Annals of Nuclear Energy 220 (2025) 111484

Contents lists available at ScienceDirect

Annals of Nuclear Energy

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

CVEC: A customized VIS-based equivalence checker for verifying commercial field-programmable gate array synthesis software in small modular reactors

Yoona Heo [a], Sejin Jung [b], Eui-Sub Kim [a], Junbeom Yoo [a,*]

[a] Konkuk University, Seoul, Republic of Korea
[b] Chinju National University of Education, Chinju, Republic of Korea

ARTICLE INFO

Keywords:
COTS software dedication
Equivalence checking
Verilog
EDIF
VIS
FPGA synthesis software

ABSTRACT

Field-programmable gate array (FPGA) is a hardware-based platform widely used in safety-related systems. FPGA development involves synthesis, placement and routing, with designs ultimately downloaded onto the device. Commercial FPGA synthesis software converts register-transfer level (RTL) designs into gate-level representations. Standards such as NUREG/CR-6421, IEEE Std 7-4.3.2, EPRI NP-5652, and EPRI TR-106439 require verification of these tools, while IEC 62566 mandates static analysis, including equivalence checking, for nuclear power plant applications. This paper introduces *CVEC*, a customized VIS-based equivalence checker that verifies the correctness of FPGA synthesis software. It performs equivalence checking between RTL designs and gate-level designs synthesized using *Synopsys Synplify Pro* within the *Libero IDE*. When verification is successful, it ensures that these softwares operate correctly at the synthesis level. Two case studies demonstrate the effectiveness of *CVEC* in verifying the functional correctness of commercial FPGA synthesis softwares.

1. Introduction

A small modular reactor (SMR) is an advanced nuclear reactor currently gaining attention. Several documents and studies discuss the potential use of field-programmable gate arrays (FPGAs) in SMRs (NuScale, 2020; U.S. Nuclear Regulatory Commission, 2016; Cummins and Quinn, 2021). FPGA-based controllers are being utilized not only in SMRs but also in the broader nuclear power plant (NPP) domain, with several studies exploring their applications (Farias et al., 2016; Zhang and Wu, 2024; Piggin and Sampson, 2016). Moreover, FPGA technology is being applied in various applications, including cyber–physical systems (CPSs) (Gautham, 2020), electric vehicles (Bukya et al., 2024), internet of things (IoT) (Lee and Park, 2021), autonomous vehicles (Ahn et al., 2019), grid systems (Allani et al., 2021), neural networks (Wang et al., 2022), quantum processing (Xu et al., 2021), and more.

As stated in IEEE Std 1012-2016 (Institute of Electrical and Electronics Engineers (IEEE), 2016a), IEEE Std 7–4.3.2-2016 (Institute of Electrical and Electronics Engineers (IEEE), 2016b) and IEC 60880:2006 (International Electrotechnical Commission (IEC), 2006), digital devices used in safety systems of NPP should be thoroughly verified and validated throughout the entire software development life-cycle (SDLC). However, an FPGA is a hardware-based platform and works with a thoroughly different SDLC from micro-processor based platforms such as programmable logic controller (PLC). The FPGA software is first modeled with hardware description languages (HDLs) such as Verilog (Institute of Electrical and Electronics Engineers (IEEE), 2001) and VHDL (Institute of Electrical and Electronics Engineers (IEEE), 2008), and then subsequently synthesized into gate-level designs. Commercial logic synthesis tools and electronic design automation (EDA) tools (e.g. *Synopsys Synplify Pro* (Synopsys, 2015), etc.) automate the FPGA logic synthesis process. Then these designs are placed and routed into physical layouts by EDA tools.

According to NUREG/CR-6421 (U.S. Nuclear Regulatory Commission, 1996b), IEEE Std 7–4.3.2 (Institute of Electrical and Electronics Engineers (IEEE), 2016b), Electric Power Research Institute (EPRI) NP-5652 (Electric Power Research Institute (EPRI), 2014) and EPRI TR-106439 (Electric Power Research Institute (EPRI), 1996), commercial grade items (*i.e.,* commercial-off-the-shelf (COTS) items) used in nuclear safety-related application must be verified through sufficient acceptance methods to assure their functionality. Also, in IEC 62566-2:2020 (International Electrotechnical Commission (IEC), 2020), it is stated that static analysis including equivalence checking is required for the HDL-programmed devices (HPD) (*e.g.,* FPGA) used in the nuclear safety applications. While the logic synthesis tools can be formally verified with compiler verification techniques (Hoare, 2003) directly, it is hard to apply them to the products of 3rd-party developers. An alternative solution is to do the tool verification indirectly as part of the COTS software dedication (Yoo et al., 2015).

* Corresponding author.
E-mail address: jbyoo@konkuk.ac.kr (J. Yoo).

https://doi.org/10.1016/j.anucene.2025.111484
Received 6 January 2025; Received in revised form 23 March 2025; Accepted 12 April 2025

"*For a specific input program (e.g., Verilog program), if a synthesis tool produces a program (e.g., Netlist) which shows the same behavior* ***for all possible cases****, we can claim that the synthesis tool works correctly at least for the input program.*"

There are several studies or commercial formal verification tools which can be used for our purpose, *i.e., the correctness verification (dedication) of commercial FPGA logic synthesis tools*, such as *FormalPro* (Siemens, 0000a), *Conformal Equivalence Checker* (Cadence, 2017) and *Formality* (Synopsys, 2019). They are, however, too case-sensitive to be comfortable with various combinations of EDA and verification tools, as we summarized in Yoo et al. (2015). For instance, we cannot use *FormalPro*, a formal equivalence checking tool for FPGA design, in combination with *Libero IDE* (Microchip, 2014), an IDE for FPGA design, and *Synopsys Synplify Pro* (Synopsys, 2015) synthesizer, a logic synthesis tool for FPGA. This was the combination of the project we worked on. Since we cannot expect the vendors to provide a specific extension that cannot make a profit, we need to develop a new customized solution for this combination.

This paper proposes a **C**ustomized **V**IS-based **E**quivalence **C**hecker (*CVEC*), to check behavioral equivalence of a Verilog program and a Netlist synthesized from it under the combination of the *Synopsys Synplify Pro* synthesizer in the *Libero IDE* EDA tool. In the FPGA industry, *Synopsys Synplify Pro* and *Libero IDE* are already being integrated and used together (Synopsys, 2009). *CVEC* uses verification engine, verification interacting with synthesis (*VIS*) (Brayton et al., 1996), a widely-used open-source system for formal verification, synthesis, and simulation of finite-state systems. It reads two input programs (*i.e.,* Verilog and electronic design interchange format (EDIF)) and checks their behavioral equivalence for all possible input combinations using *VIS*. This paper also proposes a set of assumptions, translation rules, and a translator to make the VIS-based equivalence checking possible, since VIS cannot read the Verilog and EDIF files directly.

In order to demonstrate the effectiveness and applicability of the proposed technique, *CVEC*, we performed a case study with two examples of bistable processor (BP) programs in reactor protection system (RPS), excerpted from a preliminary version of Korean nuclear power plants. We also tried to compare the performance of *CVEC* with a commercial verification tool *FormalPro*, although we had to change the logic synthesis tool into *Precision FPGA* (Siemens, 0000b) instead.

The paper is organized as follows: Section 2 provides background information about the conventional FPGA development process, EDIF and the VIS verification system. Section 3 surveys related researches, and Section 4 overviews the *CVEC* and explains the details of its verification process. The case studies with two RPS BPs of Korean nuclear power plants is presented in Section 5. In conclusion, Section 6 concludes the paper and provides remarks on future research extensions. Appendix includes examples of translations from EDIF to BLIF-MV (Brayton, 1991).

2. Background

2.1. Development and V&V of FPGA

The development life-cycle of FPGA-based digital I&Cs basically follows IEC 61513 (International Electrotechnical Commission (IEC), 2011). An FPGA-based system, however, has a specific feature to consider further. The phase of the development life cycle involving HDL programming is classified as software, whereas the phase that begins after downloading it onto a chip is classified as hardware. Therefore, FPGA development should adhere to both IEC 60880 (International Electrotechnical Commission (IEC), 2006) for software compliance and IEC 60987 (International Electrotechnical Commission (IEC), 2007) for hardware compliance. ⟨Fig. 1⟩ (Jung et al., 2016) illustrates the V-model of FPGA development life-cycle described in IEC 62566 (International Electrotechnical Commission (IEC), 2012), integrating both software and hardware aspects. The left-hand side of the ⟨Fig. 1⟩ represents the software aspect of a typical FPGA development life-cycle, as guided in NUREG/CR-7006 (U.S. Nuclear Regulatory Commission, 1996a).

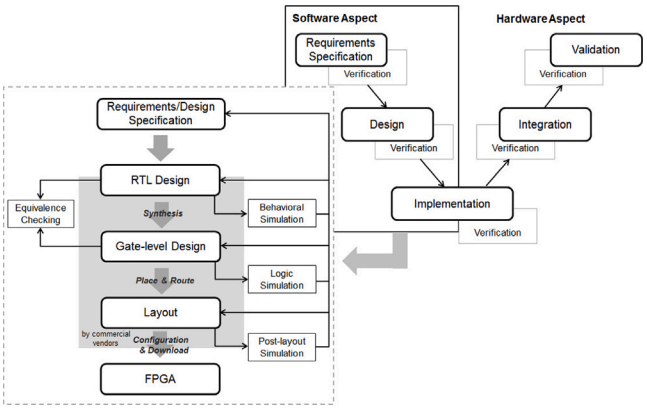

Fig. 1. The V-shaped life-cycle of FPGA development.

The FPGA software development is fully automated by synthesis softwares (*i.e.,* FPGA logic synthesis tools or EDA tools) of FPGA vendors. After programming a register transistor logic (RTL) design with HDLs, the design is transformed into a gate-level design (*i.e.,* netlist) by synthesis softwares such as *Synopsys Synplify Pro* (Synopsys, 2015), *Precision FPGA* (Siemens, 0000b) and *Cadence Virtuoso Digital Implementation* (Cadence, 2019).

For the verification and validation (V&V) of the FPGA development process, designers perform *simulation-based verification* at each stage of the FPGA software development life-cycle to ensure that each artifact complies with its requirements specification. The first simulation performed on RTL designs is *behavioral simulation*. The goal of *behavioral simulation* is to ensure that all requirements are correctly and accurately implemented into the RTL design. Since RTL designs are typically created manually by most designers, the process is time-consuming. After synthesis from RTL design to gate-level design, designers perform *logic simulation* to confirm that all functionalities were preserved during the synthesis. After the place & route (P&R) process, designers can validate the layout via *post-layout simulation* to ensure that the layout meets all timing requirements. Simulators such as *ModelSim* (Siemens, 0000c) and *Questa Simulator* (Siemens, 0000d) are widely used for the simulation-based verification. Every simulation-based verification at each step is performed individually and independently, and it is considered as one of the key factors for efficient FPGA development.

The V&V process for the FPGA development also includes equivalence checking (Burch et al., 1994). *Equivalence checking* is one of the formal verification techniques, which uses formal techniques to determine whether two versions of a design are behaviorally equivalent or not. Equivalence checking can prove that two given designs have the same functionality, *i.e.,* it determines "*whether they show the same behavior for all possible input sequences.*" It can ensure that an RTL design and the gate-level design synthesized from the RTL design always show the same behavior. This verification technique can be performed quickly and without the need for test vectors. As the automated synthesis by synthesis softwares becomes increasingly sophisticated, unintended and unexpected behaviors in FPGA designs may arise. At this point, equivalence checking can help us ensure that the synthesis process was executed correctly. Commercial tools such as *FormalPro*, *Conformal Equivalence Checker*, and *Formality* can be used for equivalence checking; however, their applicability is limited.

2.2. EDIF

EDIF (Electronic Industries Association, 1998) is a vendor-neutral format in which to store netlists and schematics. It was one of the

first attempts to establish a neutral data exchange format for the EDA industries.

The latest version of EDIF is 4.0.0, but most FPGA vendors still use the version 2.0.0 which was first approved as the standard ANSI/EIA-548-1998. Nevertheless the effort for the neutral data exchange, FPGA vendors keep modifying the EDIF format slightly and appropriately for their own tools. The EDIFs of various FPGA vendors are now not compatible with each other, unfortunately. This paper uses the EDIF of *Libero IDE*.

2.3. VIS verification system

VIS (Brayton et al., 1996) is a verification system that integrates formal verification, simulation and synthesis for hardware systems modeled as finite-state machines (FSMs). The VIS system supports fair computational tree logic (CTL) model checking (Clarke et al., 1999), language emptiness checking, sequential equivalence checking, cycle-based simulation, and hierarchical synthesis. It processes hardware designs described in (Synchronous) Verilog (Chouy, 1997) or BLIF-MV (Kukimoto, 1996) formats, using an in-house translator *vl2mv* to convert Verilog designs into the *BLIF-MV* representation for efficient analysis and synthesis.

In our earlier work (Yoo et al., 2009), we proposed a verification technique that translates a function block diagram (FBD), a graphical programming language defined in IEC 61131-3 (International Electrotechnical Commission (IEC), 2013), into a semantically equivalent Verilog program. We then attempted to prove the behavioral equivalence between two successive revisions using the sequential equivalence checking capabilities of VIS. We also developed *VIS Analyzer* (Jung et al., 2010) to provide a graphical interface designed to help domain experts fully utilize the powerful features of VIS without being overwhelmed by its primitive, text-based interface.

VIS has since been upgraded to newer versions, such as academic industrial-strength verification tool (ABC) (Brayton and Mishchenko, 2010) and incremental inductive model checker (IImc) (University of Colorado at Boulder, 2016). There are several differences between VIS and ABC, IImc. For model checking algorithms, ABC is based on and-inverter graphs (AIGs) and IImc is based on IC3, while VIS is based on FSMs. Therefore, VIS can perform sequential equivalence checking based on FSM, while the other two tools cannot. Instead, ABC performs combinational equivalence checking, and IImc cannot perform equivalence checking, since it is optimized for solving model checking problems. Also, while VIS focuses on both synthesis and verification, ABC focuses more on synthesis than verification, and IImc focuses more on verification than synthesis. Many successful studies have demonstrated the value of VIS as one of the most mature equivalence checkers to date.

2.4. COTS software dedication and equivalence checking

As mentioned in Section 1, various standards and guidelines demand the verification of COTS items (especially software in this paper) to accept them in safety-related applications. In EPRI NP-5652 and EPRI TR-106439, there are 4 methods that can be used to accept COTS items as shown:

(1) Special Tests and Inspections
(2) Commercial Grade Survey of Supplier
(3) Source Verification
(4) Acceptable Supplier/Item Performance Record

One of the ways to verify the COTS items is using the formal verification techniques such as equivalence checking. This can be an example for method (1) presented in EPRI NP-5652 and EPRI TR-106439. Also, in IEC 62566, equivalence checking is mentioned as one of the static analysis techniques to verify HPD used in nuclear safety applications.

In recent years, the verification by equivalence checking has become widely accepted in integrated circuits (ICs) fields, in place of the simulation-based techniques. There also exists some studies for logic equivalence checking with formal verification techniques (Hu and Chu, 2023; Ni et al., 2023). Many international standards (RTCA, 2000; International Electrotechnical Commission (IEC), 2000) gradually require to use the formal verification techniques, too. There are several well-known formal verification tools such as *FormalPro* (Siemens, 0000a), *Conformal Equivalence Checker* (Cadence, 2017), *Formality* (Synopsys, 2019) and *VIS* (Brayton et al., 1996) which is embedded in *CVEC*. Times (2001) provides further detailed information on the tools.

3. Related works

There are some research related to dedication of COTS software. In Kim et al. (2010), authors select safety features as functional and performance requirements for method (1). They applied functional benchmarking test method for the COTS dedication of QNX real time operating system (RTOS) to evaluate its sustainability for safety-related applications in the NPP domain. The results demonstrated that QNX RTOS has significant potential as a commercial operating system capable of meeting safety and performance requirements necessary for deployment in digital systems within NPPs. This study also established valuable criteria for comparing QNX RTOS with other commercial RTOS options, providing a reliable basis for selection.

In Kim et al. (2007), authors discuss the process and results of the COTS dedication of the PROFIBUS fieldbus message specification (FMS) driver software. PROFIBUS FMS driver is a high-level communication module used in NPP safety systems such as POSAFE-Q. In this research, methods 1, 2, and 4 presented in EPRI NP-5652 and TR-106439 are applied to PROFIBUS FMS driver software. These methods performed by telecommunication technology association (TTA) and Hilscher Company successfully validated the PROFIBUS FMS driver for safety-related applications. Also, a certification from PROFIBUS national organization (PNO) confirmed the reliability of software. The study highlights the importance of combining multiple methodologies to ensure comprehensive validation of COTS software.

In Jung et al. (2016), one of our previous works, a process for evaluating the suitability of commercial grade softwares indirectly used for FPGA logic synthesis in FPGA development is proposed. COTS software as the target of evaluation are identified and both the safety category of the target system and the usage category of the COTS software are determined. This categorization enables them to assign an appropriate safety category to the COTS software. Based on this, they define a list of dedication criteria for each category. Subsequently, they refine the acceptance criteria and select suitable acceptance methods, including specific V&V techniques. Finally, the selected acceptance methods and techniques are applied to the target software through the dedication process to verify compliance with the acceptance criteria. A case study on *Synopsys Synplify Pro* demonstrates that the proposed evaluation criteria and acceptance process are effective for assessing indirect tools used in FPGA development.

4. A customized VIS-based equivalence checker

⟨Fig. 2⟩ shows an overview of the VIS-based equivalence checker *CVEC*, customized for the *Synopsys Synplify Pro* synthesizer in the *Libero IDE* EDA. There are 4 phases to perform equivalence checking in *CVEC* and each of them is described in the following subsections.

(Phase I) *VerilogtoV4VIS* transformation
(Phase II) *EDIFtoBLIF-MV* transformation
(Phase III) VIS equivalence checking
(Phase IV) Post-analysis and visualization

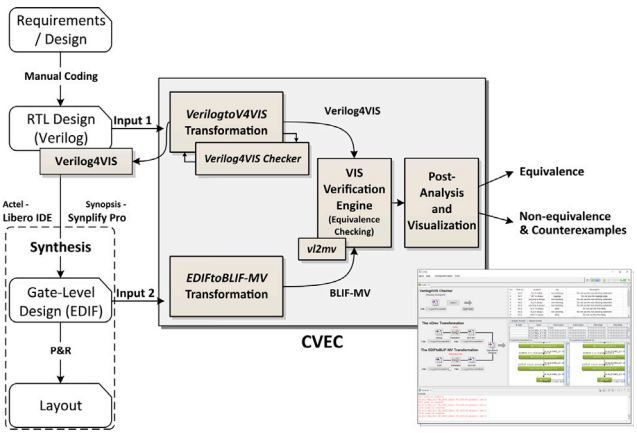

Fig. 2. A customized VIS-based equivalence checker: *CVEC*.

4.1. [Phase I] VerilogtoV4VIS transformation

In the first phase, we first provide Verilog program as the first input for *CVEC*. In VIS, the Verilog HDL front-end called *vl2mv* translator reads most elements of Verilog, then it compiles a subset of Verilog into BLIF-MV. While *vl2mv* reads Verilog, it either ignores or rejects some Verilog elements without notification. Therefore, *CVEC* transforms a typical Verilog program into a customized Verilog for VIS (*i.e., Verilog4VIS (V4VIS)* (Kim et al., 2015)) program, in order to make *vl2mv* read a typical Verilog programs correctly. Also, *CVEC* includes a rule checker called *Verilog4VIS Checker* to verify whether the assumptions and constraints for a Verilog program are satisfied, and recommends appropriate modifications to transform Verilog program into a *Verilog4VIS* program. A *Verilog4VIS* program can be developed through iterative verification and refinement. We introduce below a total of 7 assumptions and constraints for *VerilogtoV4VIS* transformation that must be followed in order to ensure the correct functioning of the VIS verification engine.

Assumptions and constraints
(1) Use the clock *clk* only at the statement *always @(posedge clk)*
(2) Do not use the time delay
(3) Do not use the non-blocking statement
(4) All *reg* variables should be initialized with 0
(5) Do not use the *integer* type variable
(6) Do not use the size of bits when defining *parameter*
(7) Do not use the *negedge* edge

The assumptions and constraints on the Verilog modeling are as follows:

The first assumption states that the VIS interprets a Verilog program as a discrete-time finite state machine with an implicit global clock, *clk*. This alignes the Verilog program with the input format of Symbolic Model Verifier (SMV) used in SMV (Cimatti et al., 1999) or PROMELA language used in SPIN (SPIN, 2016). Since all behaviors within the *always* statements are synchronized to a single clock, *clk*, VIS can execute (or simulate) a BLIF-MV model without requiring the *clk*. Consequently, clocks are not utilized in BLIF-MV, and all *clk*s are ignored. Additionally, all clock-related elements and conditions in *always @(…)* statement are ignored by *vl2mv*. Therefore, the *clk* should only be used for synchronization purposes, such as in *always @(posedge clk)*, and not for any other purposes.

If constraints **(2)** and **(3)** are not satisfied, *vl2mv* may produce a compilation error. According to assumption **(1)**, no time delays for gates or wires are allowed yet, since the BLIF-MV does not contain an element to implement time delays. Additionally, non-blocking statements are not permitted; all statements must be written as blocking statements.

Constraint **(4)** requires that all *reg* variables to be initialized with 0. No other value is allowed, as *Libero IDE* assigns a default value of 0 to the EDIF elements corresponding to the *reg* variables, regardless of their initialization. However, VIS requires all *reg* variables to be initialized with an appropriate value. Our solution is to initialize all *reg* variables to 0, regardless of their original values. We anticipate that this will not result in any side-effect, since all DI&C systems have a distinguished procedure for the system boot-up, such as temporarily ignoring all output values (*i.e.,* shutdown alarms) during the initial phase of boot-up.

We cannot use *integer* type variables in the VIS verification, since the VIS lacks a mechanism to initialize them, as specified in constraint **(5)**. Our suggestion is to replace an *integer* variable with a *reg [31:0]* variable during the initialization phase, as it functions in exactly the same way. Constraint **(6)** also restricts the detailed definition of *parameter* variables (*e.g.,* using [15:0] of bit array), as it may implicitly lead to a loss of information. We cannot use the *negedge* edge as constraint **(7)**, since VIS only supports the *posedge* edge.

4.2. [Phase II] EDIFtoBLIF-MV transformation

In the second phase, we provide the second input, an EDIF program for *CVEC*. *Synopsys Synplify Pro* in *Libero IDE* synthesizes an EDIF program from the *Verilog4VIS* program. Then *CVEC* transforms mechanically synthesized gate-level design of EDIF (*i.e.,* Netlist) into a BLIF-MV program. We provide a transformation process of 3 steps and translation rules of 6 categories, which are specialized for the EDIF format of *Libero IDE*. We also provide the mechanical *EDIFtoBLIF-MV* translator in *CVEC* to implement the process and rules.

The transformation from EDIF to BLIF-MV includes a three-step process, consisting of *(II-1) Parsing*, *(II-2) Pre-processing* and *(II-3) Translation*. The first step (II-1) parses an EDIF file and stores it in an internal data structure. All EDIF elements except auxiliary information such as the *cell* information are converted into an internal data structure. The next step (II-2) performs a pre-processing such as deleting unnecessary information in the internal data structure. For example, as all *clk*s in Verilog are ignored in BLIF-MV, we need to delete these information. Also, all ports not used but defined in an EDIF should explicitly have a default value of 0. The last step (II-3) transforms the pre-processed internal data into a BLIF-MV format according to the translation rules we developed.

The 3-step transformation transforms all of the information of *cells* in an EDIF into an equivalent BLIF-MV format, which serves as an internal input front-end of the VIS verification system. The whole process was also implemented into a CASE tool *EDIFtoBLIF-MV*, embedded in *CVEC*. The details of each transformation step, except (II-1 *Parsing*), is as follows.

4.2.1. (II-2) Pre-processing

After parsing an EDIF file into an internal data structure of *CVEC*, we need to perform a few pre-processing. First, all informations related to the clock *clk* should be deleted in accordance with the assumption (1) in *[Phase I]*. As EDIF is close to hardware layout, it includes the voltage *VCC* and ground *GND* cells, even if they are not presented in Verilog programs. We need to explicitly assign 1 and 0 to their output port, respectively. All unused ports (*i.e.,* bits) should have an explicit value by connecting them to the *GND* cell. If a port is not connected to any net, the translated BLIF-MV cannot be processed by the VIS. The processes included in this step is represented below.

Pre-processing

(1) Delete all information (*e.g.,* port, instance and net) related to the clock *clk*
(2) Assign 1 to *VCC*
(3) Assign 0 to *GND*
(4) Assign 0 to all unused ports

4.2.2. (II-3) Translation into BLIF-MV

CVEC introduces translation rules from EDIF to BLIF-MV, and implements an automatic translation, EDIFtoBLIF-MV, which is a follow-up study of Lee (2013). The translation rules are divided into 5 categories, and each category is explained and described below as pseudo-code. The examples of these translations are demonstrated in Appendix: Examples of The Translations from EDIF to BLIF-MV.

[Rule 1] *Translation of cells*

Rule 1 defines a rule to express external structure of an EDIF unit, called *cell*, in BLIF-MV format. Each *cell* in working library is translated directly into a *.model* of BLIF-MV. The name of cell in EDIF would be the name of model in BLIF-MV. In pseudo-code, it would be expressed as in ⟨Fig. 3⟩.

```
1:  FUNCTION cellToModel(edifCell):
2:    cellName = edifCell.name
3:    blifModel = ".model" + cellName + "\n"
4:    RETURN blifModel
```

Fig. 3. Translation rule for cells.

[Rule 2] *Translation of ports and arrays*

Rule 2 defines the translation of each *port* in EDIF into *.inputs* or *.outputs* of BLIF-MV. In pseudo-code, it would be expressed as in ⟨Fig. 4⟩. The input and output ports in an EDIF cell are converted into .inputs and .outputs in BLIF-MV. If the port type of an input port is an *array*, the port name becomes the base name of the .inputs in BLIF-MV; EDIF does not have an array of the output ports. Additionally, in EDIF, an array of the input ports has a starting index (`startIndex`) and an ending index (`endIndex`). In BLIF-MV, the port names and their corresponding index values within the specified range are added to the .inputs section. On the other hand, if the port type is *not* an array, the port name is simply converted into the name of .inputs or .outputs in BLIF-MV.

```
1:  FUNCTION portToBLIFMV(edifCell):
2:    blifModel = ""
3:
4:    FOR port IN edifCell.ports:
5:      IF port.type == "array":      # only input port can have array
6:        baseName = port.name
7:        startIndex = port.startIndex IF port.startIndex IS NOT NONE ELSE 0
8:        endIndex = port.endIndex IF port.endIndex IS NOT NONE ELSE (port.size - 1)
9:
10:       blifModel += ".inputs "
11:
12:       IF startIndex <= endIndex:
13:         FOR i FROM startIndex TO endIndex:
14:           blifModel += baseName + "[" + i + "]"
15:       ELSE:
16:         FOR i FROM startIndex DOWNTO endIndex:
17:           blifModel += baseName + "[" + i + "]"
18:
19:       blifModel += "\n"
20:
21:     ELSE:
22:       direction = port.direction + "s"
23:       blifModel += "." + direction + " " + port.name + "\n"
24:
25:   RETURN blifModel
```

Fig. 4. Translation rule for ports and arrays.

[Rule 3] *Translation of property functions*

Rule 3 defines two ways of translating output ports of EDIF into BLIF-MV. In pseudo-code, it can be expressed as in ⟨Fig. 5⟩. The *output port* in the cell can include a keyword *property function*, which defines the function of the cell. If the cell is a sequential cell, function in an EDIF is translated into *.latch* of BLIF-MV, and a keyword *.reset* has to be defined before the *.latch*. This is implemented by the function `sequentialToBLIFMV(edifCell)`, shown from line 14 to line 28 in ⟨Fig. 5⟩. An example for the sequential D-Flip-Flop (DFF) cell is included in ⟨Fig. A.3⟩ in Appendix.

On the other hand, if the cell is a combinational cell, the function is expressed as a *truth table* using the keyword *.table*. The *property function* in a cell defines the *truth table of functionality*. In the case of combinational cells, logic operators such as + (OR), &(AND), ∧(XOR) and !(NOT) can be used. Also, multiple logic operators can be used to define the property function. For example, combinational cell can have the property function (*string* "A & (B + C)"), which can be expressed as *ABC* + & in postfix notation to be translated into BLIF-MV. The examples for the combinational cell are demonstrated in the second and the third example in ⟨Fig. A.3⟩.

```
1:  FUNCTION getPortsByFunction(edifCell, functionName):
2:    inputPorts = []
3:    outputPorts = []
4:
5:    FOR port IN edifCell.ports:
6:      IF port.function == functionName:
7:        IF port.direction == "input":
8:          inputPorts.append(port.name)
9:        ELSE IF port.direction == "output":
10:         outputPorts.append(port.name)
11:
12:   RETURN (inputPorts, outputPorts)
13:
14: FUNCTION sequentialToBLIFMV(edifCell):
15:   blifModel = ""
16:   functionName = ""
17:
18:   FOR property IN edifCell.properties:
19:     IF property.name == "function":
20:       functionName = property.value
21:
22:     IF property.type == "isSequential" && property.value == "TRUE":
23:       inputPorts, outputPorts = getPortsByFunction(edifCell,functionName)
24:       inputOutputPorts = " ".join(inputPorts + outputPorts)
25:
26:       blifModel += ".reset " + inputOutputPorts + " 0\n"
27:       blifModel += ".latch " + functionName + " " + inputOutputPorts + "\n"
28:
29:     RETURN blifModel
30:
31:
32: FUNCTION functionPropertiesToBLIFMV(edifCell):
33:   blifModel = ""
34:   functionName = ""
35:
36:   FOR property IN edifCell.properties:
37:     IF property.name == "function":
38:       functionName = property.value
39:       inputPorts, outputPorts = getPortsByFunction(edifCell, functionName)
40:
41:       FOR inputPort IN inputPorts:
42:         FOR outputPort IN outputPorts:
43:           blifModel += ".table " + inputPort + " " + outputPort + "\n"
44:
45:       blifModel += ".default 0\n"
46:
47:     RETURN blifModel
```

Fig. 5. Translation rule for property functions.

[Rule 4] *Translation of cells including instances*

Each cell in EDIF has *contents* to implement its functionality. **[Rule 4]** defines the case where an EDIF cell includes *instance* inside its contents. The psuedo-code to express **Rule 4** is shown in ⟨Fig. 6⟩. `cellRefName` in line 6 of the code represents the referenced cell type in EDIF. When an instance is present within the cell, its name and the cell it represents determine the name of `.subckt` in the BLIF-MV format, as seen in line 8 of the code. From line 13 to line 20, each *connection* in the instance is represented, considering the multiple input and output ports in EDIF.

[Rule 5] *Translation of cells including connections*

Rule 5 connects the called cells with each other from input ports to output ports. The pseudo-code of **Rule 5** is represented in ⟨Fig. 7⟩ If the *net* in EDIF contains only the *joined* keyword, the code from line 7 to line 18 is applied. The case of calling *member* cells using the *renamed* keyword is represented in from line 20 to line 31 of the code.

```
1:  FUNCTION instanceToBLIFMV(edifCell):
2:    blifModel = ""
3:
4:    FOR instance IN edifCell.instances:
5:      instanceName = instance.name
6:      cellRefName = instance.cellRef
7:
8:      subcktLine = ".subckt " + cellRefName + " " + instanceName
9:
10:     inputPorts = []
11:     outputPorts = []
12:
13:     FOR connection IN instance.connections:
14:       portName = connection.portName
15:       netName = instanceName + "_" + portName
16:
17:       IF connection.type == "input":
18:         inputPorts.append(portName + " = " + netName)
19:       ELSE IF connection.type == "output":
20:         outputPorts.append(portName + " = " + netName)
21:
22:     subcktLine += " " + " ".join(inputs) + " " + " ".join(outputs)
23:     blifModel += subcktLine + "\n"
24:
25:   RETURN blifModel
```

Fig. 6. Translation rule for cells including instances.

```
1:  FUNCTION netsToBLIFMV(edifCell):
2:    blifModel = ""
3:
4:    FOR net IN edifCell.nets:
5:      netName = net.name
6:
7:      IF net.type == "joined":
8:        inputInstances = net.inputInstances
9:        inputPorts = net.inputPorts
10:       outputInstances = net.outputInstances
11:       outputPorts = net.outputPorts
12:
13:       blifModel += ".table " + " ".join([
14:         inputInstances[i] + "_" + inputPorts[i] for i IN range(len(inputInstances))
15:       ]) + " -> "
16:       blifModel += " ".join([
17:         outputInstances[i] + "_" + outputPorts[i] for i IN range(len(outputInstances))
18:       ]) + "\n"
19:
20:     ELSE IF net.type == "renamed":
21:       originalName = net.originalName
22:       order = str(net.order)
23:       members = net.members
24:
25:       blifModel += ".table " + originalName + order + " -> \n"
26:
27:       FOR member IN members:
28:         memberName = member.name
29:         memberOrder = str(member.order)
30:
31:         blifModel += memberName + memberOrder + "\n"
32:
33:   RETURN blifModel
```

Fig. 7. Translation rule for cells including connections.

⟨Fig. 8⟩ depicts the *EDIFtoBLIF-MV* translator, implementing the proposed transformation process from EDIF to BLIF-MV. As other translators and compilers, it has simple GUI to read an input EDIF file and to store the transformed output BLIF-MV file. The console at the bottom shows the transformation process in steps, *i.e., EDIF Opening ⇒ Parsing ⇒ Pre-processing ⇒ Translation ⇒ BLIF-MV Saving*. It is now embedded into *CVEC*.

4.3. [Phase III] VIS equivalence checking

In this phase, *CVEC* first transforms *Verilog4VIS* program into BLIF-MV program using *vl2mv* translator in VIS. Then it performs VIS-based equivalence checking upon two BLIF-MV programs, transformed from the *Verilog4VIS* program and the EDIF program, respectively. Then it automatically executes a series of VIS commands to perform sequential equivalence checking. ⟨Fig. 9⟩ shows a set of VIS commands to perform the sequential equivalence checking upon the *Verilog4VIS* and the EDIF programs used in the case study in Section 4. *CVEC* automates a series of all commands with a simple graphical user interface as depicted in

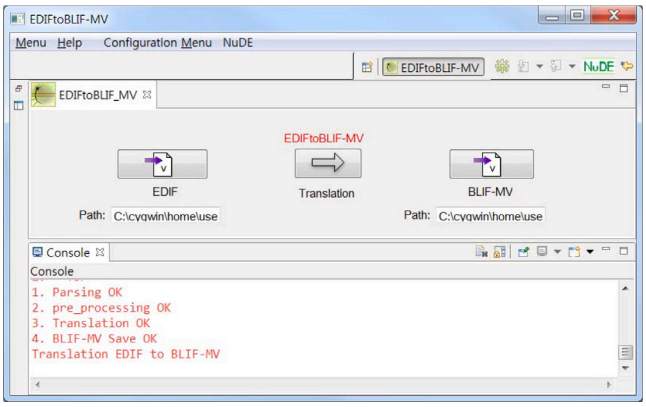

Fig. 8. The *EDIFtoBLIF-MV* translator.

⟨Fig. 10⟩. In ⟨Fig. 10⟩, VIS equivalence checking determines that the two programs are not sequentially equivalent as "*Networks are NOT sequentially equivalent.*". It also shows a verification result of 6 steps (*i.e.,* a counterexample) which simulates an inequivalent state from the initial state step-by-step.

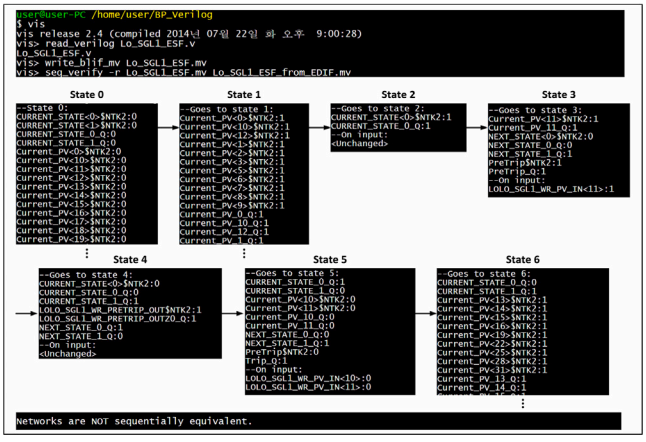

Fig. 9. A series of the VIS commands and a verification result from the VIS (excerpted).

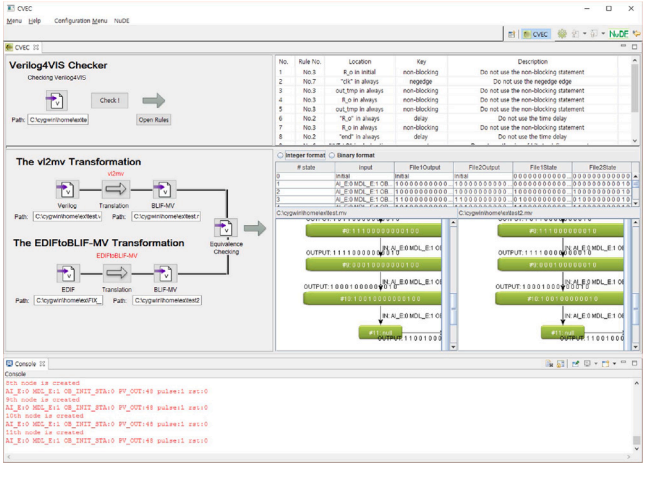

Fig. 10. The CVEC's main window.

An in-depth analysis of the counterexample allows a detailed examination of the cause of the inequivalence. The example demonstrates that the VIS discards overflowed values during the *vl2mv* translation,

whereas the *Synopsys Synplify Pro* synthesizes the overflowed parameter values fairly well. To enhance understanding, we excerpted and modified the verification results graphically, as VIS lacks a graphical interface. We can notice that all repeated information is omitted and we have to reorganize it in order to understand the error trace clearly. The verification results are then analyzed in the following phase.

4.4. [Phase IV] Post-analysis and visualization

Lastly, *CVEC* processes the results of the VIS-based equivalence checking and performs additional analyses. If the verification succeeds (*i.e.,* two BLIF-MV programs are equivalent), we can assure that the synthesis process from Verilog to EDIF (*i.e.,* Netlist) was performed correctly for the given Verilog program. If the verification fails (*i.e.,* two BLIF-MV programs are *not* equivalent), it indicates that there was an issue in the synthesis process. In such cases, VIS provides a scenario comprising a sequence of inputs and internal states/variables that leads to the inequivalent state, *i.e.,* a counterexample. Unfortunately, VIS presents it on the console in a textual format, omitting repeated or unchanged details, which makes understanding the prompt almost impossible. However, *CVEC* offers post-analysis functions to reorganize and visualize the data in various formats to enhance understanding.

⟨Fig. 10⟩ shows the *CVEC* GUI performing all phases on a single window. *Verilog4VIS Checker* and *EDIFtoBLIF-MV* are embedded, allowing all transformations and equivalence checking to be performed mechanically and seamlessly with just a few button clicks. The series of the VIS commands and analysis on the verification results shown in ⟨Fig. 9⟩ can be performed mechanically by *CVEC*, as illustrated.

If the verification fails, *CVEC* analyzes the counterexample (*i.e.,* reconstructs skipped information) and visualizes it in various formats such as flow-chart, table and text on console. The visualization helps identifying the cause of the inequivalence between the Verilog design and the synthesized EDIF. We are working on improving the post-analysis and visualization to provide more convenience and useful information to analyzers.

5. Case study

We performed a case study with two types of RPS BP software in DI&C systems of Korean nuclear power plants, in order to demonstrate the effectiveness and potential of the customized VIS-based equivalence checker, *CVEC*. Using *CVEC*, the case study aimed to demonstrate the correct functioning of the commercial FPGA logic synthesis tool *Synopsys Synplify Pro* in the *Libero IDE* EDA environment. If *CVEC* succeeds to prove the behavioral equivalence between Verilog and EDIF, we can claim that the tools function correctly, at least for the given input program. ⟨Fig. 11⟩ summarizes the whole process of the case study.

The first example is an FBD program for a preliminary version of KNICS APR-1400 RPS BP (Korea Atomic Energy Research Institute (KAERI), 2006), while the second example is a Verilog program of the PLD-based RPS BP (Choi and Lee, 2012). The first one is much complicated and detailed than the second one, since it is kind of a mock-up of commercial nuclear power plants. They both consist of 18 independent shutdown logics. The BP softwares read 18 external sensor inputs and make a decision whether to shutdown nuclear reactors or not, periodically. Since the BP software is one of the most safety-critical components in DI&C of nuclear power plants, standards and regulations (U.S. Nuclear Regulatory Commission, 1996b, 2012; Electric Power Research Institute (EPRI), 2014, 1996) strictly encourage developers to verify its correct and safe functioning.

`[Case study I]` begins with an FBD program, as the implementation platform for the KNICS APR-1400 RPS BP is a programmable logic controller (PLC). We first had to transform the FBD program into a

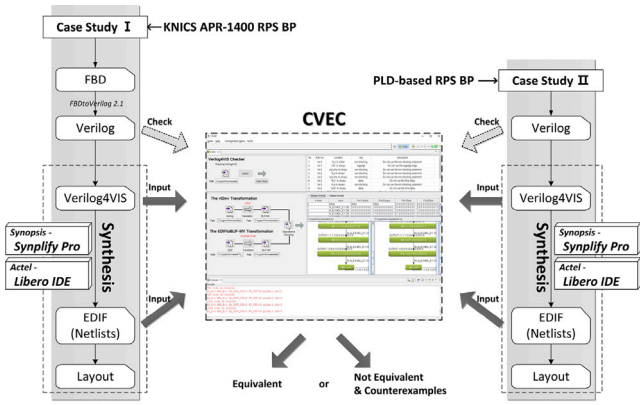

Fig. 11. An overview of the case study with two examples.

behaviorally-equivalent Verilog program. *FBDtoVerilog v2.1* embedded in *FBD Editor* (Lee et al., 2014) reads the FBD program and mechanically translates it into a behaviorally-equivalent Verilog program. On the other hand, `[Case study II]` starts from a Verilog program, and we do not need such transformation.

5.1. [Case study I] KNICS APR-1400 RPS BP

It uses 5 shutdown logics of the KNICS APR-1400 RPS BP, *e.g., fixed set-point rising/falling trip*, *variable set-point rising/falling trip* and *manual reset trip*. They can represent the whole 18 shutdown logics in BP. The maximum size of all variables in the BP is 16 bits. ⟨Table 1⟩ summarizes important features of the input Verilog program and the synthesized EDIF.

The table also presents the verification result and the time taken by the VIS verification system. In case of the simple logics such as *fixed set-point rising/falling logics*, the VIS could perform the equivalence checking in reasonable time, *e.g.,* 198.22 and 30.11 s, respectively. The transformations time is not considered here. On the other hand, for more complex logics, such as *variable set-point rising/falling trip*, the maximum size of all variables is 9 bits. The VIS successfully completed the equivalence checking for these logics in 5613 s (about 90 min). For all 5 logics, the VIS proved that the Verilog and the subsequently synthesized EDIF are equivalent for all cases.

We also performed a comparison analysis with the commercial equivalence checker *FormalPro*. As *Formal Pro* cannot work for the *Synopsys Synplify Pro*, we had to use *Precision FPGA*. Although the comparison is not entirely precise, we expect that it is generally acceptable. *FormalPro* produced the same result as *CVEC* for all logic cases. ⟨Fig. 12⟩ indicates that the commercial tool is faster than the proposed technique in several orders of magnitude. As shown in ⟨Fig. 12⟩ and ⟨Table 1⟩, the *CVEC* verification performance decreases sharply as the number of *reg* and *latch* increase. As *CVEC* uses the VIS verification engine, it is not straightforward to improve the verification performance (*i.e.,* time) dramatically, but we are planning to improve the verification performance of *CVEC*, including development of an equivalence checking engine from scratch.

In summary, *CVEC* could judge that "*The FPGA logic synthesis tool worked correctly for the Verilog programs of 5 trip logics in KNICS APR-1400 RPS BP.*" Further consideration to improve the verification performance will be addressed in Section 4.3.

Table 1
A summary information of the case study I.

	model	comb	pi	po	latch	const	edges	time(sec)	Equivalent?
Fixed Set-Point Rising Trip									
10bits	Verilog	461	16	22	37	39	1075	3.48	YES
	EDIF	1216	16	22	36	38	1709		
11bits	Verilog	501	17	24	40	42	1170	5.55	YES
	EDIF	1528	17	24	40	42	2150		
12bits	Verilog	541	18	26	43	45	1265	8.90	YES
	EDIF	1786	18	26	44	46	2511		
13bits	Verilog	583	19	28	46	50	1362	18.49	YES
	EDIF	1782	19	28	45	47	2504		
14bits	Verilog	623	20	30	49	53	1457	67.64	YES
	EDIF	2035	20	30	49	51	2865		
15bits	Verilog	663	21	32	52	56	1552	162.17	YES
	EDIF	2148	21	32	52	54	3016		
16bits	Verilog	703	22	34	55	59	1647	198.22	YES
	EDIF	2341	22	34	56	58	3285		
Fixed Set-Point Falling Trip									
10bits	Verilog	792	15	44	58	61	1812	4.58	YES
	EDIF	2224	15	44	58	60	3109		
11bits	Verilog	863	16	48	63	66	1978	5.70	YES
	EDIF	2506	16	48	63	65	3500		
12bits	Verilog	938	17	52	68	75	2148	6.86	YES
	EDIF	2703	17	52	66	68	3767		
13bits	Verilog	1009	18	56	73	80	2314	8.81	YES
	EDIF	2784	18	56	71	73	3880		
14bits	Verilog	1080	19	60	78	85	2480	15.45	YES
	EDIF	3160	19	60	79	78	4403		
15bits	Verilog	1151	20	64	83	90	2646	18.46	YES
	EDIF	3531	20	64	82	84	4928		
16bits	Verilog	1222	21	68	88	95	2812	30.11	YES
	EDIF	3682	21	68	85	87	5131		
Manual Reset Falling Trip									
7bits	Verilog	650	15	30	44	50	1467	16.56	YES
	EDIF	1994	15	30	44	46	2801		
8bits	Verilog	726	16	34	49	51	1649	640.05	YES
	EDIF	2009	16	34	49	51	2828		
9bits	Verilog	813	17	38	54	63	1842	1160.17	YES
	EDIF	2039	17	38	54	56	2849		
10bits	Verilog	893	18	42	59	68	2028	1295.10	YES
	EDIF	2234	18	42	60	62	3129		
11bits	Verilog	971	19	46	64	71	2212	11216.6	YES
	EDIF	2806	19	46	67	69	3943		
Variable Set-Point Falling Trip									
7bits	Verilog	724	12	44	55	65	1664	89.65	YES
	EDIF	1597	12	44	48	60	2214		
8bits	Verilog	820	13	50	62	75	1866	285.10	YES
	EDIF	1885	13	50	54	56	2629		
9bits	Verilog	916	14	56	69	85	2088	4677.80	YES
	EDIF	2026	14	56	60	62	2810		
Variable Set-Point Rising Trip									
7bits	Verilog	726	12	44	55	67	1646	323.53	YES
	EDIF	2745	12	44	49	51	3853		
8bits	Verilog	819	13	50	62	74	1865	126.88	YES
	EDIF	2167	13	50	54	56	3029		
9bits	Verilog	916	14	56	69	85	2088	5613.66	YES
	EDIF	2442	14	56	63	65	3395		

5.2. [Case study II] The RPS trip logics based on PLD technology

In this case study, *CVEC* uses the Verilog programs developed for an experimental RPS (Choi and Lee, 2012), based on Programmable Logic Device (PLD) technology such as FPGA. The target is a system

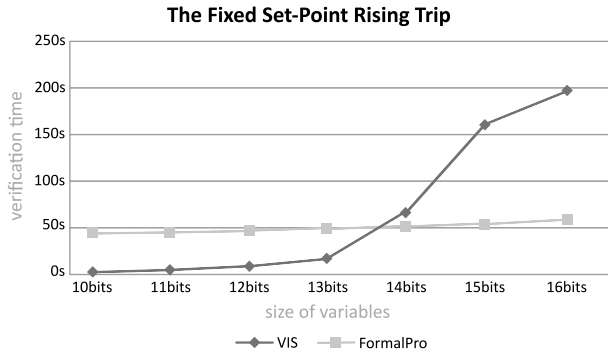

Fig. 12. The comparison of verification time in seconds (*CVEC* with *VIS* vs. *FormalPro*) for *fixed set-point rising trip logic*.

with 18 trip logics as commercial RPS BPs, but it is an experimental system consisting of fundamental functionalities. ⟨Table 2⟩ summarizes important features of all inputs and outputs and verification results of 18 trip logics. All logics were verified through *CVEC*, and the comparison with *Formal Pro* was also performed.

CVEC found two non-equivalent cases for *Lo_SG1_ESF_FSF* and *Lo_SG2_ESF_FSF* trip logics, whereas *FormalPro* with *Precision FPGA* judged that they were equivalent. An in-depth analysis found the reason that the VIS (actually *vl2mv* in the VIS) has an error at interpreting `parameters` in Verilog. If a conditional statement includes an arithmetic operation of `parameters`, such as + or -, the VIS restricts the size of the result of the arithmetic operation into the maximum size of the parameters, without notification.

For example, the Verilog program in ⟨Fig. 13⟩ reads an input *in* [3:0] and checks "*if(in > a+b)*". The parameter *a* and *b* are defined with 3 of 2 bits (*i.e,* b11). The condition statement is then equivalent to "*if(in > 6)*". However, the *vl2mv* translator interprets the statement as "*if(in > 2)*" since the maximum size of two parameters is 2 bits and the maximum size of the + operation is also regarded as 2 bits. 6 (b1010) is truncated into 2 (b10).

The figure also shows the simulation trace of the Verilog program with the VIS and the subsequently synthesized EDIF with *ModelSim*. In the VIS simulation of the Verilog program, *out* gets changed into 1 when the input *in* becomes greater than 2 not 6. The value of *out* (*i.e.*, the calculation result) can be recognized at the next simulation step in the VIS. On the other hands, *ModelSim* shows that the *out* is only changed into 1 when *in* is greater than 6, as intended by the Verilog program.

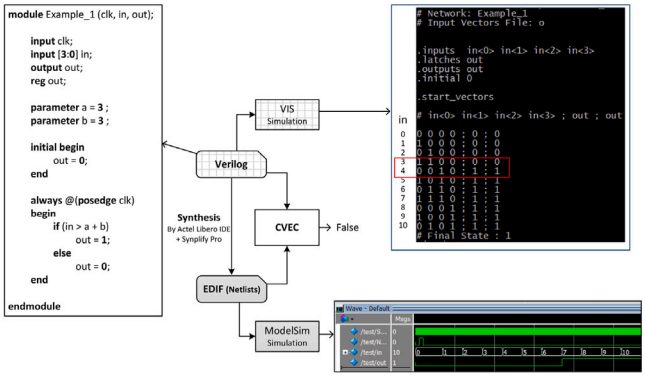

Fig. 13. A Verilog program demonstrating the reason of the non-equivalence.

In summary, *CVEC* checked 18 trip logics of Verilog program for an experimental RPS logics in Korea, and found 2 cases of non-equivalence between the Verilog programs and subsequently synthesized EDIFs.

Table 2
A summary information of the case study II.

RPS BP											
	name	model	comb	pi	po	latch	const	edges	time (sec)	Equivalent? (FormalPro)	Etc.
1	Hi_CNT_PRS_FSR	Verilog	417	33	2	40	42	558	2.419	YES	32 bits
		EDIF	17	4	2	5	5	34		(YES)	
2	Hi_Local_Power _Density_OF	Verilog	17	4	2	5	5	34	0.074	YES	1 bits
		EDIF	43	4	2	5	7	53		(YES)	
3	Hi_Log_Power_FSR	Verilog	550	36	3	45	56	1311	3.443	YES	32 bits
		EDIF	472	36	3	45	47	636		(YES)	
4	Hi_PZR_Pressure_FSR	Verilog	527	33	2	40	45	1259	3.208	YES	32 bits
		EDIF	448	33	2	40	42	606		(YES)	
5	Hi_SGL1_NR_FSR	Verilog	525	33	2	40	47	1227	2.260	YES	32 bits
		EDIF	417	33	2	39	41	561		(YES)	
6	Hi_SGL2_NR_FSR	Verilog	463	33	2	40	46	1083	2.480	YES	32 bits
		EDIF	440	33	2	39	41	597		(YES)	
7	HIHI_CPRS_NR_FSR	Verilog	540	33	2	40	58	1314	3.148	YES	32 bits
		EDIF	417	33	2	40	42	560		(YES)	
8	Lo_DNBR_OF	Verilog	17	4	2	5	5	34	0.094	YES	1 bits
		EDIF	43	4	2	5	7	53		(YES)	
9	Lo_PZR_Pressure_MRF	Verilog	613	13	4	68	72	1443	Over	YES	Size Down
		EDIF	1362	13	4	67	69	1914	10h...	(YES)	(32 --> 8)
10	Lo_RC1_FLW_VSF	Verilog	746	11	2	48	56	2007	46.636	YES	Size Down
		EDIF	1421	11	2	48	50	2010		(YES)	(48 --> 10)
11	Lo_RC2_FLW_VSF	Verilog	746	11	2	48	56	2007	42.566	YES	Size Down
		EDIF	1421	11	2	48	50	2010		(YES)	(48 --> 10)
12	Lo_SG1_PRS_MRF	Verilog	542	11	2	59	61	1279	473.498	YES	Size Down
		EDIF	1260	11	2	58	60	1776		(YES)	(48 --> 8)
13	Lo_SG2_PRS_MRF	Verilog	542	11	2	59	61	1279	589.266	YES	Size Down
		EDIF	1260	11	2	58	60	1776		(YES)	(48 --> 8)
14	Lo_SGL1_ESF_FSF	Verilog	525	33	2	40	47	1277		NO	32 bits
		EDIF	466	33	2	40	42	610		(YES)	
15	Lo_SGL1_RPS_FSF	Verilog	525	33	2	40	47	1227	3.20	YES	32 bits
		EDIF	457	33	2	40	42	598		(YES)	
16	Lo_SGL2_ESF_FSF	Verilog	525	33	2	40	47	1277		NO	32 bits
		EDIF	466	33	2	40	42	610		(YES)	
17	Lo_SGL2_RPS_FSF	Verilog	525	33	2	40	47	1227	3.18	YES	32 bits
		EDIF	457	33	2	40	42	598		(YES)	
18	Variable_OverPower_VSR	Verilog	1185	17	2	56	71	3100	437.49	YES	Size Down
		EDIF	2912	17	2	56	58	4128		(YES)	(48 --> 16)

* Note
FSR - Fixed Set-Point Rising Trip
FSF - Fixed Set-Point Falling Trip
MRF - Manual Reset Falling Trip
OF - On/OFF
VSR - Variable Set-Point Rising Trip
VSF - Variable Set-Point Falling Trip

An in-depth analysis, however, found that the two non-equivalence resulted from incorrect operation of the VIS (actually the *vl2mv* translator), and a simple remedy such as replacing "$a + b$" with 6 could resolve the non-equivalence. Therefore, *CVEC* could judge that "*The FPGA logic synthesis tool functioned correctly for the Verilog programs of 18 trip logics in an experimental RPS BP.*" The verification performance was reasonably acceptable for the PLD-based 18 trips logics of RPS BP, since the system do not include implementation details such as operation bypass, periodic tests or heart bits.

5.3. Further consideration on the verification performance

The proposed technique *CVEC* uses the verification engine of the VIS to prove the behavioral (*i.e.,* sequential not combinational) equivalence between RTL designs in Verilog and Netlists in EDIF. VIS is an open-source, widely-used tool that has been successfully utilized and validated by many researchers in public. Those success stories clearly demonstrate the value of VIS as one of the mature equivalence checker among various tools in HWMCC (Hardware Model Checking Competition) (Johannes Kepler University Linz, 2015), even though the tool is since been upgraded to newer versions (ABC (Brayton and Mishchenko, 2010) and IImc (University of Colorado at Boulder, 2016)).

There are several commercial formal verification tools which can be used for our purpose – "*Correctness verification of commercial FPGA logic synthesis tools.*" *FormalPro* (Siemens, 0000a), *Conformal Equivalence Checker* (Cadence, 2017) and *Formality* (Synopsys, 2019) are the candidates. However, they require additional information such as register/variable matching or libraries from synthesis tools. Which means, we cannot use the tools without vendor's support. For instance, we cannot use *FormalPro* for *Libero IDE* with *Synopsys Synplify Pro* synthesizer, which was the combination of the project we were working on.

Since *CVEC* is based on the VIS verification engine, it is tightly coupled with the input front-end BLIF-MV as well as the sequential equivalence checking engine of the VIS. As sequential equivalence checking has inherent limitations on the size of targets and verification speed (Brand, 2003), commercial tools often use combinational equivalence checking hierarchically through decoupling sequential logics into combinational ones (Rahim et al., 2012). They also use additional information such as Formal Verification Interface (FVI), Tcl script, Synopsys setup Verification File (SVF) and various libraries to make the strategy possible. *CVEC*, on the other hand, requires no additional information. It only needs transformation of a Verilog program into a *Verilog4VIS* program, which is the results from the limitation due to the in-house translator *vl2mv* in the VIS.

We are planning to improve the verification performance of *CVEC* in two ways. First, we plan to explore the use of combinational equivalence checking instead of sequential one, following the approach commonly adopted by typical commercial equivalence checkers. The decision to use the VIS verification engines or to develop a new solution from scratch should be made after conducting an in-depth analysis. Secondly, the input front-end BLIF-MV can be replaced with AIGER (Johannes Kepler University Linz, 2011), which can handle various input formats successfully. AIGER is a format, library, and set of utilities for And-Inverter Graphs (AIGs), providing a convenient and compact representation for circuits. It is known to describe combinational and sequential circuits efficiently.

6. Conclusion and future work

This paper proposes a customized VIS-based equivalence checker *CVEC* which can contribute to the correctness demonstration of the combination – the *Synopsys Synplify Pro* synthesizer in the *Libero IDE* EDA. *CVEC* can formally check the behavioral equivalence between a Verilog HDL and a Netlist (*i.e.,* EDIF) synthesized in this environment. It uses the VIS verification system as a verification engine, and also provides two model transformations and a rule checker. It also provides a graphical interface to perform all transformations mechanically and analyze verification results visually and efficiently. If the formal verification with *CVEC* succeeds, then we can claim that the logic synthesis from Verilog into EDIF worked correctly at least for the given Verilog program. Therefore, *CVEC* can verify the functional correctness of FPGA synthesis tools through equivalence checking, which is one of the ways to dedicate COTS items used in safety-related applications.

In order to demonstrate the effectiveness and applicability of the proposed technique, we performed a case study with two examples of RPS BP (Bistable Processor) programs excerpted from Korean nuclear power plants. We also tried to compare the performance of *CVEC* with a commercial verification tool *FormalPro*, although we had to change the logic synthesis tool into *Precision FPGA*. The comparison indicates that the commercial tool is faster than the proposed technique in many orders of magnitude. It is not straightforward to improve the verification performance dramatically, since *CVEC* uses the VIS verification engine and the input front-end if VIS, *vl2mv*. We are planning to strategically adopt combinational equivalence checking, similar to other commercial tools, and are also considering replacing BLIF-MV with AIGER. We also have a plan to develop a dedication of different commercial grade FPGAs, focusing on addressing different types of FPGAs in the SMR design to mitigate the common cause failure (CCF) problem.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Nuclear Safety Research Program through the Regulatory Research Management Agency for SMRS (RMAS) and the Nuclear Safety and Security Commission (NSSC) of the Republic of Korea (No. RS-2024-00509643).

Appendix. Examples of the translations from EDIF to BLIF-MV

In this appendix, the examples of the translations according to EDIF to BLIF-MV translation rules, **Rule 1** to **Rule 5**, presented in Section 4.2.2 are demonstrated. There are total of 11 examples, especially more than one example for **Rule 2**, **Rule 3**, and **Rule 5**, considering various cases dealth with in the same rule.

[Case 1] Cells

⟨Fig. A.1⟩ demonstrates an example of translation of cells according to **Rule 1**. When the name of cell in EDIF is `example_cell`, then the name of the model in BLIF-MV is also `example_cell`. This cell would be the cell inside working library, `example_library`.

EDIF	BLIF-MV

(library example_library
  (cell example_cell
  ...
)
)
	.model example_cell
	  ...
	.end

Fig. A.1. Example of translation of cells.

EDIF	BLIF-MV

(cell example_cell
  (view example_view
  (interface
  (port input_A (direction INPUT))
  (port input_B (direction INPUT))
  (port output_X (direction OUTPUT))
  ...
)
)
)
	.model example_cell
	.inputs input_A
	.inputs input_B
	.outputs output_X
	  ...
	.end

EDIF	BLIF-MV

(cell example_cell
  (view example_view
  (interface
  (port
  (array input 4 (direction INPUT))
  (port output (direction OUTPUT))
  ...
)
)
)
	.model example_cell
	.inputs input[0] input[1] input[2] input[3]
	.outputs output
	  ...
	.end

EDIF	BLIF-MV

(cell example_cell
  (view example_view
  (interface
  (port
  (array input 3 downto 0 (direction INPUT))
  (port output (direction OUTPUT))
  ...
)
)
)
	.model example_cell
	.inputs input[3] input[2] input[1] input[0]
	.outputs output_port
	  ...
	.end

EDIF	BLIF-MV

(cell example_cell
  (view example_view
  (interface
  (port
  (array in_bus 1 to 4 (direction INPUT))
  (port output (direction OUTPUT))
  ...
)
)
)
	.model example_cell
	.inputs input[1] input[2] input [3] input [4]
	.outputs output
	  ...
	.end

Fig. A.2. Examples of translation of ports and arrays.

[Case 2] Ports and arrays

⟨Fig. A.2⟩ depicts 4 examples of translations according to **Rule 2**. When there are only input and output ports in EDIF file, these input and output ports can simply be translated into `.inputs` and `.outputs` in BLIF-MV, as shown in the first example. Also, there are 3 examples of using the keyword *array* inside the port. However, we do not have an example for array of the output ports, since such expression is not used in EDIF.

There is an *array* for input port, with index of 4, in the second example. If there is only one index, the starting index *i* automatically becomes 0, and the ending index becomes (*i* - 1). So in the second example, input ports named *input* in array with index of 4 would become `.inputs input[0] input[1] input[2] input[3]` in BLIF-MV.

In the third example, there is *downto* keyword in the array, with the starting index of 3 and the ending index of 0. In this case, input ports named *input* would become `.inputs input[3] input[2] input[1] input[0]`, since the keyword *downto* shows the index would be in descending order.

The last example, includes the case of *to* keyword used in the array, which gives an index in ascending order. It might seem the same as the second example, but we can designate the starting index and the ending index from 1 to 4, as shown in the last example of ⟨Fig. A.2⟩. The case of using the *downto* keyword also works the same.

[Case 3] Property functions

⟨Fig. A.3⟩ shows 3 examples of using the keyword *property function* in the cell. If the function is a sequential logic, there would exist an expression `property isSequential ''TRUE''` in EDIF. Then this function is simply expressed with `.reset` and `.latch` keywords in BLIF-MV.

In the first example in ⟨Fig. A.3⟩, there is a function `''DFF''`, which means *D-Flip-Flop*. In this cell, the input ports are `D`, `CLK` and `RST`, and the output port is `Q`. The `RST` port works as reset input signal, while `CLK` port works as clock input signal and `D` port means data input signal. Using the given input and output ports, it can be expressed as `.reset RST Q 0` and `.latch D Q DFF` in BLIF-MV. `.reset RST Q 0` means that when `RST` signal is activated, `Q` has to be reset into initial value `0`. `.latch D Q DFF` means that according to data input `D`, `DFF` is activated and the result is saved in output `Q`. The reason for omitting the input port `CLK` is due to the first step of pre-processing before translation into BLIF-MV, as stated in Section 4.2.1.

On the other hand, if there is no expression of `isSequential "TRUE"`, then the function would be combinational logic, *i.e.,* the cell is a combinational cell. If the cell is a combinational cell, then the ⟨*truth table of functionality*⟩ must be represented.

In the second example of ⟨Fig. A.3⟩ uses `AND2` gate, which means there are 2 operands with AND operator. After the keyword `.table`, the name of input ports (*i.e.,* `A B`) are listed and then output port (*i.e.,* `X`) is placed after them. The truth table is represented in translation result in the right-hand side of the table, under the `.default 0` line. In here, an example of `AND` gate using operand `A` and `B` is expressed.

In the last example of ⟨Fig. A.3⟩, multiple operators are used. There is a function of `A & (B + C)`. In this case, an expression of `.table A B C X` is used. A postfix notation, which would express the function `A & (B + C)` in ABC+&, is used internally during the translation from EDIF to BLIF-MV. Then the truth table is represented after the `.default 0` line, same as the second example.

[Case 4] Cells including instances

⟨Fig. A.4⟩ demonstrates an example of **Rule 4**. When there is an *instance* in the cell, it would be translated into `.subckt` in BLIF-MV. In the example, there are input ports `A` and `B`, and output port `X`. Also there is instance `U1` in the contents, and cellRef `AND2` is in the viewRef inside the instance `U1`. These are translated as `.subckt AND2 U1 A = U1_A B = U1_B X = U1_X` by following **Rule 4**.

EDIF	BLIF-MV
(cell example_cell (view example_view (interface (port D (direction INPUT)) (port Q (direction OUTPUT)) (port CLK (direction INPUT)) (port RST (direction INPUT))) (property function "DFF") (property isSequential "TRUE") ...)))	.model example_cell .inputs D CLK RST .outputs Q .reset RST Q 0 .latch D Q DFF ... .end

EDIF	BLIF-MV
(cell example_cell (view example_view (interface (port A (direction INPUT)) (port B (direction INPUT)) (port X (direction OUTPUT))) (property function "AND2") ...)))	.model example_cell .inputs A B .outputs X .table A B X .default 0 0 0 0 0 1 0 1 0 0 1 1 1 ... .end

EDIF	BLIF-MV
(cell example_cell (view example_view (interface (port A (direction INPUT)) (port B (direction INPUT)) (port C (direction INPUT)) (port X (direction OUTPUT))) (property function (string "A & (B + C)")) ...)))	.model example_cell .inputs A B C .outputs X .table A B C X .default 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 1 1 1 1 0 1 1 1 1 1 ... .end

Fig. A.3. Examples of translation of property functions.

EDIF	BLIF-MV
(cell example_cell (view example_view (interface (port A (direction INPUT)) (port B (direction INPUT)) (port X (direction OUTPUT))) (contents (instance U1 (viewRef view_1 ((cellRef AND2 (libraryRef lib_1))) ...)))	.model example_cell .inputs A B .outputs X .subckt AND2 U1 A = U1_A B = U1_B X = U1_X ... .end

Fig. A.4. Example of translation of cells including instances.

[Case 5] cells including connections

⟨Fig. A.5⟩ shows examples following **Rule 5**. When a keyword *joined* only exists in the *net*, and keywords *portRef* and *instanceRef* are used, it would be also translated into `.table` in BLIF-MV.

In the first example, there is a net `N1`, portRefs `input1` and `output1`, and instanceRefs `inst1` and `inst2`. When it is translated into BLIF-MV format, it would be expressed as `.table inst1 input1 inst2 output1`, and in the next sentence, the name of instanceRef of input instance (*i.e.,* `inst1`) has to be placed after "- =".

The second example deals with the case where the keyword *rename* is used. In this case, net is renamed and there can exist members inside it. In the second example, there is a net renamed as `original[1]`, and there are members `memberA` and `memberB`, and order for each of them is 1 and 2. The renamed name of the net becomes `original1` in BLIF-MV, and the name of members become `memberA1` and `memberB2`.

EDIF	BLIF-MV
(cell example_cell (view example_view (contents (net N1 (joined (portRef input1 (instanceRef inst1)) (portRef output1 (instanceRef inst2)))) ...)))	.model example_cell .table inst1 input1 inst2 output1 - = inst1 ... .end

EDIF	BLIF-MV
(cell example_cell (view example_view (contents (net (rename net_renamed "original[1]" (joined (portRef (member memberA 1)) (portRef (member memberB 2)))) ...)))	.model example_cell .table original1 → memberA1 memberB2 ... .end

Fig. A.5. Examples of translation of cells including connections.

Data availability

Data will be made available on request.

References

Ahn, D.-r., Shin, S.-g., Baek, Y.-s., Lee, H.-k., Park, K.-h., Choi, I.-s., 2019. A study on simulation based fault injection test scenario and safety measure time of autonomous vehicle using STPA. J. Korea Inst. Intell. Transp. Syst. 18 (2), 129–143.

Allani, M.Y., Riahi, J., Vergura, S., Mami, A., 2021. FPGA-based controller for a hybrid grid-connected PV/wind/battery power system with AC load. Energies 14 (8), 2108.

Brand, D., 2003. Verification of large synthesized designs. In: The Best of ICCAD. Springer, pp. 65–72.

Brayton, R.K., 1991. Blif-mv: an interchange format for design verification and synthesis. Tech. Rep., University of California.

Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S.A., Khatri, S.P., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T., 1996. VIS : A system for verification and synthesis. In: The Eighth International Conference on Computer Aided Verification. CAV '96, pp. 428–432.

Brayton, R., Mishchenko, A., 2010. ABC: An academic industrial-strength verification tool. In: Computer Aided Verification. Springer, pp. 24–40.

Bukya, M., Padma, B., Kumar, R., Mathur, A., Prasad, N., 2024. FPGA-based VFF-RLS algorithm for battery insulation detection in electric vehicles. World Electr. Veh. J. 15 (4), 129.

Burch, J.R., Clarke, E.M., Long, D.E., McMillan, K.L., Dill, D.L., 1994. Symbolic model checking for sequential circuit verification. Comput.- Aided Des. Integr. Circuits Syst. IEEE Trans. on 13 (4), 401–424.

Cadence, Conformal Equivalence Checker, https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/logic-equivalence-checking/conformal-equivalence-checker.html.

Cadence, Cadence Virtuoso Digital Implementation, https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/virtuoso-digital-implementation.html.

Choi, J., Lee, D., 2012. Development of RPS trip logic based on PLD technology. Nucl. Eng. Technol. 44 (6), 697–708.

Chouy, C.-T., 1997. Synchronous verilog: A proposal.

Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M., 1999. Nusmv: A new symbolic model verifier. In: 11th International Conference on Computer Aided Verification. CAV '99, pp. 495–499.

Clarke, E.M., Grumberg, O., Peled, D.A., 1999. Model Checking. MIT Press.

Cummins, D., Quinn, E.T., 2021. Instrumentation and control technologies for small modular reactors (SMRs). In: Handbook of Small Modular Nuclear Reactors. Elsevier, pp. 117–145.

Electric Power Research Institute (EPRI), 1996. Guideline on evaluation and acceptance of commercial grade digital equipment for nuclear safety applications (EPRI TR-106439).

Electric Power Research Institute (EPRI), 2014. Plant engineering: Guideline for the acceptance of commercial-grade items in nuclear safety-related applications (EPRI NP-5652). Revision 1 to EPRI NP-5652 and TR-102260.

Electronic Industries Association, 1998. Electronic design interchange format (EDIF). EIA-548, Version 2.0.0.

Farias, M.S., Martins, R.H., Teixeira, P.I., Carvalho, P., 2016. FPGA-based I&C systems in nuclear plants. Chem. Eng. Trans. 52.

Gautham, S.M., 2020. Multilevel runtime verification for safety and security critical cyber physical systems from a model based engineering perspective.

Hoare, T., 2003. The verifying compiler: A grand challenge for computing research. J. ACM 50 (1), 63–69.

Hu, K., Chu, Z., 2023. An efficient circuit-based SAT solver and its application in logic equivalence checking. Microelectron. J. 142, 106005.

Institute of Electrical and Electronics Engineers (IEEE), 2001. IEEE std 1364-2005: Verilog - hardware description language.
Institute of Electrical and Electronics Engineers (IEEE), 2008. IEEE std 1076-2019: VHDL language reference manual.
Institute of Electrical and Electronics Engineers (IEEE), 2016a. IEEE Std 1012-2016: IEEE Standard for System, Software, and Hardware Verification and Validation. Tech. Rep..
Institute of Electrical and Electronics Engineers (IEEE), 2016b. IEEE Std 7-4.3.2-2016: IEEE Standard Criteria for Programmable Digital Devices in Safety Systems of Nuclear Power Generating Stations. Tech. Rep..
International Electrotechnical Commission (IEC), 2000. IEC 61508, functional safety of electrical, electronic and programmable electronic (e/e/PE) safety-related systems.
International Electrotechnical Commission (IEC), 2006. Nuclear Power Plants – Instrumentation and Control Systems Important to Safety - Software Aspects for Computer-Based Systems Performing Category a Functions (IEC 60880). Tech. Rep..
International Electrotechnical Commission (IEC), 2007. Nuclear power plants – instrumentation and control important to safety - hardware design requirements for computer-based systems (iec 60987). Tech. Rep., International Electrotechnical Commission (IEC).
International Electrotechnical Commission (IEC), 2011. Nuclear power plants – instrumentation and control important to safety - general requirements for systems (iec 61513). Tech. Rep., International Electrotechnical Commission (IEC).
International Electrotechnical Commission (IEC), 2012. Nuclear Power Plants – Instrumentation and Control Important to Safety - Development of HDL-Programmed Integrated Circuits for Systems Performing Category a Functions (IEC 62566). Tech. Rep., International Electrotechnical Commission (IEC).
International Electrotechnical Commission (IEC), 2013. IEC 61131-3:2013 Programmable controllers - Part 3: Programming Languages. Tech. Rep., IEC.
International Electrotechnical Commission (IEC), 2020. Nuclear Power Plants – Instrumentation and Control Important to Safety – Development of HDL-Programmed Integrated Circuits – Part 2: HDL-Programmed Integrated Circuits for Systems Performing Category B or C Functions (IEC 62566-2). Tech. Rep..
Johannes Kepler University Linz, AIGER, http://fmv.jku.at/aiger/.
Johannes Kepler University Linz, Hardware Model Checking Competition' 15, http://fmv.jku.at/hwmcc15/index.html.
Jung, S., Kim, E.-S., Yoo, J., Kim, J.-Y., Choi, J.G., 2016. An evaluation and acceptance of COTS software for FPGA-based controllers in NPPs. Ann. Nucl. Energy 94, 338–349.
Jung, S., Yoo, J., Cha, S., 2010. VIS analyzer : A visual assistant for VIS verification and analysis. In: The 13th IEEE Computer Society Symposium Dealing with the Rapidly Expanding Field of Object/Component/Service-Oriented Real-Time Distributed Computing (ORC) Technology, ISORC 2010 Symposium.
Kim, E.-S., Jung, S., Kim, J., Yoo, J., Chang, C.-H., 2015. Verilog4VIS-EC: A manipulated verilog format for VIS equivalence checking. In: KIISE 2015 Winter Conference. pp. 461–463.
Kim, J.Y., Lee, Y.J., Cha, K.H., Cheon, S.W., Lee, J.S., Kwon, K.C., 2007. Experience on the COTS software dedication of the PROFIBUS FMS-driver. In: Transactions of the Korean Nuclear Society Spring Meeting Jeju, Korea, May. pp. 10–11.
Kim, J.Y., Lee, Y.J., Cheon, S.W., Lee, J.S., Kwon, K.C., 2010. A commercial-off-the-shelf (COTS) dedication of a QNX real time operating system (RTOS). In: 2010 2nd International Conference on Reliability, Safety and Hazard-Risk-Based Technologies and Physics-of-Failure Methods. ICRESH, IEEE, pp. 123–126.
Korea Atomic Energy Research Institute (KAERI), 2006. Software Design Specification for Reactor Protection System KNICS-RPS-SD231-01. Tech. Rep., Rev.02.
Kukimoto, Y., 1996. Blif-mv. The VIS Group, University California, Berkely.
Lee, J.-H., 2013. Automatic Translation for Equivalence Checking between Verilog and EDIF Netlist with VIS Master's thesis. Konkuk University.
Lee, D.-A., Kim, E.-S., Seo, Y.-J., Yoo, J., 2014. Fbdeditor: An FBD design program for developing nuclear digital i&c systems. In: Korea Conference on Software Engineering. KCSE 2014, in: 315–318 (Ed.)in Korean.
Lee, D., Park, D., 2021. Hardware and software co-design platform for energy-efficient FPGA accelerator design. J. Korea Inst. Inf. Commun. Eng. 25 (1), 20–26.
Microchip, Libero IDE, https://www.microchip.com/en-us/products/fpgas-and-plds/fpga-and-soc-design-tools/fpga/libero-ide.
Ni, L., Yang, Z., Zhang, J., Feng, C., Liu, J., Luo, G., Li, H., Xie, B., Li, X., 2023. MEC: An open-source fine-grained mapping equivalence checking tool for FPGA. In: 2023 International Symposium of Electronics Design Automation. ISEDA, IEEE, pp. 131–136.
NuScale, 2020. NuScale Standard Plant Design Certification Application Chapter 7 Instrumentation and Controls. Tech. Rep., NUSCALE.
Piggin, R., Sampson, C., 2016. Security and safety of FPGAS in nuclear safety systems: benefits and challenges. In: 11th International Conference on System Safety and Cyber-Security (SSCS 2016). IET, pp. 1–6.
Rahim, S., Rouzeyre, B., Torres, L., 2012. A flip-flop matching engine to verify sequential optimizations. Comput. Inform. 23 (5–6), 437–460.
RTCA, 2000. DO-254: Design assurance guidance for airborne electronic hardware.
Siemens, FormalPro, https://eda.sw.siemens.com/en-US/ic/formalpro-equivalence-checking/.
Siemens, Precision FPGA, https://eda.sw.siemens.com/en-US/ic/precision/.
Siemens, ModelSim, https://eda.sw.siemens.com/en-US/ic/modelsim/.
Siemens, Questa Simulator, https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.
SPIN, http://spinroot.com/.
Synopsys, Synopsys and Actel Renew OEM Relationship for FPGA Design Software, https://news.synopsys.com/home?item=123057.
Synopsys, Synopsys Synplify Pro, http://www.synopsys.com/.
Synopsys, Formality, http://www.synopsys.com/.
Times, E., 2001. Survey compares formal verification tools. http://www.eetimes.com/document.asp?doc_id=1216123.
University of Colorado at Boulder, IImc, https://github.com/mgudemann/iimc.
U.S. Nuclear Regulatory Commission, 1996a. NUREG/CR-7006 : Review Guidelines for Field-Programmable Gate Arrays in Nuclear Power Plant Safety Systems. Tech. Rep., U.S. Nuclear Regulatory Commision (NRC).
U.S. Nuclear Regulatory Commission, 1996b. A proposed acceptance process for commercial off-the-shelf (COTS) software in reactor applications (NUREG/CR-6421, UCRL-ID-112526).
U.S. Nuclear Regulatory Commission, 2012. Reporting of defects and noncompliance (10 CFR 21).
U.S. Nuclear Regulatory Commission, 2016. Instrumentation and Controls - Introduction and Overview of Review Process. Tech. Rep., U.S. Nuclear Regulatory Commission.
Wang, J., Li, M., Jiang, W., Huang, Y., Lin, R., 2022. A design of FPGA-based neural network PID controller for motion control system. Sensors 22 (3), 889.
Xu, Y., Huang, G., Balewski, J., Naik, R., Morvan, A., Mitchell, B., Nowrouzi, K., Santiago, D.I., Siddiqi, I., 2021. QubiC: An open-source FPGA-based control and measurement system for superconducting quantum information processors. IEEE Trans. Quantum Eng. 2, 1–11.
Yoo, J., Cha, S., Jee, E., 2009. Verification of PLC programs written in FBD with VIS. Nucl. Eng. Technol. 41 (1), 79–90.
Yoo, J., Kim, E.-S., Jung, S., 2015. Verification techniques for COTS dedication of commercial FPGA tools. In: The 10th International Symposium on Embedded Technology. ISET 2015, pp. 150–151.
Zhang, D., Wu, W., 2024. Radiation environment-constrained FPGA reinforcement technology and reliability research utilizing error control coding. IEEE Access.