Translation Validation of Transformations of Embedded System Specifications using Equivalence Checking

Kunal Banerjee
Supervisors: Prof. C Mandal, Prof. D Sarkar

Dept of Computer Sc & Engg
IIT Kharagpur
Motivation

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)
Translation Validation

Translation Validation

Specification

Equivalence Checker

Implementation

Compiler

Syntax

CodeGen

Semantics

Types

Kunal Banerjee (IIT Kharagpur)

IRISS 2015

Feb 5, 2015
Program as a combination of paths

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

Representing a program using CDFG

```plaintext
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: $p_1.p_3$, $p_1.p_2.p_3$, $p_1.p_2.p_2.p_3$, $p_1.(p_2)^* . p_3$
Equivalence checking of FSMDs: A basic example

Two Finite State Machines with Datapath (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.

$$\{ q_{0,0} \xrightleftharpoons{x<y} q_{0,3} \simeq q_{1,0} \xrightleftharpoons{x<y} q_{1,3}, \quad q_{0,0} \xrightleftharpoons{!x<y} q_{0,3} \simeq q_{1,0} \xrightleftharpoons{!x<y} q_{1,3}, \quad q_{0,0} \simeq q_{1,3} \Rightarrow q_{1,0} \}$$
A major challenge: Code motions across loops

A path, by definition, cannot be extended beyond a loop.
The method of symbolic value propagation

\[ \langle \ldots, v, \ldots \rangle \quad q_{0,s} \]
\[ \langle \ldots, v, \ldots \rangle \quad q_{1,s} \]
\[ -/v \leftarrow f(x) \]
\[ -/v \leftarrow g(y) \]

\[ \langle \ldots, f(x), \ldots \rangle \quad q_{0,t} \]
\[ \langle \ldots, g(y), \ldots \rangle \quad q_{1,t} \]

(a) \( M_0 \)  
(b) \( M_1 \)

An example of value propagation
The method of symbolic value propagation

An example of value propagation with dependency between propagated values
The method of symbolic value propagation

\[
\begin{align*}
q_0, a & \xrightarrow{\beta} q_0, b \\
& \xrightarrow{\beta'} q_0, c \\
& \xrightarrow{(a) M_0} q_0, z \\
q_1, a & \xrightarrow{\alpha} q_1, b \\
& \xrightarrow{\alpha'} q_1, c \\
& \xrightarrow{(b) M_1} q_1, z
\end{align*}
\]

\[-v_i \leftarrow f(v_n, v_j)\]
\[-v_j \leftarrow h(v_k, v_l)\]
\[-v_i \leftarrow g(v_m)\]
\[-v_j \leftarrow h(v_k, v_l)\]
\[c_1 / v_i \leftarrow v_i + g(v_m)\]
\[c_1 / v_i \leftarrow v_i + f(v_n, v_j)\]

An erroneous decision taken
The method of symbolic value propagation

\[ q_0, a \rightarrow \langle \ldots, v_i, \ldots, v_j, \ldots \rangle \]
\[ \beta \rightarrow q_0, b \]
\[ \beta' \rightarrow q_0, c \]
\[ q_0, z \rightarrow \langle \ldots, g(v_m) + f(v_n, v_j), \ldots, h(v_k, v_l), \ldots \rangle \]

\[ q_1, a \rightarrow \langle \ldots, v_i, \ldots, v_j, \ldots \rangle \]
\[ \alpha \rightarrow q_1, b \]
\[ \alpha' \rightarrow q_1, c \]
\[ q_1, z \rightarrow \langle \ldots, g(v_m) + f(v_n, h(v_k, v_l)), \ldots, h(v_k, v_l), \ldots \rangle \]

Correct decision taken
The method of symbolic value propagation

Equivalence checking using value propagation

At the reset states

\((a) M_0\)

\(q_{0,a} \langle T, \langle x, y, i, N, t_1, t_2, h \rangle \rangle\)

\(-/i \leftarrow 1\)

\(q_{0,b}\)

\(q_{0,c}\)

\(i < N/\)

\(x \leftarrow t_1 + t_2 + x \ast i,\)

\(i \leftarrow i + 1\)

\(i < N/\)

\(y \leftarrow t_1 - t_2\)

\((b) M_1\)

\(q_{1,a} \langle T, \langle x, y, i, N, t_1, t_2, h \rangle \rangle\)

\(-/i \leftarrow 1, h \leftarrow t_1 + t_2,\)

\(y \leftarrow t_1 - t_2\)

\(q_{1,b}\)

\(q_{1,c}\)

\(i < N/\)

\(x \leftarrow h + x \ast i,\)

\(i \leftarrow i + 1\)

\(-i < N/\)
The method of symbolic value propagation

Equivalence checking using value propagation

At the beginning of the loops

(a) $M_0$

- $/i \leftarrow 1$
- $i < N$
- $x \leftarrow t_1 + t_2 + x \ast i$
- $i \leftarrow i + 1$

(b) $M_1$

- $/i \leftarrow 1, h \leftarrow t_1 + t_2$
- $y \leftarrow t_1 - t_2$
- $i < N$
- $x \leftarrow h + x \ast i$
- $i \leftarrow i + 1$

Kunal Banerjee (IIT Kharagpur)

IRISS 2015

Feb 5, 2015 9 / 26
Equivalence checking using value propagation

The method of symbolic value propagation

At the end of the loops
Equivalence checking using value propagation

At the end states

(a) $M_0$

- $i \leftarrow 1$
- $x \leftarrow t_1 + t_2 + x \times i$
- $i \leftarrow i + 1$

(b) $M_1$

- $i \leftarrow 1$, $h \leftarrow t_1 + t_2$
- $y \leftarrow t_1 - t_2$
- $x \leftarrow h + x \times i$
- $i \leftarrow i + 1$

Kunal Banerjee (IIT Kharagpur)
The method of symbolic value propagation

**Experimental Results – 1**

(a) BB-based  
(b) Path-based  
(c) SPARK


## Experimental Results – 1 (contd.)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Original FSMD</th>
<th>Transformed FSMD</th>
<th>#Variable</th>
<th>#across</th>
<th>Maximum</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#state</td>
<td>#path</td>
<td>#state</td>
<td>#path</td>
<td>com</td>
<td>uncom</td>
</tr>
<tr>
<td>BARCODE</td>
<td>33</td>
<td>54</td>
<td>25</td>
<td>56</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>DCT</td>
<td>16</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>41</td>
<td>6</td>
</tr>
<tr>
<td>DIFFEQ</td>
<td>15</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>EWF</td>
<td>34</td>
<td>1</td>
<td>26</td>
<td>1</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>LCM</td>
<td>8</td>
<td>11</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>IEEE754</td>
<td>55</td>
<td>59</td>
<td>44</td>
<td>50</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>LRU</td>
<td>33</td>
<td>39</td>
<td>32</td>
<td>38</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>MODN</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>PERFECT</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>QRS</td>
<td>53</td>
<td>35</td>
<td>24</td>
<td>35</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>TLC</td>
<td>13</td>
<td>20</td>
<td>7</td>
<td>16</td>
<td>13</td>
<td>1</td>
</tr>
</tbody>
</table>


Verifying Code Motions of Array-Handling Programs

The FSMD model does not provide formalism to capture arrays. Steps taken to overcome this limitation:

- Proposed a new model, namely Finite State Machine with Datapath having Arrays (FSMDA), which allows representation of data computation involving arrays using McCarthy’s access/change functions.
- Improvised the normalization process to represent arithmetic expressions involving arrays in normalized forms.
- Updated the previously mentioned equivalence checking method to accommodate extra rules for propagation of index and array variables.
- Detected a bug in the implementation of copy propagation for array variables in the SPARK compiler.

## Deriving Bisimulation Relations from Path Based Equivalence Checkers

<table>
<thead>
<tr>
<th>Bisimulation based verification</th>
<th>Path based equivalence checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ Conventional</td>
<td>× Unconventional</td>
</tr>
<tr>
<td>✓ Can handle loop shifting</td>
<td>× Cannot handle loop shifting</td>
</tr>
<tr>
<td>× Limited support for non-structure preserving transformations</td>
<td>✓ Adept in handling non-structure preserving transformations</td>
</tr>
<tr>
<td>× Termination not guaranteed</td>
<td>✓ Termination guaranteed</td>
</tr>
</tbody>
</table>

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**

```plaintext
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];
    }
}
```

```plaintext
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.
Array Data Dependence Graphs (ADDGs)

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

```c
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 <= 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. \(\{i \mid 2 \leq i \leq 101\}\)
Array Data Dependence Graphs

Construction of ADDG-1

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

```plaintext
definitions:

\begin{align*}
&M_z = \{ k \rightarrow k + 1 \mid 0 \leq k \leq N - 1 \} = M_{t1} = M_{t2} \\
&zM_{t1} = M_z^{-1} \circ M_{t1} = \{ k \rightarrow k \mid 1 \leq k \leq N \} = zM_{t2} \\
r_\alpha: z = t_1 + t_2
\end{align*}
```
Construction of ADDG-1

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

```plaintext
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];
}
```

\[ t_2 M_a = \{ j \rightarrow j \mid 1 \leq j \leq N \} = t_2 M_b \]
\[ z M_{t1} = \{ k \rightarrow k \mid 1 \leq k \leq N \} \]
\[ z M_a = \{ j \rightarrow j \mid 1 \leq j \leq N \} = z M_b \]
\[ r_\alpha : z = t1 + (a - b) \]
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];
}
```

$\{i \rightarrow i \mid 1 \leq i \leq N\}$ = $t_1 M_a$

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

$z = (a + b) + (a - b) = 2 \times a$ - simplification possible since domains match
Construction of ADDG-2

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

\[ \lambda M_z = \{ i \to i \mid 1 \leq i \leq N \} = \lambda M_a \]
\[ z M_a = \{ i \to i \mid 1 \leq i \leq N \} \]
\[ r_\beta : z = 2 \ast a \]
Equivalence of ADDGs

Two ADDGs are said to be equivalent if their characteristic formulae \( r_\alpha \) and \( r_\beta \), and corresponding mappings between the output arrays with respect to input array(s) \( z M_\alpha^\alpha \) and \( z M_\beta^\beta \), match. Hence, these two ADDGs are declared equivalent.

\[ a + b = t_1 \]
\[ a - b = t_2 \]
\[ t_1 + t_2 = z \]

\[ a \times 2 = z \]

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

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

Presence of recurrences leads to cycles in the ADDG and hence a closed form representation of $r_\alpha$ 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++ ) {
}
for ( i = 1; i < N; i++ ) {
    Z[i] = A[i];
}

ADDG

Try to establish equivalence of the separated ADDG portions.
Handling recurrences

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];
Handling recurrences

ADDGs with cycles

ADDG-1

ADDG-2
Handling recurrences

ADDGs with cycles (contd.)

In\_f + g = A

Out

ADDG-1

In\_f + g = B + C

B = C

A

Out

ADDG-2
ADDGs without cycles

ADDG-1

ADDG-2
Handling recurrences

ADDGs without cycles (contd.)

ADDG-1

\[ S1: \ A[0] = \text{In}[0]; \]

ADDG-2

\[ S1: \ A[0] = \text{In}[0]; \]
Handling recurrences

**ADDGs without cycles (contd.)**

ADDG-1

\[
\text{for (i = 1; i < N; ++i)} \\
\text{A[i] = f(In[i]) + g(A[i-1]);}
\]

ADDG-2

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

ADDGs without cycles (contd.)

ADDG-1

S3: \( \text{Out} = A[N-1] \);

ADDG-2

S7: \( \text{Out} = A[N-1] \);
## Experimental Results – 2

<table>
<thead>
<tr>
<th>Sl No</th>
<th>Benchmark</th>
<th>C lines</th>
<th>loops</th>
<th>arrays</th>
<th>slices</th>
<th>Exec time (sec) [TOPLAS]</th>
<th>Exec time (sec) [TCAD]</th>
<th>Exec time (sec) [Our]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACR1</td>
<td>14 20</td>
<td>1 3</td>
<td>6 6</td>
<td>1 1</td>
<td>0.18</td>
<td>0.76</td>
<td>0.55</td>
</tr>
<tr>
<td>2</td>
<td>LAP3</td>
<td>12 28</td>
<td>1 4</td>
<td>2 4</td>
<td>1 2</td>
<td>0.28</td>
<td>9.25</td>
<td>4.40</td>
</tr>
<tr>
<td>3</td>
<td>LIN1</td>
<td>13 13</td>
<td>3 3</td>
<td>4 4</td>
<td>2 2</td>
<td>0.12</td>
<td>0.62</td>
<td>0.52</td>
</tr>
<tr>
<td>4</td>
<td>LIN2</td>
<td>13 16</td>
<td>3 4</td>
<td>4 4</td>
<td>2 3</td>
<td>0.13</td>
<td>0.74</td>
<td>0.41</td>
</tr>
<tr>
<td>5</td>
<td>SOR</td>
<td>26 22</td>
<td>8 6</td>
<td>11 11</td>
<td>1 1</td>
<td>0.18</td>
<td>1.08</td>
<td>0.86</td>
</tr>
<tr>
<td>6</td>
<td>WAVE</td>
<td>17 17</td>
<td>1 2</td>
<td>2 2</td>
<td>4 4</td>
<td>0.31</td>
<td>6.83</td>
<td>3.82</td>
</tr>
<tr>
<td>7</td>
<td>ACR2</td>
<td>24 14</td>
<td>4 1</td>
<td>6 6</td>
<td>2 1</td>
<td>×</td>
<td>0.98</td>
<td>0.47</td>
</tr>
<tr>
<td>8</td>
<td>LAP1</td>
<td>12 21</td>
<td>1 3</td>
<td>2 4</td>
<td>1 1</td>
<td>×</td>
<td>2.79</td>
<td>1.06</td>
</tr>
<tr>
<td>9</td>
<td>LAP2</td>
<td>12 14</td>
<td>1 1</td>
<td>2 2</td>
<td>1 2</td>
<td>×</td>
<td>4.82</td>
<td>1.67</td>
</tr>
<tr>
<td>10</td>
<td>LOWP</td>
<td>13 28</td>
<td>2 8</td>
<td>2 4</td>
<td>1 2</td>
<td>×</td>
<td>9.17</td>
<td>3.90</td>
</tr>
<tr>
<td>11</td>
<td>SOB1</td>
<td>27 19</td>
<td>3 1</td>
<td>4 4</td>
<td>1 1</td>
<td>×</td>
<td>1.79</td>
<td>0.85</td>
</tr>
<tr>
<td>12</td>
<td>SOB2</td>
<td>27 27</td>
<td>3 3</td>
<td>4 4</td>
<td>1 1</td>
<td>×</td>
<td>1.85</td>
<td>1.08</td>
</tr>
<tr>
<td>13</td>
<td>EXM1</td>
<td>8 15</td>
<td>1 1</td>
<td>3 5</td>
<td>2 4</td>
<td>0.16</td>
<td>×</td>
<td>2.60</td>
</tr>
<tr>
<td>14</td>
<td>EXM2</td>
<td>8 13</td>
<td>1 1</td>
<td>3 5</td>
<td>2 3</td>
<td>0.12</td>
<td>×</td>
<td>2.12</td>
</tr>
<tr>
<td>15</td>
<td>SUM1</td>
<td>13 18</td>
<td>3 3</td>
<td>5 6</td>
<td>2 4</td>
<td>×</td>
<td>×</td>
<td>2.56</td>
</tr>
<tr>
<td>16</td>
<td>SUM2</td>
<td>13 20</td>
<td>3 4</td>
<td>5 7</td>
<td>2 4</td>
<td>×</td>
<td>×</td>
<td>2.68</td>
</tr>
</tbody>
</table>

Thank you!

http://cse.iitkgp.ac.in/~kunban/
 kunalb@cse.iitkgp.ernet.in