# An End-to-end Formal Verifier for Parallel Programs

Soumyadip Bandyopadhyay[1], Santonu Sarkar[1] and Kunal Banerjee[2]
[1]*BITS Pilani, K K Birla Goa Campus, Goa, India*
[2]*Indian Institute of Technology, Kharagpur, India*

Keywords: Equivalence Checking, Petri Net based Representation for Embedded Systems (PRES+) Model, Finite State Machine with Datapath (FSMD) Model, High-level Language.

Abstract: Among the various models of computation (MoCs) which have been used to model parallel programs, Petri net has been one of the mostly adopted MoC. The traditional Petri net model is extended into the PRES+ model which is specially equipped to precisely represent parallel programs running on heterogeneous and embedded systems. With the inclusion of multicore and multiprocessor systems in the domain of embedded systems, it has become important to validate the optimizing and parallelizing transformations which system specifications go through before deployment. Although PRES+ model based equivalence checkers for validating such transformations already exist, construction of the PRES+ models from the original and the translated programs was carried out manually in these equivalence checkers, thereby leaving scope for inaccurate representation of the programs due to human intervention. Furthermore, PRES+ model tends to grow more rapidly with the program size when compared to other MoCs, such as FSMD. To alleviate these drawbacks, we propose a method for automated construction of PRES+ models from high-level language programs and use an existing translation scheme to convert PRES+ models to FSMD models to validate the transformations using a state-of-the-art FSMD equivalence checker. Thus, we have composed an end-to-end fully automated equivalence checker for validating optimizing and parallelizing transformations as demonstrated by our experimental results.

## 1 INTRODUCTION

Embedded systems are becoming increasingly complex and pervasive with each passing day. Applications running in embedded devices demand large compute resources and they have started exploiting the parallelism as the underlying compute infrastructure is becoming more and more powerful (Gay et al., 2003; Marwedel, 2006). Designer of such applications uses compiler's optimizing transformation on the code (Smith et al., 1992; Raghavan, 2010); which if carried out by untrusted compilers, can result in software bugs[1]. Thus, for embedded systems, there is a growing concern to validate the applications before its deployment.

It is important to verify whether the implemented code faithfully represents the intended functionality, which is commonly known as the translation validation process. Here, each individual translation is followed by a validation phase to establish the behavioural equivalence of the source code and the target code (Pnueli et al., 1998; Necula, 2000; Kundu et al., 2008; Rinard and Diniz, 1999). Verification techniques of applications for embedded systems based on formal models have been well researched over the last two decades (Edwards et al., 1997; Lee and Parks, 1995). Out of several models proposed, Petri net based models, specially the PRES+ model has been found to be highly suitable for modeling concurrent behavior, simple computation over basic data types (integer, real), modeling general data structure and modeling timing behavior of a parallel application. This model allows tokens to carry information (Cortés et al., 2000) and it has a well-defined semantics for precise representation of systems.

A major limitation of these methods is that they can verify only structure preserving transformations and invariably fails for schedulers that alter the control structure of a program (Camposano, 1991). To alleviate this shortcoming, a path based equivalence checker for the FSMD models (which are essentially sequential control and data flow graphs (CDFGs)) was proposed in (Karfa et al., 2012) which was later modified to handle more sophisticated uniform, non-uniform code motions and code motions across loops

[1]As a case in point, consider an unpredictable bug in gcc v4.9.2 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=64490



Bandyopadhyay, S., Sarkar, S. and Banerjee, K.
An End-to-end Formal Verifier for Parallel Programs.
DOI: 10.5220/0006464503880393
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 388-393
ISBN: 978-989-758-262-2

in (Banerjee et al., 2014). They, however, cannot handle thread-level parallelizing transformations mainly because FSMD, being a sequential mode of computation, cannot capture a parallel behaviour straightway.

The work described in (Bandyopadhyay et al., 2012) proposed a translation algorithm from a PRES+ model to an FSMD model and then used the existing FSMD equivalence checker of (Karfa et al., 2012) to establish an equivalence between the initial and the optimized versions of a program, modeled using the PRES+ formalism. However, in this method, the model construction from the original programs was a manual process. The authors of (Voron and Kordon, 2008) reported a method for automated construction of Petri net models from a high-level language program where the source program is converted into an intermediate representation form such as, abstract syntax tree, for various modules. However, in their method, only control structure is captured, and the data flow analysis is not performed. We present a technique for automated construction of value based PRES+ models (Cortés et al., 2000) from parallel programs capturing both control and data flow. Subsequently the PRES+ model is used for formal verification of two programs. The major contributions in this paper are as follows:

1. The proposed approach captures maximal block level and instruction level parallelism during PRES+ model construction.

2. Our verifier is generic and portable as the underlying model is generic. For instance, we have demonstrated that it is possible to integrate an FSMD based equivalence checker(Banerjee et al., 2014) to accept our PRES+ model for program validation. For this purpose, we have used a PRES+ to FSMD translator (Bandyopadhyay et al., 2012). One can seamlessly integrate a PRES+ based checker with our model instead of an FSMD based one to build yet another validator.

The experimental results demonstrate the efficiency of the tool.

Rest of the paper has been organized as follows. The automated construction of PRES+ model from high-level language programs is mentioned in section 2. The results obtained when the procedure was tested on some examples can be found in section 3. The paper is finally concluded in section 4.

# 2 AUTOMATED PRES+ CONSTRUCTION METHOD

We demonstrate our automated model construction method in the following subsection. The functional modules are depicted in Algorithm 2 – Algorithm 5 with Algorithm 1 being the top level module.

## 2.1 A Brief Example

Figure 1 (a) depicts a simple parallel program which can be easily converted to a 3 address code along with the basic block information as shown in Figure 1(b) using tools like flex and bison. The function `creatPRES` (Algorithm 1) checks the properties of each basic block. For the basic block *bb*2, the function `creatPRES` calls the function `subNetForAssignMentBB` (Algorithm 2). The function `subNetForAssignMentBB` constructs a data dependency graph (DDG) for *bb*2. Then, the function performs the reaching definition analysis on the *DDG* that results in sets of instruction-level parallel statements in *bb*2. For each member in the set of parallel statements, the method creates places for each operands, i.e., $\{p_1, p_2\}$ shown in Figure 2. Next, the function constructs the transitions and out-places. Now, the transitions and out-places of the sub-net of the PRES+ model are $\{t_1,t_2\}$ and $\{p_3, p_4\}$, respectively. Then the function `subNetForAssignMentBB` (Algorithm 2) identifies that there are two basic block information associated with `goto` statement, i.e., *bb*4 and *bb*6. The function `subNetForAssignMentBB` identifies the basic blocks *bb*4 and *bb*6 as parallel blocks. Then the control goes to the caller function, `creatPRES` (Algorithm 1). which checks that set of parallel blocks is non empty. Hence, it calls the function `subnetForParallelBB` (Algorithm 5). In this example, there are also two parallel basic blocks, *bb*4 and *bb*6 respectively. For *bb*4, the function `subnetForParallelBB` identifies that *bb*4 is a condition containing block. Therefore, it calls the function `subnetForCondBB` (Algorithm 3). The function `subnetForCondBB` identifies the conditional statement in three address code and the operator used in the condition. For each operands of the condition, two mutually exclusive transitions are created having one pre-place. Then for each transition one post is created. Then it identifies that the basic block information associated with `goto` statement, e.g., *bb*3 whose id is less than the id of currently processed basic block i.e., *bb*4. Hence *bb*3 is inferred as loop containing basic block. For the block *bb*3, the function `subnetForCondBB` (Algorithm 3) calls the function `subnetForLoopBB` which in turn calls the function `subNetForAssignMentBB` (Algorithm 2). Then for each element in *bb*3, the function constructs a loop variable sets. For each of these member in the set, the function computes the used-defined variable pairs. In this example, the variable associated with

```
int i=1,j=1,k;
#pragma scop
while (i<=10)
    i++;
while (j<=10)
    j++;
#pragma scop
 k=i+j;
return k;
```

(a)

```
int k,j,i;
<bb2>:  i = 1;j = 1;
goto <bb3>;
goto <bb5>;
<bb3>:  i = i + 1;
<bb4>:
  if (i <= 9) goto <bb3>;
  else       goto <bb7> ;
<bb5>:   j = j + 1;
<bb6>:
  if (j <= 9)  goto <bb5>;
  else         goto <bb7>;
<bb7>:  k = i + j;
```

(b)

Figure 1: A simple parallel program and the corresponding basic blocks.

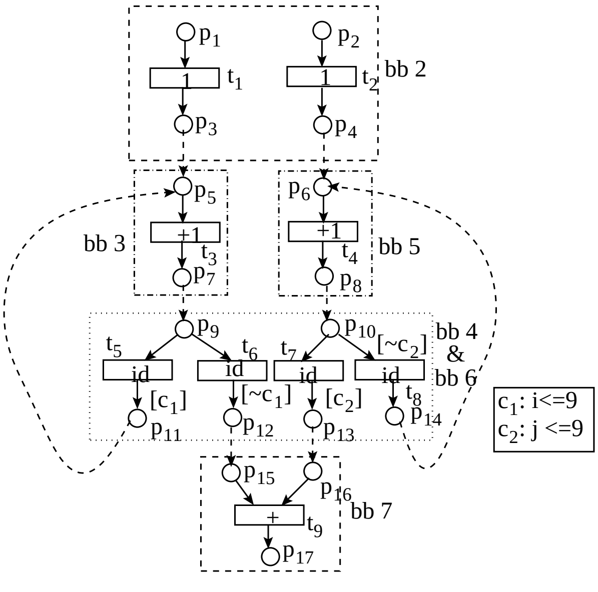
the place $p_5$ corresponds to the used-defined pair; therefore, the place $p_5$ contains a back edge. Next the function `subnetForParallelBB` (Algorithm 5) processes the basic block *bb*6 in identical manner. Then the control goes to the caller function. For the block *bb*7, the caller function calls the function `subNetForAssignMentBB` (Algorithm 2) and then it constructs the corresponding subnet. Finally, all the PRES+ subnets are attached according to the updated symbol table information. If the same variable is associated with two different places, those two places are then merged into a single place. In Figure 2, the place $p_3$ is merged with $p_5$ as the variable *i* is associated with both the place $p_3$ and $p_5$. Symbolically, it is represented as $p_3 \hookleftarrow p_5$. In Figure 2, the dotted arrow between the place to place indicates the merging operation. The graphical representation is depicted in Figure 2.

## 3 EXPERIMENTAL RESULTS

The tool has been tested on parallel examples on a 2.0 GHz Intel(R) Core(TM)2 Duo CPU machine (using only a single core). We have carried out the experiments on a set of parallel examples in a systemic manner. Here, we have transformed five sequential programs into parallel programs using a prominent thread-level parallelizing compiler PLuTo (Bondhugula et al., 2008). The experimental set up is as follows:

1. **Preparation of the example suite:** We have taken five sequential source programs. The list of the source programs and their functionality are as follows:

   *a) BCM :* A toy example on basic code motion without writable shared variables which illustrates computational vs. executional optimality.




(c)

Figure 2: Constructed PRES+ model.

**Algorithm 1:** PRES+ **creatPRES** $(BB, PB)$.

**Inputs:** Set of basic blocks along with informations and parallel block.
**Outputs:** PRES+ model $N$.

> $N = \emptyset$; $PB = \emptyset$;
> **for** each basic block $b$ in $BB$ **do**
>   **if** ($b$ is normal assignment block) **then**
>     *subNetForAssignMentBB(b, N, PB)*;
>   **else**
>     **if** ($b$ is loop containing ) **then**
>       *subnetForLoopBB(b, N, PB)*;
>     **else**
>       **if** ($PB \neq \emptyset$ ) **then**
>         *subnetForParallelBB(b, N, PB)*;
>       **else**
>         **if** ($b$ is conditional basic block) **then**
>           *subnetForCondBB(b, N, PB)*;
>         **end if**
>       **end if**
>     **end if**
>   **end if**
> **end for**
> Merge all the subnet according to the symbol table information.
> **return** $N$;

   *b) MINANDMAX-P:* Computes sum of the maximum of four numbers $n_1, n_2$, $n_3, n_4$ and the minimum of the four numbers $n_1, n_5$, $n_6$ and $n_7$ (having $n_1$ as the common element).

   *c) LUP:* It computes "LU Decomposition with Pivoting". In this experimentation, we have only taken the pivoting routine which does not contain

**Algorithm 2:** STRUCT2TUPLE **subNetForAssignMentBB** ($b$, $N$, $PB$).

**Inputs:** A basic block, a PRES+ model $N$, set of parallel blocks

**Outputs:** Two tuple structures. The elements of this structure are as follows: 1.sub-net of the PRES+ model and 2. parallel block list

$G = \emptyset$;<br>
$G = G \cup$ *creatDDG* ($b$);<br>
/*Construction is carried out by GauTe Tool*/<br>
$L =$ *reachingDefinitionAnalysis* ($G$);<br>
/* The function returns a set of lists. Each list contains set of statements. Every statement in a list is independent to the other statements present in that list. This analysis is carried out by NuSMV*/<br>
**for** each list $l$ in $L$ **do**<br>
    $P = \emptyset$;<br>
    **for** each element $e$ in $l$ **do**<br>
        $P = P \cup \{p\}$;<br>
        /* The function takes an element and creates places for every used variable of that element */<br>
        $T = T \cup \{t\}$;<br>
        **for** each $t$ in $T$ **do**<br>
            /* Construct normalize expression and guard condition using SMT solver*/<br>
        **end for**<br>
        $P_{out} = P_{out} \cup \{p_{out}\}$;<br>
        /*The function creates an output place for the transition $T$ and update the symbol table for places and transitions */<br>
        Attach $p, t$ and $p_{out}$<br>
    **end for**<br>
**end for**<br>
**if** number of block associated with goto $>1$ **then**<br>
    The blocks along with goto statement are put in to $PB_{new}$;<br>
    $PB = PB \cup PB_{new}$; //update the parallel block lists<br>
**end if**<br>
Update $N$<br>
**return** $\langle N, PB \rangle$;

any array. The detailed functionality of this source program is given in PLuTo example suite (Bondhugula et al., 2008).

*d) DEKKER's* and *PATTERSON's algorithms:* Implementations of the classical solutions to the mutual exclusion problem of two concurrent processes. Since our mechanism does not handle writable shared variables among parallel threads, we have considered a single process in each of these cases; also we have introduced a series of dummy assignment statements within the critical section.

2. **Transforming the programs:** The above five sequential programs are transformed by a prominent thread level parallelizing compiler, PLuTo (Bondhugula et al., 2008); the transformed versions accordingly have parallel structures. Table 1 depicts the type of transformations applied for each of the above examples. It is to be noted that for testing our tool, (in the context of parallelizing transformations) we have five sequential programs and the five parallel programs are obtained using PLuTo compiler.

**Algorithm 3:** STRUCT2TUPLE **subnetForCondBB** ($b$, $N$, $PB$).

**Inputs:** A basic block, a PRES+ model, a set of parallel blocks

**Outputs:** Two tuple structures. The elements of this structure are as follows: 1.sub-net of the PRES+ model and 2. parallel block list

*cond* = *getCond* ($b$);<br>
/* The function gets condition of execution of the block $b$. The condition is easily obtainable from 3 address code using SMT solver */<br>
*expr* = *getExpr* ($b$);<br>
/* The function returns operator used in the condition */<br>
$P = \emptyset$;<br>
$P = P \cup \{p\}$;<br>
/* The function determines input places for condition */<br>
$T_1 = T_1 \cup \{t_1\}$;<br>
$P_{out_1} = P_{out_1} \cup \{p_{out_1}\}$;<br>
Attach $p, t_1, p_{out_1}$;<br>
$T_1 = T_2 \cup \{t_2\}$<br>
$P_{out_2} = P_{out_2} \cup \{p_{out_2}\}$;<br>
Attach $p, t_2, p_{out_2}$;<br>
$P_{out} = P_{out_1} \cup P_{out_2}$;<br>
/* The function accumulates all the output places */<br>
**if** current process block Id $>$ block Id associated with goto **then**<br>
    *subnetForLoopBB*($b, N, PB$);<br>
**end if**<br>
**if** number of block associated with goto $>1$ **then**<br>
    The blocks along with goto statement are put in to $PB_{new}$;<br>
    $PB = PB \cup PB_{new}$; //update the parallel block lists<br>
**end if**<br>
Update $N$<br>
**return** $\langle N, PB \rangle$;

3. The automated model constructor constructs two PRES+ models – one from the original code and the other from the transformed program. The two PRES+ models are then translated into corresponding FSMD models using (Bandyopadhyay et al., 2012) and finally equivalence checking is carried out using the FSMD equivalence checker of (Banerjee et al., 2014). It is to be noted that all the above parallel examples do not contain any writable shared variable.

**Algorithm 4:** STRUCT2TUPLE **subnetForLoopBB** (*b*, *N*, *PB*).

**Inputs:** A basic block, a PRES+ model *N*, set of parallel blocks

**Outputs:** Two tuple structures. The elements of this structure are as follows: 1.sub-net of the PRES+ model and 2. parallel block list

```
subNetForAssignMentBB(b,N,PB);
L = ∅
for each element e in b do
    L = L ∪ getVariables(e);
end for
for each variable v in L do
    P1 = findLastPlace(v);
    /* Find the last place of the variables updated in the loop */
    P2 = findReturnPlace(v);
    /* Values need to be sent back through back edges in case of loop. It finds the place to which values are sent back. */
    P1 = P2
end for
if number of block associated with goto >1 then
    The blocks along with goto statement are put in to PBnew;
    PB = PB ∪ PBnew; //update the parallel block lists
end if
Update N
return ⟨N,PB⟩;
```

Table 1: Transformations carried out using parallelizing compilers.

| Example | Transformations |
|---|---|
| BCM | Boosting up |
| MINANDMAX-P | Thread level parallelization |
| LUP | Thread level parallelization |
| DEKKER | Thread level parallelization |
| PATTERSON | Thread level parallelization |

## Analysis

Table 2 depicts the size of PRES+ models in terms of number of places and transitions, PRES+ model construction time from both original and transformed programs and PRES+ to FSMD translation times for both original and transformed PRES+ models. In this experimentation, we have also carried out a comparative study between FSMD and PRES+ equivalence checking methods. The last two columns indicate the FSMD equivalence checking and direct PRES+ equivalence checking times (Bandyopadhyay et al., 2015a; Bandyopadhyay et al., 2015b; Bandyopadhyay et al., 2016a), respectively. It is to be noted that FSMD equivalence checking includes PRES+ to FSMD translation time, path construction time and the equivalence checking time. On the other hand, direct PRES+ equivalence checking includes path construction time and equivalence checking time. By comparing the numbers in the columns `FSMD Eqiv` and `PRES+ Eqiv`, we notice that FSMD equivalence checking time is faster than the PRES+ equivalence checking time because the path construction method of the PRES+ model is complicated compared to the FSMD model (Bandyopadhyay et al., 2016b). It is also to be noted that earlier FSMD equivalence checker reported in (Banerjee et al., 2014) is not capable of validating thread-level parallelizing transformation. However, FSMD equivalence checking module which is integrated within our automated model constructor is capable of handling those parallelizing transformations. As PRES+ to FSMD translation module uses both symbolic execution as well as serialization technique, FSMD captures parallelism using serialized form.

**Algorithm 5:** STRUCT2TUPLE **subnetForParallelBB** (*b*, *N*, *PB*).

**Inputs:** A basic block, a PRES+ model *N*, set of parallel blocks

**Outputs:** Two tuple structures. The elements of this structure are as follows: 1.sub-net of the PRES+ model and 2. parallel block list

```
for each b in PB do
    if (b is normal assignment block). then
        subNetForAssignMentBB(b, N,PB);
    else
        if (b is loop containing ) then
            subnetForLoopBB(b, N,PB);
        else
            if (b is conditional basic block) then
                subnetForCondBB(b, N, PB);
            end if
        end if
    end if
end for
return ⟨N,PB⟩;
```

## 4 CONCLUSION

In this work, an automated model construction method is presented for obtaining PRES+ models from high-level languages. Our tool when integrated with the PRES+ to FSMD translator of (Bandyopadhyay et al., 2012) and the FSMD equivalence checker of (Banerjee et al., 2014) provides an end-to-end fully automated verifier for optimizing and parallelizing transformations. An overview of this entire tool is provided here. Through experiments over a set of parallel examples, the efficacy of the verifier, and es-

Table 2: Experimentation using parallel examples.

| Example | Original | | | | | Transformed | | | | | Time ($\mu$s) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lines | Place | Trans | Time ($\mu$s) | | Lines | Place | Trans | Time ($\mu$s) | | FSMD Eqiv | PRES+ Eqiv |
| | | | | Model Const. | Translation | | | | Model Const. | Translation | | |
| BCM | 11 | 34 | 21 | 650 | 234 | 11 | 32 | 20 | 628 | 232 | 3001 | 8345 |
| MINANDMAX-P | 20 | 42 | 20 | 983 | 312 | 20 | 41 | 20 | 982 | 312 | 3421 | 8451 |
| LUP | 32 | 80 | 73 | 1025 | 775 | 32 | 80 | 71 | 1028 | 771 | 5152 | 10678 |
| DEKKERS | 29 | 53 | 45 | 994 | 702 | 29 | 53 | 41 | 979 | 623 | 4352 | 10414 |
| PETTERSON | 29 | 53 | 45 | 992 | 710 | 29 | 53 | 41 | 989 | 673 | 4758 | 10612 |

pecially that of the automatic PRES+ constructor, is demonstrated. Some of the possible future extensions of our work are as follows. Optimization of the constructed PRES+ model is a short-term goal. Moreover, PRES+ models permit specifications of timing behaviours too; enhancing the present tool for timing analyses seems promising as well.

## ACKNOWLEDGMENTS

Santonu Sarkar has been partially supported by Science and Engineering Research Board, Govt. of India project (SB/S3/EECE/0170/2014) for this research work.

## REFERENCES


Bandyopadhyay, S., Banerjee, K., Sarkar, D., and Mandal, C. A. (2012). Translation validation for pres+ models of parallel behaviours via an fsmd equivalence checker. In *VDAT*, pages 69–78.

Bandyopadhyay, S., Sarkar, D., Banerjee, K., and Mandal, C. A. (2015a). A path-based equivalence checking method for petri net based models of programs. In *ICSOFT-EA*, pages 319–329.

Bandyopadhyay, S., Sarkar, D., and Mandal, C. (2015b). An efficient equivalence checking method for petri net based models of programs. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015 (Poster), Florence, Italy, May 16-24, 2015, Volume 2*, pages 827–828.

Bandyopadhyay, S., Sarkar, D., and Mandal, C. (2016a). An efficient path based equivalence checking for petri net based models of programs. In *Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016*, pages 70–79.

Bandyopadhyay, S., Sarkar, D., Mandal, C. A., Banerjee, K., and Duddu, K. R. (2016b). A path construction algorithm for translation validation using PRES+ models. *Parallel Processing Letters*, 26(2):1–25.

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

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008). PLuTo: A practical and fully automatic polyhedral program optimization system. In *PLDI 08*.

Camposano, R. (1991). Path-based scheduling for synthesis. *IEEE transactions on computer-Aided Design of Integrated Circuits and Systems*, Vol 10 No 1:85–93.

Cortés, L. A., Eles, P., and Peng, Z. (2000). Verification of embedded systems using a petri net based representation. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 149–155, Washington, DC, USA. IEEE Computer Society.

Edwards, S., Lavagno, L., Lee, E. A., and Sangiovanni-Vincentelli, A. (1997). Design of embedded systems: Formal models, validation, and synthesis. In *Proceedings of the IEEE*, pages 366–390.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11.

Karfa, C., Mandal, C. A., and Sarkar, D. (2012). Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM Trans. Design Autom. Electr. Syst.*, 17(3):30.

Kundu, S., Lerner, S., and Gupta, R. (2008). Validating high-level synthesis. In *Computer Aided Verification*, CAV '08, pages 459–472.

Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799.

Marwedel, P. (2006). *Embedded System Design*. Springer(India) Private Limited, New Delhi, India.

Necula, G. C. (2000). Translation validation for an optimizing compiler. In *PLDI*, pages 83–94.

Pnueli, A., Siegel, M., and Singerman, E. (1998). Translation validation. In *TACAS*, pages 151–166.

Raghavan, V. (2010). *Principles of Compiler Design*. Tata McGraw Hill Education Private Limited, New Delhi.

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

Smith, M. D., Horowitz, M., and Lam, M. S. (1992). Efficient superscalar performance through boosting. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 248–259, New York, NY, USA. ACM Press.

Voron, J.-B. and Kordon, F. (2008). Transforming sources to Petri nets: A way to analyze execution of parallel programs. In *SimuTools*, page 13.