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

Rakshit Mittal\textsuperscript{1,2}, Rochishnu Banerjee\textsuperscript{1,3}, Dominique Blouin\textsuperscript{2,3} and Soumyadip Bandyopadhyay\textsuperscript{1,3}

\textsuperscript{1}Birla Institute of Technology and Science Pilani, KK Birla Goa Campus, Goa, India
\textsuperscript{2}Telecom Paris, Institut Polytechnique de Paris, Paris, France
\textsuperscript{3}Hasso Plattner Institut, Potsdam, Germany

Keywords: Translation Validation, Equivalence Checking, Colored Petri Net, Z3 Theorem Prover.

Abstract: Software applications often require the transformation of an input source program into a translated one for optimization. In this process, preserving the semantics across the transformation also called equivalence checking is essential. In this paper, we present ongoing work on a novel translation validation technique for handling loop transformations such as loop swapping and distribution, which cannot be handled by state-of-the-art equivalence checkers. The method makes use of a reduced size Petri net model integrating SMT solvers for validating arithmetic transformations. The approach is illustrated with two simple programs and further validated with a programs benchmark.

1 INTRODUCTION

Software applications often require the transformation of an input source program into a translated version while preserving the semantics across the transformation. These kinds of translation are performed to efficiently utilize the intrinsic computer architecture, such as multiple cores and vector registers. Researchers have developed various optimizing transformations such as code motions, common sub-expression elimination, etc. (Bacon et al., 1994)

The task of performing these translations can be automated or be done manually by design experts. For the case of safety-critical systems, these translations need to be formally validated before they can be used to certify system reliability and accuracy.

Checking the equivalence of the functional behaviors of source and translated programs is thus an important step. This process of verification by proving the semantic equivalence between source and translated programs is called translation validation. The conventional method for translation validation is to symbolically check for computational equivalence between the source and translated programs.

Instruction-level parallelism is one such transformation that is widely used in high level synthesis during the scheduling phase. Petri nets are a popular modeling paradigm that can capture and express instruction-level parallelism. The classical Petri net model has been extended in many different ways to better serve the purpose of modeling different application scenarios. Colored Petri Nets (CPNs) (Jensen and Kristensen, 2009) are one such extension that employ the concept of distinct classes of tokens (named colors) in the net.

Path-Based Equivalence Checking (PBEC) is a popular method for translation validation, which is based on graphical models/representations of code. PBEC methods rely on capturing the computations along the paths of a graph. The changes in data and control flow when traversing from one node to another along these paths represent the computations of the program. Petri net PBEC methods have been proposed in (Mittal et al., 2020; Bandyopadhyay et al., 2018) but they are not able to validate code with complex arithmetic expressions. CDFG PBEC (Banerjee et al., 2014) methods are not able to validate parallelizing transformations either.

Satisfiability Modulo Theories (SMT) solvers are tools used to solve constraint satisfaction problems.
They are used in verification as a means of analyzing the symbolic execution and semantics of programs. Z3 Theorem Prover is an industry-standard SMT Solver developed by Microsoft Research to solve such problems.

In this paper, we propose an approach for translation validation of several loop-involving code optimizing transformations. The approach, which is a work-in-progress, has three major parts: a Petri net model constructor, a Petri net path constructor, and an equivalence checker which consists of a path analyzer and the Z3 Theorem Prover (de Moura and Bjørner, 2008).

The major contributions of this paper are as follows:

- Approach to validate several transformations such as loop swapping and distribution, and parallelization which cannot be handled by state-of-the-art CDFG-based equivalence checkers.
- Refinement and reduction in size of Petri net model from that employed in (Mittal et al., 2020), which enhances the efficiency of the equivalence checking mechanism and helps with scalability issues.
- Integration of SMT solvers in the approach to check equivalence between two programs.

This paper is organized as follows: Section 2 presents an overview of the entire workflow of the approach. Through a motivating example, the workflow is explained in Section 3. Through a small set of experimentation, we have compared our method with (Bandyopadhyay et al., 2017; Mittal et al., 2020) and various other CDFG based equivalence checkers. Section 4 compares the experimental results of our approach with these other equivalence checkers. Section 5 describes the state of the art. Finally, we conclude our paper in Section 6.

2 WORKFLOW

The workflow of the proposed approach is illustrated in Fig. 1. Initially, a source program \( P_s \), is subjected to a series of semantic preserving transformations, either manual or automated, that result in a translated program \( P_t \). To validate these transformations, we need to express the code through a formal model. In our approach, we have used Colored Petri Nets (CPN) as the intermediary modeling paradigm. This task is performed by the Model Constructor module which outputs two CPNs: \( N_0 \) and \( N_1 \) corresponding to the source and translated programs respectively.

To formally check behavioral equivalence between programs, there is a necessity to characterise the computations. However, in the case of loop(s), we do not know how many times the loop(s) will be executed. To overcome this computational barrier (seemingly infinite number of loop traversals), we represent the CPN model computations as a finite set of paths. A path is characterised by the data transformation functions and their condition(s) of execution along the path. This task of extracting the set of paths is performed by Path Constructor module, which gives the set \( \pi_0 \) from \( N_0 \) and \( \pi_1 \) from \( N_1 \).

Using the path-cover data, the process of equivalence checking is carried out by the Path-Based Equivalence Checking (PBEC) module that is composed of the Path Analyzer and Z3 Theorem Prover. We state the principle of equivalence checking as follows: “\( \forall \) paths \( \pi_0 \in N_0 \), \( \exists \) an equivalent path \( \pi_1 \in N_1 \) \( \Rightarrow \pi_0 \sim \pi_1 \Rightarrow P_0 \sim P_1 \)” . The symbol \( \sim \) represents asymptotic equivalence between the models/nets. The equivalence checking process is dynamically performed by the Path Analyzer through place, variable, and transition correspondence.

This establishment of equivalence (or non-equivalence) of the data-flow characteristics of the two programs (rather, their corresponding path covers) is facilitated by the Z3 Theorem Prover. Z3 is a powerful SMT Solver that can easily validate arithmetic transformations. To enable Z3, the Path Analyzer module generates a set of Z3-compatible input expressions from the path cover data (\( I_0 \) from \( \pi_0 \) and \( I_1 \) from \( \pi_1 \)).

Z3 return a Yes/No (Boolean) result to the PBEC module. In the case of a ‘Yes’ answer, the paths are equivalent. After all candidate paths have been checked, a ‘Yes’ answer from the Path Analyzer implies equivalence but a ‘No’ answer is interpreted as ‘Can’t Say’, since the proposed equivalence checking method is sound but not complete. The same will be discussed in Section 4. In the case of ‘Yes’, the Path Analyzer also returns the equivalent pairs of paths from \( N_0 \) and \( N_1 \).

3 MOTIVATING EXAMPLE

In this section, we detail the major steps of the equivalence checking workflow using a simple example source program \( P_s \) and its transformed version \( P_t \) as given in Listings 1 and 2 respectively.

The program \( P_t \) takes five inputs \( a, b, l, m, \) and \( n \), and computes the function:

\[
k = (m \times 10^l) + (n \div 10^l) + (a - b)
\]  

(1)
3.1 Formalism

A Petri net model $N$, is a bipartite directed graph; one subset $P$, say, of vertices comprises places and the other subset $T$, say, comprises transitions. If there is an arc $(p,t)$ from a place $p$ to a transition $t$, then $p$ is called a pre-place of $t$ and the arc is called in-coming arc of $t$. The set of all pre-places of $t$ is denoted as $^tP$. If there is an arc $(t,p')$ from a transition $t$ to a place $p'$, then $p'$ is called a post-place of $t$; the set of all post-places of $t$ is denoted as $^tP$. The arc is called an out-going arc of $t$.

The set $P_{in} \subseteq P$ is designated as the set of in-ports of the model. It comprises all places that are not post-places of any transition. Similarly, another set $P_{out} \subseteq P$ is called the set of out-ports, which comprises the places that are not pre-places of any transition.

A place can hold an entity called token. A token is a set of variable-value pairs that can hold the values for multiple associated program variables. The marking of a net is a particular distribution of tokens over the net.

Each place $p \in P$ is associated with a vector of program variables $V_p$, say. For places that are in-ports, the vector consists of no variables. For places that are out-ports, the vector consists of no variables. For places that are neither in- nor out-ports, the vector contains variables that are associated with the marking.

Each out-going arc is associated with a set of functions. This function-set $F$, say, is a set of arithmetic expressions over (a subset of) the program variables. Each transition $t$ is associated with a guard condition $g_t$, which is a Boolean function over (a subset of) the program variables. A transition $t$ is said to be enabled when all its pre-place(s) have token(s) and they are associated with set(s) of values which satisfy $g_t$. Consequent to the firing of an enabled transition $t$, token(s) is(are) removed from all $p \in ^tP$ and token(s) is(are) placed in all $p \in ^tP$. The value vector of the token(s) in the post-place(s) depends respectively, on the associated function-set $F$.

Each place $p \in P$ is associated with a vector of program variables $V_p$, say. For places that are in-ports, the vector consists of no variables. For places that are out-ports, the vector consists of no variables. For places that are neither in- nor out-ports, the vector contains variables that are associated with the marking.

The corresponding transformed program $P_t$ is obtained by loop distribution followed by thread level parallelizing transformation of $P_s$; the independent sub-expressions $m \times 10^l$ and $n \div 10^l$ are computed separately in two parallelized loops.

In the following subsection, we introduce some basic terminologies to describe the example through which we will explain the equivalence checking workflow.
Figure 2: CPN model $N_0$ corresponding to the source program in Listing 1.

Table 1: Informal transformation mapping.

<table>
<thead>
<tr>
<th>Control Flow Graph</th>
<th>Petri net</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>place</td>
</tr>
<tr>
<td>transition</td>
<td>in-coming arc, transition, out-going arc</td>
</tr>
<tr>
<td>transition condition</td>
<td>guard condition associated with transition</td>
</tr>
<tr>
<td>transition function</td>
<td>function-set associated with out-going arc</td>
</tr>
</tbody>
</table>

that are neither in-ports nor out-ports, there are two kinds of such variables: changed variables and unchanged variables. Changed variables are those variables whose values are changed from when the token was last in the place. Similarly, unchanged variables are those whose values don’t change. The partition between changed and unchanged variables for each place, is defined dynamically during the computations of the Petri net and the same will be illustrated in the next subsection. Out-ports have no changed variables in the associated variable vector.

3.2 Model Construction

Using compiler internal infrastructure, the program can be transformed into an intermediate representation. This representation can be transformed to a Control Flow Graph (CFG) using the `fdump` process of the GCC compiler. In Table 1, we present an informal mapping from CFG to the proposed Petri net model.

3.3 Computation Methodology

Fig. 2 depicts the CPN model $N_0$ for the source program in Listing 1. The program is initialized with a token in place $p_1$, which is the in-port. The transition $t_i$ is now enabled and consequently fired. For example, the user inputs values $a = 4, b = 2, l = 3, m = 1$ and $n = 7425$. The token is removed from $p_1$ (and $p_2$), and moved to $p_3$. The function-set $F_1$ contains the arithmetic expressions that initialize the variables $i, a, b, c, d, e, k, l, m, n$. The variable vector is an ordered pair, the first element represents the set of changed variables, and the second element represents the set of unchanged variables. When a place is marked for the first time, all associated variables are considered as unchanged variables. The variable vector $V_{p_2}$ at this point is $\langle \{i\}, \{a, b, c, d, e, i, k, l, m, n\} \rangle$.

Now, the guard conditions of $t_2$ and $t_3$ are evaluated. Since initially $g_{t_2}$ (i.e. $i = 0 < l = 2$) is true, it is fired and the token moves from $p_2$ back to $p_2$, but with values changed according to the expressions in $F_2$ (now $i = 1, m = 10, n = 7422$). Consequently, since the values of $i, m, n$ differ from their value associated previously with $p_2$, they are considered as changed variables. The variable vector is updated to $V_{p_2} = \langle \{i, m, n\}, \{a, b, c, d, e, i, k, l, m, n\} \rangle$.

This cycle is repeatedly traversed until $i = 3$ (and $m = 1000, n = 7$) and the guard condition $g_{t_3}$ ceases to be true. This captures the while loop in the program that calculates the values of $m$ and $n$ while incrementing the value of $l$. The termination of the loop is captured by transition $t_4$ with the guard condition $i \geq l$. Now that $t_4$ is disabled and $t_3$ is enabled, the token is moved from $p_2$ to $p_3$. The value vector of the token is changed according to the expressions in $F_3$. The variable vector for $p_3$, $V_{p_3} = \langle \{i\}, \{a, b, c, d, e, i, k, l, m, n\} \rangle$ This signals the termination of the program since no more transitions are enabled. Finally, $c = 56, d = 28, e = 2, k = 1009$.

Fig. 3 depicts the CPN model $N_1$ for the translated program in Listing 2. As before, $t'_i$ is fired and the token is removed from $p'_1$ and added to $p'_2$ and $p'_3$ simultaneously. This is how the Petri net modelling paradigm captures parallelism. At this point, the variable vectors $V_{p'_2} = V_{p'_3} = \langle \{i, j, a, b, e, k, l, m, n\} \rangle$. Now, $g_{t'_2}$ and $g_{t'_3}$ are simultaneously evaluated. Since $g_{t'_2}$ and $g_{t'_3}$ are satisfied, transitions $t'_0$ and $t'_4$ are both fired simultaneously.

The value vectors in the tokens are updated according to the respective associated functions-sets, and sent back to the respective places. Since the values of $i, m$ have changed for $p'_2$ and $j, n$ have changed for $p'_3$, the variable vectors for these places are updated as follows: $V_{p'_2} = \langle \{i\}, \{a, b, e, k, l, m\} \rangle$ and $V_{p'_3} = \langle \{j, n\}, \{i, a, b, c, d, e, k, m\} \rangle$ respectively. These loops are traversed simultaneously until both $g_{t'_2}$ and $g_{t'_3}$ are false.
At this point, to evaluate $g_4'$, we have two sources of values for the three variables $i, j, l$ coming from the tokens in $p_2'$ and $p_3'$. In such cases of conflict, the variable value from the place which has the variable as a changed variable in its variable vector is given precedence. That is, since $i$ is a changed variable for $p_2'$, the value is selected from $p_2'$. Similarly, the value of $j$ is selected from $p_2'$. In the case of $l$, since it is an unchanged variable in both the places, its value in both tokens is compared. If equal (which is the case), there is no conflict; if unequal, an error is thrown. Since $t_4'$ is now enabled, it is fired, and as before, $p_4'$, now has the token with a value vector updated according to the function-set $F_S$. The values of the variables are the same as that in the source Petri net $N_0$.

A computation $\mu_{P_{in}}$ in a Petri net is defined as the sequence of markings from in-port to out-port. So the computations in the above examples can be written mathematically as:

$$\mu_{P_{in}} = \{(p_1), (p_2)\}$$

### 3.4 Notion of Path on CPN Model

In the previous section, we have seen a set of computations involving loops. In a general program, the number of loop traversals is unbounded. Therefore, we cannot characterize the set of computations and we cannot establish computational equivalence between two models. From the classical program verification techniques, we introduce the concept of finite paths such that any computation can be represented in terms of a finite set of paths. To construct the path, we need to introduce the notion of cut-points. Using (at least one) cut-points we ‘cut’ each loop to construct a finite number of paths. The notion of cut-points in our CPN model is as follows:

1. All in-ports, $p \in P_{in}$ are cut-points.
2. All out-ports, $p \in P_{out}$ are cut-points.
3. All places that have back-edges are cut-points.

A path is a sequence of out-going arcs from a set of cut-points to a cut-point, while having no cut-point in between. Through the backward cone of foci method, we construct the paths in the Petri net model. The detailed discussion of the path construction algorithm is given in (Bandyopadhyay, 2016). It is to be noted that if an out-going arc is covered in one path, it need not be considered in another path.

From the above rules for cut-points, the set of cut-points in the source model $N_0$ in Fig. 2 is $\{p_1, p_2, p_3\}$. Starting from $p_1$, we obtain the path $\alpha_1 = \{(t_1, p_2)\}$ from $p_1$ to $p_2$. Similarly, we obtain two more paths: $\alpha_2 = \{(t_2, p_2)\}$ from $p_2$ back to $p_2$ and $\alpha_3 = \{(t_3, p_3)\}$ from $p_2$ to $p_3$.

In the translated model $N_1$ in Fig. 3 the set of cut-points is $\{p_1', p_2', p_3'\}$. Starting from $p_1'$, we can construct two paths $\beta_1 = \{(t_1', p_2')\}$ to $p_2'$ and $\beta_2 = \{(t_1', p_3')\}$ to $p_3'$. Now, from $p_2'$, we obtain two paths: $\beta_3 = \{(t_2', p_3')\}$ which captures the loop from $p_2'$ back to $p_2'$ and $\beta_4 = \{(t_3', p_3')\}$ from $p_2'$ and $p_3'$ to $p_4'$. Similarly, we obtain the path $\beta_5 = \{(t_3', p_3')\}$ from $p_3'$ back to $p_3'$.

### 3.5 Validity of PBEC

To prove the validity of the path-based equivalence checker, we show that any computation can be repre-
sented as a concatenation of parallel paths.

As an example, taking the translated model $N_1$ in Fig. 3, we can express the computation as follows:

$$
\mu_{p_4} = \langle \{p'_1\}, \{p'_2, p'_3\}\rangle^{i+1}, \{p'_4\}\rangle
$$

We can express the same computation in terms of the sequence of transitions that are fired. In order to do so, the $i^{th}$ element of the computation in terms of the transitions, is (are) the transition(s) that fire(s) when moving from the $i^{th}$ to $i+1^{th}$ marking. Following this principle, we obtain the computations for the translated model as:

$$
\mu_{p_4'} = \langle \{t'_1\}, \{t'_2, t'_3\}\rangle, \{t'_4\}\rangle
$$

We can now express the computation in terms of the out-going arcs. To do this, we can simply replace each transition with its corresponding set of out-going arc(s):

$$
\mu_{p_4''} = \langle \{(t'_1, p'_2)\}, \{t'_1, p'_3\}\rangle, \{t'_2, p'_3\}, \{t'_3, p'_4\}\rangle
$$

For any computation $\mu_{p_{out}}$ of an out-port $p_{out}$ of a CPN model $N$ with path-cover $\pi$, there exists a reorganized sequence $\mu_{p_{r}}$ of paths in $\pi$, such that $\mu_{p} \simeq \mu_{p_{r}}$.

The set of paths of $N_1$, $\pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4\}$, is the transition(s) that fire(s) when

For any computation $\mu_{p_{out}}$ of an out-port $p_{out}$ of a CPN model $N$ with path-cover $\pi$, there exists a reorganized sequence $\mu_{p_{r}}$ of paths in $\pi$, such that $\mu_{p} \simeq \mu_{p_{r}}$.

The set of paths of $N_1$, $\pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4\}$, is the transition(s) that fire(s) when

Two paths $\alpha$ and $\beta$ are considered equivalent when $\alpha \simeq \beta$. The equivalence checking mechanism is based on the principle: “$\forall \alpha \in \pi_0 \exists \beta \in \pi_1 | \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \Rightarrow N_0 \simeq N_1$.” During checking, the algorithm constructs correspondence relationships between the places, variables, and transitions, respectively.

To check two arithmetic or logical expressions, we integrate the Z3 Theorem Prover with the equivalence checker, to further extend the equivalence checking capability. Following are the informal algorithmic steps for checking equivalence between $N_0$ and $N_1$:

In our motivating example, the set of paths in $N_0$ and $N_1$ are $\{\alpha_1, \alpha_2, \alpha_3\}$ and $\{\beta_1, \beta_2, \beta_3, \beta_4\}$ respectively. Also, $R_{\alpha_1} = g_1$, $R_{\alpha_2} = g_2$, $R_{\alpha_3} = g_3$, and $\alpha_1 \simeq \alpha_2 \implies \alpha_1 \simeq \alpha_3 \implies R_{\alpha_1} \simeq R_{\alpha_2} \implies R_{\alpha_3} \simeq R_{\alpha_4} = F_1$. Similarly, $R_{\beta_1} = R_{\beta_2} = F_1, R_{\beta_3} = F_2, R_{\beta_4} = F_3$. Similarly, $R_{\beta_1} = G_1, R_{\beta_2} = G_2, R_{\beta_3} = G_3$. Hence, $\alpha_1 \simeq \beta_1$ and $\alpha_3 \simeq \beta_3$.

From this information we also infer that the post-places of these paths correspond to each other, i.e. $p_1$ corresponds to $p_2$ and $p_3$.

Two paths $\alpha$ and $\beta$ are considered equivalent when $\alpha \simeq \beta$. The equivalence checking mechanism is based on the principle: “$\forall \alpha \in \pi_0 \exists \beta \in \pi_1 | \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \Rightarrow N_0 \simeq N_1$.” During checking, the algorithm constructs correspondence relationships between the places, variables, and transitions, respectively.

To check two arithmetic or logical expressions, we integrate the Z3 Theorem Prover with the equivalence checker, to further extend the equivalence checking capability. Following are the informal algorithmic steps for checking equivalence between $N_0$ and $N_1$:

1. **Condition of Execution.** $R_\alpha$, which is associated with the guard conditions $g_i$, of the transitions associated with the path.

2. **Data Transformation.** $r_\alpha$, which is associated with the function-set $F$ of the transitions associated with the path.
In the following subsection, we briefly describe the Z3 Theorem Prover, its internal equivalence checking mechanism, and how it will be integrated with the path analyzer.

### 3.6.1 Z3 Theorem Prover

For two candidate paths \( \alpha \) and \( \beta \), the Z3 Theorem Prover (Z3) receives the conditions of execution, \( R_\alpha \) and \( R_\beta \), and the data transformation, \( r_\alpha \) and \( r_\beta \), from the path analyzer. All the program statements are encoded as Static Single Assignments to preserve the order of execution. The sub-scripts ‘s’ and ‘t’ are appended for variables of \( P_s \) and \( P_t \) respectively. The input to Z3 consists of:

1. Variables and corresponding type declarations.
2. Functions in the form of `assert` statements.
3. Test statements asserted as negations. Z3 returns a sat (true) answer if it finds even one case (from the entire model space) that satisfies equivalence. Using the negation, we can test that equality is satisfied over the entire model space. Mathematically: for \( \xi \) (the model space) and \( c \) (the cases), by De Morgan’s Law, \( \neg (\bigcup c \in \xi) = \bigcap c \in \xi \).

So, an unsat output from Z3 actually corresponds to equivalence and a sat output implies non-equivalence. Also, the test statements check for equality only between the common variables of \( P_s \) and \( P_t \). In case of multiple assignment of the same variable, only the last executed variable is considered (i.e. the variable with highest numerical suffix).

As an example, in **Step 3** for checking equivalence between the paths \( \alpha_3 \) and \( \beta_4 \), the Z3 input is as follows:

```plaintext
(* a_s b_s *)

(g_t4_t t = (div c_s d_s ))

(g_t3_s t >= i_0_s && l_s t)

(g_t4_t t = (div c_s d_s ))

(g_t3_s t >= i_0_s && l_s t)

(g_t4_t t = (div c_s d_s ))

(g_t3_s t >= i_0_s && l_s t)

(check-sat)
```

Listing 3: Checking equivalence of \( R_\alpha \) and \( R_\beta \).

In Listing 3, the first two lines define the guard conditions as Boolean functions. Lines 3-7 define the associated variables. The next two lines 8-10 define

\[
\begin{align*}
g_{3,s} &= i \geq l \\
g_{4,t} &= j \geq l
\end{align*}
\]

To facilitate equivalence checking, equivalence between variables is asserted in lines 11-13. \( i_{0,t} = i_{0,s} \) is inferred from \( F_1 \) and \( F_2 \). Next is the assert statement for equivalence checking defined as a negation. In the last statement we check equivalence. Z3 returns unsat which implies \( R_\alpha = R_\beta \). Similarly, to check the data transformation equivalence between the two paths, the corresponding code is given in Listing 4.

```plaintext
(* a_s b_s *)

(\begin{align*}
\text{assert} &= (c_s \text{-} (* a_s \text{-} (* a_s \text{-} a_s s_s))) \\
\text{assert} &= (b_s \text{-} (* b_s \text{-} b_s s_s)))
\end{align*})

\text{assert} = (a_s \text{-} (c_s d_s ))

\text{assert} = (k_s \text{-} (n_1_s t s_s))

\text{assert} = (c_t \text{-} (a_t b_t))

\text{assert} = (k_t \text{-} (n_1_t t e_t))

\text{assert} = \text{not}(a_s \text{-} a_t)

\text{assert} = (b_s \text{-} b_t)

\text{assert} = (n_1_s \text{-} n_1_t)

\text{assert} = (e_s \text{-} e_t)

\text{assert} = (k_s \text{-} k_t)

(check-sat)
```

Listing 4: Checking equivalence of \( r_\alpha \) and \( r_\beta \).

In Listing 4, the lines 1-14 declare the associated variables. Lines 15-18 assert the equivalence for \( a, b, m, n \) between the source and translated program. Lines 19-24 serve as an assertion for the computations in \( F_1 \) and lines 25-26 assert the computations in \( F_2 \). Finally, lines 27-30 comprise the assertion statement in negation for equivalence checking between all the corresponding defined variables and the statement in line 31 checks for equivalence. Z3 gives the result unsat for the check, which implies that \( r_\alpha = r_\beta \).
Table 2: Model size for different Petri-net PBEC.

<table>
<thead>
<tr>
<th>Example</th>
<th>ST-1</th>
<th>ST-2</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCM</td>
<td>34</td>
<td>28</td>
<td>6</td>
</tr>
<tr>
<td>MINMAX</td>
<td>31</td>
<td>27</td>
<td>7</td>
</tr>
<tr>
<td>PETERSON</td>
<td>11</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>DEKKERS</td>
<td>19</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>LUP</td>
<td>28</td>
<td>21</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3: Capabilities of different PBEC.

<table>
<thead>
<tr>
<th>Example</th>
<th>FSMD-VP</th>
<th>FSMD-EVP</th>
<th>ST-1</th>
<th>ST-2</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCM</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>MINMAX</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PETERSON</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>DEKKERS</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LUP</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4 EXPERIMENTAL RESULTS

We have manually tested our equivalence checking algorithm on five examples, where parallelising transformations are applied using Pluto (Bondhugula et al., 2008) and Par4All (Amini et al., 2012) compilers. BCM is a toy code for validating Basic Code Motion technique, where some polynomial arithmetic operations are applied in the basic blocks. PETERSON and DEKKERS are implementations of classical solutions to the mutual exclusion problem of two concurrent processes. In the critical section, some polynomial expression computation is present. LUP computes the LU-decomposition with Pivoting, for a matrix. We have only taken the pivoting routine which does not contain any array. The details for LUP are given in the PLuto example suite (Bondhugula et al., 2008). MINMAX computes the sum of the maximum of four numbers ($n_1, n_2, n_3, n_4$) and the minimum of four numbers ($n_1, n_3, n_5, n_7$). The programs and their descriptions can be found in (Bandyopadhyay, 2016).

A comparative study of the model size of our proposed approach with the models of two other Petri net-based equivalence checking tools ST-1 (Bandyopadhyay et al., 2017) and ST-2 (Mittal et al., 2020). It is to be noted that the model size of the current method is comparable with ST-2.

Table 3 presents several parallelising and arithmetic transformation verification capabilities of the proposed approach, compared with ST-1, ST-2 and two CDFG (Control Data Flow Graph) based PBEC namely, FSMD-VP (Finite State Machine with Datapath and Value Propagation) (Banerjee et al., 2014) and FSMD-EVP (Finite State Machine with Datapath and Extended Value Propagation) (Chouksey et al., 2019). It is to be noted that both FSMD based PBEC cannot handle the parallelising transformations because FSMD is a sequential model of computation. ST-1 and ST-2 cannot handle arithmetic transformations. They have their own normalizer, which affects their limitations. These limitations are overcome by Z3.

5 RELATED WORK

Translation validation was introduced in (Pnueli et al., 1998) and was demonstrated in (Necula, 2000) and (Rinard and Diniz, 1999). The approach was further enhanced in (Kundu et al., 2008) where they verified the high-level synthesis tool named SPARK. All these techniques are bisimulation based methods. A loop parallelizing transformation validation method comprising rewrite rules has been reported in (Bell, 2013). A bisimulation method for parallel programs is also reported in (Milner, 1989). Another equivalence checking method is the inductive-inferring based technique reported in (Felsing et al., 2014). The method only works for scalar handing programs. It compares the coupling predicates between two programs.

A major limitation of these methods is that the termination is not guaranteed. To alleviate this shortcoming, a path based equivalence checker for the FSMD model was proposed for uniform and non-uniform code motions, code motion across loop and loop invariant code optimizations in (Karfa et al., 2012; Banerjee et al., 2014; Chouksey et al., 2019). However, these methods cannot handle loop swapping transformations and many thread-level parallelizing transformations because FSMDs cannot capture parallel behaviors easily.

The literature records no significant attempts for devising formal equivalence checking methods using Petri net based models which is essentially a parallel model of computation; although, there are several works on property verification using Petri net modelling paradigm (Lime et al., 2009; Charron-Bost et al., 2013; Corradini et al., 2013; Westergaard, 2012). In (Bandyopadhyay et al., 2018), the validation of loop swapping and thread level parallelising transformations using Petri net based models of programs was reported. However, the model size of the method is not tractable. The major limitation of this method is it cannot handle loop invariant code motion as well as polynomial arithmetic transformations. To overcome the limitations, a modification in the model construction as well as equivalence checking was reported in (Mittal et al., 2020). However, the method cannot handle polynomial arithmetic transformations.
6 CONCLUSION

In this paper we presented our ongoing work on developing an approach to check the equivalence of software programs using a novel translation validation technique for handling loops. In addition, our approach makes use of SMT solvers to validate arithmetic transformations. Such constructions cannot be handled by state-of-the-art equivalence checkers.

We presented an initial validation of the approach for a standard benchmark. Currently this validation was performed manually. Therefore, our future work is to implement a tool-chain supporting the approach and validate it on a larger benchmark. For this, we will reuse existing compiler front-ends (e.g. GCC) and automatically construct the Petri Net models from the generated intermediate code representation so that the approach can be tested on different programming languages, potentially including existing architecture description languages such as UML, SysML and AADL. This will also allow us to further characterize the domain of applicability of the approach; i.e. which language constructions and translations are handled by our approach and to evaluate scalability for large programs.

ACKNOWLEDGEMENT

The authors would like to thank Didier Buchs and Dipankar Sarkar for initial ideas regarding the refinement of the Petri net model.

REFERENCES


