Translation Validation using Path Based Equivalence Checkers Augmented with SMT Solvers

Kunal Banerjee

Dept of Computer Sc & Engg
IIT Kharagpur

Outline

1. **Background**
   - Translation validation
   - Path based equivalence checkers
   - SMT solvers
2. **Normalization technique**
3. **Deploying SMT solvers**
4. **Experimental results**
5. **Conclusion and future works**

Outline

1. **Background**
   - Translation validation
   - Path based equivalence checkers
   - SMT solvers
2. Normalization technique
3. Deploying SMT solvers
4. Experimental results
5. Conclusion and future works

Background

Program: An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner.
(source: `webopedia.com`)

We are not always happy with the programs we write.

Objectives of program 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: `webopedia.com`)

We are not always happy with the programs we write.

Objectives of program 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: `webopedia.com`)

We are not always happy with the programs we write.

Objectives of program optimization:

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

So, we need a **compiler**.

Can you trust your compiler?

Erroneous loop reversal

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

```
sum = 0;
for (i=N; i>=0; i--) {
  sum = sum + a[i];
} /* a[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?

Erroneous loop reversal

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

```
sum = 0;
for (i=N; i>=0; i--) {
  sum = sum + a[i];
} /* a[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.

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 of program verification restricts the scope of the input language supported by the verified compiler.
- 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.)

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 of program verification restricts the scope of the input language supported by the verified compiler.
- 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.)

What is the remedy?

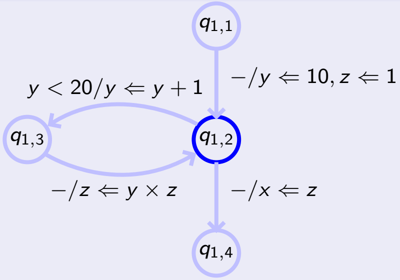
- Verified Compiler – All optimized programs will be *correct by construction*.
  Example: CompCert, INRIA
  Limitations:
  - Very hard to formally verify all passes of a compiler.
  - Undecidability of the general problem of program verification restricts the scope of the input language supported by the verified compiler.
- 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.)

How to verify programs?

Break a program into smaller chunks — 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: $p_1.p_3$, $p_1.p_2.p_3$, $p_1.p_2.p_2.p_3$, $p_1.(p_2)^*.p_3$

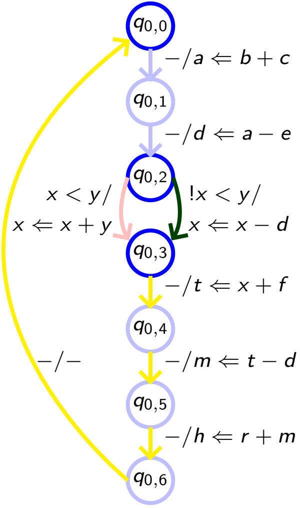
Finite State Machine with Datapath (FSMD)

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

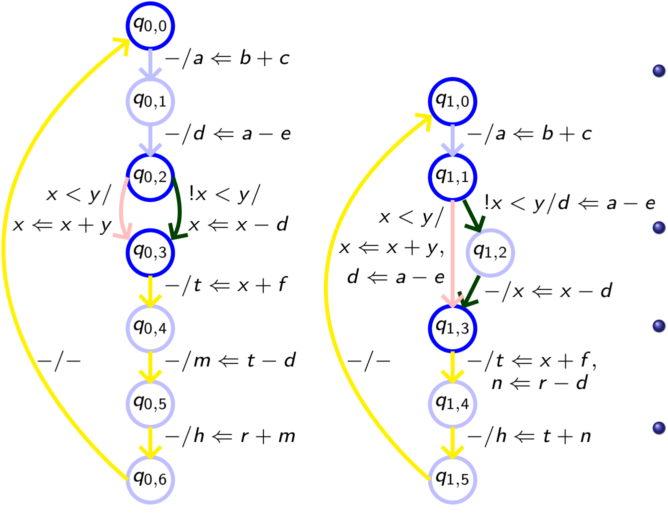
Equivalence checking of FSMDs: A basic example



(a)M_0

- Any computation in an FSMD can be represented by a concatenation of its computation paths
- A path is an alternating sequence of states and transitions, starting and ending at cutpoints
- Identification of suitable cutpoints and the path segments between them leads to a finite path cover P_0 in M_0
- For an FSMD, the reset state and all states with multiple incoming/outgoing transitions can be considered as the cutpoints
- Length and number of computations of an FSMD can both be *infinite*
- Since any computation corresponds to a concatenation of paths, it is enough to establish path equivalences

Equivalence checking of FSMDs: A basic example



(a) M_0 (b) M_1

- Two FSMDs M_0 and M_1 are equivalent if for every path in P_0 there is an equivalent path in P_1 and vice versa
- Code transformations can make this job difficult
- Paths may be extended, and the path covers are updated accordingly
- $\{q_{0,0} \xRightarrow{x<y} q_{0,3} \simeq q_{1,0} \xRightarrow{x<y} q_{1,3},\ q_{0,0} \xRightarrow{!x<y} q_{0,3} \simeq q_{1,0} \xRightarrow{!x<y} q_{1,3},\ q_{0,3} \Longrightarrow q_{0,0} \simeq q_{1,3} \Longrightarrow q_{1,0}\}$

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

Outline

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?

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

Convert both expressions into sum-of-minterms

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

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

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

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

✓

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

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

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

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

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

✓

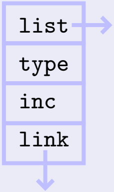
A normalization technique for integers

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;
};
```



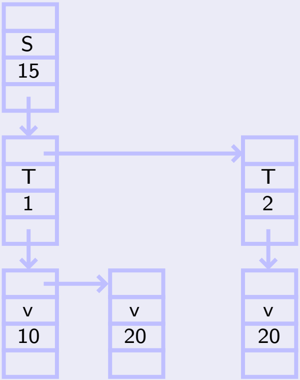
Proposed by *J. C. King, "A Program Verifier," PhD thesis, Carnegie-Mellon University, 1969*.

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) $P \rightarrow \mathrm{abs}(S) \mid (S)\,\mathrm{mod}(S) \mid S \div C_d \mid v \mid c_p$, 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.

Outline

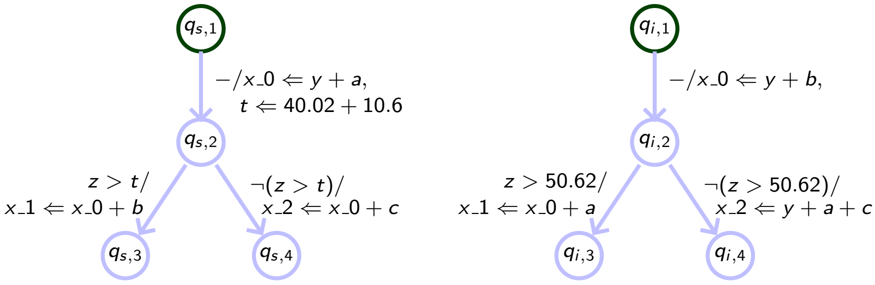
Single assignment form: A prerequisite for SMT Solvers

An example to highlight single assignment form

(a)	(b)	(c)
`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;`

- 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.

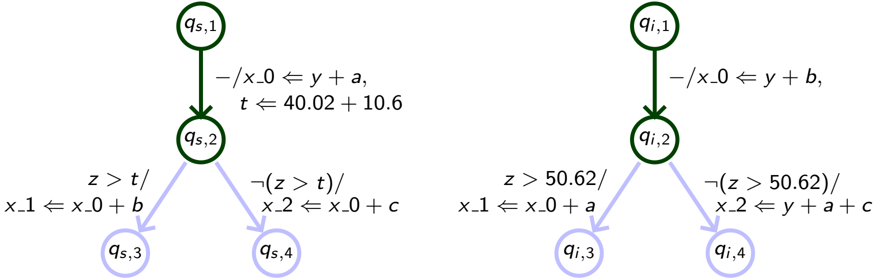
Formula generation for SMT solvers



(a) M_s (b) M_i

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

Formula generation for SMT solvers



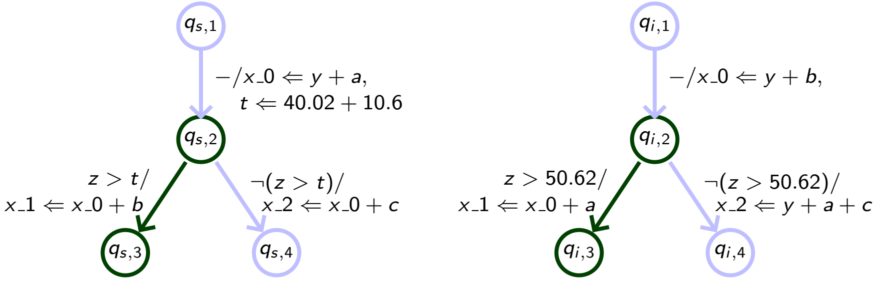
(a) M_s (b) M_i

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



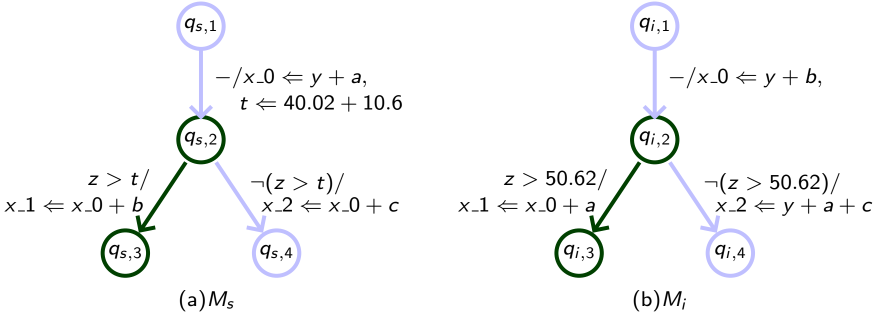
(a)M_s (b)M_i

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

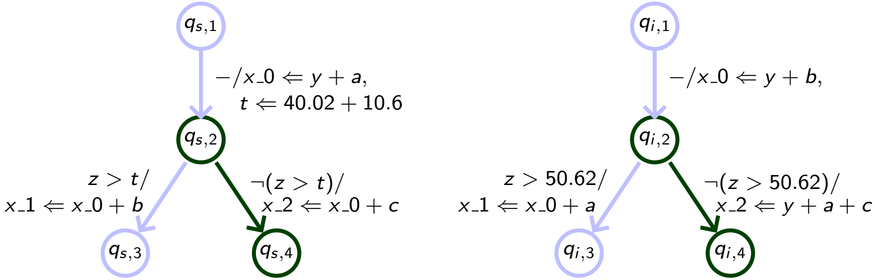


Encoding in CVC4 input language (appended with the earlier one)

```
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



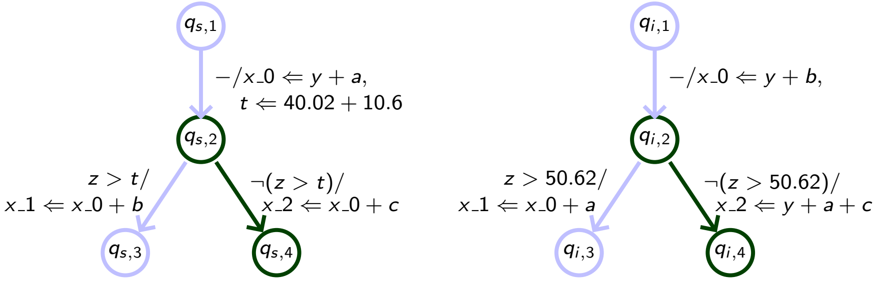
(a) M_s (b) M_i

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
#];
```

Outline

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.

Outline

Conclusion and future works

Conclusion

- 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.

References

[ISQED06] Karfa et al, "A Formal Verification Method of Scheduling in High-level Synthesis," ISQED 2006

[TCAD08] Karfa et al, "An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis," TCAD 2008

[TODAES12] Karfa et al, "Formal Verification of Code Motion Techniques using Data-flow-driven Equivalence Checking," TODAES 2012

[ISED12] Banerjee et al, "A Value Propagation Based Equivalence Checking Method for Verification of Code Motion Techniques," ISED 2012

[TCAD13] Banerjee et al, "Verification of Code Motion Techniques using Value Propagation," TCAD 2013

[COINS] `http://coins-compiler.sourceforge.jp/international/`

Thank you!

kunalb@cse.iitkgp.ernet.in