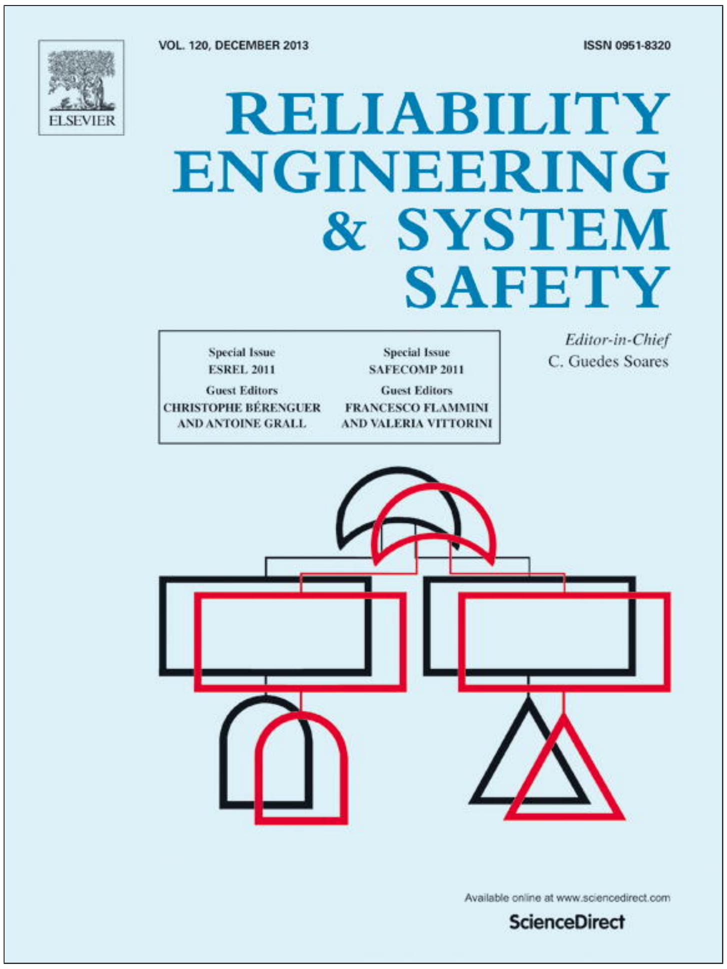

VOL. 120, DECEMBER 2013
ISSN 0951-8320
ELSEVIER
RELIABILITY ENGINEERING & SYSTEM SAFETY
Special Issue
ESREL 2011
Guest Editors
CHRISTOPHE BÉRENGUER AND ANTOINE GRALL
Special Issue
SAFECOMP 2011
Guest Editors
FRANCESCO FLAMMINI AND VALERIA VITTORINI
Editor-in-Chief
C. Guedes Soares
Available online at www.sciencedirect.com
ScienceDirect

Reliability Engineering and System Safety 120 (2013) 139–149

Contents lists available at ScienceDirect

Reliability Engineering and System Safety

journal homepage: www.elsevier.com/locate/ress

A systematic verification of behavioral consistency between FBD design and ANSI-C implementation using HW-CBMC

Dong-Ah Lee [a], Junbeom Yoo [a,*], Jang-Soo Lee [b]

[a] Division of Computer Science and Engineering, Konkuk University, Republic of Korea
[b] Man-Machine Interface System Team, Korea Atomic Energy Research Institute, Republic of Korea

ARTICLE INFO

Available online 2 July 2013

Keywords:
Verification process
Behavioral consistency
Model checking
FBD
ANSI-C
HW-CBMC

ABSTRACT

Controllers in safety critical systems such as nuclear power plants often use the Function Block Diagram (FBD) to design software embedded in the PLC (Programmable Logic Controller). Software engineers develop FBD programs manually, while engineering tools provided by PLC vendors translate them into ANSI-C programs mechanically. Every new PLC and its software engineering tool should demonstrate the so-called *FBD-to-C* translator's correctness thoroughly. This paper proposes a verification process which can efficiently verify the translator's correctness using the model checking technique. The HW-CBMC model checker verifies the behavioral consistency between FBD and ANSI-C programs formally according to the process and templates which this paper proposes. We also developed a CASE tool '*CWrapper*' and performed a case study with simplified examples of the APR-1400 (Advanced Power Reactor-1400) nuclear reactor protection system in Korea.

1. Introduction

Safety is an important property for real-time embedded systems [1] such as nuclear power plants to obtain permissions for operation and export from government authorities. As the nuclear reactor protection system (RPS) makes decisions for emergent reactor shutdown, RPS software should be verified throughout its entire development life-cycle. RPS software is typically modeled with IEC-61131 FBD (Function Block Diagram) [2] in the design phase, and then in the implementation phase, translated into ANSI-C programs and compiled into an executable machine code for RPS hardware—PLC (programmable Logic Controller). Compiler expert companies typically provide C compilers with a thorough demonstration of functional correctness. On the other hand, PLC vendors usually develop translators which perform FBDs into C programs by themselves. They should demonstrate the translator's correctness and functional safety [3] sufficiently.

In the PLC industry for RPS, vendors such as *AREVA* [4], *invensys* [5] and *POSCO ICT* [6] have provided safety-level PLCs and their own software engineering tool-sets. '*SPACE*' [7] is a software engineering tool-set for *AREVA*'s PLC '*TELEPERM XS*' [8]. It stores FBD programs into a database '*INGRES*' and generates ANSI-C programs to perform code-based testing and simulation ('*TXS SIVAT*' [9]). *ISTec GmbH* [10] also has developed a reverse engineering tool '*RETRANS*' [11] for checking consistency between FBD programs and generated C programs. The mechanical translator in '*SPACE*' has been validated in such ways, and the software engineering tool-sets have been used successfully for more than a decade. PLCs of *invensys* also have been widely used. '*TriStation 1131*' [12] is its software engineering tool-set. It provides enhanced emulation-based testing and real-time simulation of FBDs, but does not include a C translator yet.

KNICS (Korea Nuclear Instrumentation and Control System R&D Center) [13] and *POSCO ICT* in Korea have recently developed a safety-level PLC '*POSAFE-Q*' and its software engineering tool-set '*pSET*' [14]. The tool-set provides a graphical editor for FBD and LD (Ladder Diagram) programming languages [2], and also generates ANSI-C programs automatically. However, sufficient demonstration of correctness and functional safety of the so-called '*FBD-to-C*' translator is still in progress. Thus, it must be one of the most critical obstacles needed to pass inspection in order to obtain permissions for the export of the new Korean nuclear power plant [15] as a whole, *i.e.*, including control software—I&C (Instrumentation & Control).

This paper proposes a systematic way to demonstrate functional correctness of the '*FBD-to-C*' translator using the model checking techniques [16]. We use the '*HW-CBMC*' [17] model checker which can verify the behavioral equivalence between FBD and ANSI-C programs. We first translate a FBD program into a behaviorally equivalent Verilog program based on translation rules in [18]. We modify the rules to translate it into a suitable Verilog program for HW-CBMC, because the Verilog program as an input of HW-CBMC is different from that of the VIS verification

* Corresponding author. Tel.: +82 2 450 3258.
E-mail address: jbyoo@konkuk.ac.kr (J. Yoo).

0951-8320/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.ress.2013.06.006

system [19]. The next step is to prepare an ANSI-C program which is the other input of HW-CBMC. We provide a '*CWrapper*' program which wraps the ANSI-C program with template-based statements to help users perform the HW-CBMC verification mechanically. The HW-CBMC model checker then verifies the behavioral consistency between the Verilog program translated from FBDs and the wrapped ANSI-C program. This paper uses a part of the FBD programs for ARP-1400 (Advanced Power Reactor-1400) RPS BP (Bistable Processor) to demonstrate feasibility and efficiency of the proposed verification technique.

The remainder of the paper is organized as follows: Section 2 briefly explains the basic elements of the proposed verification techniques, such as FBD, Verilog and HW-CBMC. It also details the typical software development life-cycle of PLC-based systems. The whole verification process is introduced in Section 3. In Section 4, we apply the proposed verification technique to a part of PLC software for APR-1400 RPS BP in Korea. Section 5 overviews related work and we conclude the paper in Section 6.

2. Background

2.1. PLC-based software development process

RPS is a real-time embedded system, implemented on the hardware—PLC. The RPS software is designed in FBD/LD languages and then translated into C programs which will be compiled and loaded on PLCs. Fig. 1 explains a typical software development process for RPS as a waterfall model [20].

SRS (Software Requirements Specification) is written in natural languages or formal specification languages [21–23]. Experts on PLC programming languages then translate the requirements specification into design models programmed in FBD or LD manually. PLC vendors provide their own automatic translators from the FBD/LD programs into ANSI-C programs, while typically using COTS (Commercial Off-the-Shelf) software such as '*TMS320C55x*' of *Texas Instruments* [24] for the C compilers. The COTS compilers were well verified and certified, and sufficient to be used for implementing the RPS software without additional efforts.

The lower part of the figure shows V&V (Verification and Validation) techniques which have been used to demonstrate correctness and functional safety of the '*Automatic Translator*.' '*TXS SIVAT*' [9] from *AREVA*'s *TELEPERM XS* [25] is an example of the C code-based simulation technique, while the '*RETRANS*' [11] is that of the bi-simulation technique. Structural testing techniques with coverage criteria [26] are also applied into the automatically translated C programs. The KNICS project in Korea used a testing tool '*IBM Rational Rhapsody*' [27] for C program testing. The equivalence checking is a verification technique which this paper proposes [28]. It uses a model checker *HW-CBMC* [17], which reads Verilog and ANSI-C programs and checks their behavioral equivalence [29]. It first translates FBD programs into behaviorally equivalent Verilog programs [18]. These various techniques spanning from simulation and testing to formal verification have all been used to guarantee the correct functioning of the PLC vendor-specific '*Automatic Translator*,' *i.e.*, the *FBD-to-C* translator.

2.2. Function Block Diagram

FBD (Function Block Diagram) is one of five standard PLC programming languages defined in the IEC 61131-3 standard [2]. It consists of an arbitrary number of function blocks connected together with wires similar to that of a circuit diagram. FBD has been widely used for developing software controllers of plants and machines because of its graphical notations and usefulness in implementing data flow based applications. For example, the FBD in Fig. 2 consists of 4 function blocks, and the first executed function block is `GE_DINT` while the last one is `SEL_DINT`. `GE_DINT` is the function block calculating logical '≥' with two decimal integer inputs. The whole FBD program is a set of FBDs interconnected with each other according to their predefined sequential execution order.

2.3. Verilog

Verilog is one of the most common HDLs (Hardware Description Languages) used by IC (Integrated Circuit) designers. Designs modeled in Verilog are technology independent, easy to develop and debug, and considered more readable than schematics. For this reason, Verilog is being increasingly used to specify software

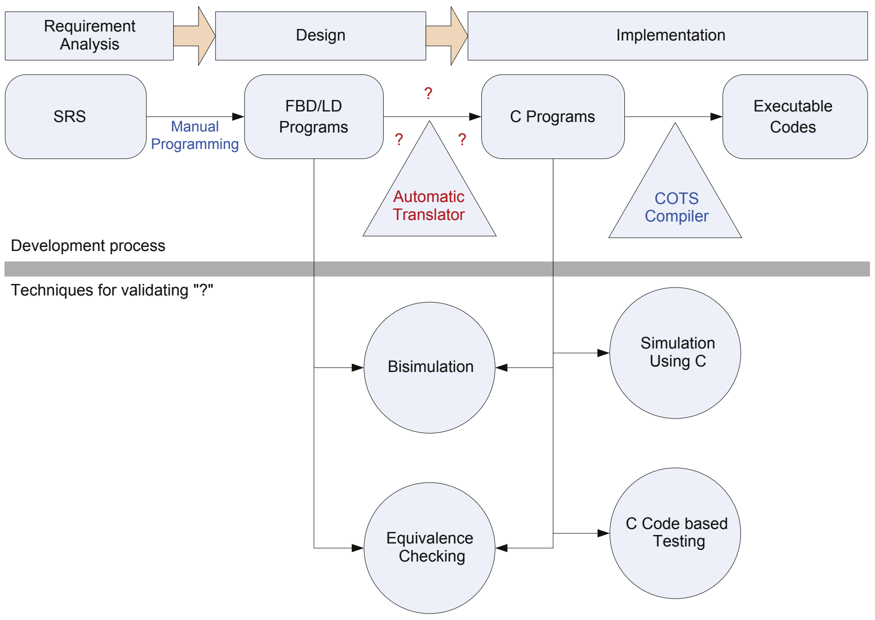


Fig. 1. RPS software development process using PLCs.

logic for process control systems. Verilog has several variable types. A `wire`, similar to a physical wire in a circuit, is used to connect modules in software development. A wire does not store its value and must be driven by a continuous assignment statement or by connecting it to an output of a module. On the other hand, a `reg`, used in a procedural assignment block beginning with `always`, represents a data object which holds its value from the current execution cycle to the next.

2.4. HW-CBMC

HW-CBMC [17] is a testing and debugging tool in IC (Integrated Circuit) industry. It can verify behavioral consistency between two implementations of the same design: one written in ANSI-C, typically for simulation, and the other written in register transfer level HDL, an actual product. Motivation of HW-CBMC is to reduce additional time needed for debugging and testing of the HDL implementations in order to produce chips as soon as possible. HW-CBMC reduces cost by providing an automated way of establishing the consistency of HDL implementation using the ANSI-C implementation as a reference, because the debugging and testing cost of the ANSI-C implementation is usually lower.

This paper proposes to use the HW-CBMC's behavioral consistency checking function for verifying the behavioral consistency between FBDs and ANSI-C programs. It requires FBDs to be translated into Verilog programs first [30]. The semantics of FBD are similar to that of Verilog in our previous research [31]. However, HW-CBMC does not allow all grammars and structures of Verilog, and we provide modified translation rules for translating FBD programs into appropriate Verilog programs for HW-CBMC.

3. Verification of FBD-to-C translator using HW-CBMC

3.1. Overview

Verification of the '*FBD-to-C*' translator using HW-CBMC consists of three steps as depicted in Fig. 3. The first step translates FBD programs into behaviorally equivalent Verilog programs. It then wraps ANSI-C programs using templates this paper provides in the second step. Finally HW-CBMC verifies behavioral equivalence between those programs, i.e., Verilog and ANSI-C programs. This subsection briefly explains these three steps.

`(Step 1)` HW-CBMC reads two inputs—Verilog and ANSI-C programs. We have to translate FBD programs into equivalent Verilog programs in order to use the HW-CBMC verification. Our previous research [18] proposed translation rules from FBD to Verilog for the formal verification using the VIS verification system [19]. This paper modified several translation rules since HW-CBMC reads Verilog

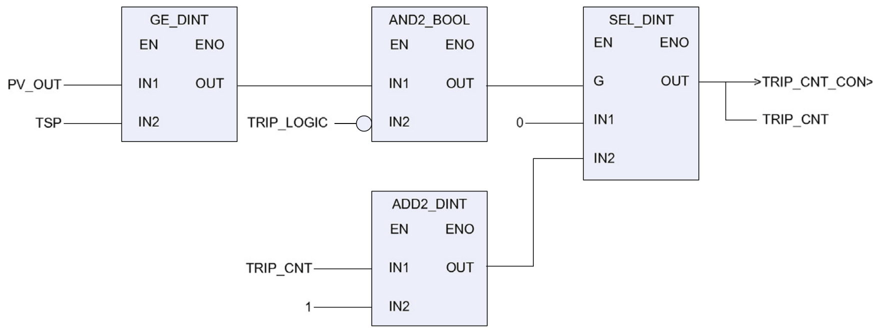


Fig. 2. An example of FBD program.

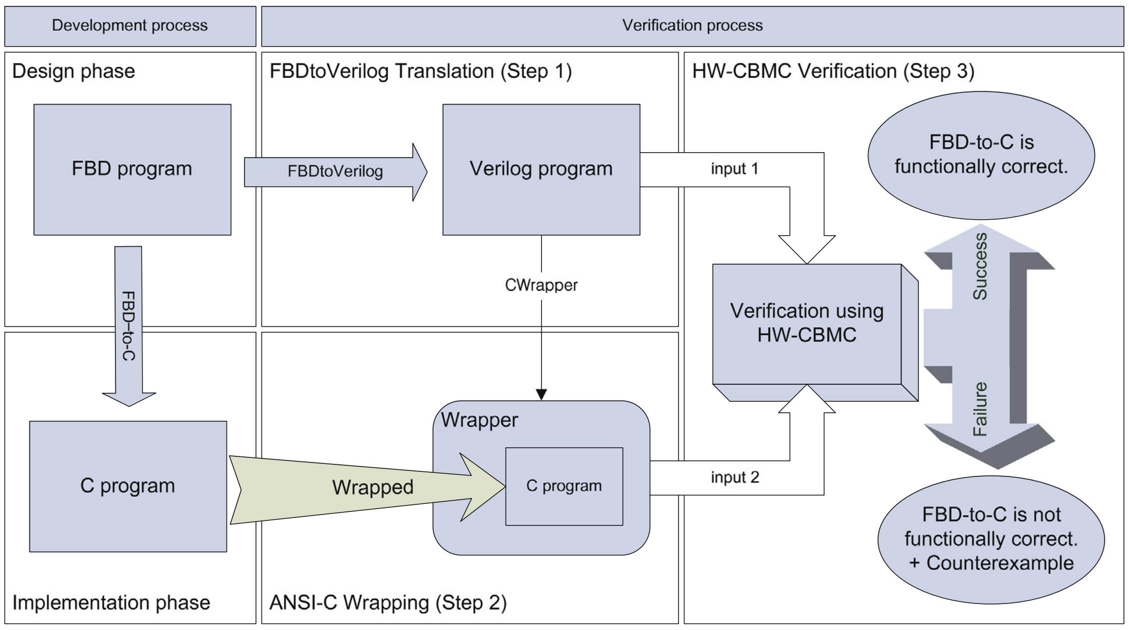


Fig. 3. An overall verification process for '*FBD-to-C*' translator using HW-CBMC.

programs which are slightly syntactically different from those for VIS. We implemented the modified rules into '*FBDtoVerilog 1.0H*' which are explained in detail in Section 3.2.

`(Step 2)` HW-CBMC requires users to insert into ANSI-C programs verification properties and codes for reading programs and initiating the verification. The insertion may incur unexpected modification on irrelevant parts, and is a potential threat to the validity of the HW-CBMC verification. It is also an error-prone activity to insert properties case-by-case. Thus, we provide a '*CWrapper*' program, wrapping the ANSI-C program with template-based statements to prohibit unnecessary modifications and to help users perform the HW-CBMC verification mechanically. Section 3.3 describes the templates-based wrapping process.

`(Step 3)` Users perform the *HW-CBMC verification* with two programs, a translated Verilog program and a wrapped ANSI-C program. HW-CBMC decides on their behavioral equivalence and produces '*success*' or '*fail*' with a counterexample. The former means that the FBD-to-C translator works functionally correctly, while the latter does not at least for the specific case of the counter-example.

3.2. (Step 1) The FBDtoVerilog translation

3.2.1. Well-formed FBDs

We assume that all FBDs should be well-formed FBDs. If a FBD is not well-formed, we cannot apply the systematic verification process using HW-CBMC to the FBD, since it is too biased from typical FBD programming schemes. An FBD is well-formed, if it satisfies the assumptions below:

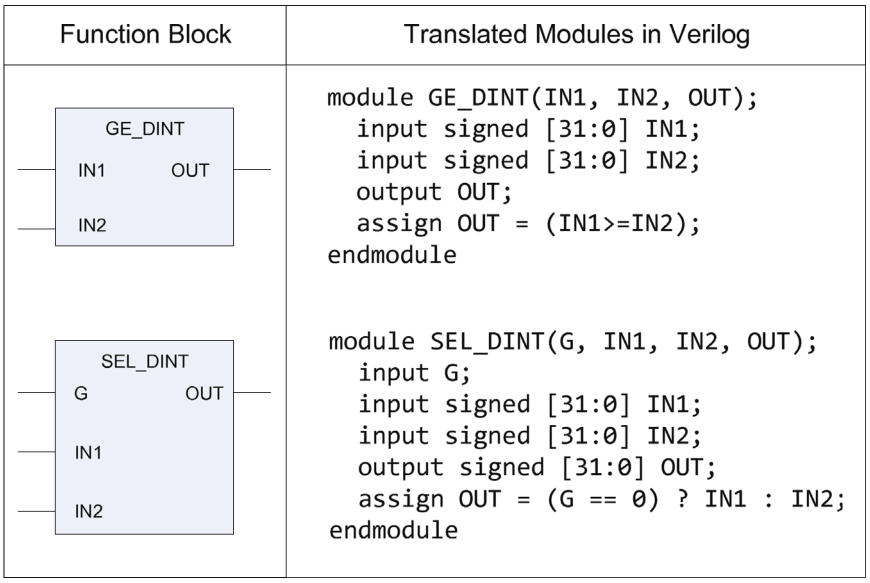
- *Assumption* 1. `EN` port of all FBs should be set to *enable*.
  FBD programming engineers often use `EN` and `ENO` ports as control signals to enable or disable other function blocks. The IEC 61131-3 standard does not explain the case clearly when function blocks are enabled and disabled subsequently by controlling the ports. Furthermore it is not an appropriate usage of FBDs that programmers use the `EN` port to control other function blocks as control-flow based languages such as C and JAVA, because FBD is a data-flow based language. We assume that all `EN` ports are set to 1 (TRUE) and they are not allowed to have a connection with others.
- *Assumption* 2. Explicit data-type conversions should be used.
  Some FBD software engineering tools allow implicit data-type conversions such as from `INT` to `BOOL`. However, arithmetic or logical computation among variables which have different types may cause unexpected errors, unless explicit type conversions do not proceed. We assume that all data-type conversions are defined explicitly.
- *Assumption* 3. Output variables should not be overwritten.
  An output variable must have a unique name and be assigned only once in a cycle. FBD evaluates outputs at every execution cycle. If an FBD generates several different values for the same output variable in a cycle, it is a potential cause of unexpected behaviors.

3.2.2. Translation of FBs and FBDs

Translation from (well-formed) FBDs to Verilog programs includes two steps: translations for function blocks (FBs) and function block diagrams (FBDs). A basic unit of the translation is a FB which is translated into a `module` of Verilog, whereas it is translated into a `function` of Verilog for the VIS verification system [18]. Fig. 4 presents two function blocks `GE_DINT` and `SEL_DINT` and corresponding module definitions written in Verilog. The translation of a FB starts declaring name and input/output ports of the module. Its body includes definitions of types and sizes of input/output ports. It also includes an `assign` statement which defines the behavior of a FB in accordance with IEC 61131-3 standard [2].

The translation of a FBD assumes that all modules for FBs are defined beforehand. It also starts with defining the name and input/output ports of the module too. If the FBD has feedback transitions or stores variables in order to use value at the later execution cycles (see [18] for details), then the variables are translated into the `reg` type variables. Other cases are translated into the `wire` type variables. All module definitions for FBs are called according to their execution order sequentially. At last, the `assign` statements assign values to the output variables of the FBD, and new values are also assigned to `reg` variables in the `always` block. Fig. 5 shows an example FBD program in KNICS RPS BP and the corresponding Verilog code.

We implemented the translation (from FBDs to Verilog programs for HW-CBMC) in a tool '*FBDtoVerilog 1.0H*.' It uses a *de facto* standard XML format of FBDs—PLCopen TC6 XML [32]. PLCopen is a vendor- and product-independent worldwide association. FBDtoVerilog 1.0H translates FBDs into Verilog programs which can be read by HW-CBMC, not VIS nor SMV. In our previous work, '*FBDtoVerilog*



```
module GE_DINT(IN1, IN2, OUT);
  input signed [31:0] IN1;
  input signed [31:0] IN2;
  output OUT;
  assign OUT = (IN1>=IN2);
endmodule


module SEL_DINT(G, IN1, IN2, OUT);
  input G;
  input signed [31:0] IN1;
  input signed [31:0] IN2;
  output signed [31:0] OUT;
  assign OUT = (G == 0) ? IN1 : IN2;
endmodule
```

Fig. 4. An example of function blocks and translated Verilog modules.

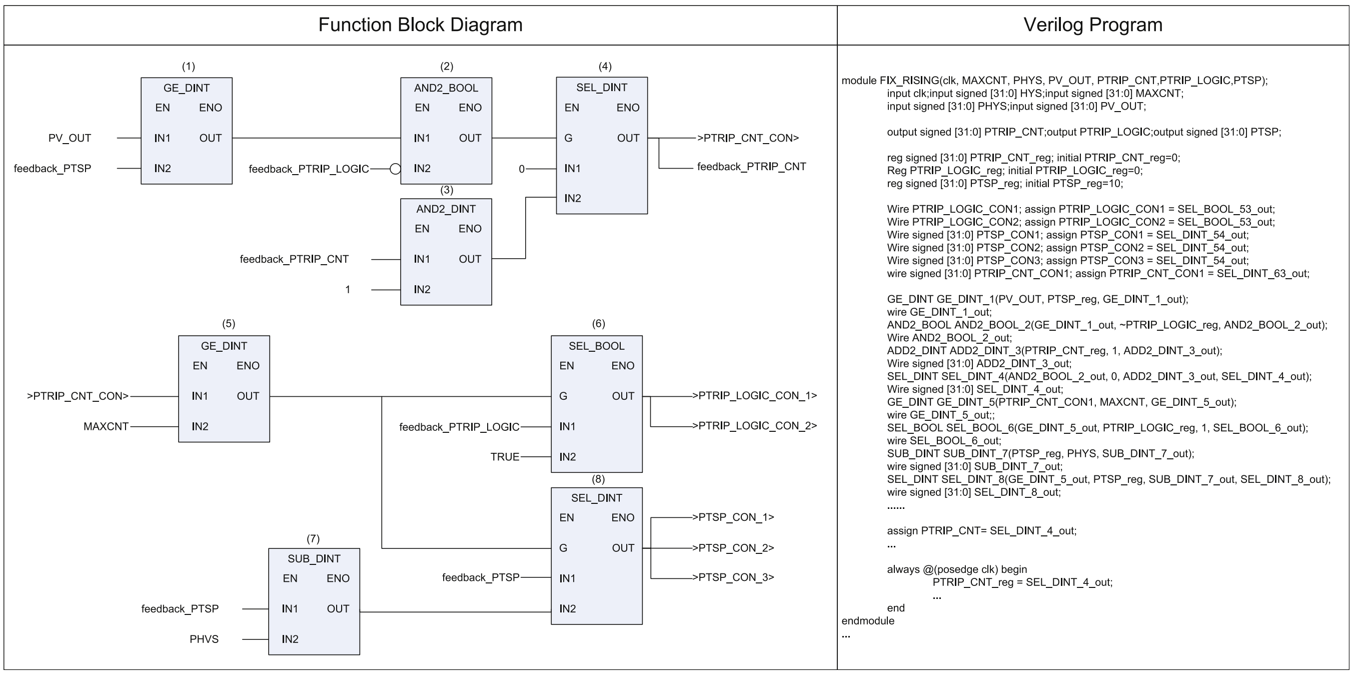
1.0' [18,33] translates FBDs into Verilog programs for verification using VIS and SMV [34]. The Verilog programs, however, cannot be used for the HW-CBMC verification, since the model checkers have their own restrictions and rules. For example, HW-CBMC cannot handle `functions` of Verilog whereas VIS and SMV can.

3.3. *(Step 2) The ANSI-C wrapping*

HW-CBMC reads an ANSI-C program to which verification properties and commands are inserted. Step 2 aims to produce the ANSI-C program in which all necessary information and commands are included, using a concept of wrapping with templates. Fig. 6 presents the wrapping process, and the output of the process is an ANSI-C program wrapped with all necessary information for the HW-CBMC verification. It is structured with 11 templates from T1 to T11 while 3 templates (T3, T7, T10) are optional. It helps users perform the verification mechanically and prevents frequent non-careful modifications on the ANSI-C programs.

T1. *Inclusion of the target ANSI-C file*

The wrapping process starts by including a target ANSI-C file. HW-CBMC can execute functions in a ".c" file which has the body of



```
module FIX_RISING(clk, MAXCNT, PHYS, PV_OUT, PTRIP_CNT,PTRIP_LOGIC,PTSP);
        input clk;input signed [31:0] HYS;input signed [31:0] MAXCNT;
        input signed [31:0] PHYS;input signed [31:0] PV_OUT;

        output signed [31:0] PTRIP_CNT;output PTRIP_LOGIC;output signed [31:0] PTSP;

        reg signed [31:0] PTRIP_CNT_reg; initial PTRIP_CNT_reg=0;
        Reg PTRIP_LOGIC_reg; initial PTRIP_LOGIC_reg=0;
        reg signed [31:0] PTSP_reg; initial PTSP_reg=10;

        Wire PTRIP_LOGIC_CON1; assign PTRIP_LOGIC_CON1 = SEL_BOOL_53_out;
        Wire PTRIP_LOGIC_CON2; assign PTRIP_LOGIC_CON2 = SEL_BOOL_53_out;
        Wire signed [31:0] PTSP_CON1; assign PTSP_CON1 = SEL_DINT_54_out;
        Wire signed [31:0] PTSP_CON2; assign PTSP_CON2 = SEL_DINT_54_out;
        Wire signed [31:0] PTSP_CON3; assign PTSP_CON3 = SEL_DINT_54_out;
        wire signed [31:0] PTRIP_CNT_CON1; assign PTRIP_CNT_CON1 = SEL_DINT_63_out;

        GE_DINT GE_DINT_1(PV_OUT, PTSP_reg, GE_DINT_1_out);
        wire GE_DINT_1_out;
        AND2_BOOL AND2_BOOL_2(GE_DINT_1_out, ~PTRIP_LOGIC_reg, AND2_BOOL_2_out);
        Wire AND2_BOOL_2_out;
        ADD2_DINT ADD2_DINT_3(PTRIP_CNT_reg, 1, ADD2_DINT_3_out);
        Wire signed [31:0] ADD2_DINT_3_out;
        SEL_DINT SEL_DINT_4(AND2_BOOL_2_out, 0, ADD2_DINT_3_out, SEL_DINT_4_out);
        Wire signed [31:0] SEL_DINT_4_out;
        GE_DINT GE_DINT_5(PTRIP_CNT_CON1, MAXCNT, GE_DINT_5_out);
        wire GE_DINT_5_out;;
        SEL_BOOL SEL_BOOL_6(GE_DINT_5_out, PTRIP_LOGIC_reg, 1, SEL_BOOL_6_out);
        wire SEL_BOOL_6_out;
        SUB_DINT SUB_DINT_7(PTSP_reg, PHYS, SUB_DINT_7_out);
        wire signed [31:0] SUB_DINT_7_out;
        SEL_DINT SEL_DINT_8(GE_DINT_5_out, PTSP_reg, SUB_DINT_7_out, SEL_DINT_8_out);
        wire signed [31:0] SEL_DINT_8_out;
        ......

        assign PTRIP_CNT= SEL_DINT_4_out;
        ...

        always @(posedge clk) begin
                PTRIP_CNT_reg = SEL_DINT_4_out;
                ...
        end
endmodule
...
```

Fig. 5. A part of FIX_RISING module in KNICS RPS BP and translated Verilog program.

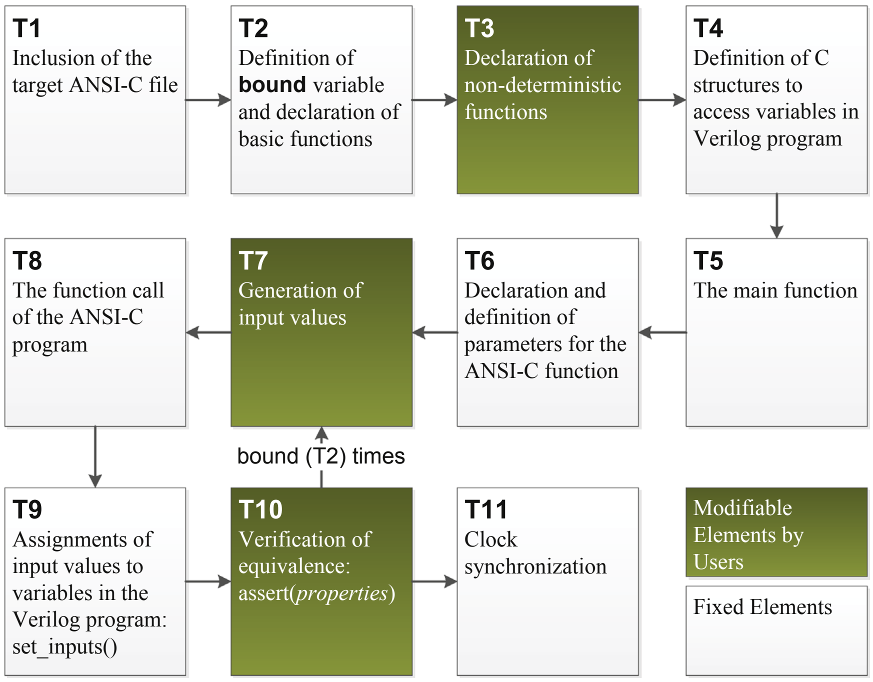

T1
Inclusion of the target ANSI-C file
T2
Definition of bound variable and declaration of basic functions
T3
Declaration of non-deterministic functions
T4
Definition of C structures to access variables in Verilog program
T8
The function call of the ANSI-C program
T7
Generation of input values
T6
Declaration and definition of parameters for the ANSI-C function
T5
The main function
bound (T2) times
T9
Assignments of input values to variables in the Verilog program: set_inputs()
T10
Verification of equivalence: assert(properties)
T11
Clock synchronization
Modifiable Elements by Users
Fixed Elements

Fig. 6. A template-based process of wrapping ANSI-C program.

the function, not a header, ".h", file. The inclusion, therefore, should include a ".c" file as shown below.

```
#include "ModuleName.c"
```

T2. *Definition of bound variable and declaration basic functions*

Definition of a bound variable and declaration of two basic functions, `next_timeframe()` and `set_inputs()`, are followed. The `bound` defines the number of repeated executions for the `main` function (T7–T10). Users have to set the value within a verification command at Step 3. The two functions are used for synchronizing two programs (i.e., Verilog and ANSI-C programs). '`next_timeframe()`' sets clock signals at T11 and '`set_input()`' sets assigned variables at T9.

```
extern const unsigned int bound;
void next_timeframe();
void set_input();
```

T3. *Declaration of non-deterministic functions*

T3 is a modifiable element. If users want to perform the verification with random values as input values, then it should be filled with non-deterministic functions. The functions return nondeterministic values to input variables, which mean that the value of inputs is not specified. Users may use them to assign the values to inputs at T7. A name of non-deterministic functions begins with the prefix `nondet_`. For instance, the following function returns a nondeterministically chosen integer:

```
int nondet_int();
```

T4. *Definition of C structures to access variables in Verilog program*

HW-CBMC needs `structure` data-type variables to access variables defined in the Verilog program. The structure types are declared at T4 and accessed at T9 to set values or at T10 to verify the equivalence. Information of input and output ports in Verilog programs makes variables in the structure types.

struct module ModuleName{int $\texttt{input}_n$;…_Bool $\texttt{output}_m$;};
extern struct module ModuleName ModuleName;

T5. *The main function*

The `main` function starts with declarations of temporary variables. The variables have temporary values generated at T7. They are not mandatory elements if the verification uses only deterministic values. It, however, is necessary for non-deterministic value generation, because two inputs—one for Verilog and the other for ANSI-C—must have equivalent values in a cycle. To provide equivalent inputs, the non-deterministic function should be called once in a cycle and return the random value to the temporary variables.

void main {
int temp_$\texttt{input}_0$, …, temp_$\texttt{input}_n$;
int temp_$\texttt{output}_0$, …, temp_$\texttt{output}_m$;

T6. *Declaration and definition of parameters for the ANSI-C function*

Execution of the ANSI-C program requires the definition of parameters for its function. Information in regard to the parameters decides the definition. For example, they may be various variables such as integer and boolean, or a single structure data-type. The parameters are used at T8 as parameters for the ANSI-C function and T10 as resources of verification properties.

int $\texttt{input}_0, \ldots, \texttt{input}_n$;
int $\texttt{output}_0, \ldots, \texttt{output}_m$;

T7. *Generation of input values*

Input variables can be assigned in two ways, deterministic and non-deterministic. Users may assign a value to an input variable deterministically by using an arithmetic formula. On the other hand, the non-deterministic assignment of input values should use the functions defined at T3. It returns values of specific types non-deterministically. The following assignment statements demonstrate an example of the two types of input variables, respectively.

$\texttt{input}_0 = \texttt{input}_0 + 10$;
$\texttt{input}_1 =$ nondet_int();

T8. *The function call of the ANSI-C program*

To execute the included ANSI-C program at T1, T8 calls a function in the ANSI-C program. The function is executed with the parameters defined at T6. The values which are generated at T7 have been assigned to the parameters before the function is executed. The below example assumes that a name of the function is the same as the name of the included ANSI-C program.

ModuleName($\texttt{input}_0, \ldots, \texttt{output}_m$);

T9. *Assignments of input values to variables in the Verilog program*

Executing a Verilog program also requires assigning values to the Verilog input variables. The values should equate to those used by the ANSI-C function at T8. We use the C structures defined at T4 to assign the value generated at T7 to the Verilog input variables. `set_inputs()` is then called to synchronize the assigned values with the Verilog program.

ModuleName.$\texttt{input}_n = \texttt{input}_n$;…
set_inputs();

T10. *Verification of equivalence*

The HW-CBMC verification uses assertion statements such as `assert(`**`property`**`)`. The **`property`** is an equality equation between two outputs, i.e., from ANSI-C and Verilog programs. The outputs of the ANSI-C program are defined at T6 while those of the Verilog program are defined at T4. The HW-CBMC verification succeeds when the **`property`** is true. If it is not true, i.e., these two outputs are not equivalent, the verification fails and it produces a counter example.

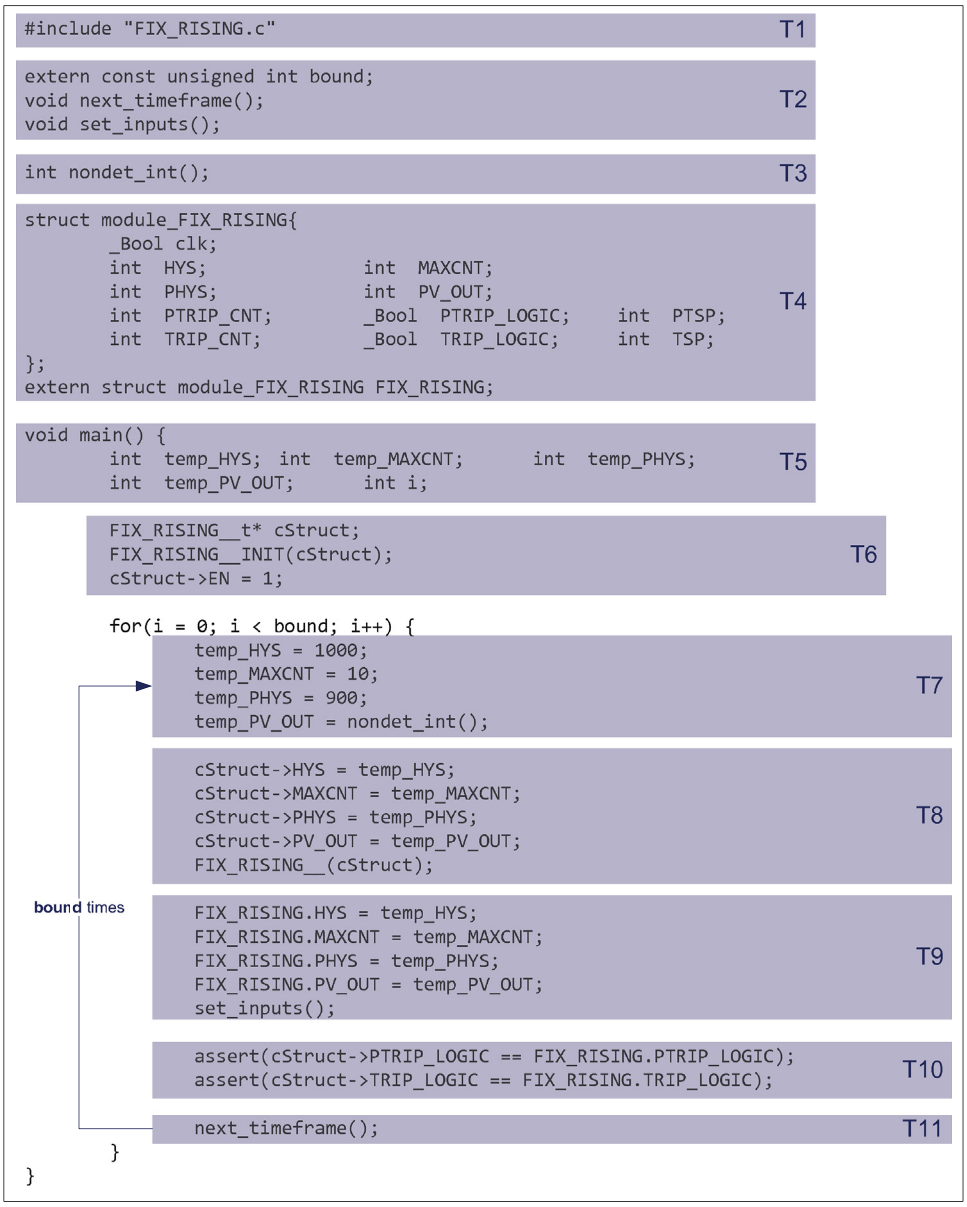
assert(ModuleName.$\texttt{output}_0 = = \texttt{output}_0$);

T11. *Clock synchronization*

The last step in the wrapping process is to make the Verilog program proceed by one execution cycle. We use the function `next_timeframe()`. Then it proceeds to T7 and generates inputs again. This iteration repeats for the `bound` times defined at T2.

```
next_timeframe();
```

Fig. 7 is an example of the wrapped ANSI-C program for the *FIX_RISING* module in the KNICS RPS BP. Hereafter we call it



```
#include "FIX_RISING.c"

extern const unsigned int bound;
void next_timeframe();
void set_inputs();

int nondet_int();

struct module_FIX_RISING{
        _Bool clk;
        int  HYS;                int  MAXCNT;
        int  PHYS;               int  PV_OUT;
        int  PTRIP_CNT;          _Bool  PTRIP_LOGIC;     int  PTSP;
        int  TRIP_CNT;           _Bool  TRIP_LOGIC;      int  TSP;
};
extern struct module_FIX_RISING FIX_RISING;

void main() {
        int  temp_HYS; int  temp_MAXCNT;      int  temp_PHYS;
        int  temp_PV_OUT;        int i;

        FIX_RISING__t* cStruct;
        FIX_RISING__INIT(cStruct);
        cStruct->EN = 1;

        for(i = 0; i < bound; i++) {
                temp_HYS = 1000;
                temp_MAXCNT = 10;
                temp_PHYS = 900;
                temp_PV_OUT = nondet_int();

                cStruct->HYS = temp_HYS;
                cStruct->MAXCNT = temp_MAXCNT;
                cStruct->PHYS = temp_PHYS;
                cStruct->PV_OUT = temp_PV_OUT;
                FIX_RISING__(cStruct);

                FIX_RISING.HYS = temp_HYS;
                FIX_RISING.MAXCNT = temp_MAXCNT;
                FIX_RISING.PHYS = temp_PHYS;
                FIX_RISING.PV_OUT = temp_PV_OUT;
                set_inputs();

                assert(cStruct->PTRIP_LOGIC == FIX_RISING.PTRIP_LOGIC);
                assert(cStruct->TRIP_LOGIC == FIX_RISING.TRIP_LOGIC);

                next_timeframe();
        }
}
```

Fig. 7. An example of the wrapped ANSI-C program for the *FIX_RISING* module in KNICS RPS BP.

Wrapper. The *Wrapper* includes an ANSI-C file (T1) first and declares basic functions and variables (T2). Declaration of the non-deterministic input function `nondet_int()` is followed (T3). The C structure, `module_FIX_RISING`, for the target Verilog programs is defined (T4).

The `main` function begins with the declaration of temporary variables (T5). The program uses the structure data-type to define input and output variables as a parameter (T6). The `for` loop iterates for the `bound` times. One non-deterministic and three deterministic assignments are followed (T7), and the ANSI-C and Verilog programs with assigned inputs are executed (T8 and T9). The *Wrapper* has two assertion statements for `trip` and `pretrip` outputs of the KNICS RPS BP (T10). Finally, the `for` loop ends with the function `next_timeframe()` (T11). We developed a CASE tool '*CWrapper*' generating the wrapped ANSI-C program in accordance with the templates-based process. The details will be introduced in Section 4.

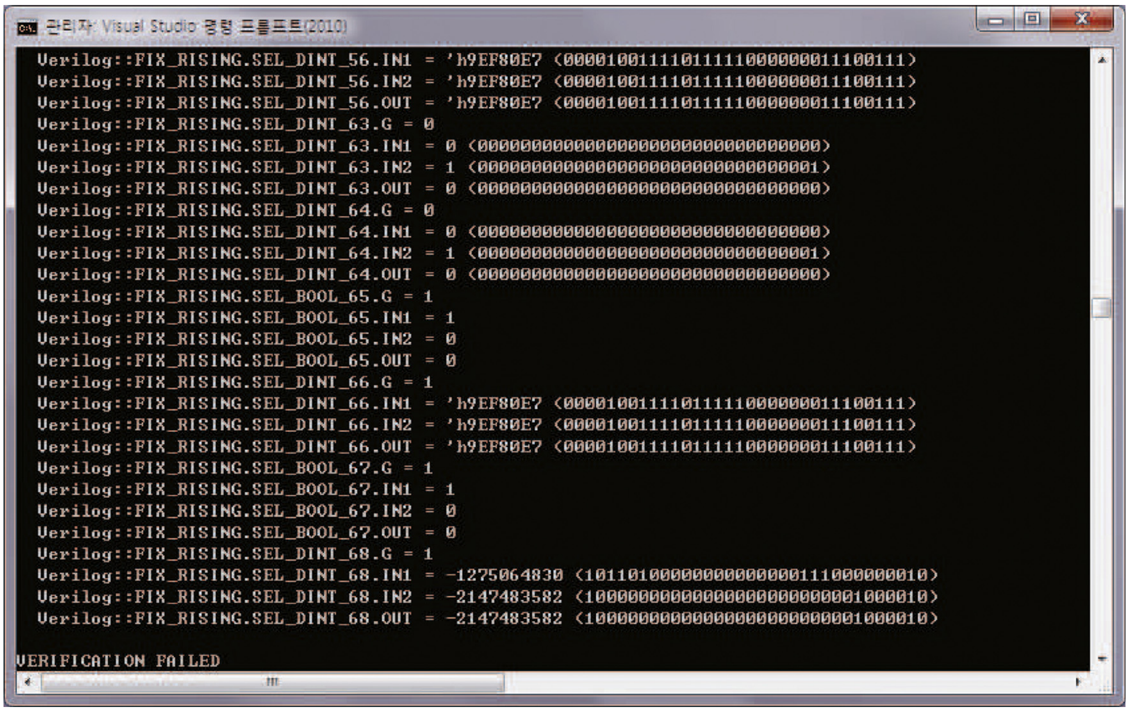
3.4. (Step 3) The HW-CBMC verification

The HW-CBMC verification aims to verify the behavioral equivalence between outputs from two programs (such as Verilog and ANSI-C) which have the same combination of inputs. It requires all related files (Verilog, ANSI-C, etc.) to be located at the same directory. Users can also execute the HW-CBMC verification within the Visual Studio Command Prompt.[1] The command for the verification can be seen below:

```
> hw-cbmc.exe ModuleName.v Wrapper.c
  –module ModuleName –bound N
```

ModuleName.v is the Verilog file translated from an FBD, while Wrapper.c is the wrapped file of a target ANSI-C program. The command has two parameters. `–module ModuleName` declares

[1] http://msdn.microsoft.com/en-us/library/ms229859.aspx.



```
 Verilog::FIX_RISING.SEL_DINT_56.IN1 = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_DINT_56.IN2 = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_DINT_56.OUT = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_DINT_63.G = 0
 Verilog::FIX_RISING.SEL_DINT_63.IN1 = 0 (00000000000000000000000000000000)
 Verilog::FIX_RISING.SEL_DINT_63.IN2 = 1 (00000000000000000000000000000001)
 Verilog::FIX_RISING.SEL_DINT_63.OUT = 0 (00000000000000000000000000000000)
 Verilog::FIX_RISING.SEL_DINT_64.G = 0
 Verilog::FIX_RISING.SEL_DINT_64.IN1 = 0 (00000000000000000000000000000000)
 Verilog::FIX_RISING.SEL_DINT_64.IN2 = 1 (00000000000000000000000000000001)
 Verilog::FIX_RISING.SEL_DINT_64.OUT = 0 (00000000000000000000000000000000)
 Verilog::FIX_RISING.SEL_BOOL_65.G = 1
 Verilog::FIX_RISING.SEL_BOOL_65.IN1 = 1
 Verilog::FIX_RISING.SEL_BOOL_65.IN2 = 0
 Verilog::FIX_RISING.SEL_BOOL_65.OUT = 0
 Verilog::FIX_RISING.SEL_DINT_66.G = 1
 Verilog::FIX_RISING.SEL_DINT_66.IN1 = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_DINT_66.IN2 = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_DINT_66.OUT = 'h9EF80E7 (00001001111011111000000011100111)
 Verilog::FIX_RISING.SEL_BOOL_67.G = 1
 Verilog::FIX_RISING.SEL_BOOL_67.IN1 = 1
 Verilog::FIX_RISING.SEL_BOOL_67.IN2 = 0
 Verilog::FIX_RISING.SEL_BOOL_67.OUT = 0
 Verilog::FIX_RISING.SEL_DINT_68.G = 1
 Verilog::FIX_RISING.SEL_DINT_68.IN1 = -1275064830 (10110100000000000000111000000010)
 Verilog::FIX_RISING.SEL_DINT_68.IN2 = -2147483582 (10000000000000000000000001000010)
 Verilog::FIX_RISING.SEL_DINT_68.OUT = -2147483582 (10000000000000000000000001000010)

VERIFICATION FAILED
```

Fig. 8. An example of a verification failure and a counterexample produced.

the module name of the Verilog program to read. `-bound N` declares the number of iterating cycles from T11 to T7 in the template.

HW-CBMC returns "*VERIFICATION SUCCESS*" when the two programs are behaviorally equivalent. Users can be confident that the ANSI-C program generated from an FBD program always shows the same behavior with its origin—FBD program. If the verification fails, HW-CBMC returns "*VERIFICATION FAILED*" and produces a counterexample as depicted in Fig. 8.

4. Case study

We applied the proposed process to two PLC programs, FIX_RISING and FIX_FALLING developed by pSET, to demonstrate its feasibility. Table 1 shows information of the two programs for this case study. The two programs have a similar structure; however, their operations are contrariwise. For example, FIX_RISING generates trip and pretrip signals where one of the inputs has a value over a set point for more than specific times. The pretrip signal is a warning signal for the trip signal, thus the limitation of the pretrip is lower.

We used an automatic translator, '*pSET2TC6*' [35], which translates the program files of pSET as the PLCopen TC6 XML standard, because pSET does not support the standard. pSET has two different data formats: one is a binary type which is unreadable, and the other is a ASCII type. Although readable, the ASCII type is not commonly used. Since we developed FBDtoVerilg according to the PLCopen standard for vendor- and product-independence, it is necessary to translate the ASCII format to the PLCopen TC6 XML standard format.

4.1. Translation of FBD and generation of wrapper

We translated the programs in Table 1 using FBDtoVerilog. *FBDtoVerilog* has an input file which follows PLCopen TC6 XML standard. It produces Verilog files with the same name as the name of the FBD program according to translation rules as described in Section 3.2. For instance, if a name of the FBD file is FIX_RISING.xml, then *FBDtoVerilog* produces a FIX_RISING.v file.

Table 1
Information of programs for case study.

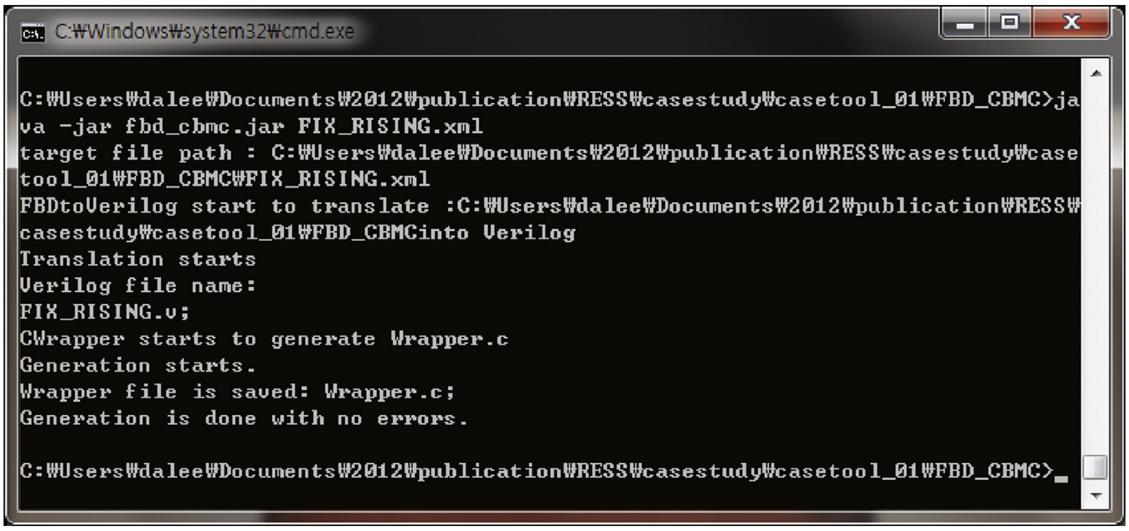
Module name	# Blocks	# Inputs	# Outputs (Feedback)
FIX_RISING	26	4	6 (4)
FIX_FALLING	26	4	6 (4)

CWrapper automatically produces a *Wrapper* file, Wrapper.c. When *CWrapper* performs the producing *Wrapper*, it refers to the translated Verilog program to define the C structure as described at T4 in Fig. 6. *CWrapper* considers how FBD-to-C translates the FBD to C, because the parameters and function call depend on the C program. pSET translates input and output ports of FBD into a structure data-type of C with suffix `__t`; it also translates the FBD program into a function of C with suffix __ (double underscores). For instance, the input and output ports of `FIX_RISING` are defined as a structure data-type `FIX_RISING__t`; the `FIX_RISING` is translated into `FIX_RISING__(FIX_RISING__t* a__)`. The current version of *CWrapper* is implemented with respect to the ANSI-C program translated by pSET.

We implemented an execution program to execute *FBDtoVerilog* and *CWrapper*. The program is executed with a FBD program, and it executes *FBDtoVerilog* to translate the FBD program into Verilog. After the translation, it executes *CWrapper* to produce *Wrapper*. Fig. 9 presents the execution with the *FIX_RISING* program.

4.2. Results of HW-CBMC verification

We performed the verification using HW-CBMC in deterministic and non-deterministic ways. Verification using non-deterministic functions usually takes more time and additional memory space. Furthermore, if number of the bound is too great, then HW-CBMC is shut down abnormally. We chose the bound


```
C:₩Windows₩system32₩cmd.exe

C:₩Users₩dalee₩Documents₩2012₩publication₩RESS₩casestudy₩casetool_01₩FBD_CBMC>ja
va -jar fbd_cbmc.jar FIX_RISING.xml
target file path : C:₩Users₩dalee₩Documents₩2012₩publication₩RESS₩casestudy₩case
tool_01₩FBD_CBMC₩FIX_RISING.xml
FBDtoVerilog start to translate :C:₩Users₩dalee₩Documents₩2012₩publication₩RESS₩
casestudy₩casetool_01₩FBD_CBMCinto Verilog
Translation starts
Verilog file name:
FIX_RISING.v;
CWrapper starts to generate Wrapper.c
Generation starts.
Wrapper file is saved: Wrapper.c;
Generation is done with no errors.

C:₩Users₩dalee₩Documents₩2012₩publication₩RESS₩casestudy₩casetool_01₩FBD_CBMC>_
```

Fig. 9. A screen dump of performing CASE tool with regard to *FIX_RISING* program.

Table 2
Verification results.

Module name	Result
FIX_RISING	VERIFICATION SUCCESS
Deterministic (bound)	73.851 s (100)
Non-deterministic (bound)	192.036 s (30)
FIX_FALLING	VERIFICATION SUCCESS
Deterministic (bound)	67.828 s (100)
Non-deterministic (bound)	272.391 s (30)

empirically, because there are not criterion that users can decide it.

The deterministic verification fixes three inputs; PTSP, TSP, and MAXCNT. We used a formula to assign values to another one, PV_OUT, according to a scenario. The scenario is that the three inputs are fixed and PV_OUT can increase or decrease. We also perform non-deterministic verification with respect to the scenario; however, we assigned a non-deterministic value to PV_OUT. Table 2 describes the verification results of two modules in BP of the KNICS project.

HW-CBMC returns "VERIFICATION SUCCESSFUL" for both cases, which means that the translator of pSET is functionally correct in the cases. Deterministic and non-deterministic verification of FIX_RISING took 73.851 s and 190.773 s respectively. It shows that non-deterministic verification takes more time than deterministic one even if it has less bound. Fig. 10 presents a screen dump which is the result of the non-deterministic verification of FIX_RISING.

5. Related work

5.1. Equivalence checking

Equivalence checking is a technique used to check the functional equivalence between two programs. The VIS (Verification Interacting with Synthesis) [19] is a widely used tool for the formal verification, synthesis, and simulation of finite state systems. It uses Verilog as a front-end and provides combinational and sequential equivalence checking of two Verilog programs. The combinational equivalence of the VIS provides a sanity check when re-synthesizing portions of a network, and its sequential verification is done by building the product finite state machine. Yoo et al. [18] uses VIS to verify the equivalence of PLC programs between successive revisions. The revision for optimization must exhibit the same functions as the former one. The approach helps to guarantee that the revision does not change the functions.

On the other hand, there is a study for equivalence checking between two different descriptions. Bombieri et al. [36] presents a formal definition of equivalence between the Transaction Level Modeling (TLM) and Register Transfer Level (RTL) is presented. The TLM is the reference modeling style for hardware/software design and verification of digital systems, and the RTL is a level of abstraction used in describing the operation of a synchronous digital circuit. The definition is based on events, and it shows how such a definition can be used for proving the equivalence between both.

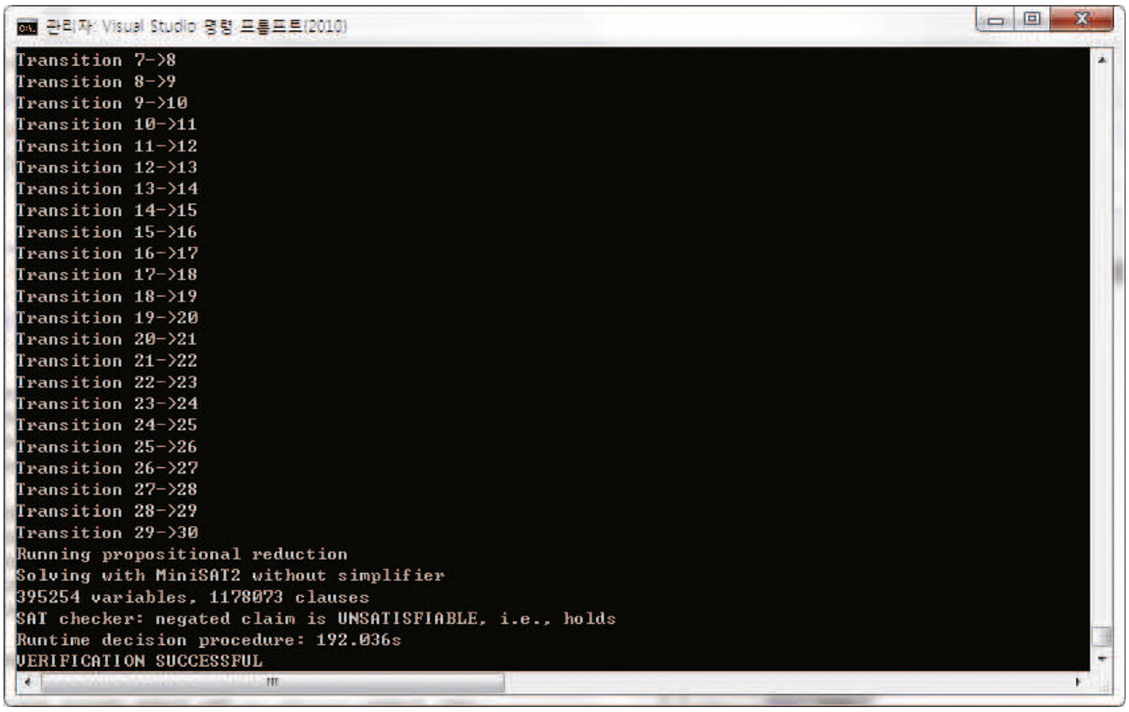
5.2. Verification of compilers, code-generators and translators

Verifying the correctness of compilers directly including code generators or translators is one of the most difficult topics in computing research [37]. There are researches to verify them through various techniques; see [38] for a survey. One of them uses two model checkers to verify untrusted code generators [39]. It specifies the same properties in two expressions, LTL (Linear Temporal Logic) for the NuSMV [34] model checker and user-specified assertion in CBMC. NuSMV verified the original program with the LTL properties and CBMC verified the automatically generated program with the user-specified assertion properties. They verified the code generator using the two programs with the two same properties about correctness.

Ref. [40] shows verification of compiling specification for a Lisp compiler. They verified compilation from ComLisp to the Stack-based intermediate language SIL, which is the first phase of the compilation. They specified two languages formally, and specified the compilation rules formally. The correctness of the compilation process is verified using a PVS specification and verification system. Verification of a C compiler is also performed in [41]. This is also focused on a partial C compiler, which is its front-end. They verified the observational semantic equivalence between the source and generated code using Coq, which is proof assistant.

6. Conclusion

This paper describes a systematic technique to verify behavioral consistency between FBD design and ANSI-C implementation using HW-CBMC. We introduced the process of the verification technique and gave explanations for the steps of each process. The



```
Transition 7->8
Transition 8->9
Transition 9->10
Transition 10->11
Transition 11->12
Transition 12->13
Transition 13->14
Transition 14->15
Transition 15->16
Transition 16->17
Transition 17->18
Transition 18->19
Transition 19->20
Transition 20->21
Transition 21->22
Transition 22->23
Transition 23->24
Transition 24->25
Transition 25->26
Transition 26->27
Transition 27->28
Transition 28->29
Transition 29->30
Running propositional reduction
Solving with MiniSAT2 without simplifier
395254 variables, 1178073 clauses
SAT checker: negated claim is UNSATISFIABLE, i.e., holds
Runtime decision procedure: 192.036s
VERIFICATION SUCCESSFUL
```

Fig. 10. The result of non-deterministic verification of FIX_RISING.

first step is translation of the target FBD program into a semantically equivalent Verilog program. Second, it produces a wrapping program to wrap the target ANSI-C program. Finally, HW-CBMC verifies behavioral consistency between the Verilog program and the ANSI-C program. We performed two case studies developed by pSET to demonstrate its feasibility. We also implemented CASE tools, '*FBDtoVerilog 1.0H*' and '*CWrapper*.' The case studies are performed semi-automatically with the CASE tools. Our future research plan is to make a GUI (Graphic User Interface) environment for convenience to perform the process. Another plan is to provide the iterating number of the '`main`' function for the verification in a precise way. This paper uses the empirical number of the iteration, but we expect that it is possible to provide the number in a statistical or mathematical way.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-0003619) and by the MKE (The Ministry of Knowledge Economy), Korea, under the Development of Performance Improvement Technology for Engineering Tool of Safety PLC (Programmable Logic Controller) program supervised by the KETEP (Korea Institute of Energy Technology Evaluation And Planning) (KETEP-2010-T1001-01038). It was also supported, in part, by a grant from the Korea Ministry of Strategy, under the development of the integrated framework of I&C conformity assessment, sustainable monitoring, and emergency response for nuclear facilities.

References

[1] Leveson N. SafeWare: system safety and computers. Computer science and electrical engineering series. Addison-Wesley; 1995.
[2] I. IEC, 61131-3. Programmable controllers-part 3: programming languages, International Standard, Second Edition, International Electrotechnical Commission, Geneva, vol. 1; 2003.
[3] I. IEC, IEC 61508: Functional safety of electrical, electronic and programmable electronic (E/E/PE) safety-related systems, International Standard, Second Edition, International Electrotechnical Commission, Geneva, vol. 1; 2003.
[4] Areva, last accessed: Jan/4/2013 ⟨http://www.areva.com⟩.
[5] invensys, last accessed: Jan/4/2013 ⟨http://iom.invensys.com⟩.
[6] Posco ict, last accessed: Jan/4/2013 ⟨http://www.poscoict.co.kr⟩.
[7] SIEMENS, SPACE, Engineering system of Teleperm XS PLC, Technical Report. KWU NLL1-1026-76-V1.0/11.96, Germany, last accessed: Jan/4/2013; 1996.
[8] SIEMENS, Teleperm xs, brief description, Technical Report. KWU NLL1-1004-76-V2.2/04.98, Germany; 1998.
[9] Richter S, Wittig J-U. Verification and validation process for safety I&C systems. Nuclear Plant Journal 2003;21(3):36–40.
[10] Istec: Industrielle software-technik gmbh, last accessed: Jan/4/2013 ⟨http://www.istec.de⟩.
[11] iSTec, RETRANS, institute for safety technology, last accessed: Jan/4/2013 ⟨http://www.istec-gmbh.de/leistungen/qualifizierung/produkte⟩.
[12] Safety software suite, last accessed: Jan/4/2013 ⟨http://iom.invensys.com/⟩.
[13] KNICS, Korea nuclear instrumentation and control system r&d center, last accessed: Jan/1/2010 ⟨http://www.knics.re.kr/english/eindex.html⟩.
[14] Cho S, Koo K, You B, Kim T.-W, Shim T, Lee JS. Development of the loader software for PLC programming. In: Proceedings of conference of the institute of electronics engineers of Korea, vol. 30; 2007. p. 995–60.
[15] Wikipedia, Nuclear power in South Korea, last accessed: Jan/4/2013 [⟨http://en.wikipedia.org/wiki/Nuclear_power_in_South_Korea⟩].
[16] Clarke E, Grumberg O, Peled D. Model checking. MIT Press; 1999.
[17] Clarke E, Kroening D. Hardware verification using ANSI-C programs as a reference. In: Design automation conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific, IEEE; 2003. p. 308–11.
[18] Yoo J, Cha S, Jee E. Verification of PLC programs written in FBD with VIS. Nuclear Engineering and Technology 2009;41(1):79–90.
[19] Brayton R, Hachtel G, Sangiovanni-Vincentelli A, Somenzi F, Aziz A, Cheng S, et al. Vis: a system for verification and synthesis. In: Computer aided verification. Springer; 1996. p. 428–32.
[20] Sommerville I. Software engineering. International computer science series. Addison-Wesley; 2007.
[21] Heitmeyer C, Jeffords R, Labaw B. Automated consistency checking of requirements specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 1996;5(3):231–61.
[22] Yoo J, Kim T, Cha S, Lee J, Seong Son H. A formal software requirements specification method for digital nuclear plant protection systems. Journal of Systems and Software 2005;74(1):73–83.
[23] Yoo J, Jee E, Cha S. Formal modeling and verification of safety-critical software. IEEE Software 2009;26(3):42–9.
[24] TEXAS INSTRUMENTS, TMS320C55x Optimizing C/C++ Compiler User Guide; 2003.
[25] AREVA, Teleperm xs system overview, Online, last accessed: Jan/4/2013 ⟨http://de.areva.com/EN/customer-397/teleperm-xs-system-overview.html⟩.

[26] Pezzè M, Young M. Software testing and analysis: process, principles, and techniques. Wiley; 2008.

[27] I. Rational, Rational rhapsody, last accessed: Jan/4/2013 ⟨http://www-01.ibm.com/software/awdtools/rhapsody/⟩.

[28] Lee D, Yoo J, Lee J. Equivalence checking between function block diagrams and C programs using HW-CBMC. Computer Safety, Reliability, and Security 2011:397–8.

[29] Huang H, Cheng K. Formal equivalence checking and design debugging, Frontiers in electronic testing; FRET 12. Kluwer Academic; 1998.

[30] IEC/IEEE behavioural languages—part 4: Verilog hardware description language (adoption of IEEE STD 1364-2001), IEC 61691-4 First edition 2004-10; IEEE 1364 (2004) 0–860.

[31] Jee E, Jeon S, Cha S, Koh K, Yoo J, Park G, et al. FBDVerifier: interactive and visual analysis of counter-example in formal verification of function block diagram. Journal of Research and Practice in Information Technology 2010;42 (3):171–88.

[32] PLCopen for efficiency in automation, last accessed: Jan/4/2013 ⟨http://www.plcopen.org⟩.

[33] Yoo J, Lee J, Jeong S, Cha S. FBDtoVerilog: a vendor-independent translation from fbds into verilog programs. In: The twenty-third international conference on software engineering and knowledge engineering (SEKE 2011); 2010. p. 48–51.

[34] Cimatti A, Clarke E, Giunchiglia F, Roveri M. Nusmv: a new symbolic model verifier. In: Halbwachs N, Peled D, editors. Computer aided verification. Lecture notes in computer science, vol. 1644. Berlin, Heidelberg: Springer; 1999. p. 495–9.

[35] Lee D-A, Yoo J. pSET2TC6: a translation tool to standardize the output format of pSET. In: Korean institute of information scientists and engineers (KIISE) 2011, vol. 38; 2011. p. 105–7.

[36] Bombieri N, Fummi F, Pravadelli G, Marques-Silva J. Towards equivalence checking between TLM and RTL models. In: Fifth IEEE/ACM international conference on formal methods and models for codesign, 2007. MEMOCODE 2007, IEEE; 2007. p. 113–22.

[37] Hoare T. The verifying compiler: a grand challenge for computing research. Modular Programming Languages 2003:25–35.

[38] Dave M. Compiler verification: a bibliography. ACM SIGSOFT Software Engineering Notes 2003;28(6) 2–2.

[39] Staats M, Heimdahl M. Partial translation verification for untrusted code-generators. Formal Methods and Software Engineering 2008:226–37.

[40] Dold A, Vialard V. A mechanically verified compiling specification for a lisp compiler. FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science 2001:144–55.

[41] S. Blazy, Z. Dargaye, X. Leroy, Formal verificationof a C compiler front-end, FM 2006: Formal Methods 2006: 460–75.