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 \textit{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

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 \textit{infinite}.

Since any computation corresponds to a concatenation of paths, it is enough to establish path equivalences.
Equivalence checking of FSMDs: A basic example

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.

$\{q_{0,0} \xleftrightarrow{x<y} q_{0,3} \simeq q_{1,0} \xleftrightarrow{x<y} q_{1,3}, q_{0,0} \xleftrightarrow{!x<y} q_{0,3}, q_{1,0} \xleftrightarrow{!x<y} q_{1,3}, q_{0,3} \Rightarrow q_{0,0} \simeq q_{1,3} \Rightarrow q_{1,0}\}$
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

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
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

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

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

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \]?

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \]
Normalization technique

Equivalence of expressions

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

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

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

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z ? \]

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \]

✓
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

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

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

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.Y.Z + \bar{X}.\bar{Y}.Z \equiv X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z? \]

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \]
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

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

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

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z ? \]

\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv \]
\[ X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \]
Normalization technique
Equivalence of expressions

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

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

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

\[
X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + X.\bar{Y}.Z + \bar{X}.\bar{Y}.Z \equiv \\
X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z?
\]

\[
X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z \equiv \\
X.Y.Z + X.Y.\bar{Z} + X.\bar{Y}.Z + X.\bar{Y}.\bar{Z} + \bar{X}.\bar{Y}.Z
\]
Normalization technique

Equivalence of expressions

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;
};

An example of normalized expression over integers

**Expression:** \(1 \times y \times z + 2 \times z + 15\)

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) \( T \rightarrow T \ast P \mid c_t \), where \( c_t \) is an integer.
3) \( P \rightarrow \text{abs}(S) \mid (S) \mod(S) \mid S \div C_d \mid v \mid c_p \),
   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

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

- 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

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
### An example to highlight single assignment form

<table>
<thead>
<tr>
<th>S1: x := a + b;</th>
<th>x_0 := a + b;</th>
<th>ASSERT x_0 = a + b;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := x + c;</td>
<td>x_1 := x_0 + c;</td>
<td>ASSERT x_1 = x_0 + c;</td>
</tr>
<tr>
<td>y := x + d;</td>
<td>y := x_1 + d;</td>
<td>ASSERT y = x_1 + d;</td>
</tr>
</tbody>
</table>

(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

Here, we have considered the path based equivalence checker of [ISED12].
Formula generation for SMT solvers

(a) $M_s$

- $q_{s,1}$
- $q_{s,2}$
  - $x_0 \leftarrow y + a$, $t \leftarrow 40.02 + 10.6$
  - $z > t/
  - $x_1 \leftarrow x_0 + b$
- $q_{s,3}$
- $q_{s,4}$

(b) $M_i$

- $q_{i,1}$
- $q_{i,2}$
  - $x_0 \leftarrow y + b$
  - $z > 50.62/
  - $x_1 \leftarrow x_0 + a$
- $q_{i,3}$
- $q_{i,4}$

Encoding in CVC4 input language

```plaintext
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) $M_s$

- $q_{s,1}$
  - $\neg/x_0 \Rightarrow y + a,$
  - $t \Leftarrow 40.02 + 10.6$
- $q_{s,2}$
  - $z > t/$
  - $x_1 \Leftarrow x_0 + b$
- $q_{s,3}$
- $q_{s,4}$

(b) $M_i$

- $q_{i,1}$
  - $\neg/x_0 \Rightarrow y + b,$
- $q_{i,2}$
  - $z > 50.62/$
  - $x_1 \Leftarrow x_0 + a$
- $q_{i,3}$
- $q_{i,4}$

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

```plaintext
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) $M_s$

- $q_{s,1}$
  - $\neg/x_0 \leftarrow y + a,$
  - $t \leftarrow 40.02 + 10.6$

- $q_{s,2}$
  - $z > t$/
  - $x_1 \leftarrow x_0 + b$

- $q_{s,3}$

- $q_{s,4}$

(b) $M_i$

- $q_{i,1}$
  - $\neg/x_0 \leftarrow y + b,$

- $q_{i,2}$
  - $z > 50.62$/
  - $x_1 \leftarrow x_0 + a$

- $q_{i,3}$

- $q_{i,4}$

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

```plaintext
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) $M_s$

- $q_{s,1}$
- $q_{s,2}$
  - $x_0 \leq y + a$
  - $t \leq 40.02 + 10.6$
- $q_{s,3}$
- $q_{s,4}$

(b) $M_i$

- $q_{i,1}$
- $q_{i,2}$
  - $z > 50.62$
  - $z > t$
- $q_{i,3}$
- $q_{i,4}$

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) $M_s$

- $\neg (z > t) /\ x.1 \leftarrow x.0 + b$
- $z > t /\ x.1 \leftarrow x.0 + b$
- $/x.0 \leftarrow y + a,$
- $t \leftarrow 40.02 + 10.6$

(b) $M_i$

- $\neg (z > 50.62) /\ x.2 \leftarrow y + a + c$
- $z > 50.62 /\ x.2 \leftarrow y + a + c$
- $/x.0 \leftarrow y + b,$

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

```plaintext
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

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

Encoding in SMT2 input language

```smt
(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

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
## Experimental results

**Table:** Results for our method on different benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Benchmark Characteristics</th>
<th>Formulae</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#op</td>
<td>#BB</td>
<td>#if</td>
</tr>
<tr>
<td>DCT</td>
<td>42</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>DIFFEQ</td>
<td>20</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>EWF</td>
<td>52</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PERFECT</td>
<td>12</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>PRIMEFAC</td>
<td>10</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>BV-DEMORGAN</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>BV-BOOLRULE</td>
<td>9</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>UD-SIMPLIFY</td>
<td>15</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>UD-MINMAX</td>
<td>15</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

× – Normalization technique is not applicable for these cases.

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


Thank you!

kunalb@cse.iitkgp.ernet.in