VIS Analyzer: A Visual Assistant for VIS Verification and Analysis

Sehun Jeong
College of Info. and Comm.
Korea University
Seoul, South Korea
gifaranga@korea.ac.kr

Junbeom Yoo
College of Info. and Comm.
Konkuk University
Seoul, South Korea
jbyoo@konkuk.ac.kr

Sungdeok Cha
College of Info. and Comm.
Korea University
Seoul, South Korea
scha@korea.ac.kr

***Abstract*—Formal verification plays an important role in demonstrating the quality of safety-critical [1] systems such as nuclear power plants. We have used the VIS verification system [2] to determine behavioral equivalence between two successive revisions in developing the KNICS RPS (Reactor Protection System) [3] in Korea. The VIS accepts a high-level programming language Verilog [4] as input, and its verification results contain valuable information about one reason of the failure. However the VIS offers no graphical interface, and partially displays relevant information necessary to understand the full verification scenario accurately. Many nuclear engineers and verification experts found the information insufficient, and it makes hard to the wide use of the VIS verification system in industry. This paper proposes the VIS Analyzer, a visual assistant for VIS verification and analysis, which can help nuclear engineers take full benefits of VIS without being overwhelmed by incomplete and low-level details. The VIS Analyzer automates the VIS verification processes such as equivalence checking and model checking, and displays the verification results in visual formats. We used a recent case study introduced in [5] to demonstrate its effectiveness and usefulness.**

I. Introduction

Benefits of formal methods [6] cannot be emphasized enough. Regulatory bodies, such as KINS (Korea Institute of Nuclear Safety), often mandate that developers or SQA (Software Quality Assurance) teams apply formal methods as a means of safety demonstration. Among many tools and techniques for formal methods, model checking techniques [7] have received the most attention of research communities as well as practitioners. Model checker performs exhaustive search of systems behavior without any users intervention in the checking process, and also generates useful output called a counterexample for the failed case. Engineers can get insights into the system behavior which is difficult to get through other methods (e.g. inspection and testing). We can also get a high level of safety assurance if a model checker reports success, indicating the system behavior satisfies the required properties.

Outputs from formal verification tools such as the VIS and the SMV [10] are often quite difficult for practitioners to understand. The reasons are as follows. First, semantic gap between domain knowledge of practitioners and model checkers' output is too significant for practitioners to understand them sufficiently. Most model checkers have their own programming languages as input, and practitioners have to understand the input program translated from its requirements specifications or models. Second, counterexamples may contain too excessive or redundant information to reason about the source of failures. Efficient visualization of the counterexamples is a key to take full advantage of model checking. Third, counterexamples from such model checkers as the VIS often contain partial information on states and values. The information partiality includes omission of unchanged values between two successive steps in counterexamples. Automatic reorganization of the partial information into complete one is also crucial for its wide use. Fourth, verification processes of formal verification tools are overly long and detail for domain engineers. For example, the single VIS model checking task requires six to seven commands inputting in a row. Carefully abstracted verification process provides benefits of formal verification without too much efforts for retraining.

The VIS (Verification Interacting with Synthesis) is a widely used tool that integrates formal verifications, simulation, and synthesis of finite states hardware systems. It uses Verilog as a front-end and supports fair CTL (Computational Tree Logic) model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-base simulation, and hierarchical synthesis. We have used the VIS to formally verify the Verilog programs translated from FBD (Function Block Diagram) programs in our verification framework proposed in [5].

This paper proposes an assistant tool, VIS Analyzer (ver. 3.0), to automate the VIS verification processes (equivalence checking and model checking) and support visual analysis of the verification results. In case of equivalence checking, the VIS Analyzer reads two Verilog programs and executes sequential or combinational equivalence checking seamlessly. It also reorganizes omitted information in the verification results through the VIS simulation on background, and displays them in intuitive tabular and flowchart forms. When performing model checking, it reads one Verilog program and CTL properties, executes model checking, and seamlessly shows the verification result visually to aid understanding. We

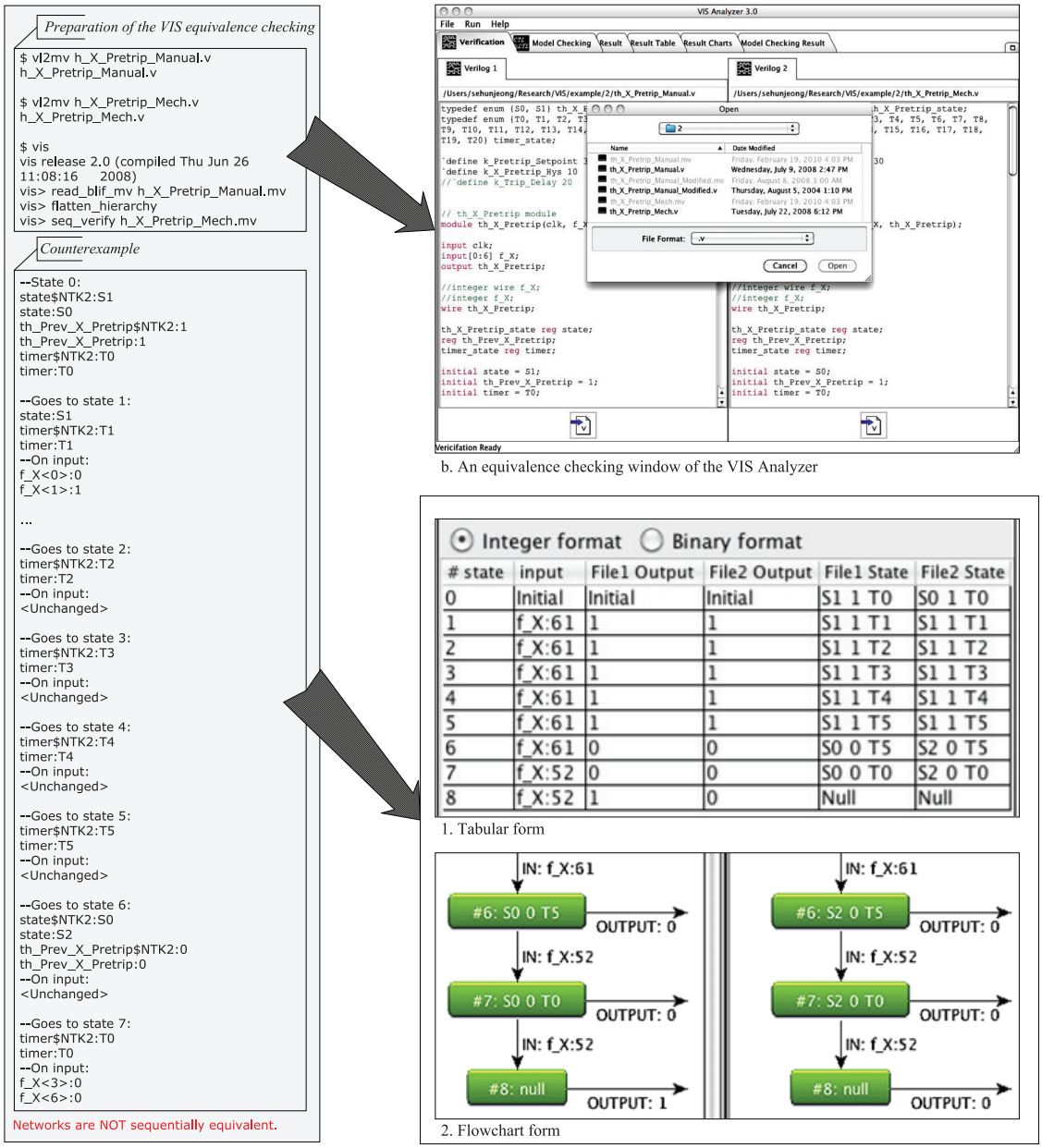

Fig. 1. A Comparison of Two Equivalence Checking Processes

performed a case study, a subset of KNICS consortium's RPS (Reactor Protection System) for APR-1400 nuclear power plant in Korea [8], to demonstrate its usefulness.

The remainder of the article is organized as follows. In section 2, we compare features of the VIS Analyzer with the typical VIS using examples from the KNICS project. Related work with the VIS Analyzer is explained in section 3. Section 4 concludes the paper and suggests future work.

II. A Comparison of Verification Processes

A. Equivalence Checking

Fig. 1 compares the VIS sequential equivalence checking process with that of the VIS Analyzer. It used a preliminary shutdown logic of the KNICS RPS in Korea. As we mentioned, the VIS Analyzer focused on three cases as follows to assist practitioners and SQA engineers. At first, the VIS Analyzer visually shows the result of the VIS equivalence checking feature. As we can see in Fig. 1(a), original counterexample is shown in plane text format which is not adequate for representing long and complex data. The

VIS Analyzer uses tabular and flowchart form as visualization method while preserving the original results in independent window for reference. Tabular and flowchart form are well-known and industry-widely used. Moreover, the VIS Analyzer automatically translates arrays of binary numbers (e.g. *f_X*) into decimal numbers to increase readability. Fig. 1(c) shows visualization results for the counterexample part of Fig. 1(a).

Second, we abstracted the process of the VIS verification itself. As we can see in top box in Fig. 1(a), the VIS requires users to input a series of commands manually for file opening, generating essential intermediate files and performing verification. However, these are unfamiliar tasks for domain engineers who are not verification experts. The VIS Analyzer abstracts these tasks into simpler one. What remains for users are file opening as usual GUI applications (see Fig. 1(b)) and selecting verification menu. In addition, the VIS Analyzer provides a comparison window for two source codes with highlighting in accordance with the Verilog grammar.

```
.inputs f_X<0> f_X<1> f_X<2> f_X<3> ...
.latches state th_Prev_X_Pretrip timer
.outputs th_X_Pretrip
.initial S1 1 T0
.start_vectors


0 1 1 1 1 0 1
0 1 1 1 1 0 1
...
0 1 1 0 1 0 0
```

Fig. 2. An Example of The Input Vector File

Third, the VIS Analyzer complements the VIS sequential equivalence checking result. When two target software revisions are not sequentially equivalent, the VIS equivalence checking feature shows one counter example path to the user in a textual format. Fig. 1(a) is the VIS sequential equivalence checking result about two example pre-trip logics. We found three information insufficiencies from the counterexample.

- The counterexample contains inputs and latches informations for seven steps counterexample path, but it skips unchanged inputs and latches values between two successive steps. In case of numeric values used are defined as arrays of numbers, only changed parts are shown. For example, seventh state in Fig. 1(a) contains *f_X<3>* and *f_X<6>* only. And the state shows only *timer* variable except *the_Prev_X_Pretrip* and *state*.
- The counterexample omits final state information which circulates inequality between two target systems. Users have to run the VIS simulation feature to know the final latches and output values.
- The counterexample doesn't show output values such as *output th_X_Pretrip* in Fig. 1(b) for all states. Although the VIS simulation feature provides output information according to particular input sequences, it requires additional effort to make input vector file (see Fig. 2) which specifies initial latch values and a list of input values to simulate.

The VIS Analyzer attempts to complement insufficiencies as we mentioned using three methods as follows. In case of partial input and latch informations, the VIS Analyzer automatically reconstructs partial values into complete one with reference to values of previous counterexample states. As shown in Fig 1(c), table and flowchart show complete *f_X* value and other latch values through all state steps. For output values, the VIS Analyzer automatically generates input vector file based on the counterexample, and seamlessly executes the simulation feature on background process using the input vector file. When the input vector file is generated, the VIS Analyzer adds one more input values to the input vector file as last sequence to show final output values which cause inequality. The output values are added on to the counterexample visualization, such as output columns in Fig. 1(c) table and output arrows in Fig. 1(c) flowchart, and both table and flowchart contain eighth state which shows output inequality, while the original counterexample in Fig. 1(a) contains seven states only.

B. Model Checking

Fig. 3 shows comparison between the VIS model checking process and that of the VIS Analyzer. In case of the VIS Analyzer model checking feature, we applied three approaches used in the VIS Analyzer equivalence checking feature such as automation of verification process, result complementation and result visualization. First, the VIS Analyzer visually shows model checking result in tabular and flowchart forms. The VIS model checking feature shows counterexample when some of verified properties are resulted as fail. We found that the counterexample of the VIS model checking is hard to understand because it has plane text format which is not adequate for representing of long and complex data. For example, Fig. 3(a) counterexample part consists of three distinct sections, but these sections are hard to distinguish at one glance because they have similar configuration. th sections in Fig. 3(a)) into one stream of states, and visualizes the stream as flowchart. The flowchart form is adequate for represent series of states stream. The loop notation in the flowchart format effectively and intuitively represents fair path cycle information in the counterexample. Fig. 3(c) shows counterexample parts of Fig. 3(a) in the flowchart form. Since Fig. 3(a) contains fair path cycle, Fig. 3(c) has loop notation accordingly. In addition, the tabular form is provided as alternative visualization form.

Second, the VIS Analyzer model checking feature provides abstracted verification process likewise the VIS Analyzer

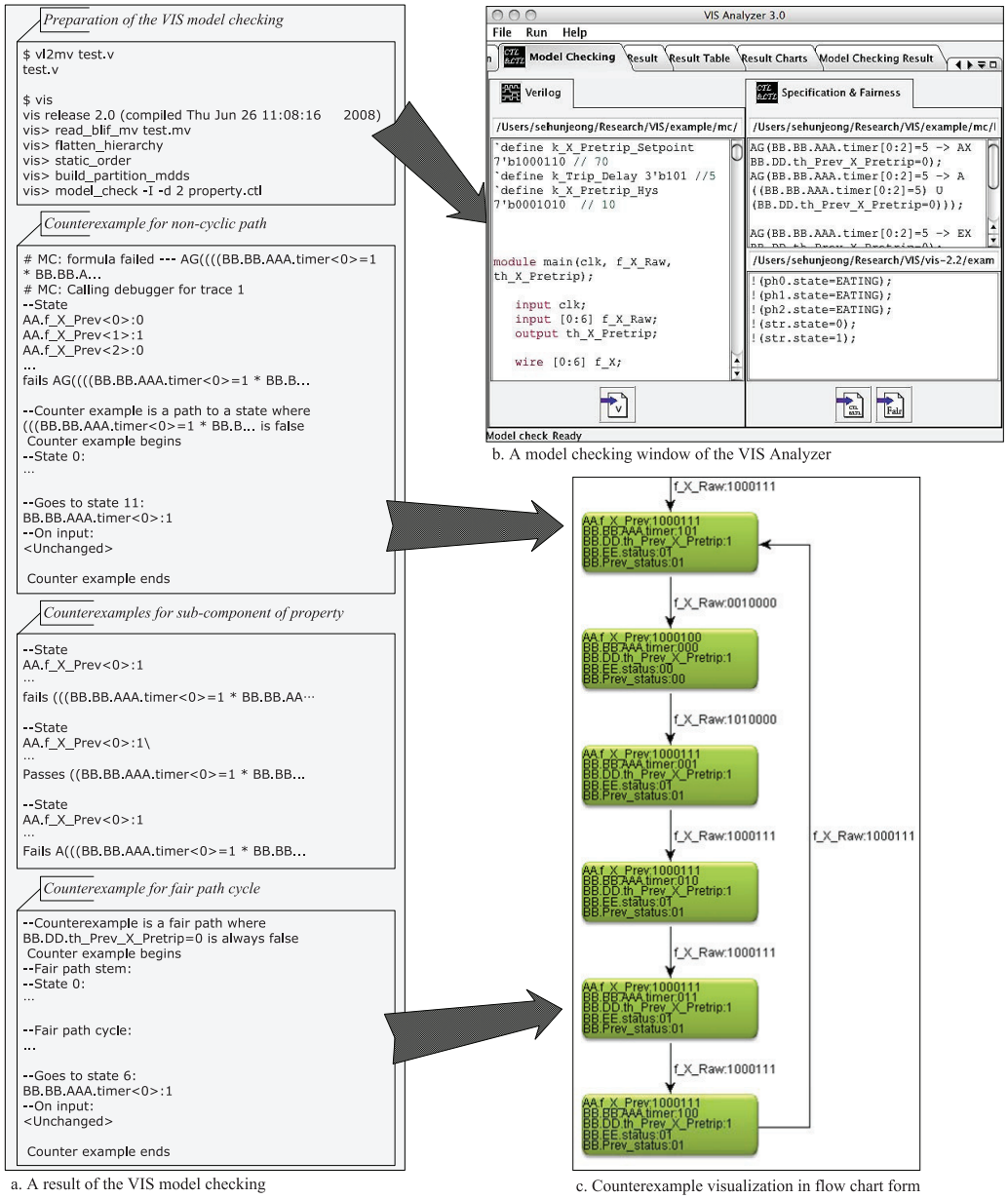

Fig. 3. A Comparison of Two Model Checking Processes

equivalence checking feature. Top section in Fig. 3(a) shows the VIS model checking process which includes intermediate file generation and a number of commands inputting. The VIS Analyzer provides abstract way to perform verification tasks with graphical user interface. The VIS Analyzer model checking window (see Fig.2(b)) consists of three sub-windows which are stand for source code, verification properties and fairness properties. Users can easily check what they are currently working on with the help of the syntax highlighting feature and the composite code windows.

Third, the VIS Analyzer model checking feature restores partial inputs and latches information of the original VIS model checking result using the same method used in the VIS Analyzer equivalence checking feature. For example, being partially displayed values in the Fig. 3(a), such as *BB.BB.AAA.timer<0>*, are fully complemented in each node of Fig. 3(c). As we mentioned before, reducing the hardness is key feature of verification supporting tools because the information partiality of the counterexample makes hard to interpret the result.

III. Related Work

Developing supporting tools which can connect gap between domain industry needs and formal methods tools is very important. Supporting tools should include abstracted

verification processes and result visualization features. The process abstraction helps users performing complex verification processes without user intervention and low-level detail about formal verification. And the verification result visualization features amplify human recognition [9].

There have been many studies about formal verification tool automation and its result visualization. They all tried to simplify tedious process and visualize complex information with intuitive and friendly forms for domain experts. RuleBase [11], SMV automation tool resemble our work. The RuleBase automates the SMV verification process and visualizes results for the hardware designer. The user can easily analyze visualized hardware verification results with the help of graphic user interface of this tool.

In the area of visualization of programing languages, PLFire [12] visualizes Phased Logic (PL) [13] in a form of block and wire diagram to help PL optimization process. And Code browser [14] adapts a tree-like graphical method to represent C program languages in web browser. These tools support analyzing and understanding complex code with informative and intuitive diagrams.

Data visualizations are also a common topic. Vis-Complete [15] suggests visualization method for constructing pipelines using database. And Shipboard Power Systems (SPS) [16] provides a visualization method for complex power status data in shipboard. When abnormal status occurred, SPS indicates error status and locations with 3D wire graphics of shipboard. Domain engineers can easily analyze current status and respond to errors.

IV. Conclusion

When developing safety-critical software such as RPS in nuclear plants, formal methods are required as a means of safety demonstration. The VIS is a useful formal verification tool as it provides equivalence checking, model checking and simulation on Verilog programs. However inconvenient user interface such as text-based console environment of VIS makes hard to domain experts, such as nuclear power plant engineers.

This paper proposes an assistant tool, the VIS Analyzer (ver. 3.0), to automate the verification processes (equivalence checking and model checking) and support visual analysis of the verification results. In case of equivalence checking, it reorganizes omitted information in the verification results through VISs simulation on background. When performing model checking, it reads one Verilog program and CTL properties, executes model checking, and seamlessly shows the verification result visually to aid understanding. We performed a case study, a subset of KNICS consortiums RPS (Reactor Protection System) for APR-1400 nuclear power plant in Korea [8], to demonstrate its usefulness. We are currently focusing on visualizing the verification result more intuitively using multi-dimensions.

Acknowledgment

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency)" (NIPA-2009-(C1090-0902-0032))

References

[1] N. G. Leveson, *SAFEWARE*, System Safety and Computers, Addison-Wesley, 1995.

[2] *VIS (Verification Interacting with Synthesis)*, http://embedded.eecs.berkeley.edu/research/vis

[3] J. Yoo, E. Jee, S. Cha, *"Formal Modeling and Verification of Safety-Critical Software"*, IEEE Software, Vol.26, No.3, pp.42-49, May/June 2009.

[4] D. E. Tomas, P. R. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1991.

[5] J. Yoo, S. Cha, E. Jee, *"A Verification Framework for FBD based Software in Nuclear Power Plants"*, The 15th Asia Pacific Software Engineering Conference (APSEC), pp.385-392, Beijing, China, Dec. 3-5, 2008.

[6] M. Pezz, M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*, Chapter 8, WILEY, 2007.

[7] E. M. Clarke, Orna Grumberg, Doron A. Peled, *Model Checking*, MIT Press, 1999.

[8] KNICS, *http://www.knics.re.kr*

[9] R. D. Meyer, D. Cook, *Visualization of data*, Current Opinion in Biotechnology, pp89-96, 2000

[10] SMV, *www.kenmcmil.com/smv.html*

[11] I. Beer, S. Ben-David, C. Eisner, A. Landver, *RuleBase: an Industry-Oriented Formal Verification Tool*, 33rd Design Automation Conference, pp 655-660, 1996.

[12] K. Fazel, M. A. Thornton, R. B. Reese, *PLFire: A Visualization Tool for Asynchronous Phased Logic Designs*, Design, Automation and Test in Europe Conference and Exhibition, 2003

[13] D. H. Linder, *Phased Logic: Supporting the Synchronous Design Paradigm with Delay Insensitive Circuitry*, IEEE Transactions on Computer, Vol.45, No.9, September 1996.

[14] J. Cherry, M. Arrieta, E. Brown, S. Ramaswamy, *An Interactive visualization tools for complex programs*, 2004 International Conference on Software Engineering, June 21-24, 2004

[15] D. Koop, C. E. Scheidegger, S. P. Callahan; J. Freire, C. T. Silva, *VisComplete: Automating Suggestions for Visualization Pipelines*, IEEE Transactions on Visualization and Computer graphics, Vol. 14, No. 6, pp 1691-1698, 2008.

[16] K. L. Butler-Purry, N. D. R. Sarma, *Visualization for Shipboard Power Systems*, Proceedings of the 35th Annual Hawaii International Conference, 2003.