Author(s):

KOH, K[wang] Y[oung]; JEE, E[un] K[young]; JEON, S[eungjae];
SEONG, P[oong] H[yun] & CHA, S[ungdeok]

This Publication has to be referred as:
A FORMAL VERIFICATION METHOD OF FUNCTION BLOCK DIAGRAMS WITH TOOL SUPPORTING: PRACTICAL EXPERIENCES

KOH, K[wangyong]; JEE, E[junkyoung]; JEON, S[jeungjae]; SEONG, P[oong] H[yun] & CHA, S[ungdeok]

Abstract: Function Block Diagram (FBD) is a standard application programming language for the Programmable Logic Controller (PLC) and currently being used in the development of a fully-digitalized Reactor Protection System (RPS) under the Korea Nuclear Instrumentation & Control System (KNICS) project. Therefore, rigorous verification of the FBD programs is indispensable. In this paper, we propose a formal verification technique of FBD programs used in KNICS, RPS and its supporting tool, FBDVerifier, which performs model checking and gives us counterexamples displayed in the form of timing graph to enhance their readability. Several important errors of FBD programs were found and reflected to the next version. Large scale case study showed that the proposed method is effective and practical.

Key words: Function Block Diagram (FBD), Programmable Logic Controller (PLC), Formal Verification, Model Checking

1. INTRODUCTION

A fully-digitalized Reactor Protection System (RPS), which is called IDIPS (Integrated Digital Protection System), is being developed under Korea Nuclear Instrument and Control System (KNICS) project in order to be used in newly constructed nuclear power plant and also in the upgrade of existing analog based RPSs. Programmable Logic Controller (PLC) and its representative programming language Function Block Diagram (FBD) are used to control many kinds of functional processes of IDIPS. Because the software embedded in a RPS is very crucial to the safety of a nuclear power plant, rigorous safety demonstration of FBD programs is indispensable.

Several different approaches to formalize PLC programs have been presented in (Bani Younis & Frey, 2003), but a few works have been performed for FBD programs. In (Baresi et al., 2000) a toolset called PLCTOOLS has been introduced. The FBD programs are modelled and described as High Level Timed Petri Nets (HLTPN) which is used for validating the design and generating the code. PLCTOOLS supports the design validation and code generation processes, but it does not support formal verification such as model checking, a major function of FBDVerifier. In (Vyatkin & Hanisch, 2000) the controller code in FBD format and the overall system are translated to Signal-Net-Systems (SNS) (Starke, 2000). On the combined model of plant and controller modelled by SNS, model checking is performed using Signal/Event System Analyzer (SESA) which is a model checker for Signal-Net models (Starke & Rosh, 2000). This approach, however, follows IEC 61449 while our approach follows IEC 61131, and it is therefore inapplicable to our FBD programs.

In this paper, we propose a formal verification technique of FBD programs; FBD programs are defined formally in compliance with IEC 61131-3, and then the programs are automatically translated into Verilog model by the supporting tool, FBDVerify, which is developed to support the FBD verification framework, and finally the model is verified using Cadence SMV, and counterexamples are displayed in the form of timing graph to enhance their readability. This approach is described in Fig. 1.

Fig. 1. FBD verification framework supported by FBDVerifier

2. FORMAL VERIFICATION OF FBDs

Verilog (IEEE, 2003) is one of the most popular Hardware Description Languages (HDL) used by integrated circuit designers. In order to verify FBD programs, we chose Verilog as a verification language because the semantics of FBD is similar to those of Verilog so that FBD can be translated into Verilog efficiently. We used model checking technique to verify FBD programs and chose Cadence SMV (McMillan, 2001) as a model checker to verify Verilog models generated from FBD programs.

2.1 Translation Rules

Rule 1. Module declaration: A module is the principal design entry in Verilog. The first line of a module declaration specifies the name of the module and list of input/output ports.

Rule 2. Variable type decision: All variables are declared with their type, bit size, and name in Rule 2. Each variable in the FBD is mapped to one of the Verilog variable types; input, reg, wire and output. A variable \( v_i \) \( \text{has input type if its value is transmitted from external input. A variable } v_i \) \( \text{has reg type if its value needs to be stored internally. The reg variables hold their values which will be used at the next scan cycle operation. Variables whose values do not need to be stored internally are declared as wire variables. A variable } v_i \) \( \text{has output type if it is designated as an external output of the module.}

Rule 3. Initialization of reg variables: The reg variables are initialized in Rule 3. Verilog operator \( \text{<=>} \) means the assignment to l-value. Usually initial values of reg variables are specified in the FBD program. If not, they are determined by a user or have default values.

Rule 4. Output assignment for each wire and output variable: In Rule 4, the target FBD represented by a set of connected function blocks is translated into assignment statements from top to bottom in accordance with the execution order of the FBD. While a function which does not have internal state is mapped into a Verilog operator, a function block which stores internal state is mapped into a Verilog module.
Rule 5. Stored value assignment for reg variables: The stored values are assigned to the reg variables in Rule 5. @ (posedge clk) means the positive edge of the clock signal, i.e., the beginnings of each cycle. As the updated value of a reg variable becomes visible at the next time unit, new value is read at next cycle.

Rule 6. Insertion of properties: The template always begin - end is generated automatically and properties are embedded by a user after automatic generation of a Verilog model.

2.2 FBDVerifier

FBDVerifier automates the FBD verification framework described in Fig. 1. It takes LDA files, which is FBD storing format of a tool pSET (POSCON R&D Center, 2007) as inputs, and then converts the FBD programs into Verilog model. User can adjust bit sizes and initial values of variables during the translation. After Verilog translation is completed, a user inserts properties to be verified into the Verilog model and executes Cadence SMV with one click in FBDVerifier. When a target FBD has a lot of variables, a generated counterexample is often lengthy and complex. It is very hard to analyze a counterexample only with the table-style trace view provided by Cadence SMV. In many cases of our case study, we had to examine several hundreds rows of variables and many columns of steps in order to analyze a counterexample. To enhance readability of counterexample, timing graph form is displayed. Variables are highlighted in different color and shape for better visualization. With this function, counterexample analysis time was significantly reduced.

3. CASE STUDY

We applied the proposed method to Bistable Processor (BP), Coincidence Processor (CP) and Automatic Test and Interface Processor (ATIP) which are safety-critical subsystems of IDiPS and their information is described in Table 1. The software design specification document for the RPS has approximately 700 pages and the FBD programs for the RPS are composed of approximately twenty thousands function blocks and nine thousands variables. The Verilog model generated from the FBD for the RPS consists of more than fourteen thousands of lines totally.

We divided found errors into 5 categories according to their causes. Incorrect logic errors are the most serious errors caused by incorrect logic. Omission means errors caused by omission of specification. Ambiguous logic errors are possible errors which can be serious or not according to the environment constraints. Incorrect FBD program represents the cases that FBD specification is wrong while SDS is correct. Incorrect SDS errors are the opposite cases. The verification results are summarized in Table 2.

<table>
<thead>
<tr>
<th>Subsystems</th>
<th>Page # of natural lang. spec.</th>
<th># of function blocks</th>
<th># of variables</th>
<th>Line # of Verilog model</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>190</td>
<td>1,335</td>
<td>1,038</td>
<td>7,862</td>
</tr>
<tr>
<td>CP</td>
<td>163</td>
<td>1,623</td>
<td>820</td>
<td>3,085</td>
</tr>
<tr>
<td>ATIP</td>
<td>365</td>
<td>18,359</td>
<td>7,024</td>
<td>3,401</td>
</tr>
</tbody>
</table>

Tab. 1. RPS subsystem information

4. CONCLUSION AND FUTHER STUDY

We proposed a formal verification technique for FBD which is a commonly used PLC programming language. We suggested the translation rules from an FBD program into a Verilog model and developed the FBDVerifier tool to automate the FBD verification framework. We performed model checking for the Verilog models generated from FBD programs successfully. It is almost impossible to verify FBD through manual translation due to huge amount of work and time. Automatic translation function of FBDVerifier made it possible for us to verify huge size of real case FBD programs, and its counterexample visualization function reduced the time and effort of analyzing counterexamples significantly. By using FBDVerifier, we could accomplish FBD verification and counterexample analysis for the whole RPS within 3 man-months which is relatively short time.

But, we used manual abstraction techniques to make the verification feasible against state explosion problems and current FBDVerifier receives only LDA file format used in pSET. It is therefore necessary to construct systematic abstraction method instead of manual abstraction and to extend FBDVerifier to be able to deal with various FBD storing formats.

5. REFERENCES


IEEE Standard Hardware Description Language Based on the Verilog hardware Description Language. IEEE, 2003


