An End-to-end Formal Verifier for Parallel Programs

Soumyadip Bandyopadhyay¹, Santonu Sarkar¹ and Kunal Banerjee²

¹BITS Pilani, K K Birla Goa Campus, Goa, India
²Indian 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 bugs¹. 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.

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

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 Gordon, 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 place 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 ++;
#pragma ...
while (j <=10)
    j ++;
#pragma ...
return i + j;

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 $bb6$ in identical manner. Then the control goes to the caller function. For the block $bb7$, 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 $p_3 \Leftarrow p_5$. 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.

   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 $n_1, n_5, n_6, n_7$ (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. subnet of the PRES+ model and 2. parallel block list

1: \( G = \emptyset; \)
2: \( G = G \cup \text{creatDDG}(b); \)
   /* Construction is carried out by GauTe Tool */
3: \( L = \text{reachingDefinitionAnalysis}(G); \)
   /* 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 */
4: for each list \( l \) in \( L \) do
5: \( P = \emptyset; \)
6: for each element \( e \) in \( l \) do
7: \( P = P \cup \{ p \}; \)
   /* The function takes an element and creates places for every used variable of that element */
8: \( T = T \cup \{ t \}; \)
9: for each \( t \) in \( T \) do
10: /* Construct normalize expression and guard condition using SMT solver */
11: end for
12: \( P_{out} = P_{out} \cup \{ p_{out} \}; \)
   /* The function creates an output place for the transition \( T \) and update the symbol table for places and transitions */
13: \( \text{Attach } p_{out} \) and \( p_{out} \)
14: end for
15: end for
16: if number of block associated with goto > 1 then
17: The blocks along with goto statement are put in to \( \text{PB}_{new}; \)
18: \( \text{PB} = \text{PB} \cup \text{PB}_{new}; // update the parallel block lists \)
19: end if
20: Update \( N \)
21: return \( (N, PB); \)

Algorithm 3: STRUCT2TUPLE subnetForConditionBB (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. subnet of the PRES+ model and 2. parallel block list

1: \( \text{cond} = \text{getCond}(b); \)
   /* The function gets condition of execution of the block \( b \). The condition is easily obtainable from 3 address code using SMT solver */
2: \( \text{expr} = \text{getExpr}(b); \)
   /* The function returns operator used in the condition */
3: \( P = \emptyset; \)
4: \( P = P \cup \{ p \}; \)
   /* The function determines input places for condition */
5: \( T_1 = T_1 \cup \{ t_1 \}; \)
6: \( P_{out1} = P_{out1} \cup \{ p_{out1} \}; \)
7: Attach \( p_{t1} \) and \( p_{out1} \)
8: \( T_2 = T_2 \cup \{ t_2 \}; \)
9: \( P_{out2} = P_{out2} \cup \{ p_{out2} \}; \)
10: Attach \( p_{t2} \) and \( p_{out2} \)
11: \( P_{out} = P_{out} \cup P_{out2}; \)
   /* The function accumulates all the output places */
12: if current process block Id > block Id associated with goto then
13: \( \text{subnetForLoopBB}(b, N, PB); \)
14: end if
15: if number of block associated with goto > 1 then
16: The blocks along with goto statement are put in to \( \text{PB}_{new}; \)
17: \( \text{PB} = \text{PB} \cup \text{PB}_{new}; // update the parallel block lists \)
18: end if
19: Update \( N \)
20: return \( (N, PB); \)

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.

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.

1: subNetForAssignMentBB(b, N, PB);
2: L = ∅
3: for each element e in b do
4: \( P_1 = \text{getVariables}(e) \);
5: end for
6: for each variable v in L do
7: \( P_2 = \text{findLastPlace}(v) \);
8: \( P_2 = \text{findReturnPlace}(v) \);
end for
9: \( P_3 = P_2 \);
10: end for
11: if number of block associated with goto > 1 then
12: The blocks along with goto statement are put in to \( PB_{new} \);
13: \( PB = PB \cup PB_{new} \); //update the parallel block lists
14: end if
15: Update N
16: return \( (N, PB) \);

Table 1: Transformations carried out using parallelizing compilers.

<table>
<thead>
<tr>
<th>Example</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCM</td>
<td>Boosting up</td>
</tr>
<tr>
<td>MINANDMAX-P</td>
<td>Thread level parallelization</td>
</tr>
<tr>
<td>LUP</td>
<td>Thread level parallelization</td>
</tr>
<tr>
<td>DEKKER</td>
<td>Thread level parallelization</td>
</tr>
<tr>
<td>PATTERSON</td>
<td>Thread level parallelization</td>
</tr>
</tbody>
</table>

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.

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-
especially 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**


