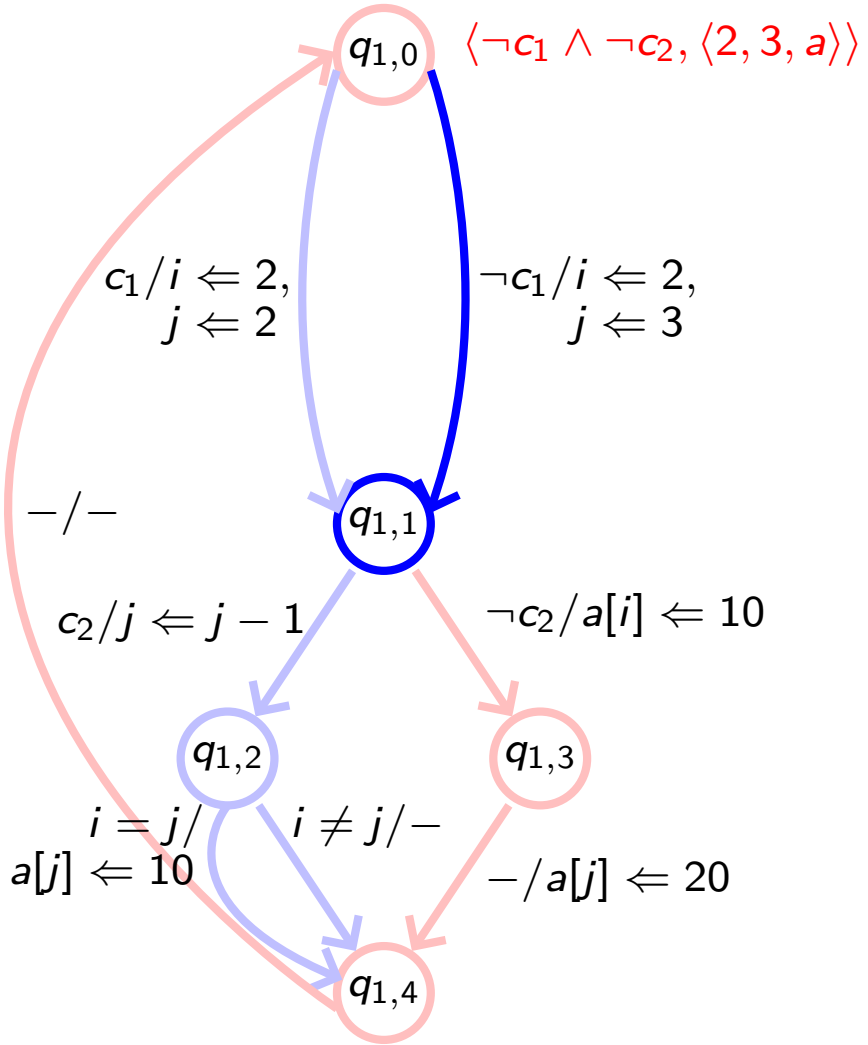


(a) M_1

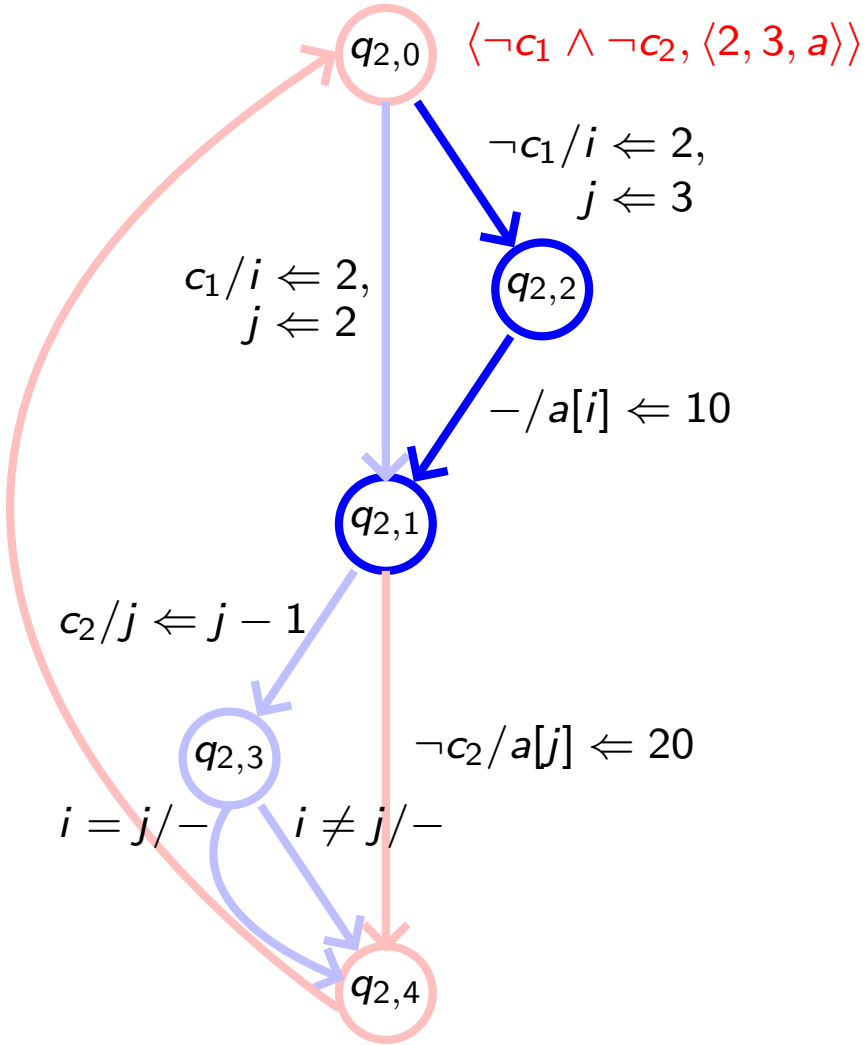


(b) M_2

An illustrative example



(a) M_1



(b) M_2

Experimental Results – 2

Benchmark	arr	op	BB	if	loop	path	Orig FSMDA		Trans FSMDA		Time
							state	scalar	state	scalar	(ms)
ASSORT	1	46	7	1	2	7	22	13	26	27	22
BLOWFISH	5	64	14	3	7	21	61	17	36	21	44
FFT	8	42	6	0	3	7	31	22	19	28	36
GCD	1	18	9	4	2	13	14	3	8	3	11
GSR	2	22	6	0	3	7	15	10	11	16	16
LU–DECOMP	2	56	18	2	8	21	47	8	44	20	21
LU–PIVOT	2	15	6	2	1	7	15	6	11	6	6
LU–SOLVE	4	34	10	0	5	11	28	4	22	14	12
MINSORT	1	22	6	1	2	7	20	7	11	7	7
SB–BALANCE	3	24	14	2	4	13	22	3	13	6	9
SB–PC	3	25	11	1	4	11	21	6	15	12	8
SB–SAC	3	21	11	1	4	11	20	5	14	8	8
SVD	4	279	82	17	36	107	250	25	220	77	93
TR-LCOST	4	74	25	4	9	27	56	14	46	31	35
TR-NWEST	5	72	25	4	9	27	54	13	47	29	31
TR-VOGEL	8	160	54	17	14	63	96	13	70	68	68
WAVELET	2	41	4	1	2	7	25	8	14	32	14
ERRONEOUS	1	22	5	1	2	7	16	6	15	10	6

☞ Detected a bug in the implementation of copy propagation for array variables in the SPARK compiler.

★ K Banerjee et al., "Extending the FSMD Framework for Validating Code Motions of Array-Handling Programs," TCAD 2014.

Outline

1 Background

2 Translation validation of code motion transformations

3 Deriving bisimulation relations from path based equivalence checkers

4 Translation validation of code motion transformations in array-intensive programs

5 **Augmenting equivalence checkers with SMT solvers**
- SMT solvers

6 Other Models of Computation

7 Conclusion and scope of future work

SMT solvers

SMT: Satisfiability Modulo Theories

The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality.
(source: `wikipedia.org`)

Example: $3x + 2y \geq 4,\ x, y \in \mathbb{N}$

SMT solvers used in this work: CVC4, Yices2, Z3

Other SMT solvers: Beaver, Boolector, MiniSmt, SONOLAR

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

✓

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

How to establish equivalence of expressions?

$$X + \bar{Y}.Z \equiv X + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

Convert both expressions into sum-of-minterms

$$X.(Y + \bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.(X + \bar{X}) \equiv$$
$$X.(Y + \bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + \bar{Y}.Z.X + \bar{Y}.Z.\bar{X} \equiv$$
$$(X.Y + X.\bar{Y}).(Z + \bar{Z}) + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \ ?$$

$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv$$
$$X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z$$

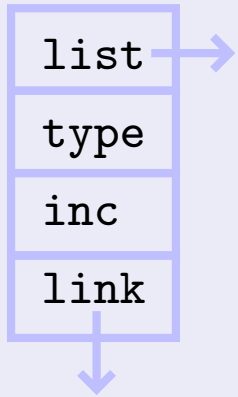
A normalization technique for integers

No canonical representation exists for expressions over integers.

Structure of a normalized cell

```
typedef struct normalized_cell NC;

struct normalized_cell {
  NC   *list;
  char type;
  int  inc;
  NC   *link;
};
```



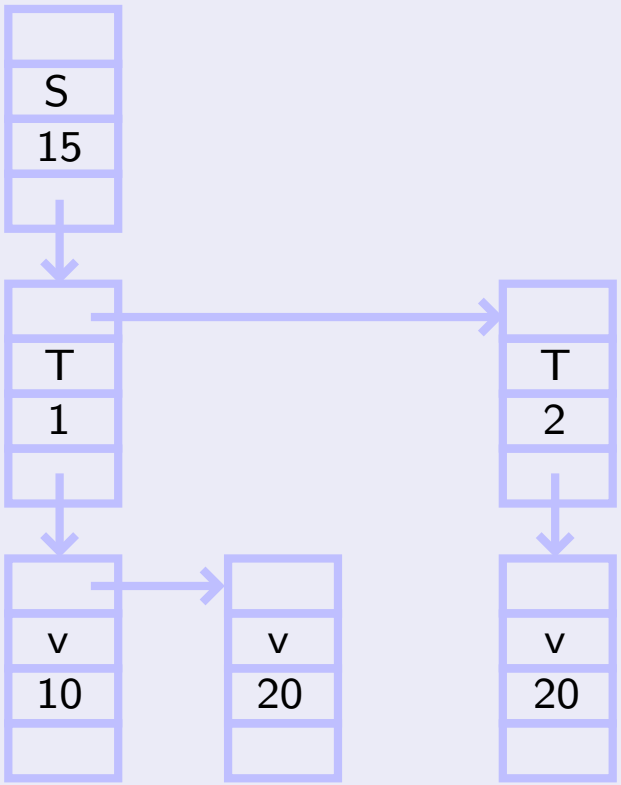
Proposed by *J. C. King, "A Program Verifier," PhD thesis, Carnegie-Mellon University, 1969.*

An example of normalized expression over integers

Expression: $1 \times y \times z + 2 \times z + 15$

Symbols

Loc	Var
y	10
z	20



Normalization can show that this expression is equivalent to $z + z \times y + z + 20 - 5$.

Normalization grammar

Grammar

1) $S \rightarrow S + T \mid c_s$, where c_s is an integer.
2) $T \rightarrow T * P \mid c_t$, where c_t is an integer.
3) $P \rightarrow \mathrm{abs}(S) \mid (S) \bmod (S) \mid S \div C_d \mid v \mid c_p$, where $v \in I \cup V$, and c_p is an integer.
4) $C_d \rightarrow S \div C_d \mid S$.

Some simplification rules for integers are given in [TCAD08].
This grammar is latter applied on reals also in [TODAES12].

Limitations of the normalization method

An example where normalization fails

```
if( a != b ) {
  n := a×a - 2×a×b + b×b;
  d := a - b;
  x := n / d;
}
```

```
if( a != b ) {
  x := a - b;
}
```

- The normalization technique resolves equivalence of expressions by reducing them to the same syntactical structure and does not actually *solve* the expressions by substituting for variables.
- The normalization technique does not account for bit-vectors and user-defined datatypes.

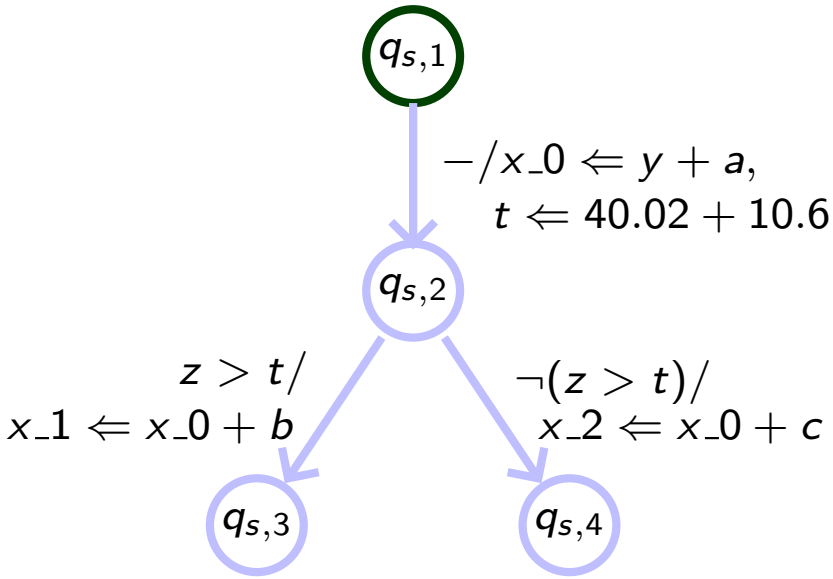
Single assignment form: A prerequisite for SMT Solvers

An example to highlight single assignment form

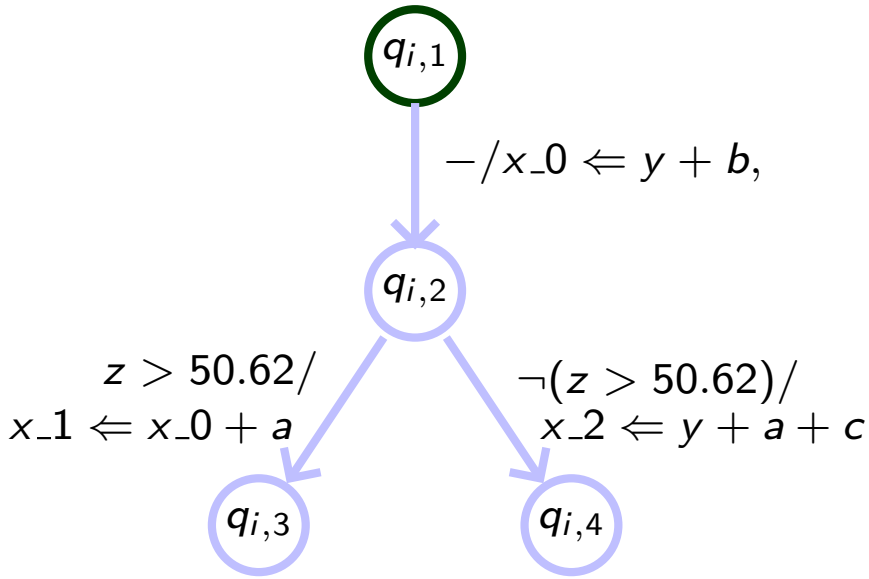
```
S1: x := a + b;     x_0 := a + b;     ASSERT x_0 = a + b;
S2: x := x + c;     x_1 := x_0 + c;   ASSERT x_1 = x_0 + c;
S3: y := x + d;     y := x_1 + d;     ASSERT y = x_1 + d;
      (a)                (b)                (c)
```

- The *order of execution* of the statements is not captured by the ordering of the *assert* statements.
- Programs in single assignment form help in producing *assert* statements whose ordering is irrelevant, that is, they can be arranged in any order to produce the same effect.

Formula generation for SMT solvers



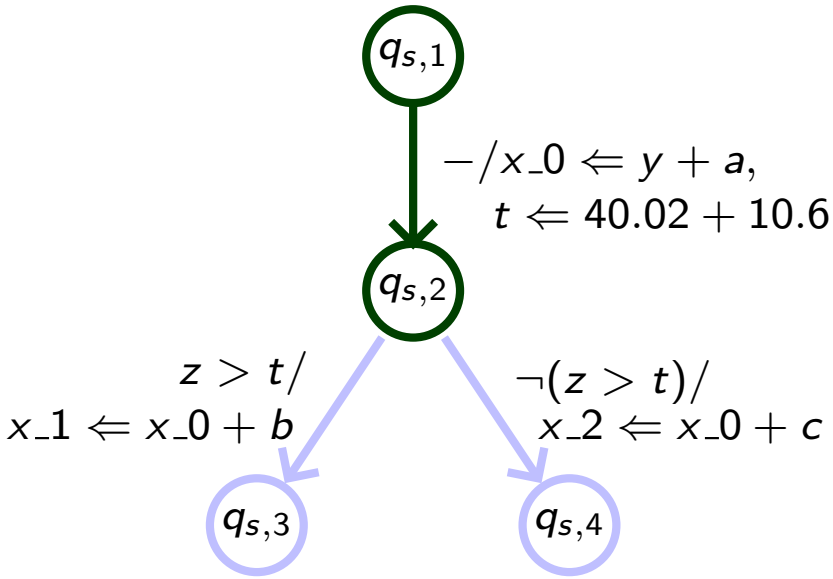
(a) M_s



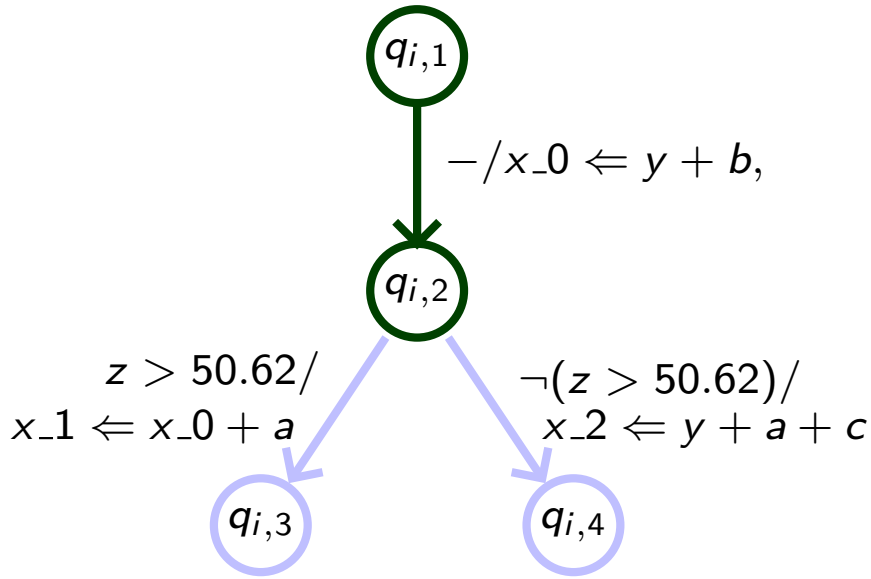
(b) M_i

Here, we have considered the path based equivalence checker of [ISED12].

Formula generation for SMT solvers



(a) M_s



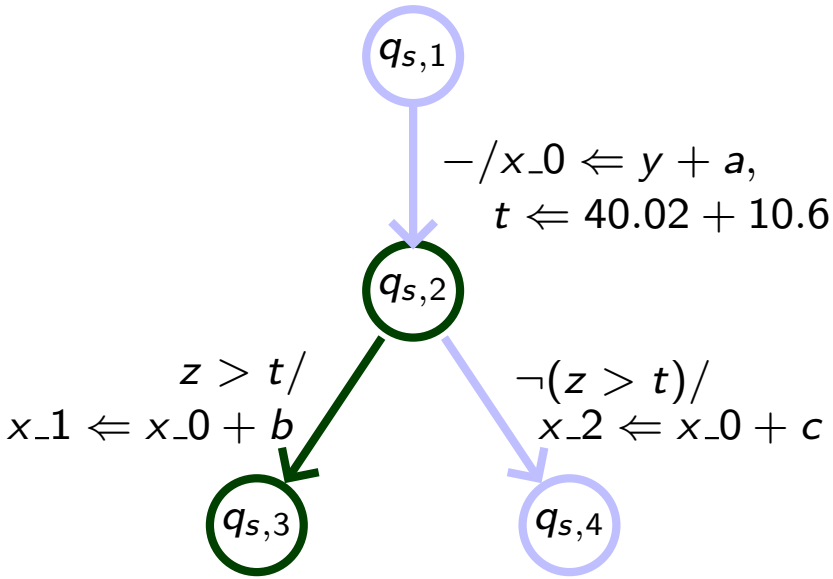
(b) M_i

Encoding in CVC4 input language

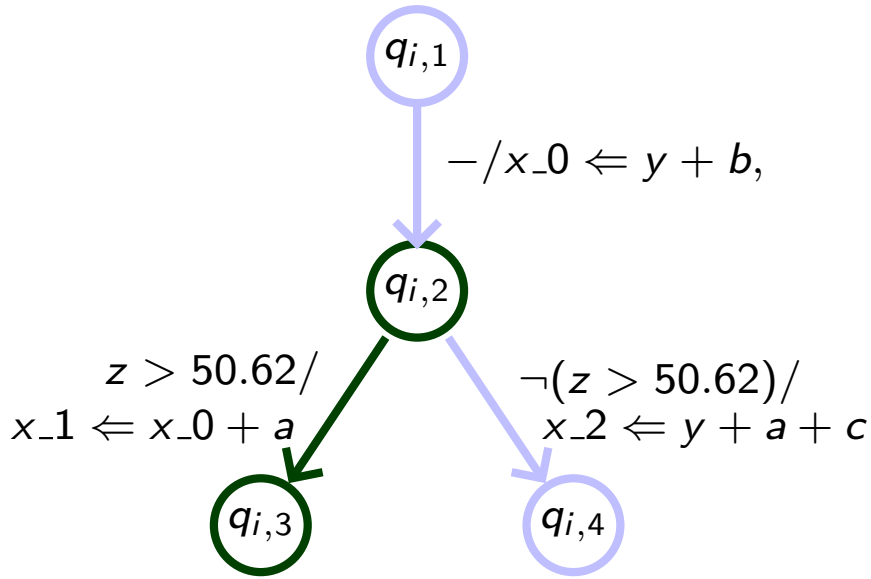
```
y_s:INT; a_s:INT; x_0_s:INT; t_s:REAL;
y_i:INT; b_i:INT; x_0_i:INT;
ASSERT y_s = y_i;
ASSERT x_0_s = y_s+a_s; ASSERT t_s = 40.02 + 10.6;
ASSERT x_0_i = y_i + b_i;
QUERY x_0_s = x_0_i;
```

Output: `invalid` (need to look beyond this basic block)

Formula generation for SMT solvers



(a) M_s



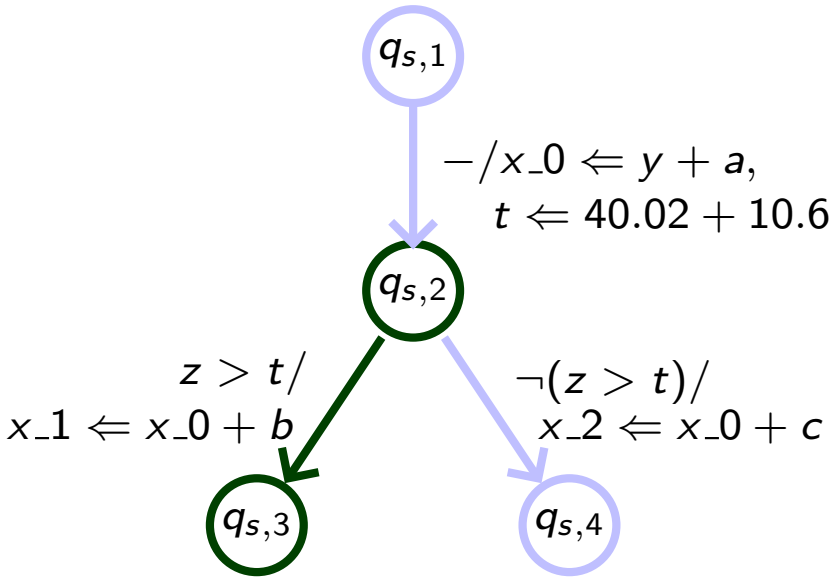
(b) M_i

Encoding in CVC4 input language (appended with the previous one)

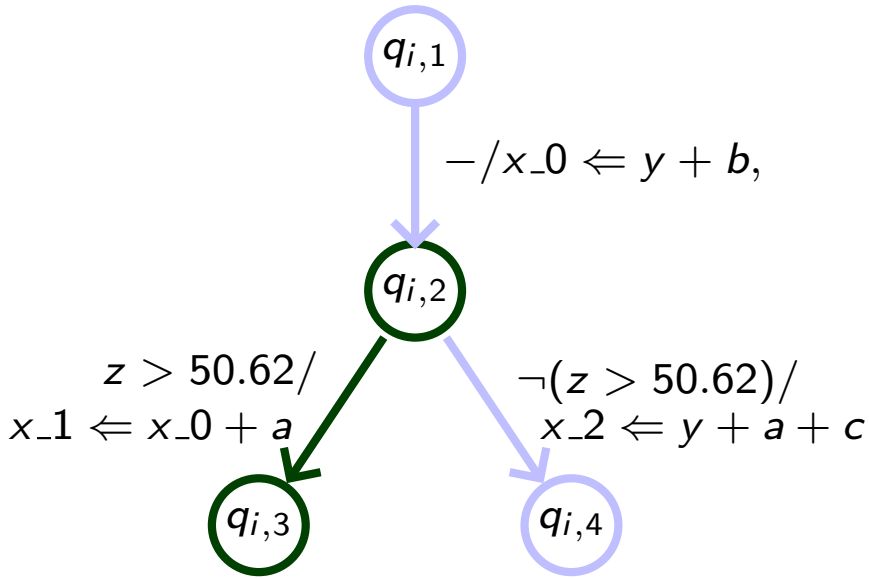
```
z_s:REAL; cond_s:BOOLEAN;
z_i:REAL; cond_i:BOOLEAN;
ASSERT z_s = z_i;
ASSERT cond_s = z_s > t_s;
ASSERT cond_i = z_i > 50.62;
QUERY cond_s = cond_i;
```

Output: `valid` (these two branches run in synchrony)

Formula generation for SMT solvers



(a) M_s



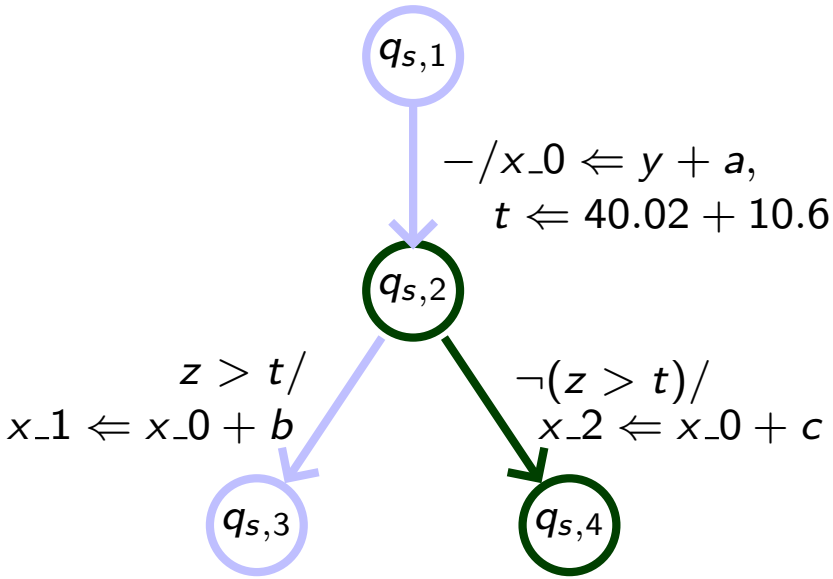
(b) M_i

Encoding in CVC4 input language (appended with the earlier one)

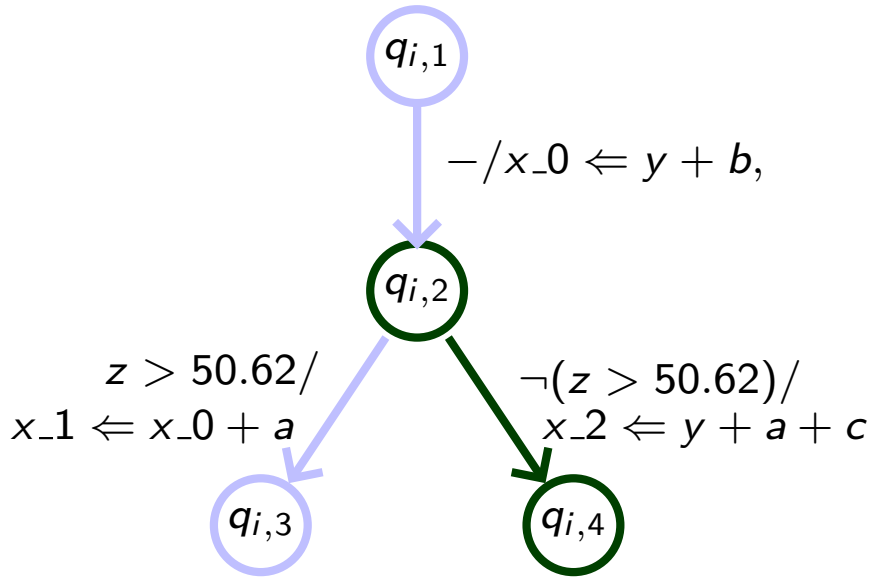
```
b_s:INT; x_1_s:INT;
a_i:INT; x_1_i:INT;
ASSERT a_s = a_i; ASSERT b_s = b_i;
ASSERT x_1_s = x_0_s + b_s;
ASSERT x_1_i = x_0_i + a_i;
QUERY x_1_s = x_1_i;
```

Output: `valid` (the computations match at states $q_{s,3}$ and $q_{i,3}$)

Formula generation for SMT solvers



(a) M_s



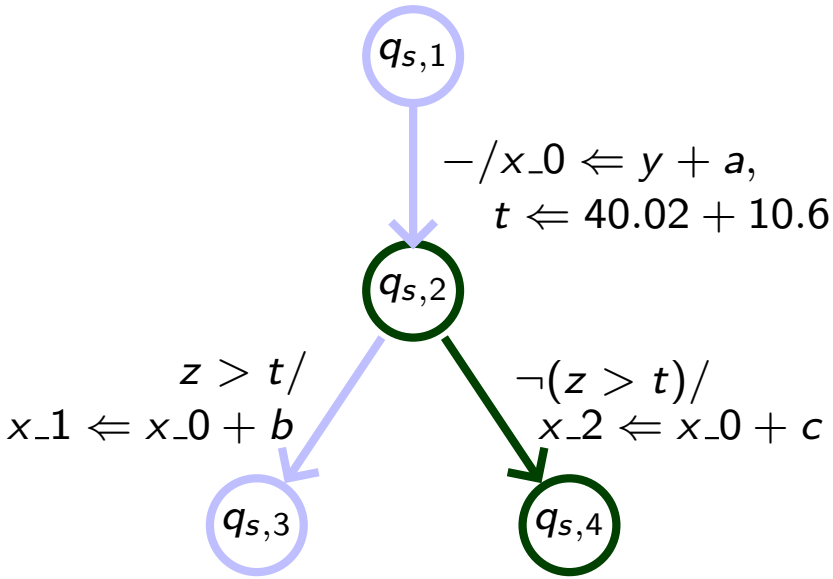
(b) M_i

Encoding in CVC4 input language (appended with the earliest one)

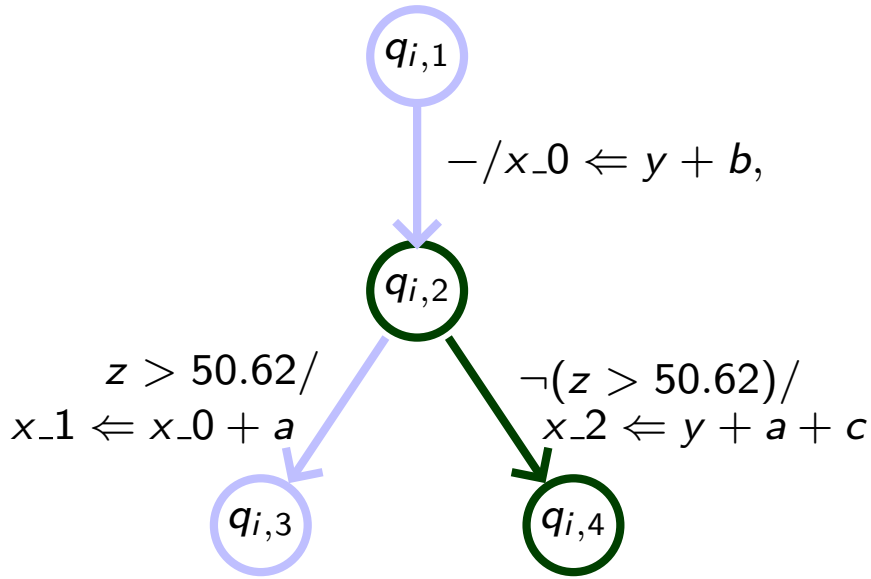
```
z_s:REAL; cond_s:BOOLEAN;
z_i:REAL; cond_i:BOOLEAN;
ASSERT z_s = z_i;
ASSERT cond_s = z_s <= t_s;
ASSERT cond_i = z_i <= 50.62;
QUERY cond_s = cond_i;
```

Output: `valid` (these two branches run in synchrony)

Formula generation for SMT solvers



(a) M_s



(b) M_i

Encoding in CVC4 input language (appended with the earliest one)

```
c_s:INT; x_2_s:INT;
a_i:INT; c_i:INT; x_2_i:INT;
ASSERT a_s = a_i; ASSERT b_s = b_i; ASSERT c_s = c_i;
ASSERT x_2_s = x_0_s + c_s;
ASSERT x_2_i = y_i + a_i + c_i;
QUERY x_2_s = x_2_i;
```

Output: `valid` (the computations match at states $q_{s,4}$ and $q_{i,4}$)

Revisiting the example where normalization fails

An example where normalization fails

```
if( a != b ) {
  n := a×a - 2×a×b + b×b;
  d := a - b;
  x := n / d;
}
```

```
if( a != b ) {
  x := a - b;
}
```

Encoding in SMT2 input language

```
(declare-const a_s Real) (declare-const b_s Real) (declare-const n_s Real)
(declare-const d_s Real) (declare-const x_s Real)
(declare-const a_i Real) (declare-const b_i Real) (declare-const x_i Real)
(assert (= a_s a_i))   (assert (= b_s b_i))
(assert (not (= a_s b_s)))
(assert (= n_s (+ (- (* a_s a_s)(* 2 a_s b_s)) (* b_s b_s))))
(assert (= d_s (- a_s b_s)))   (assert (= x_s (/ n_s d_s)))
(assert (not (= a_i b_i)))     (assert (= x_i (- a_i b_i)))
(assert (not (= x_s x_i)))
(check-sat)
```

Output of Z3: unsat

Modeling bit-vectors and user-defined datatypes

Bit-vector example for Z3: DeMorgan's law

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert (not (= (bvand (bvnot x) (bvnot y)) (bvnot (bvor x y)))))
(check-sat)
```

Declaring user-defined datatype in CVC4

```
struct recordType {
  _Bool flag;
  double r;
  int i;
};
```

```
recordType: TYPE = [#
  flag:BOOLEAN,
  r:REAL,
  i:INT
#];
```

Experimental results

Table: Results for our method on different benchmarks

Benchmarks	Benchmark Characteristics							Formulae		Execution Time (ms)			
	#op	#BB	#if	#loop	#path	#state$_\alpha$	#state$_\beta$	#assert	#query	Norm	Yices2	CVC4	Z3
DCT	42	1	0	0	1	43	12	92	8	32	54	52	42
DIFFEQ	20	3	0	1	3	18	11	67	10	13	NLA	38	39
EWF	52	1	0	0	1	30	19	113	8	63	NLA	285	161
PERFECT	12	6	3	1	7	12	10	50	14	8	NLA	22	40
PRIMEFAC	10	4	2	1	5	8	7	40	10	7	NLA	16	24
BV-DEMORGAN	9	4	1	0	3	7	6	49	15	×	13	24	34
BV-BOOLRULE	9	4	2	0	5	7	7	43	11	×	36	19	26
UD-SIMPLIFY	15	1	0	0	1	8	4	29	4	×	NLA	9	6
UD-MINMAX	15	6	3	1	7	15	11	86	22	×	32	19	33

× – Normalization technique is not applicable for these cases.

NLA – Yices2 terminated prematurely due to the presence of non-linear arithmetic.

Summary and future works

Summary

- We have augmented a path based equivalence checker [ISED12] with SMT solvers.
- Experiments carried out using three SMT solvers – Yices2, CVC4, Z3 – demonstrate that the current equivalence checker is now equipped to handle bit-vectors, user-defined datatypes and sophisticated code transformations.
- The upgraded equivalence checker will automatically benefit from the current research focusing on improving (underlying) SMT solvers.
- To reduce execution time, it may be more advantageous solution to employ an SMT solver only when normalization fails to prove the equivalence.

Future works

- Automate the whole verification process; COmpiler INfraStructure [COINS] may be helpful in this regard.
- Perform extensive experimentation to test the limits of SMT solvers.
- Since different SMT solvers excel in different fields, find out the best possible combination.

Outline

1. Background
2. Translation validation of code motion transformations
3. Deriving bisimulation relations from path based equivalence checkers
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. **Other Models of Computation**
7. Conclusion and scope of future work

A major challenge: Loop transformations for arrays

Loop and arithmetic transformations are used extensively to gain speed-ups (parallelization), save memory usage, reduce power, etc.

Loop Fusion

```
for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    a[i+1][j+1] = F(in);
} }

for (i=0; i<=7; i++) {
  for (j=0; j<=7; j++) {
    b[i][j] = c[i][j];
} }
```

```
for (l1=0; l1<=3; l1++) {
  for (l2=0; l2<=3; l2++) {
    for (l3=0; l3<=1; l3++) {
      for (l4=0; l4<=1; l4++) {
        i = 2*l1 + l3;
        j = 2*l2 + l4;
        a[i+1][j+1] = F(in);
        b[i][j] = c[i][j];
} } } }
```

For array operations, **equivalence of data transformations and index spaces** have to be ensured.

Modelling equivalence of programs as a set of formulae

Two programs

```
//Program 1
for( k = 0; k < N; k++ ) {
  temp1[k] = a[k] + b[k];
  temp2[k] = a[k] - b[k];
  out[k] = temp1[k] + temp2[k];
}
```

```
//Program 2
for( k = 0; k < N; k++ ) {
  out[k] = 2 * a[k];
}
```

Equivalence problem encoded in CVC4

```
N: INT; a: INT -> INT; b: INT -> INT; temp1: INT -> INT;
temp2: INT -> INT; out_s: INT -> INT; out_i: INT -> INT;
ASSERT FORALL (k:INT): 0<=k AND k<N => temp1(k) = a(k) + b(k);
ASSERT FORALL (k:INT): 0<=k AND k<N => temp2(k) = a(k) - b(k);
ASSERT FORALL (k:INT): 0<=k AND k<N => out_s(k) = temp1(k) + temp2(k);
ASSERT FORALL (k:INT): 0<=k AND k<N => out_i(k) = 2 * a(k);
QUERY  FORALL (k:INT): 0<=k AND k<N => out_s(k) = out_i(k);
```

Output: `Valid`

Experimental Results – 3

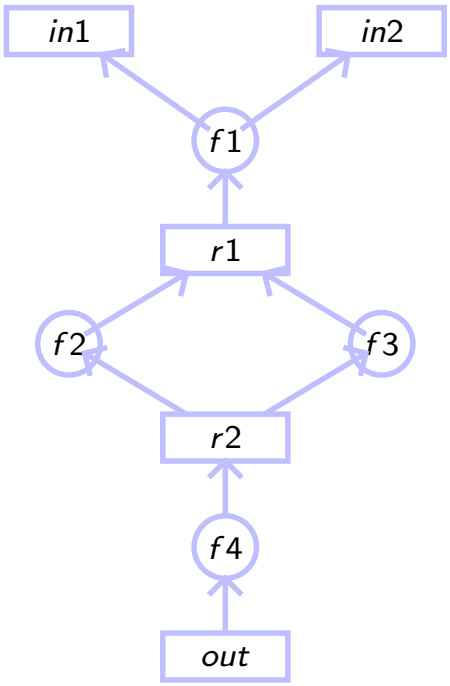
Table: Results for Yices-1.0.38, CVC4-1.1 and ACL2-6.1

	Yices1 (sec)			*CVC4 (sec)*			*ACL2 (sec)*		
Benchmark	*equiv*	*not-equiv1*	*not-equiv2*	*equiv*	*not-equiv1*	*not-equiv2*	*equiv*	*not-equiv1*	*not-equiv2*
SOB1	**0.125**	TO	U-188.151	**0.015**	U-0.036	U-0.033	**0.294**	F-0.339	F-0.384
SOB2	**0.011**	TO	U-179.435	**0.015**	U-0.025	U-0.022	**0.268**	F-0.314	F-0.335
WAVE	**0.011**	TO	TO	**0.079**	U-0.035	TO	F-0.288	F-0.228	F-0.231
LAP1	**0.012**	TO	TO	**0.012**	TO	TO	**0.231**	F-0.264	F-0.270
LAP2	TO	TO	TO	TO	TO	TO	F-0.302	F-0.255	F-0.308
LAP3	**0.014**	TO	TO	**0.224**	TO	TO	**0.222**	F-0.250	F-0.252
ACR1	TO	TO	TO	U-0.014	TO	TO	**0.250**	F-0.258	F-0.299
ACR2	TO	TO	TO	U-0.014	TO	TO	**0.282**	F-0.309	F-0.314
SOR	NLA	NLA	NLA	LE	LE	LE	F-0.304	F-0.312	F-0.295
LIN1	NLA	NLA	NLA	U-0.007	U-0.007	U-0.008	F-0.235	F-0.226	F-0.232
LIN2	NLA	NLA	NLA	U-0.009	U-0.014	U-0.007	F-0.234	F-0.229	F-0.215
LOWP	**0.013**	TO	**236.850**	**0.013**	TO	TO	**0.258**	F-0.246	F-0.359

Time Out (TO) – 5 minutes, U – Unknown, F – Failed

NLA – Non-Linear Arithmetic, LE – Logic Exception

Array Data Dependence Graphs (ADDGs)



ADDG

- Array data dependence graph (ADDG) model can capture array intensive programs [Shashidhar et al., DATE 2005]
- ADDGs have been used to verify static affine programs
- Equivalence checking of ADDGs can verify loop transformations as well as arithmetic transformations

Two equivalent array-handling programs

Loop fusion and arithmetic simplification

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```

```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```

```
for ( i = 1; i <= 100; i++ ) { out[i-1] = in[i+1]; }
```

Jargons:

Iteration domain: Domain of the index variable. $\{i \mid 1 \leq i \leq 100\}$

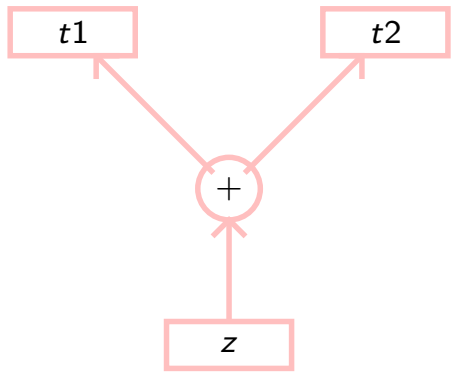
Definition domain: Domain of the (lhs) variable getting defined. $\{i \mid 0 \leq i \leq 99\}$

Operand domain: Domain of the operand variable. $\{i \mid 2 \leq i \leq 101\}$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$${}_{I}M_{z} = \{k \rightarrow k+1 \mid 0 \leq k \leq N-1\} = {}_{I}M_{t1} = {}_{I}M_{t2}$$

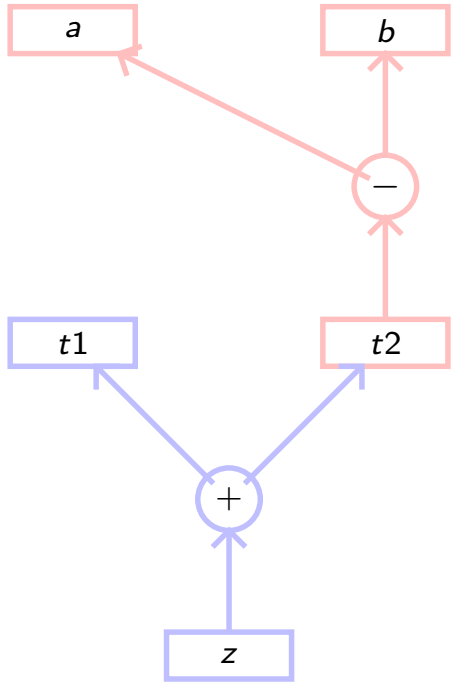
$${}_{z}M_{t1} = {}_{I}M_{z}{}^{-1} \diamond {}_{I}M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\} = {}_{z}M_{t2}$$

$$r_{\alpha} : z = t1 + t2$$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i];
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j];
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

$_{t2}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{t2}M_b$

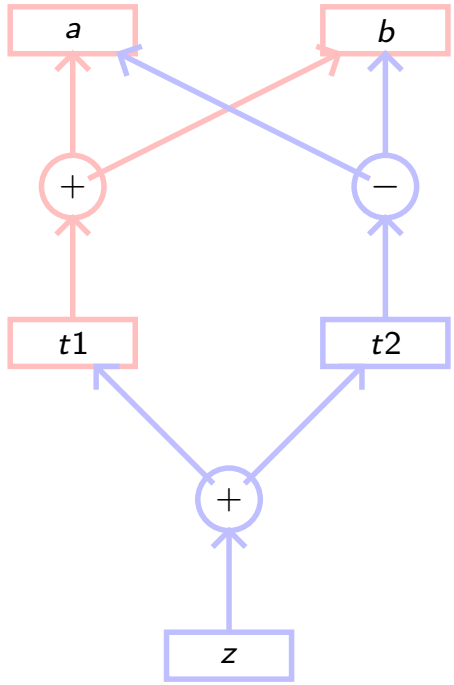
$_{z}M_{t1} = \{k \rightarrow k \mid 1 \leq k \leq N\} \quad {}_{z}M_a = \{j \rightarrow j \mid 1 \leq j \leq N\} = {}_{z}M_b$

$r_\alpha : z = t1 + (a - b)$

Construction of ADDG-1

ADDGs are constructed in reverse order, from the output array towards the input array(s).

```
for ( i = 1; i <= N; i++ ) {
  t1[i] = a[i] + b[i]
}
for ( j = N; j >= 1; j-- ) {
  t2[j] = a[j] - b[j]
}
for ( k = 0; k < N; k++ ) {
  z[k+1] = t1[k+1] + t2[k+1];
}
```



ADDG-1

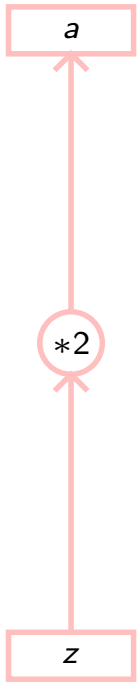
$_{t1}M_a = \{i \rightarrow i \mid 1 \leq i \leq N\} = {}_{t1}M_b$

$_{z}M_a = \{k \rightarrow k \mid 1 \leq k \leq N\} = {}_{z}M_b$

$r_\alpha : z = (a + b) + (a - b) = 2 * a$ – simplification possible since domains match

Construction of ADDG-2

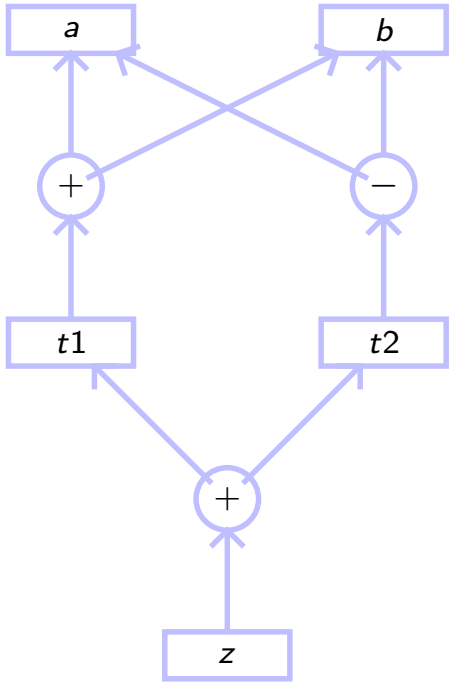
```
for ( i = 1; i <= N; i++ ) {
  z[i] = 2 * a[i];
}
```



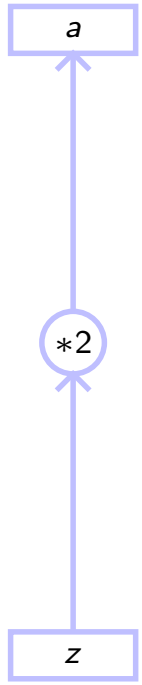
ADDG-2

$$_{I}M_{z} = \{i \rightarrow i \mid 1 \leq i \leq N\} = {}_{I}M_{a}$$
$$_{z}M_{a} = \{i \rightarrow i \mid 1 \leq i \leq N\}$$
$$r_{\beta} : z = 2 * a$$

Equivalence of ADDGs



ADDG-1



ADDG-2

Two ADDGs are said to be **equivalent** if their characteristic formulae – r_α and r_β, and corresponding mappings between the output arrays wrt input array(s) – ${}_zM_a^\alpha$ and ${}_zM_a^\beta$, match.
Hence, these two ADDGs are declared equivalent.

Experimental Results – 4

Table: Results on the earlier benchmarks

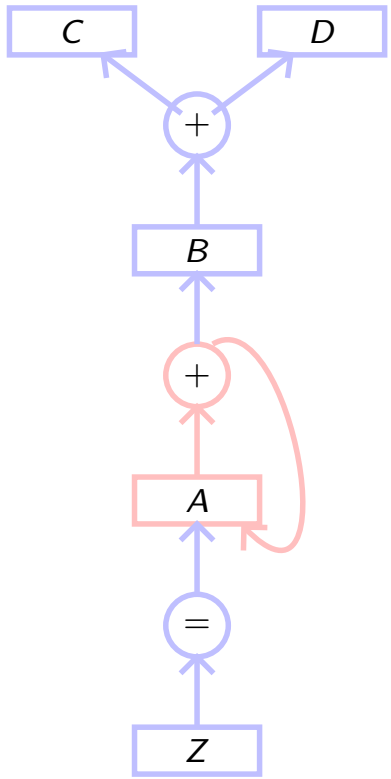
Cases	nests	C lines		loops		arrays		slices		Exec time (sec)			Exec time (sec) - ISA		
		src	trans	src	trans	src	trans	src	trans	eqv	not-eqv1	not-eqv2	eqv	not-eqv1	not-eqv2
SOB1	2	27	19	3	1	4	4	1	1	1.79	0.61	0.75	×	×	×
SOB2	2	27	27	3	3	4	4	1	1	1.85	0.90	0.62	×	×	×
WAVE	1	17	17	1	2	2	2	4	4	6.83	3.81	3.84	0.31	0.18	0.19
LAP1	2	12	21	1	3	2	4	1	1	2.79	0.57	0.65	×	×	×
LAP2	2	12	14	1	1	2	2	1	2	4.82	0.45	0.93	×	×	×
LAP3	2	12	28	1	4	2	4	1	2	9.25	1.14	4.84	0.28	0.19	0.25
ACR1	1	14	20	1	3	6	6	1	1	0.76	0.51	0.72	0.18	0.12	0.13
ACR2	1	24	14	4	1	6	6	2	1	0.98	0.46	0.39	×	×	×
SOR	2	26	22	8	6	11	11	1	1	1.08	0.61	0.62	0.18	0.20	0.17
LIN1	2	13	13	3	3	4	4	2	2	0.62	0.28	0.26	0.12	0.11	0.13
LIN2	2	13	16	3	4	4	4	2	3	0.74	0.20	0.33	0.13	0.12	0.13
LOWP	2	13	28	2	8	2	4	1	2	9.17	0.65	2.90	×	×	×

• Verdoolaege et al., "Equivalence checking of static affine programs using widening to handle recurrences," TOPLAS 2012.

★ ISVLSI 2011, I-CARE 2013 (Best Paper Award), TCAD 2013.

Handling recurrences

```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```

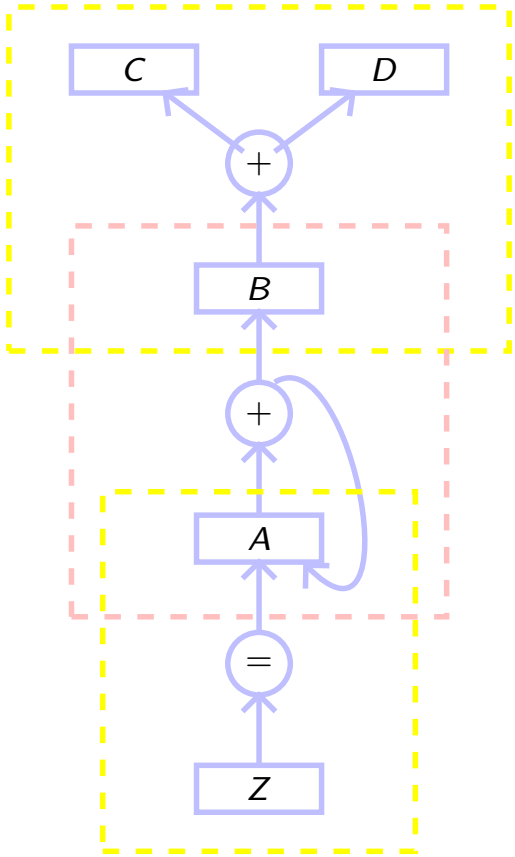


ADDG

Presence of recurrences leads to cycles in the ADDG and hence a closed form representation of r_α cannot be obtained.

Remedy – Separate DAGs from cycles

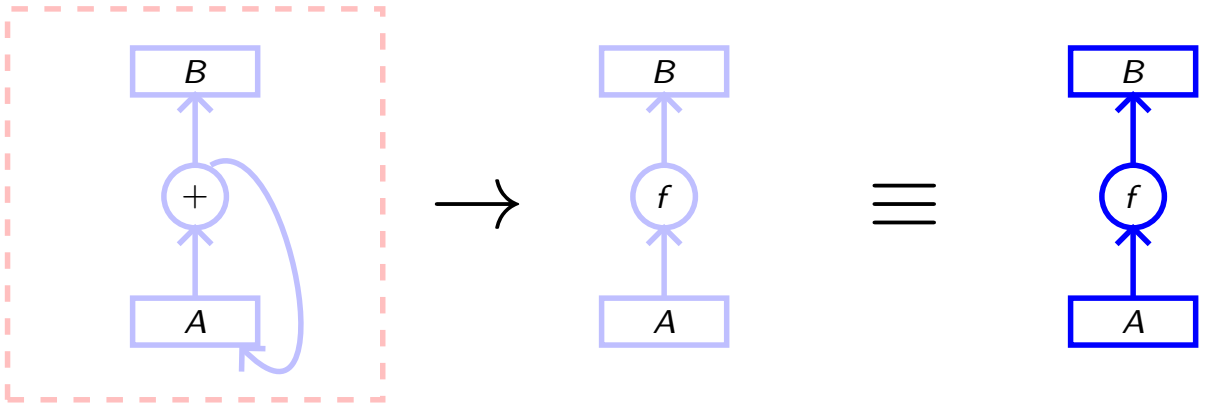
```
for ( i = 1; i < N; i++ ) {
  B[i] = C[i] + D[i];
}
for ( i = 1; i < N; i++ ) {
  A[i] = A[i-1] + B[i];
}
for ( i = 1; i < N; i++ ) {
  Z[i] = A[i];
}
```



ADDG

Try to establish equivalence of the *separated* ADDG portions.

Representing recurrences as uninterpreted functions



ADDG-1 ADDG-1 ADDG-2

- A gets functionally transformed in terms of B
- Determining the *exact* values of the elements of A (wrt B) is as hard as synthesizing invariants
- Instead we introduce an uninterpreted function and hope that we shall get an equivalent function in the other ADDG as well

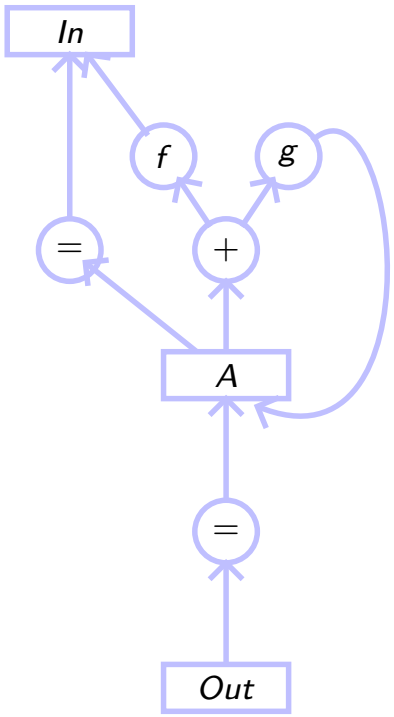
An illustrative example

A pair of programs involving recurrences

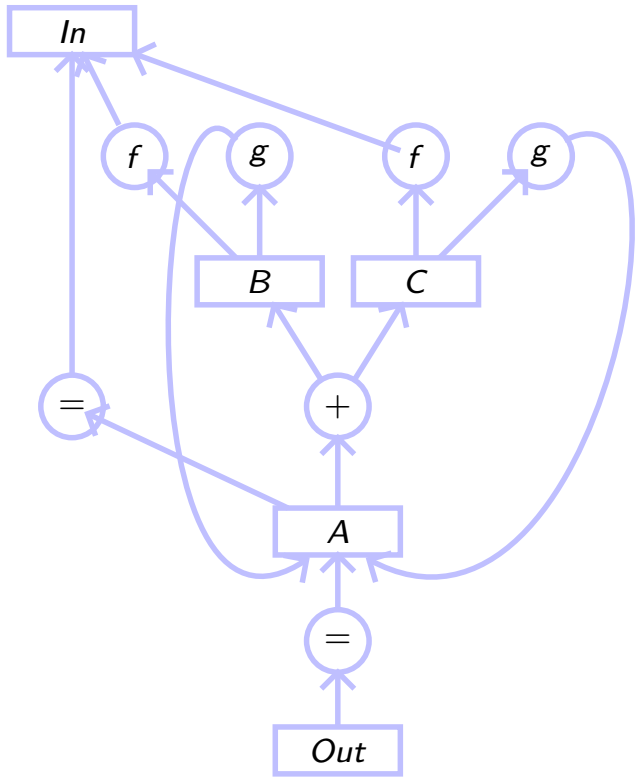
```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
  S2: A[i] = f(In[i]) + g(A[i-1]);
}
S3: Out = A[N-1];
```

```
S1: A[0] = In[0];
for (i = 1; i < N; ++i) {
  if ( i%2 == 0 ) {
    S2: B[i] = f(In[i]);
    S3: C[i] = g(A[i-1]);
  } else {
    S4: B[i] = g(A[i-1]);
    S5: C[i] = f(In[i]);
  }
  S6: A[i] = B[i] + C[i];
}
S7: Out = A[N-1];
```

ADDGs with cycles

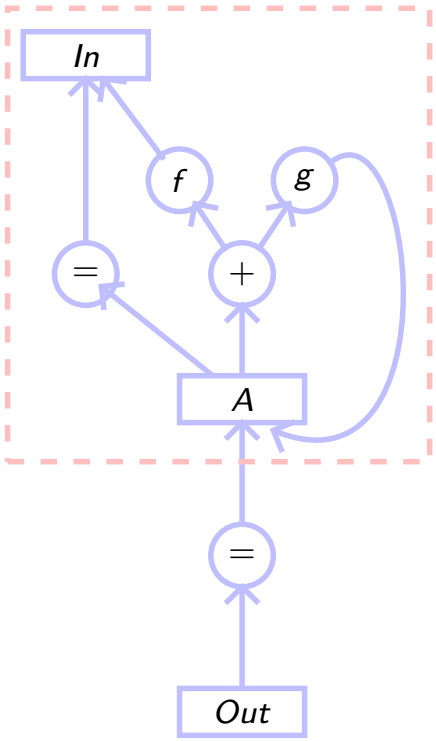


ADDG-1

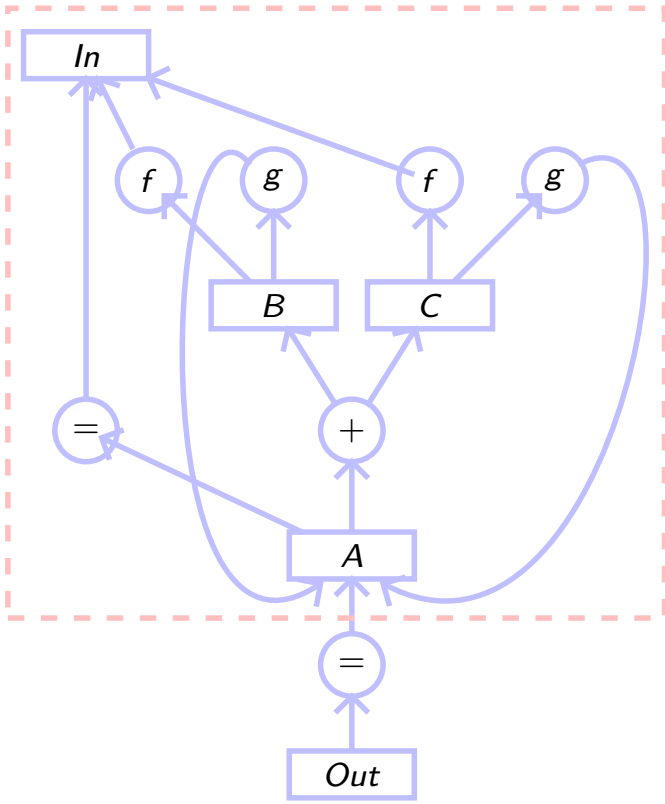


ADDG-2

ADDGs with cycles

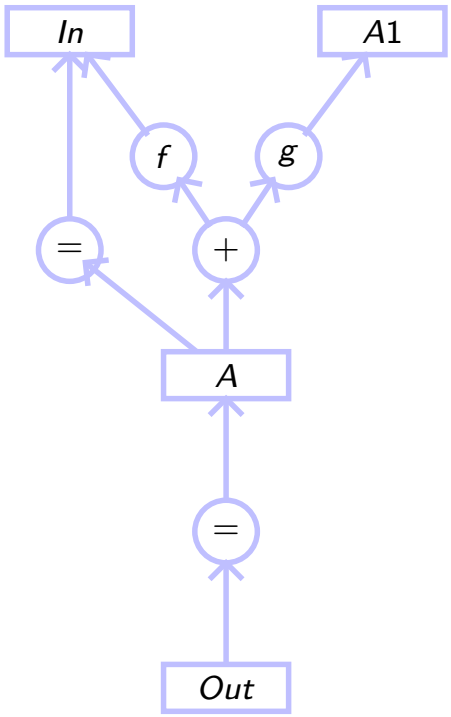


ADDG-1

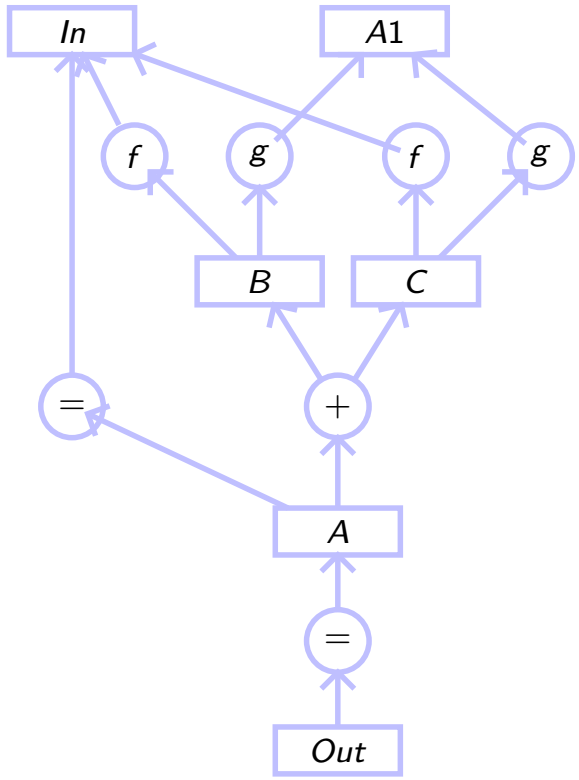


ADDG-2

ADDGs without cycles

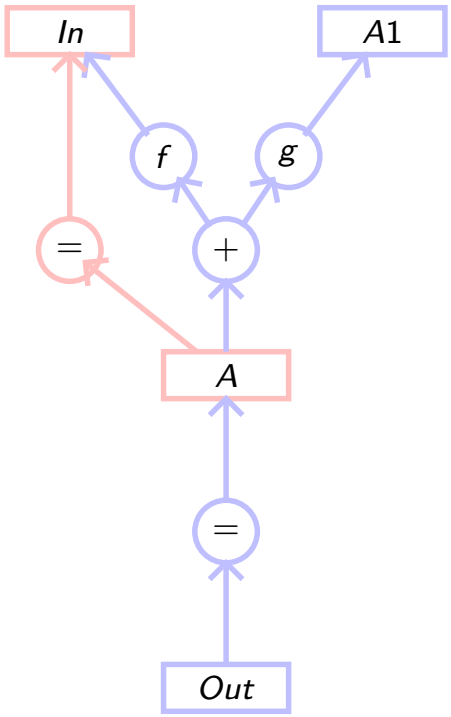


ADDG-1



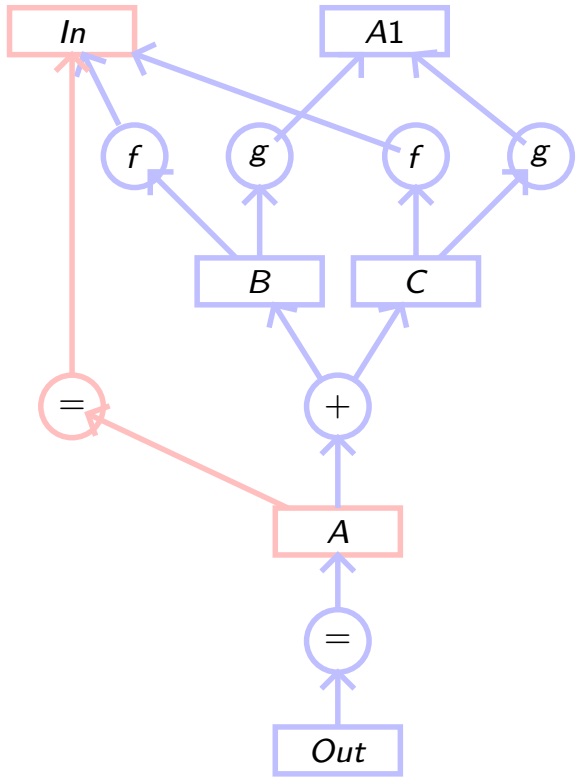
ADDG-2

ADDGs without cycles



ADDG-1

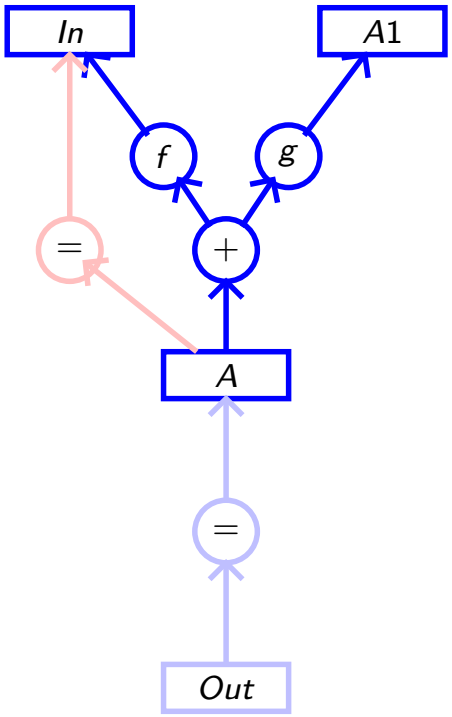
```
S1:  A[0] = In[0];
```



ADDG-2

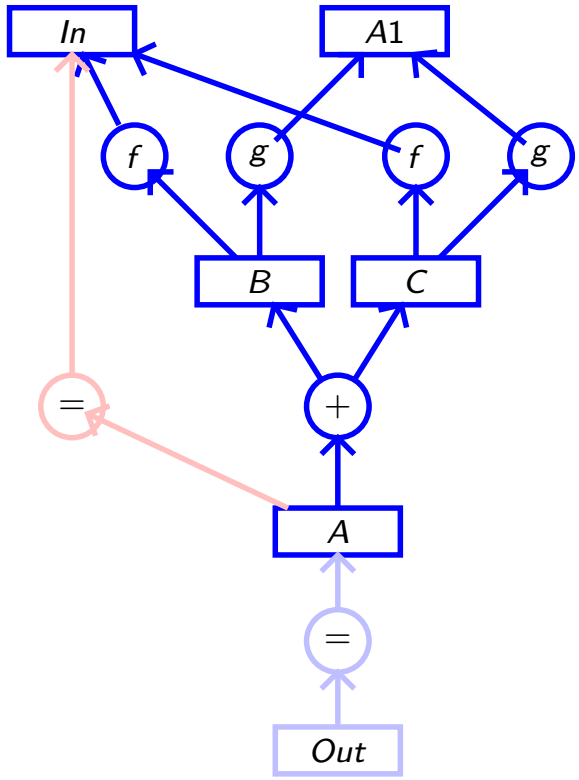
```
S1:  A[0] = In[0];
```

ADDGs without cycles



ADDG-1

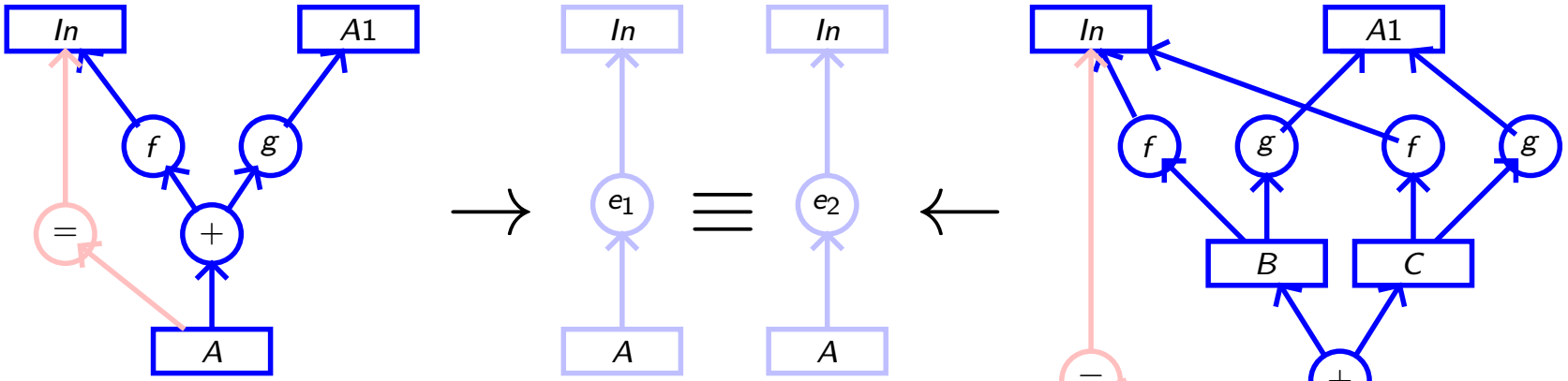
```
for (i = 1; i < N; ++i)
  A[i] = f(In[i]) + g(A[i-1]);
```



ADDG-2

```
for (i = 1; i < N; ++i) {
  if ( i%2 == 0 ) { B[i] = f(In[i]); C[i] = g(A[i-1]); }
  else { B[i] = g(A[i-1]); C[i] = f(In[i]); }
  A[i] = B[i] + C[i]; }
```

ADDGs without cycles

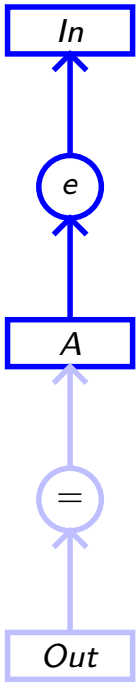


We have to show $e_1(In[i]) = e_2(In[i])$ for all $i, 0 \leq i < N$

Basis case proves $e_1(In[0]) = e_2(In[0])$

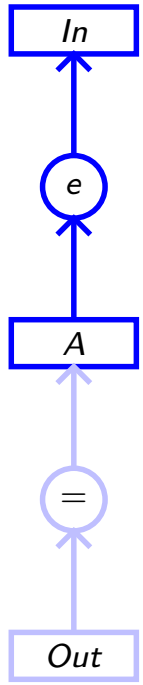
Induction hypothesis: $e_1(In[i]) = e_2(In[i]), 0 \leq i < m$, RTP: $e_1(In[m]) = e_2(In[m])$

ADDGs without cycles



ADDG-1

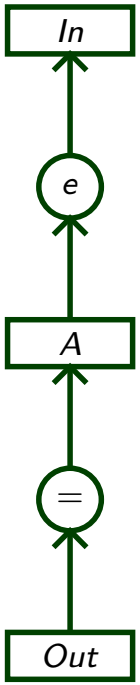
```
A = e(In)
```



ADDG-2

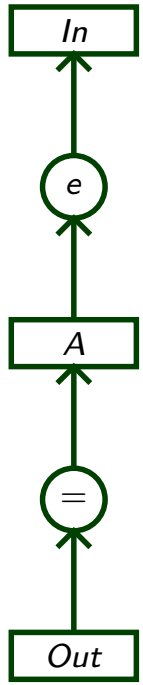
```
A = e(In)
```

ADDGs without cycles



ADDG-1

```
S3:  Out = A[N-1];
```



ADDG-2

```
S7:  Out = A[N-1];
```

Experimental Results – 5

Sl No	Benchmark	C lines src	C lines trans	loops src	loops trans	arrays src	arrays trans	slices src	slices trans	Exec time (sec) [ISA]	Exec time (sec) [Prev]	Exec time (sec) [Our]
1	ACR1	14	20	1	3	6	6	1	1	0.18	0.76	0.28
2	LAP1	12	28	1	4	2	4	1	2	0.28	9.25	0.55
3	LIN1	13	13	3	3	4	4	2	2	0.12	0.62	0.24
4	LIN2	13	16	3	4	4	4	2	3	0.13	0.74	0.30
5	SOR	26	22	8	6	11	11	1	1	0.18	1.08	0.68
6	WAVE	17	17	1	2	2	2	4	4	0.31	6.83	0.53
7	ACR2	24	14	4	1	6	6	2	1	×	0.98	0.36
8	LAP2	12	21	1	3	2	4	1	1	×	2.79	0.35
9	LAP3	12	14	1	1	2	2	1	2	×	4.82	0.56
10	LOWP	13	28	2	8	2	4	1	2	×	9.17	0.63
11	SOB1	27	19	3	1	4	4	1	1	×	1.79	0.61
12	SOB2	27	27	3	3	4	4	1	1	×	1.85	0.57
13	EXM1	8	14	1	1	2	4	1	3	0.14	×	0.36
14	EXM2	8	15	1	1	3	5	1	3	0.19	×	0.48
15	SUM1	8	14	1	1	2	4	1	3	×	×	0.40
16	SUM2	16	19	4	4	4	6	2	4	×	×	0.72

★ K Banerjee, "An Equivalence Checking Mechanism for Handling Recurrences in Array-Intensive Programs," POPL-SRC 2015.

★ K Banerjee et al., "Translation Validation of Loop and Arithmetic Transformations in the Presence of Recurrences," LCTES 2016.

Data race detection for array-intensive programs

Example

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Sequential vs Parallel

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Loop Transformation: Loop Fission

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Arithmetic Transformation: + is commutative, * distributes over +

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Introduction/elimination of temporary variables

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Program characteristic: Presence of recurrences

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Data race detection for array-intensive programs

Program characteristic: Absence of single assignment (SA) form

```
void function1(
  int A[], int Z[]) {
  int i, B1[100], B2[100];
  S0: Z[0] = A[0];
  for (int i=1; i<N; i++) {
    S1: B1[i] = A[i];
    S2: B2[i] = A[N-i-1];
    S3: Z[i] = Z[i-1] +
      B1[i]*B2[i] + B1[i];
  }
}
```

```
void function2(int A[], int Z[])
{
  int i, t, B[100];
  T0: Z[0] = A[0];
  #pragma omp parallel {
  #pragma omp for nowait
  for (i=0; i<N; i++) {
    T1: B[i] = A[i];
  }
  #pragma omp for
  for (i=0; i<N; i++) {
    T2: t = N-i-1;
    T3: Z[i] = B[t]+1;
    T4: Z[i] = Z[i]*B[i]+Z[i-1];
} } }
```

Summary of challenges

Challenges

- Loop transformations
- Arithmetic transformations
- Synchronization transformations
- Ability to handle recurrences
- Tackle non-SA form

Summary of challenges

Challenges

- Loop transformations
- Arithmetic transformations
- Synchronization transformations
- Ability to handle recurrences
- Tackle non-SA form

Types of data hazards

Write After Read (WAR)

```
S1: R4 = R1 + R5
S2: R5 = R1 + R2
```

Write After Write (WAW)

```
S1: R2 = R4 + R7
S2: R2 = R1 + R3
```

Read After Write (RAW)

```
S1: R2 = R1 + R3
S2: R4 = R2 + R3
```

Significance of single assignment form

Write After Read (WAR)

```
S1: R4 = R1 + R5
S2: R5 = R1 + R2
```

Write After Write (WAW)

```
S1: R2 = R4 + R7
S2: R2 = R1 + R3
```

Read After Write (RAW)

```
S1: R2 = R1 + R3
S2: R4 = R2 + R3
```

The C-ADDG Model

C-ADDG

A C-ADDG is an ADDG with the following additional characteristics:

- The loop labeling function $\mathcal{L} : \mathcal{F} \rightarrow \mathbb{Z}$ associates an integer number to each operator node.
- The function $\mathcal{H} : \mathcal{F} \rightarrow \mathscr{C}$ marks each operator node in the C-ADDG a colour from the set $\mathscr{C}$ of infinite colours.
- The loop labeling function $\mathcal{L}(f)$ denotes the innermost loop body in which the statement corresponding to f belongs. $\mathcal{L}(f) = 0$ indicates that f does not belong to any loop.
- A barrier statement is associated with a barrier node $b \in \mathcal{F}$. For a barrier node b, its data transformation is considered to be identity. At a barrier, for each array node corresponding to an array variable defined within that parallel construct, a duplicate array node is created with an intermediary barrier operator node b.

The C-ADDG Model (contd.)

Colour Mapping

The colour set $\mathscr{C}$ of infinite colours has two special colours – black and blue – that imposes the following constraints on the mapping function $\mathcal{H}$:

- the operator nodes (f) corresponding to statements which are executed by a single thread are marked with *black colour*,
- operator nodes corresponding to all statements in a parallel for construct are marked with a *distinct colour*,
- a barrier node (b) is marked with *blue colour*.

Data-race conditions

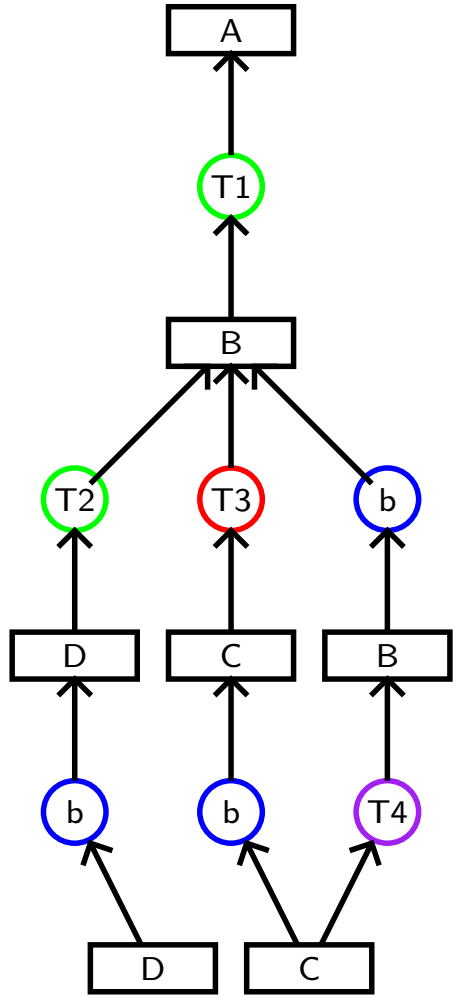
Consider two statements S and T such that T reads an element $\bar{i}$ of array A, say, which has been written into by statement S.
Then, potential RAW race between S and T occurs if any of the folowing two conditions is satisfied:

R1 S and T are in the **same** parallel loop f (colors are not black) and for two different iteration vectors $\overline{i_1}, \overline{i_2}$ in f, S and T reads and writes into the same array variable

R2 Colours of S and T are **not same** and there is no intervening barrier node

An example of C-ADDG

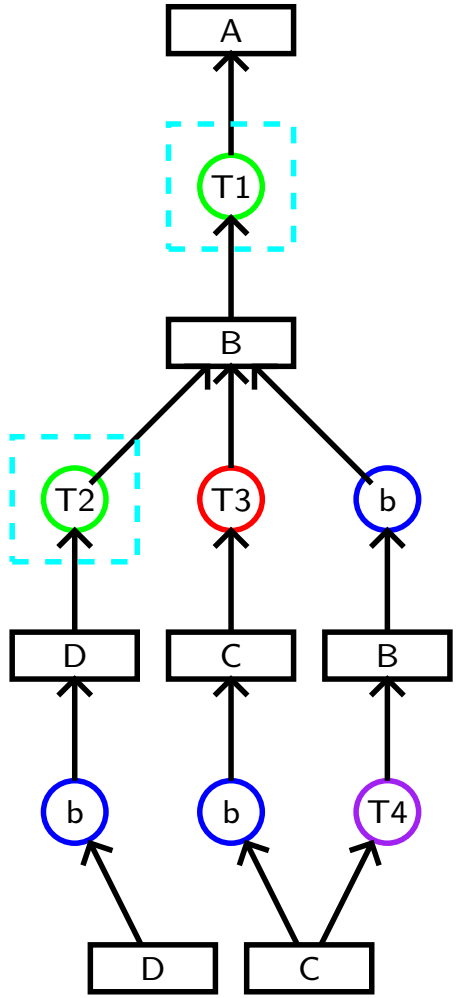
```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 1; i < 100; i++) {
    T1: B[i] = A[i] * A[i];
    T2: D[i] = g(B[i], B[i-1]);
  }
  #pragma omp for
  for (i = 1; i < 50; i++)
    T3: C[i] = B[i]*2;
} //barrier b
#pragma omp parallel for
for (i = 50; i < 100; i++)
  T4: C[i] = B[i]*2;
```



C-ADDG

An example of C-ADDG: R1 race condition

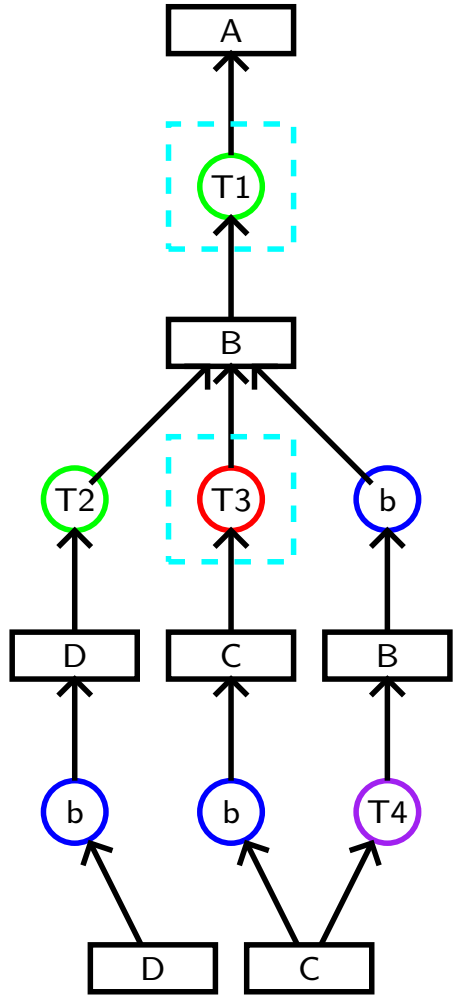
```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 1; i < 100; i++) {
    T1: B[i] = A[i] * A[i];
    T2: D[i] = g(B[i], B[i-1]);
  }
  #pragma omp for
  for (i = 1; i < 50; i++)
    T3: C[i] = B[i]*2;
} //barrier b
#pragma omp parallel for
for (i = 50; i < 100; i++)
  T4: C[i] = B[i]*2;
```



C-ADDG

An example of C-ADDG: R2 race condition

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 1; i < 100; i++) {
    T1: B[i] = A[i] * A[i];
    T2: D[i] = g(B[i], B[i-1]);
  }
  #pragma omp for
  for (i = 1; i < 50; i++)
    T3: C[i] = B[i]*2;
} //barrier b
#pragma omp parallel for
for (i = 50; i < 100; i++)
  T4: C[i] = B[i]*2;
```



C-ADDG

Experimental Results

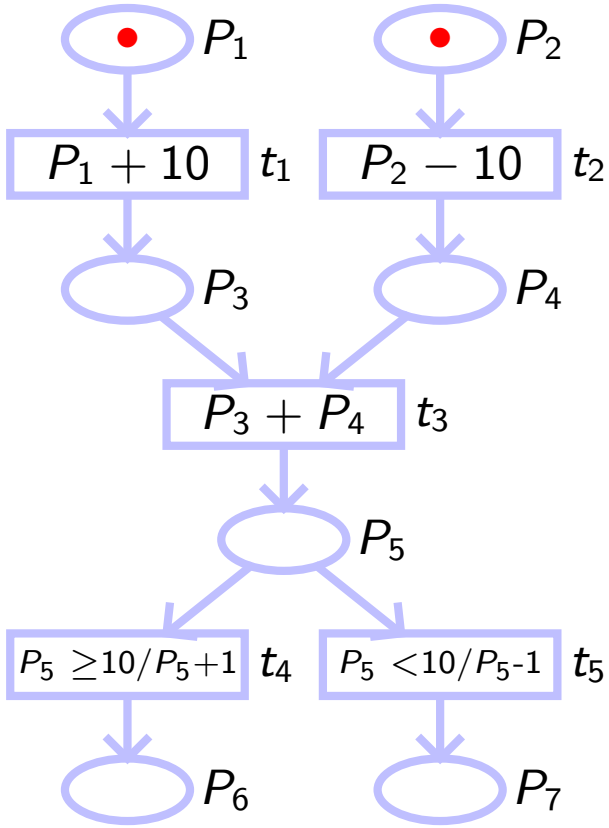
- Tested on Rodinia and PLuTo benchmarks
  - PLuTo was used to translate sequential programs into parallel, and perform loop transformations
  - Arithmetic transformations were manually introduced
- Compared our tool with [Verdoolaege et al., TOPLAS 2012]
  - Four benchmarks were erroneously reported to be non-equivalent by the tool of Verdoolaege et al., whereas ours was successful in establishing equivalence
    - Our tool can handle arithmetic transformations
  - Our tool is slower in execution
    - We have an extra overhead of calling a separate process (ISL library)
    - Our tool DOES consider parallel constructs and checks synchronization constraint

PRES$^+$: A parallel model of computation

- Petri net based Representation of Embedded Systems
- Extension of classical Petri net to support concurrency and timing behaviour
- Tokens carry information on functional transformations
- PRES$^+$ model is a seven tuple $R = \langle P, K, T, I, O, inP, outP \rangle$ where,

P: Set of places,
K: Set of all Possible token types,
T: Set of transitions
I: Set of input places, $I \subseteq P \times T$
O: Set of output places, $O \subseteq T \times P$
inP: Set of in-ports
$outP$: Set of out-ports

An example of PRES$^+$ model

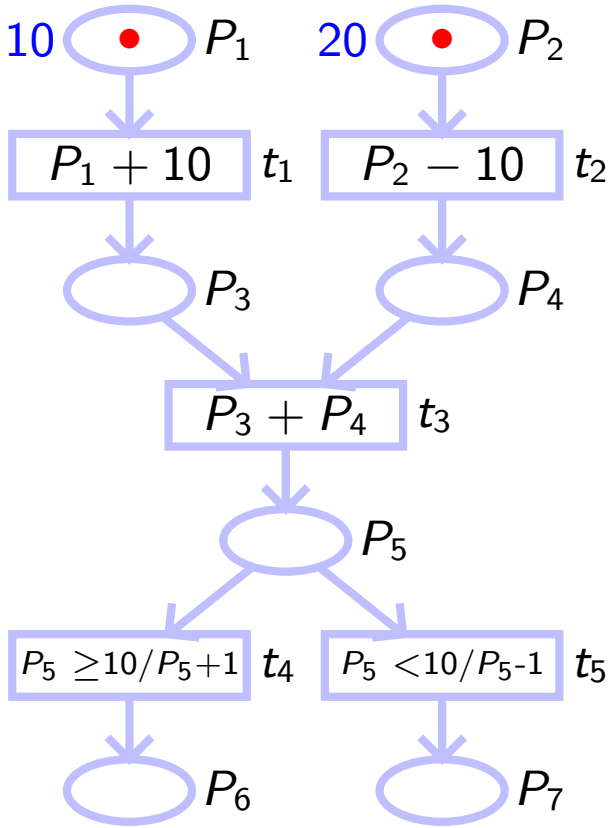


Set of places $= \{P_i, 1 \leq i \leq 7\}$, Set of transitions $= \{t_i, 1 \leq i \leq 5\}$

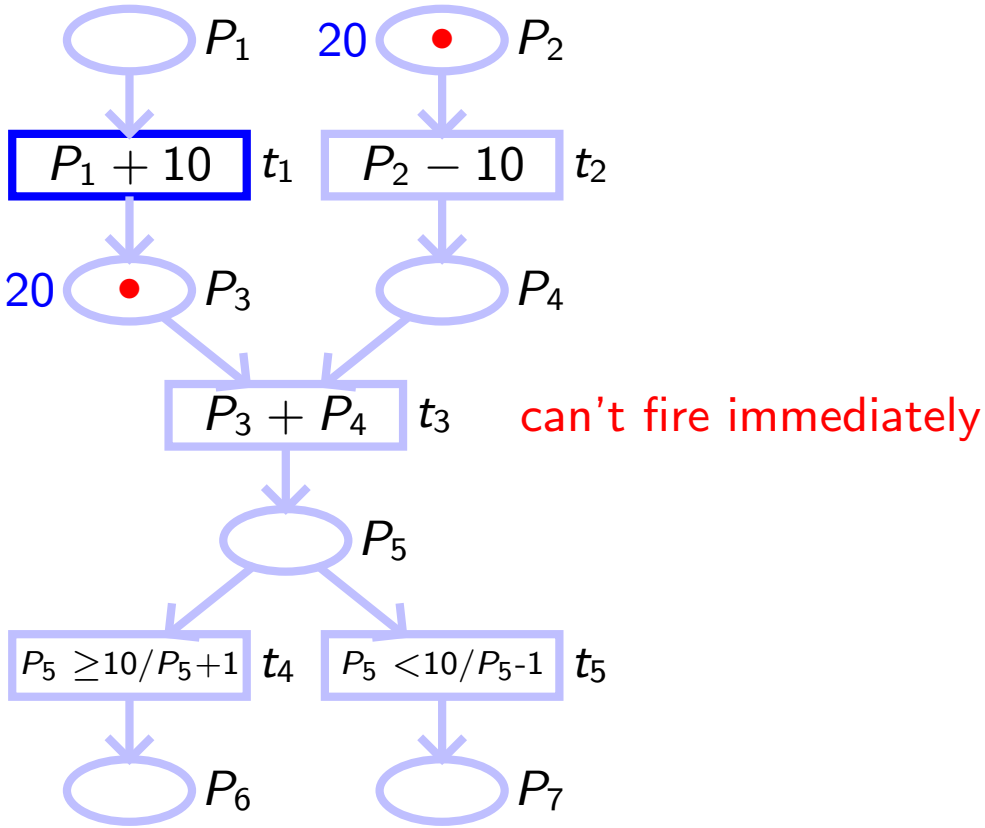
It's a *one-safe* model – only one token allowed at a place at a time

The guard conditions of the transitions are mutually exclusive – ensures deterministic computation

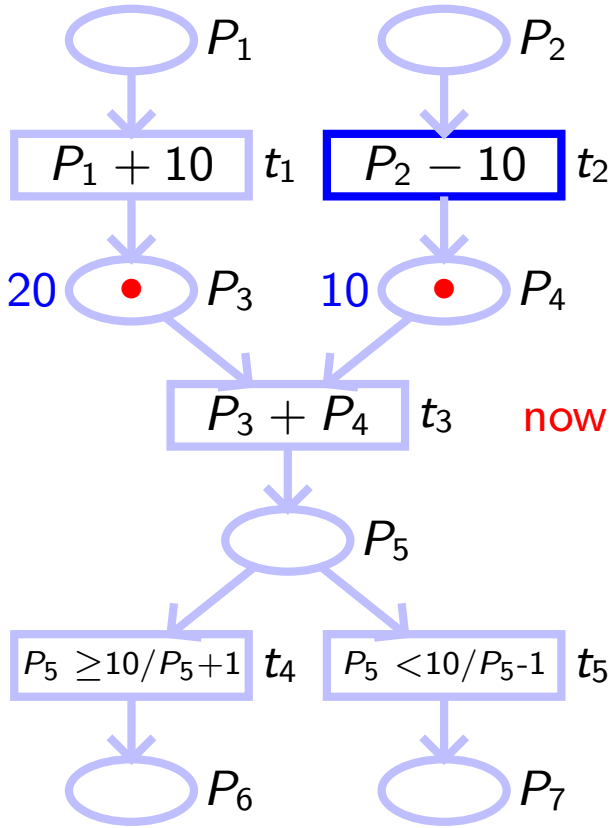
An execution instance: $P_1 = 10, P_2 = 20$



An execution instance: $P_1 = 10, P_2 = 20$

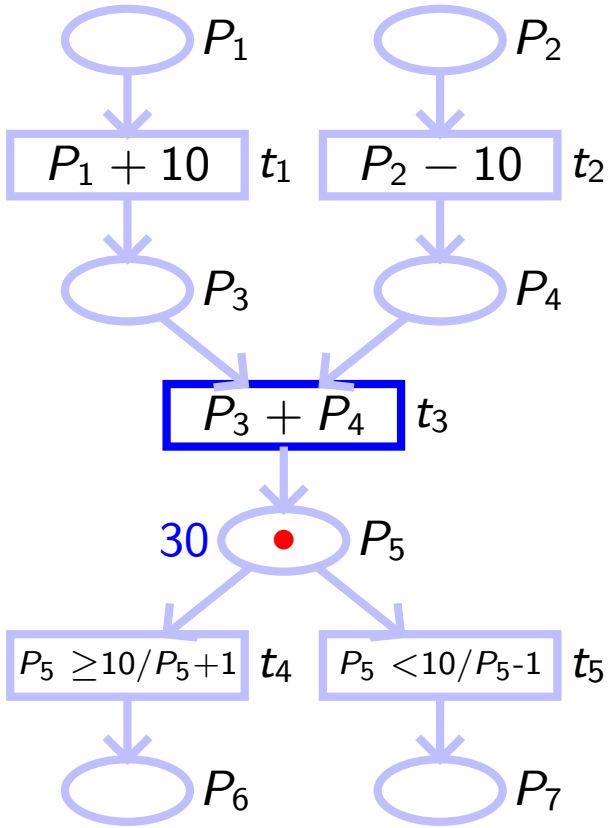


An execution instance: $P_1 = 10, P_2 = 20$



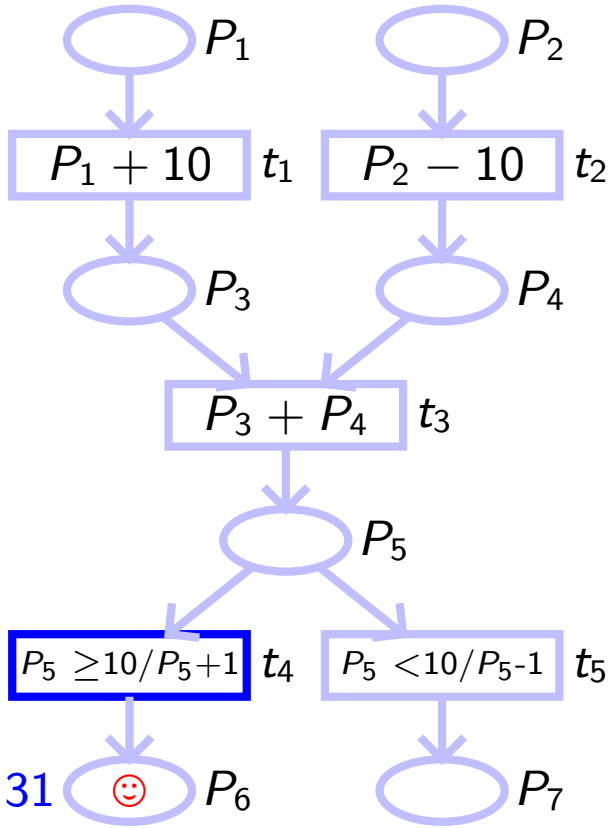
now it can fire – synchronization

An execution instance: $P_1 = 10, P_2 = 20$

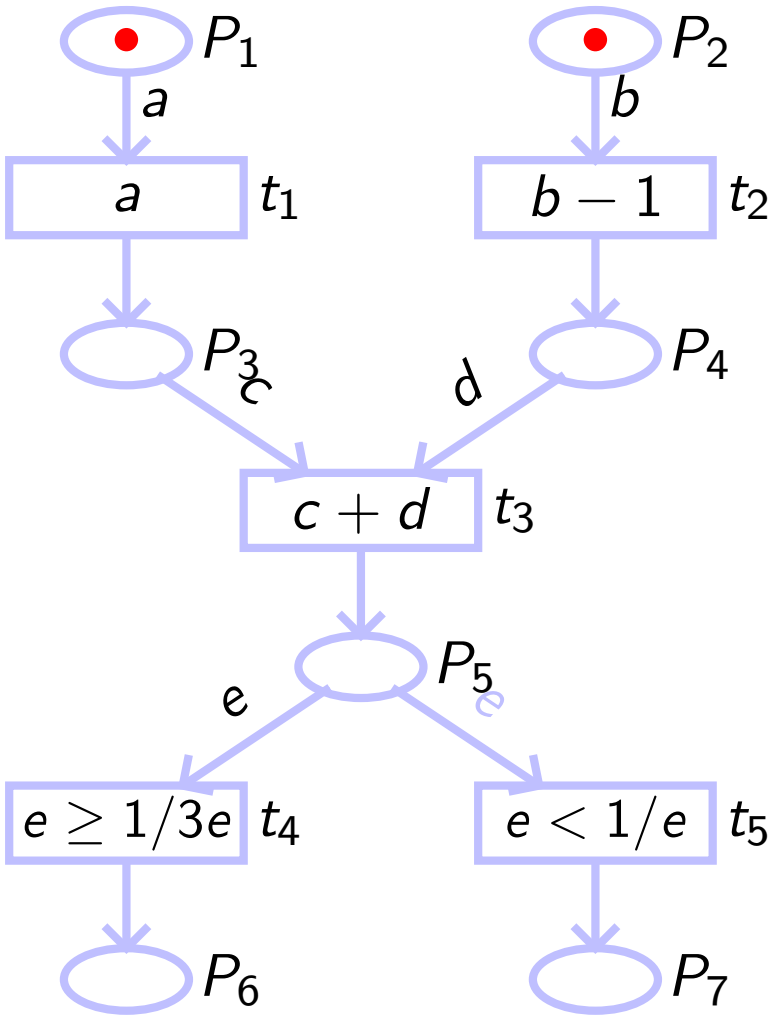


only one can fire – determinism

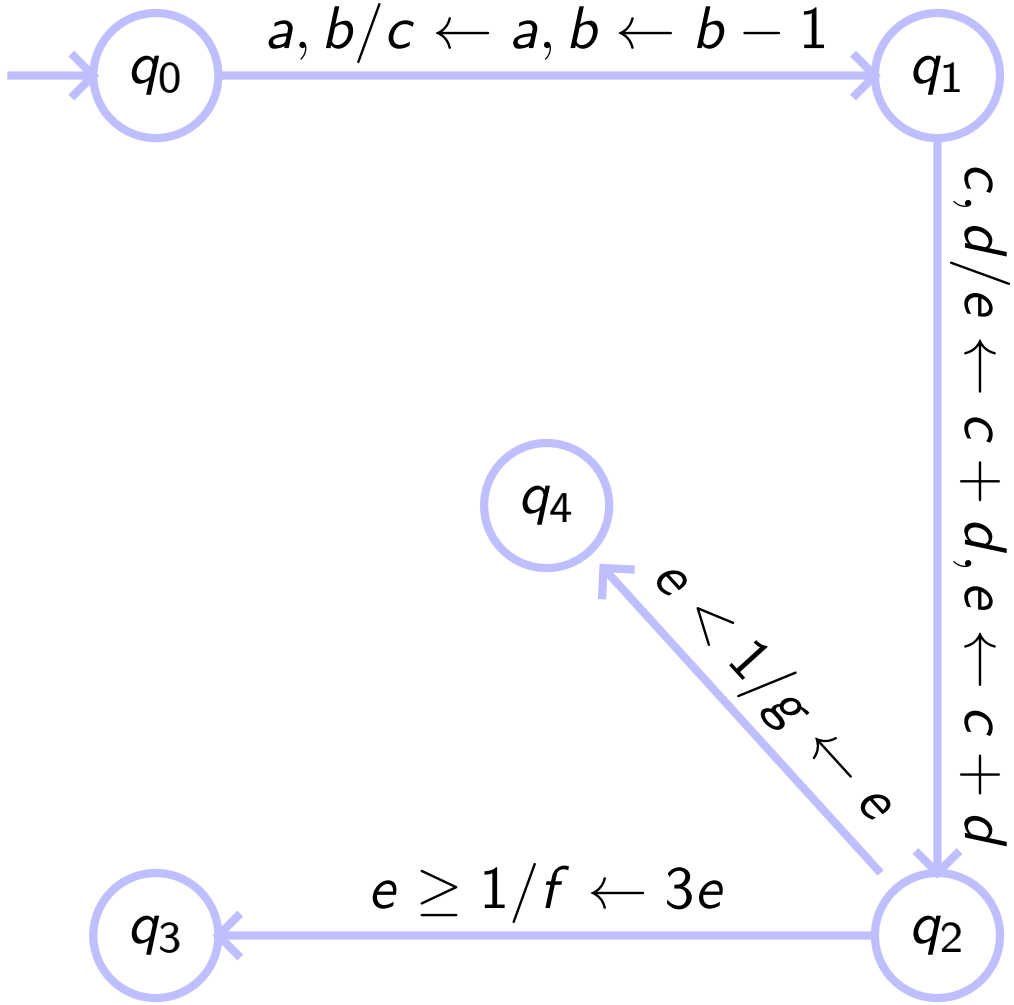
An execution instance: $P_1 = 10, P_2 = 20$



PRES+ translation to FSMD

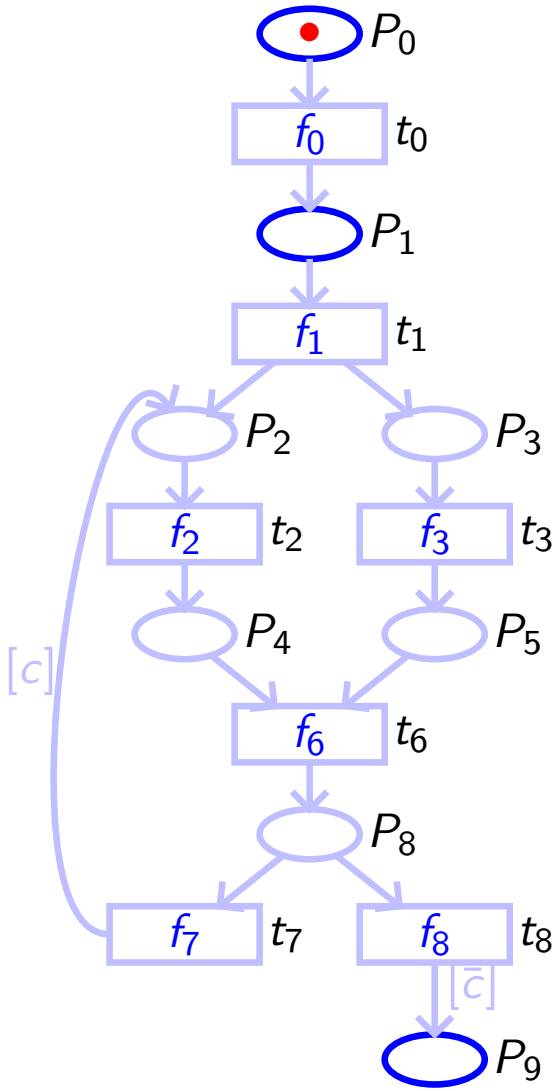


A PRES+model



The corresponding FSMD

Capturing computations using paths for direct equivalence checking

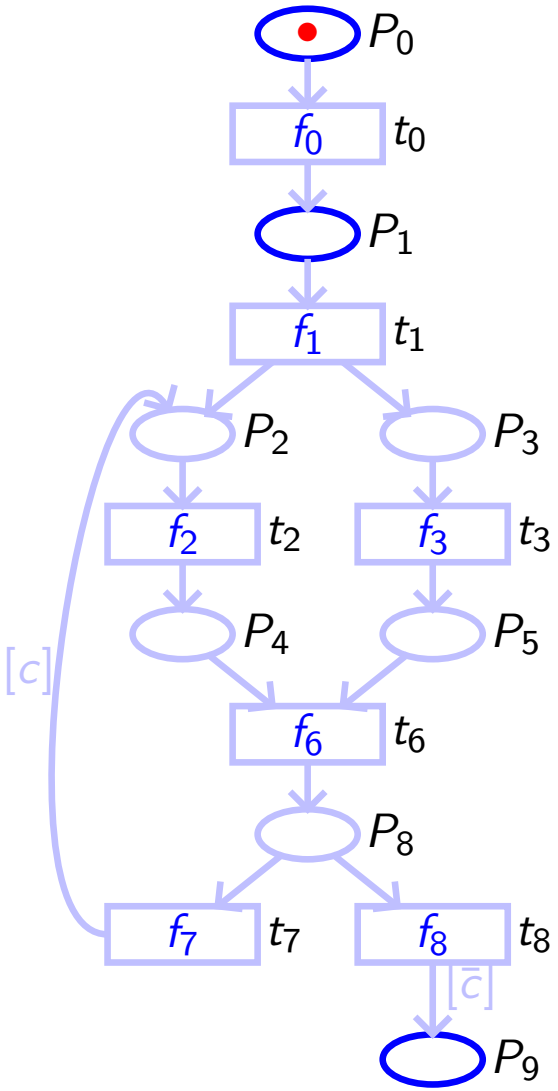


Cutpoints: To reduce computations into concatenation of paths, we need to cut the loops. This is accomplished by introducing cutpoints.

Rules:

1. An in-port is a cutpoint.
2. A place that is contained in the post-set list of more than one transition is a cutpoint.
3. An out-port is a cutpoint.

Capturing computations using paths for direct equivalence checking



Cutpoints: To reduce computations into concatenation of paths, we need to cut the loops. This is accomplished by introducing cutpoints.

Rules:

1. An in-port is a cutpoint.
2. A place that is contained in the post-set list of more than one transition is a cutpoint.
3. An out-port is a cutpoint.

Direct equivalence checking of two PRES$^+$ models

Verification method for checking equivalence of PRES$^+$ models consists of the following steps:

1. Introduce the cutpoints using the rules given in the previous slide.
2. Construct the initial path cover Π_0 of N_0 comprising paths from a cutpoint to another cutpoint without having any intermediate cutpoint. Let $\Pi_0 = \{\alpha_{0,0}, \alpha_{0,1}, \cdots, \alpha_{0,k}\}$.
3. Show that $\forall \alpha_{0,i} \in \Pi_0$, there exists a path $\alpha_{1,j}$ of N_1 such that $\alpha_{0,i} \simeq \alpha_{1,j}$; i.e. their data transformations and condition of execution match.
4. Repeat steps 2 and 3 with N_0 and N_1 interchanged.

Outline

1. Background
2. Translation validation of code motion transformations
3. Deriving bisimulation relations from path based equivalence checkers
4. Translation validation of code motion transformations in array-intensive programs
5. Augmenting equivalence checkers with SMT solvers
   - SMT solvers
6. Other Models of Computation
7. **Conclusion and scope of future work**

Summary

Objective: **Translation Validation** – since compilers cannot always be trusted

Type of transformation	*Model*
Code motions	FSMD (path extension)
Code motions across loops	**FSMD (value propagation)**
Code motions for arrays	**FSMDA**
Loop + Arithmetic transformations	ADDG (without cycles)
Loop + Arithmetic + Recurrence	**ADDG (with cycles)**

Associated bisimulation relation and path based equivalence checking methods

Reductions to resolve recurrences

Program that computes $\sum_{i=1}^{N} A[i]$

```
sum[0] = A[0];
for( i = 1; i <= N; ++i ) {
  sum[i] = sum[i-1] + A[i];
}
out = sum[N];
```

```
sum[N] = A[N];
for( i = N-1; i >= 0; --i ) {
  sum[i] = sum[i+1] + A[i];
}
out = sum[0];
```

Reductions involve accumulations of parametric number of sub-expressions using an associative and commutative operator.

- Iooss et al., "On Program Equivalence with Reductions," SAS 2014.

Limitation: Reasoning over a finite domain

What's the output?

```
if ( x+1 >= x )
  printf(``Hello'');
else
  printf(``World'');
```

What happens if `x` is the maximum representable integer?

- Output is `World` if modular arithmetic is followed
- Output is `Hello` if saturation arithmetic is followed
- C does not have a defined semantics for overflows, definitions of some other behaviours differ across different standards (ANSIC, C99) [ICSE12]

Limitation: Reasoning over a finite domain

What's the output?

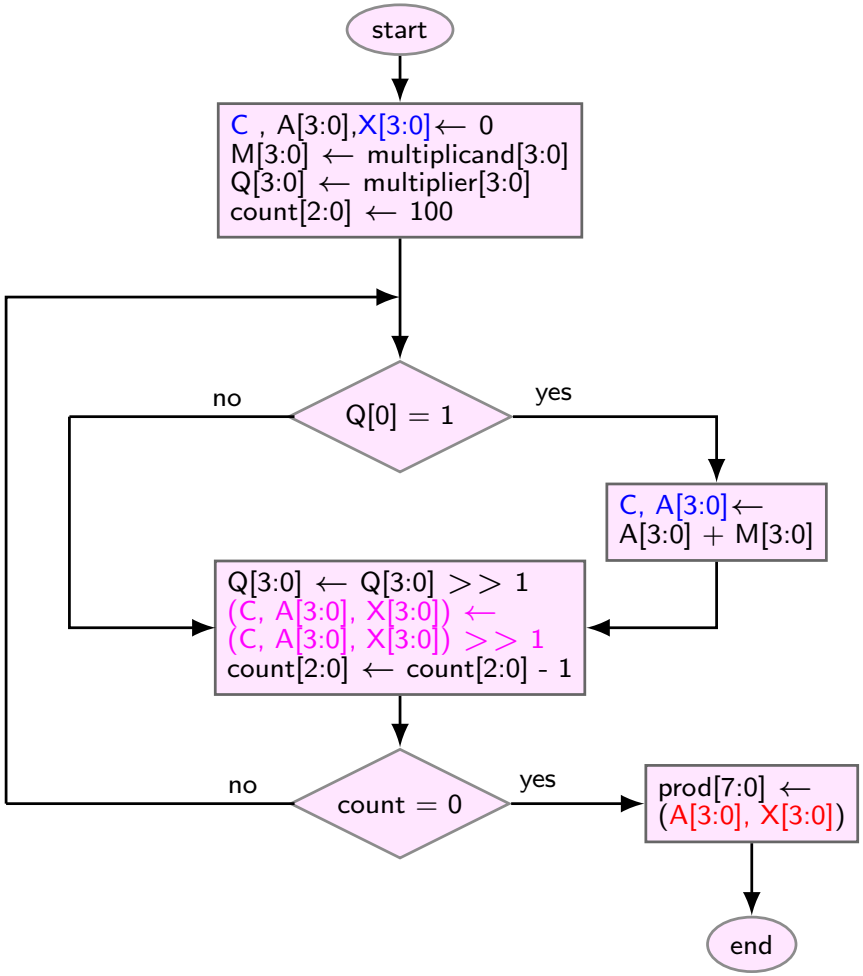
```
if ( x+1 >= x )
  printf(``Hello'');
else
  printf(``World'');
```

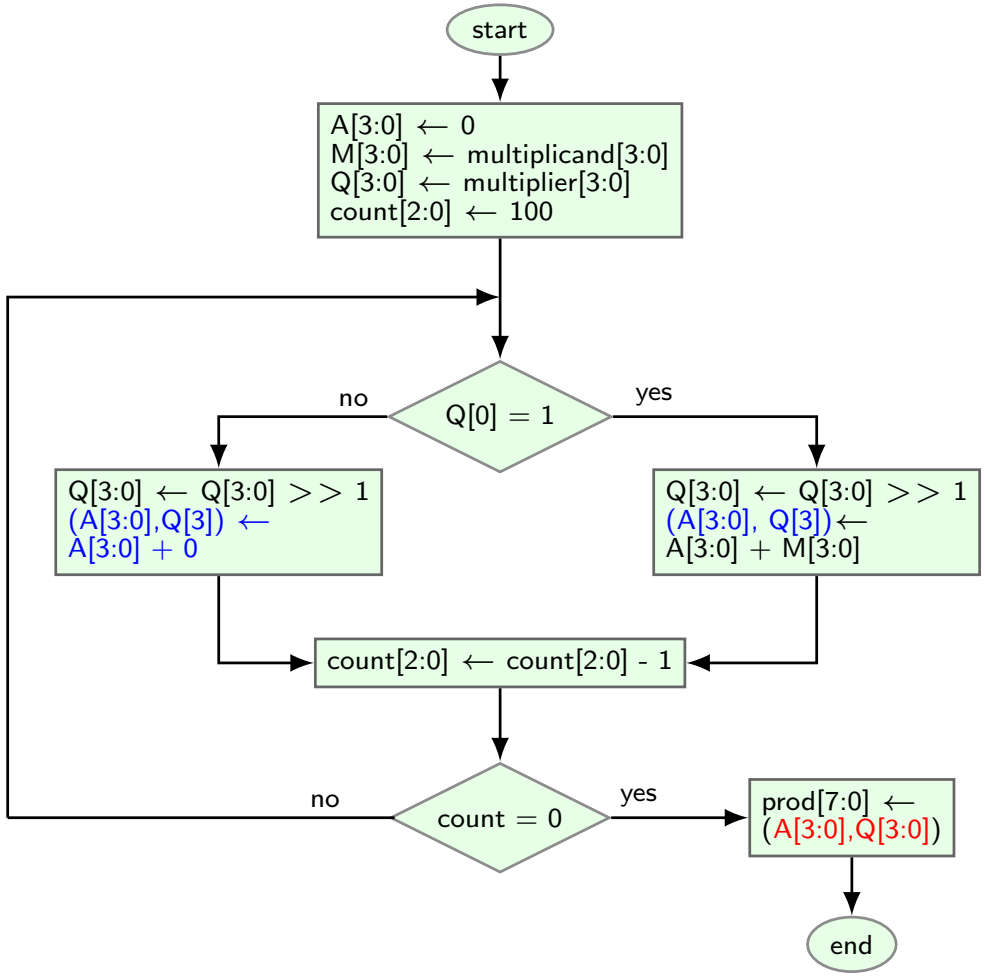
What happens if `x` is the maximum representable integer?

- Output is `World` if modular arithmetic is followed
- Output is `Hello` if saturation arithmetic is followed
- C does not have a defined semantics for overflows, definitions of some other behaviours differ across different standards (ANSIC, C99) [ICSE12]

Bit-level equivalence checking

Mismatched variables and operations, different association, mismatched shifting





Thank you!

✉ kunalb@cse.iitkgp.ernet.in ✉ ckarfa@iitg.ernet.in