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

Rakshit Mittal[1,2][a], Rochishnu Banerjee[1][b], Dominique Blouin[2,3][c]
and Soumyadip Bandyopadhyay[1,3][d]

[1] *Birla Institute of Technology and Science Pilani, KK Birla Goa Campus, Goa, India*
[2] *Telecom Paris, Institut Polytechnique de Paris, Paris, France*
[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 translation 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 modelling 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.

[a] https://orcid.org/0000-0001-9871-800X
[b] https://orcid.org/0000-0002-0114-2452
[c] https://orcid.org/0000-0001-7606-0251
[d] https://orcid.org/0000-0001-5865-9754

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 π_0 from N_0 and π_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 $\in N_0$, $\exists$ an equivalent path $\in N_1$ $\implies \pi_0 \simeq \pi_1 \implies P_s \simeq P_t$". The symbol $\simeq$ 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 π_0 and I_1 from π_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 Analayzer 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_s takes five inputs a, b, l, m, and n, and computes the function:

$$k = (m \times 10^l) + (n \div 10^l) + (a - b) \tag{1}$$

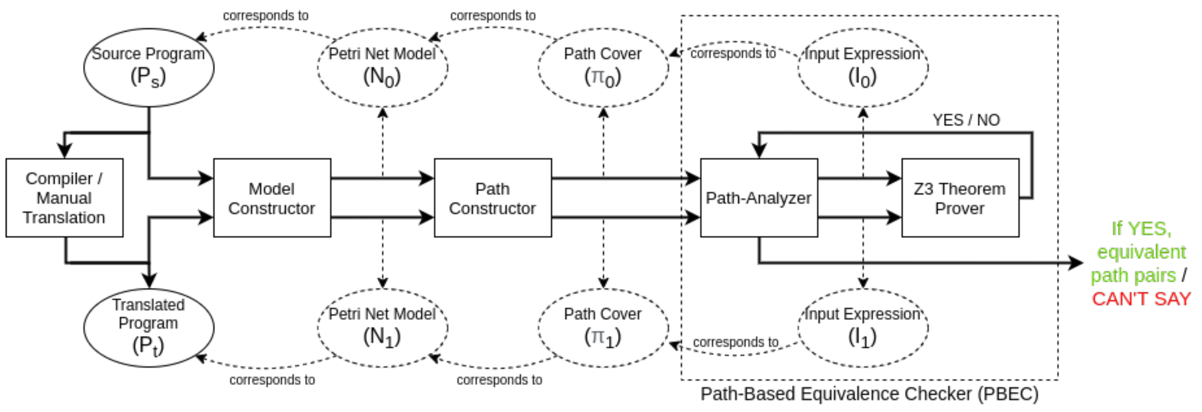


Figure 1: Workflow of proposed approach.

```
int i = 0,a,b,c,d,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);
while ( i < l ) {
    m = m * 10;
    n = n / 10;
    i++;}
c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b)
e = c / d;
k = m + n + e;
```

Listing 1: The source program P_s.

```
int i = j = 0,a,b,e,k,l,m,n;
scanf("%f,%f,%f,%f,%f",
       &a,&b,&l,&m,&n);
#parbegin scop
while ( i < l ) {
    m = m * 10;
    i++;}
||
while ( j < l ) {
    n = n / 10;
    j++;}
#parend scop
e = a - b;
k = m + n + e;
```

Listing 2: The transformed program P_t.

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.

3.1 Formalism

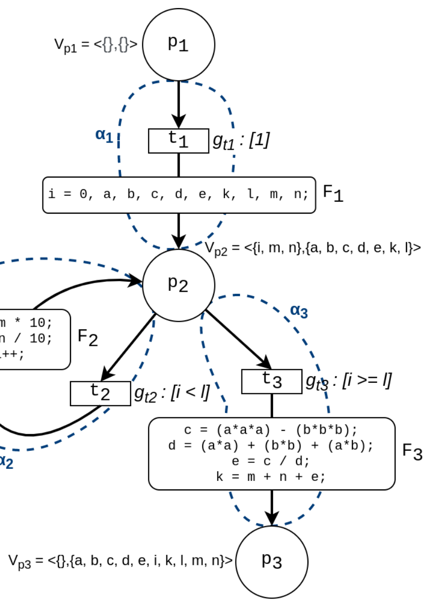
A Petri net model N, is a bipartite directed graph; one subset P, say, of vertices comprises *places* and the other subset T, say, comprises *transitions*. If there is an arc (p,t) from a place p to a transition t, then p is called a *pre-place* of t and the arc is called *in-coming arc* of t. The set of all pre-places of t is denoted as $^\circ t$. 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 t°. The arc is called an *out-going arc* of t.

The set $P_{in} \subset P$ 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 $P_{out} \subset P$ 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 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 $p \in {}^\circ t$ and token(s) is(are) placed in all $p \in t^\circ$. 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


Vp1 = <{},{}>
p1
α1
t1
gt1 : [1]
F1
Vp2 = <{i, m, n},{a, b, c, d, e, k, l}>
p2
α3
F2
t2
gt2 : [i < l]
t3
gt3 : [i >= l]
F3
α2
Vp3 = <{},{a, b, c, d, e, i, k, l, m, n}>
p3

```
i = 0, a, b, c, d, e, k, l, m, n;
```

```
m = m * 10;
n = n / 10;
i++;
```

```
c = (a*a*a) - (b*b*b);
d = (a*a) + (b*b) + (a*b);
e = c / d;
k = m + n + e;
```

Figure 2: CPN model N_0 corresponding to the source program in Listing 1.

Table 1: Informal transformation mapping.

Control Flow Graph	Petri net
state	place
transition	in-coming arc, transition, out-going arc
transition condition	guard condition associated with transition
transition function	function-set associated with out-going arc

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_1 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 ${}^{\circ}t_1 = \{p_1\}$, and moved to $t^{\circ}{}_1 = \{p_2\}$. 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 $\langle\{\},\{a,b,c,d,e,i,k,l,m,n\}\rangle$.

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 $i = 1$, $m = 10$, $n = 742$). 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 $V_{p_2} = \langle\{i,m,n\},\{a,b,c,d,e,k,l\}\rangle$.

This cycle is repeatedly traversed until $i = 3$ (and $m = 1000$, $n = 7$) and the guard condition g_{t_2} 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 i. The termination of the loop is captured by transition t_3 with the guard condition $i \geq l$. Now that t_2 is disabled and t_3 is enabled, t_3 is fired, and 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, $V_{p_3} = \langle\{\},\{a,b,c,d,e,i,k,l,m,n\}\rangle$ 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'_1 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, $g_{t'_2}$, $g_{t'_3}$, and $g_{t'_4}$ are simultaneously evaluated. Since $g_{t'_2}$ and $g_{t'_3}$ are satisfied, transitions t'_2 and t'_3 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: $V_{p'_2} = \langle\{i,m\},\{j,a,b,e,k,l,n\}\rangle$ 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.

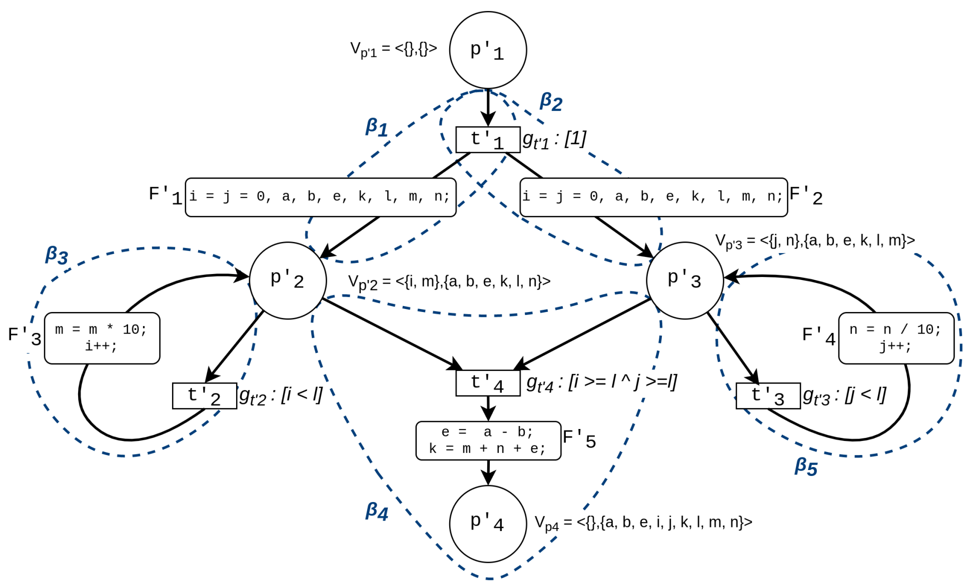


Figure 3: CPN model N_1 corresponding to program in Listing 2.

At this point, to evaluate $g_{t'_4}$, 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 for i is selected from p'_2. Similarly, the value of j is selected from p'_3. 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'_5. The values of the variables are the same as that in the source Petri net N_0.

A *computation* $\mu_{p_{out}}$ 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:

$$\mu_{p_3} = \langle \{p_1\}, \{p_2\}^4, \{p_3\} \rangle$$

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, $\forall p \in P_{in}$, are cut-points.
2. All out-ports, $\forall p \in P_{out}$, 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 $\alpha_1 = \langle(t_1, p_2)\rangle$ from p_1 to p_2. Similarly, we obtain two more paths: $\alpha_2 = \langle(t_2, p_2)\rangle$, from p_2 back to p_2 and $\alpha_3 = \langle(t_3, p_3)\rangle$ from p_2 to p_3.

In the translated model N_1 in Fig. 3 the set of cut-points is $\{p'_1, p'_2, p'_3, p'_4\}$. Starting from p'_1, we can construct two paths $\beta_1 = \langle(t'_1, p'_2)\rangle$ to p'_2 and $\beta_2 = \langle(t'_1, p'_3)\rangle$ to p'_3. Now, from p'_2, we obtain two paths: $\beta_3 = \langle(t'_2, p'_2)\rangle$ which captures the loop from p_2 back to p_2 and $\beta_4 = \langle(t'_4, p'_4)\rangle$ from p'_2 and p'_3 to p'_4. Similarly, we obtain the path $\beta_5 = \langle(t'_3, p'_3)\rangle$ 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 represented 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:

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

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:

$$\mu_{p'_4} = \langle \{t'_1\}, \{t'_2, t'_3\}^{l}, \{t'_4\} \rangle$$

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):

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

For any computation $\mu_{p_{out}}$ of an out-port p_{out} of a CPN model N with path-cover π, there exists a reorganized sequence μ^r_p, of paths in π, such that $\mu_p \simeq \mu^r_p$.

The set of paths of N_1, $\pi_1 = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$. Initially, $\mu^r_{p'_4} = \phi$. The last member of $\mu_{p'_4}$ is (t'_4, p'_4). The path β_4 has (t'_4, p'_4) as its last member. So β_4 is prepended to $\mu^r_{p'_4}$, and all the out-going arcs in β_4 (only (t'_4, p'_4)) are removed once from $\mu_{p'_4}$.

Now, the last member of $\mu_{p'_4}$ is $\{(t'_2, p'_2), (t'_3, p'_3)\}$. (t'_2, p'_2) is the last member of β_3 and (t'_3, p'_3) is the last member of β_5. So $\{\beta_3 || \beta_5\}$ is prepended to $\mu^r_{p'_4}$ and all the out-going arcs from β_3 and β_5 are removed once from $\mu_{p'_4}$. This step will be repeated $l-1$ times until the only element left in $\mu_{p'_4}$ is $\{(t'_1, p'_2), (t'_1, p'_3)\}$. Since (t'_1, p'_2) is the last element of β_1 and (t'_1, p'_3) is the last element of β_2, $\{\beta_1 || \beta_2\}$ is prepended to $\mu^r_{p'_4}$. The algorithm is now terminated since $\mu_{p'_4}$ is empty. Therefore

$$\mu^r_{p'_4} = \langle \{\beta_1 || \beta_2\}, \{\beta_3 || \beta_5\}^{l}, \{\beta_4\} \rangle$$

3.6 Equivalence Checking Mechanism

There are two entities associated with every path

1. *Condition of Execution*, R_α, which is associated with the guard conditions g_t, of the transitions associated with the path.
2. *Data Transformation*, r_α, which is associated with the function-set F of the transitions associated with the path.

Two paths α and β are considered equivalent when $R_\alpha \simeq R_\beta$ and $r_\alpha = r_\beta$. The equivalence checking mechanism is based on the principle: "$\forall\ \alpha \in \pi_0$, $\exists\ \beta \in \pi_1$ and $\forall\ \beta \in \pi_1$, $\exists\ \alpha \in \pi_0 \mid \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \implies N_0 \simeq N_1$". 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 $\{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$ respectively. Also, $R_{\alpha_1} = g_{t_1}$, $R_{\alpha_2} = g_{t_2}$, $R_{\alpha_3} = g_{t_3}$ and $r_{\alpha_1} = F_1$, $r_{\alpha_2} = F_2$, $r_{\alpha_3} = F_3$. Similarly, $R_{\beta_1} = R_{\beta_2} = g_{t'_1}$, $R_{\beta_3} = g_{t'_2}$, $R_{\beta_4} = g_{t'_4}$, $R_{\beta_5} = g_{t'_3}$ and $r_{\beta_1} = F'_1$, $r_{\beta_2} = F'_2$, $r_{\beta_3} = F'_3$, $r_{\beta_4} = F'_5$, $r_{\beta_5} = F'_4$.

Step 1). Taking the first element of π_0, i.e. α_1, we look at its pre-place p_1. Places p_1 and p'_1 correspond to each other since they are in-ports. Since p'_1 is a pre-place for paths β_1 and β_2, these two paths are candidate paths for α_1. The SMT solver tells us that $R_{\alpha_1} \simeq R_{\beta_1}$ (i.e. $g_{t_1} = g_{t'_1}$) and $R_{\alpha_1} \simeq R_{\beta_2}$ (i.e. $g_{t_1} = g_{t'_2}$). The SMT solver also tells us that $r_{\alpha_1} = r_{\beta_1}$ (i.e. $F_1 = F'_1$) and $r_{\alpha_1} = r_{\beta_2}$ (i.e. $F_1 = F'_2$). Hence, $\alpha_1 \simeq \beta_1$ and $\alpha_1 \simeq \beta_2$.

From this information we also infer that the post-places of these paths correspond to each other, i.e. p_2 corresponds to p'_2 and p'_3.

Step 2). Taking the next element of π_0 i.e. α_2. The pre-place of α_2 is p_2, which corresponds to p'_2 and p'_3. Since β_3 and β_5 have the two places respectively as their pre-place, they are candidate paths for α_2. Checking for equivalence between these paths results in a 'No' answer from the SMT solver. So, we go for *path merging*. The paths β_3 and β_5 can be merged parallelly, due to place, variable, and transition correspondence. The SMT solver tells us that $R_{\alpha_2} \simeq R_{\beta_3 || \beta_5}$. Similarly, $r_{\alpha_2} = r_{\beta_3 || \beta_5}$. Hence, $\alpha_2 \simeq (\beta_3 \parallel \beta_5)$.

Step 3). Finally, taking the path α_3, it's pre-place is p_2 which has correspondence to p'_2 and p'_3, which are the pre-places of β_4. Similarly, the post-place of α_3 corresponds to the post-place of β_4 since they are out-ports in their respective nets. Hence, β_4 is a candidate path for α_3. The SMT solver tells us that $R_{\alpha_3} \simeq R_{\beta_4}$ and $r_{\alpha_3} = r_{\beta_4}$. Hence, $\alpha_3 \simeq \beta_4$. So,

$$\alpha_1 \simeq \beta_1, \beta_2\ ;\ \alpha_2 \simeq \{\beta_3 \parallel \beta_5\}\ ;\ \alpha_3 \simeq \beta_4$$

Since "$\forall\ \alpha \in \pi_0\ \exists\ \beta \in \pi_1 \mid \alpha \simeq \beta \implies \pi_0 \simeq \pi_1 \implies N_0 \simeq N_1$". That is, the programs in Listing 1 and Listing 2 are semantically equivalent.

In the following subsection, we briefly describe the Z3 Theorem Prover, it's internal equivalence checking mechanism, and how it will be integrated with the path analyzer.

3.6.1 Z3 Theorem Prover

For two candidate paths α and β , the Z3 Theorem Prover (Z3) receives the conditions of execution, R_α and R_β, and the data transformation, r_α and r_β, 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 ξ (the model space) and c (the cases), by De Morgan's Law, $\neg(\bigcup\limits_{c\in\xi} c) = \bigcap\limits_{c\in\xi} \neg c$.

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 α_3 and β_4, the Z3 input is as follows:

```
(declare-const g_t3_s Bool)                 1
(declare-const g_t4_t Bool)                 2
(declare-const i_0_s Int)                   3
(declare-const i_0_t Int)                   4
(declare-const j_0_t Int)                   5
(declare-const l_s Int)                     6
(declare-const l_t Int)                     7
(assert (= g_t3_s(>= i_0_s l_s)))           8
(assert (= g_t4_t(and(>= i_0_t l_t)         9
                 (>= j_0_t l_t))))          10
(assert (= l_s l_t))                        11
(assert (= i_0_s i_0_t))                    12
(assert (= i_0_t j_0_t))                    13
(assert (not(= g_t3_s g_t4_t)))             14
(check-sat)                                 15
```

Listing 3: Checking equivalence of R_{α_3} and R_{β_4}.

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 $g_{t3}\_s = i \geq l$ and $g_{t4}\_t = i \geq l \ \& \ j \geq l$. To facilitate equivalence checking, equivalence between variables is asserted in lines 11-13. $i\_0\_t = i\_0\_s$ is infered 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 $R_{\alpha_3} = R_{\beta_4}$. Similarly, to check the data transformation equivalence between the two paths, the corresponding code is given in Listing 4.

```
1  (declare-const a_s Int)
2  (declare-const a_t Int)
3  (declare-const b_s Int)
4  (declare-const b_t Int)
5  (declare-const c_s Int)
6  (declare-const d_s Int)
7  (declare-const e_s Int)
8  (declare-const e_t Int)
9  (declare-const k_s Int)
10 (declare-const k_t Int)
11 (declare-const m_1_s Int)
12 (declare-const m_1_t Int)
13 (declare-const n_1_s Int)
14 (declare-const n_1_t Int)
15 (assert (= a_s a_t))
16 (assert (= b_s b_t))
17 (assert (= m_1_s m_1_t))
18 (assert (= n_1_s n_1_t))
19 (assert (= c_s (-(* a_s (* a_s a_s))
20         (* b_s (* b_s b_s)))))
21 (assert (= d_s (+(* a_s a_s) (+(* b_s b_s)
22         (* a_s b_s)))))
23 (assert (= e_s (div c_s d_s)))
24 (assert (= k_s (+ m_1_s (+ n_1_s e_s))))
25 (assert (= e_t (+ a_t b_t)))
26 (assert (= k_t (+ m_1_t (+ n_1_t e_t))))
27 (assert (not (and (= a_s a_t)
28         (and (= b_s b_t)
29         (and (= m_1_s m_1_t)
30         (and (= e_s e_t)
31         (and (= n_1_s n_1_t)
32         (= k_s k_t))))))))
33 (check-sat)
```

Listing 4: Checking equivalence of r_{α_3} and r_{β_4}.

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_5'. 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 $r_{\alpha_3} = r_{\beta_4}$.

Table 2: Model size for different Petri-net PBEC.

Example	ST-1		ST-2		Proposed	
	p	t	p	t	p	t
BCM	34	28	6	6	3	2
MINMAX	31	27	7	7	4	6
PETERSON	11	9	4	2	6	8
DEKKERS	19	14	6	4	6	8
LUP	28	21	6	4	10	16

Table 3: Capabilities of different PBEC.

Example	FSMD-VP	FSMD-EVP	ST-1	ST-2	Proposed
BCM	X	X	X	X	✓
MINMAX	X	X	✓	✓	✓
PETERSON	X	X	X	X	✓
DEKKERS	X	X	X	X	✓
LUP	X	X	✓	✓	✓

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_5, n_6, n_7). The programs and their descriptions can be found in (Bandyopadhyay, 2016).

Table 2 presents 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 parallelizing 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 parallelizing 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-inferencing 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 computationm; 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

Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., Mcmahon, J., Pasquier, F.-X., Péan, G., and Villalon, P. (2012). Par4all: From convex array regions to heterogeneous computing. *Workshop IMPACT*.

Bacon, D. F., Graham, S. L., and Sharp, O. J. (1994). Compiler transformations for high-performance computing. *ACM Computing Survey*, 26(4).

Bandyopadhyay, S. (2016). *Path based equivalence checking of Petri net representation of programs for translation validation*. PhD thesis, IIT, Kharagpur.

Bandyopadhyay, S., Sarkar, D., and Mandal, C. (2018). Equivalence checking of petri net models of programs using static and dynamic cut-points. *Acta Informatica*.

Bandyopadhyay, S., Sarkar, S., Sarkar, D., and Mandal, C. A. (2017). Samatulyata: An efficient path based equivalence checking tool. In *ATVA*.

Banerjee, K., Karfa, C., Sarkar, D., and Mandal, C. (2014). Verification of code motion techniques using value propagation. *IEEE TCAD*, 33(8).

Banerjee, K., Sarkar, D., and Mandal, C. (2014). Extending the fsmd framework for validating code motions of array-handling programs. *IEEE TCAD*, 33(12).

Bell, C. J. (2013). Certifiably sound parallelizing transformations. In *CPP*, pages 227–242.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI*.

Charron-Bost, B., Merz, S., Rybalchenko, A., and Widder, J. (2013). Formal verification of distributed algorithms (dagstuhl seminar 13141). *Dagstuhl Reports*, 3(4):1–16.

Chouksey, R., Karfa, C., and Bhaduri, P. (2019). Translation validation of code motion transformations involving loops. *IEEE TCADICS*, 38(7).

Corradini, A., Ribeiro, L., Dotti, F. L., and Mendizabal, O. M. (2013). A formal model for the deferred update replication technique. In *TGC*.

de Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *TACAS*, pages 337–340.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., and Ulbrich, M. (2014). Automating regression verification. In *ACM/IEEE International Conference on ASE*.

Jensen, K. and Kristensen, L. M. (2009). *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer.

Karfa, C., Mandal, C., and Sarkar, D. (2012). Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM TODAES*, 17(3).

Kundu, S., Lerner, S., and Gupta, R. (2008). Validating high-level synthesis. CAV.

Lime, D., Roux, O. H., Seidner, C., and Traonouez, L. (2009). Romeo: A parametric model-checker for petri nets with stopwatches. In *TACAS*.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, Inc.

Mittal, R., Banerjee, R., Sarkar, S., and Bandyopadhyay, S. (2020). Translation validation of loop involving code optimizing transformations using petri net based models of programs. In *PNSE Workshop*.

Necula, G. C. (2000). Translation validation for an optimizing compiler. In *PLDI*.

Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *TACAS*.

Rinard, M. and Diniz, P. (1999). Credible compilation. Technical Report MIT-LCS-TR-776, MIT.

Westergaard, M. (2012). Verifying parallel algorithms and programs using coloured petri nets. *Trans. on Petri Nets and Other Models of Concurrency*.