CESNET technical report number 5/2004

Verification Process of Hardware Design in Liberouter Project [1]

Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, and Pavel Šimeček

November 15, 2004

1 Abstract

This technical report analyzes the process of verification of hardware design in Liberouter project. Such an analysis had become a necessity since we had developed several tools that were difficult both to maintain and modify. This document tries to sum up our needs and to propose a way our tools could be organized. Description of verification environment Verunka is given in more detail as it is a new tool.

2 Introduction

The main aim of Liberouter project [LibWWW] is to develop a hardware accelerated router based on Combo6 PCI card. The card itself is based on FPGA (Field Programmable Gate Array) technology. Given a design or a part of it for that card our task is to formally specify its properties and to model-check them. We will not go into details regarding a particular design or specific design elements here. They are treated elsewhere [Formalize], [Results]. We will, however, concentrate on process of doing so.

3 From Design to Model-checked Formula

This section goes through the verification process. We will mention complications of respective steps and our tools and techniques to fight them.

[1] This work is supported by the FP5 project No. IST-2001-32603, the CESNET activity Programmable hardware, and the GACR grant No. 201/03/0509.

3.1 Formal Specification

A designer develops a new hardware design module in VHDL possibly suggesting properties and invariants. We formalize them which results in a formal specification to be model-checked. A system of asserts has been developed to ease this part. The system is described in the Verification Cookbook [VC].

3.2 VHDL to SMV Conversion

The model-checking tool of our choice is Cadence SMV [SMV]. Its input language, called simply SMV, is far less expressive then VHDL. Moreover, only a conversion utility from a subset of Verilog HDL to SMV is part of Cadence SMV package. Fortunately, we can convert VHDL to Verilog using Leonardo Spectrum synthesizer that is the one used by design developers.

However, there are several issues. They are caused by different expressiveness of source and target languages, the synthesis and even bugs or misbehavior of the Verilog to SMV conversion utility. Our solutions to these problems are based on substitutions and modifications at all three levels, i.e. VHDL code, Verilog HDL code and SMV code. Details can be found elsewhere.

We have developed `vhd2v` script to help with synthesis of VHDL to Verilog and `v2smv.pl` script to assist in converting Verilog to SMV.

3.3 Model-checking and Reporting Results

Once we obtain an SMV code we can try to model-check it and report the result to the developer. It is useful to be able to repeat a verification process, e.g. to compare results after changing some steps of the process. Hence a lot of information should be part of the result: VHDL code used, parameters of conversion to SMV and preconditions among others.

An XML structure [XML] has been defined to store all this information as a verification report in a tractable manner. However, XML format is not easy to be written by hand. That was the first motive for Verunka environment.

3.4 Iterations of the Verification Process

Hardware designs are evolving due to optimizations and/or enhancements. The verification process must be repeated in this case from scratch because signals used to express properties may have changed. Only a small number of properties have to be rewritten because the design must have a relatively stable interface to other modules. Many formulae can be hence reused.

Another reason is that of refining formal specifications. The design module typically rely on a correct behaviour of its environment, namely the rest of the design. Such a correct behavior has to be described by a number of preconditions. These are false counter-example guided in many cases. Formulae are only added in this case.

The verification report structure meets both these needs.

4 Making the Process More Transparent

A need to draw a line between conversion and model-checking arose as we gained more experiences. Especially the `v2smv.pl` script had accumulated many features. Many of them were based on some heuristics even if a correct solution was easy to implement. Moreover, verification report handling has become very laborious.

We have therefore strictly separated the conversion from model-checking and removed some tricks by exploiting `smv` features. The conversion output is a `smv`-ready SMV code that includes preconditions and properties from an external file. Formal specification is written down to a verification report from where it is written to an external file to be included into SMV code upon request.

Such a separation has another advantage from practical point of view. Model-checking is extremely demanding when it comes to computational power and memory. It is hence not desirable to run model-checking on the same machine as the synthesis, which is essential for design developers and cannot be run elsewhere. The clear division makes it easy to do different steps on different machines.

There still remain many tasks regarding verification reports at this point which were difficult to make by hand but which could be effectively automated. The idea of a verification environment has been hence introduced.

5 Verification Environment Verunka

Verunka is a Czech female name which was decided as an acronym for *ver*ification *run*.

Given a valid SMV code one should be able to easily control all the verification tasks, namely

- create or open a verification report
- add a verification to the report (either create a new one or copy a selected one)

- display particular information from the report
- write out formulae to be included into SMV code
- model-check a formula of a verification and store the results to the report
- write the report

It means Verunka should be able to call the *smv* model-checker and treat XML verification reports properly.

5.1 Design Decisions

User interface. The environment should primarily allow user to select a verification and a formula and to display relevant information for selected ones. A command-line interface is not suitable for such a purpose. Implementing a complete graphical user interface for the X Window System, however, seemed too complicated. An `ncurses` based terminal application has been considered the best compromise.

Implementation language. There was no doubt a scripting language would be selected. The idea behind this decision is that: once a user is able to run the application he or she should be able to modify it. Choosing Perl was a natural consequence as it is a general purpose language we are familiar with. We decided to be careful when using non-standard modules as they could be unstable or difficult to install. The latter was the case of Curses::UI module.

Implementation Architecture. Implementation should be, of course, transparent and easy to modify. We have chosen object oriented design and have strictly separated user interface from verification report processing. The two form separate sets of Perl modules that can be used without each other. The *verunka* script actually binds user interface events to verification report processing actions.

We are going to describe architecture principles in following sections. See documentation of respective modules for more technical information. It is embedded directly in sources in Plain Old Documentation format.

5.2 User Interface

User interface architecture was inspired by GTK+ [GTK+] and Curses::UI [CursesUI]. However, only a very simplified version of them is needed for Verunka.

There are currently four classes regarding user interface.

Verunka::Widget: This is a base class for all widgets.

Verunka::Label: A label widget intended to display information. It can display short single-line messages as well as a long text across several lines.

Verunka::ListBox: A list box allows user to select item(s) from a list.

Verunka::UI: This class provides a user interface framework. User interface is represented by an instance of this class. Such an instance stores global user interface properties (color definitions, for example), references to widgets that are part of the user interface and bindings of events (i.e. key presses) to actions.

To implement a user interface, one has to do the following. Create an instance of Verunka::UI and define overall properties of user interface. Create instances of widgets based on the Verunka::UI object. Define handlers and bind them to events. Call Verunka::UI->mainloop which enters the loop of interpreting events.

An event consists of two parts. The first one is the widget that currently holds focus. The second one is the key code. Hence a handler is bound to a pair widget, key code. However, it is possible to bound a default handler to a key code. Default handler takes effect if there is no handler bound to the key code and currently focused widget.

5.3 Report Handling

Reports are stored as XML data [XML]. We have written down Verunka::FullTree Perl module, a general object oriented API to access, modify and store XML data. A more detailed description of this module is given below.

Another module (or class using terminology of object oriented programming), Verunka:Report, is built on top of Verunka::FullTree. It implements all verification tasks and their impact on the report. Its functionality depends on semantics of the report necessarily, hence relying on XML data of a specific DTD. However, as many as possible tasks are done syntactically based on the DTD. This includes creation of new parts of the report, supplying default attribute values etc.

5.3.1 Handling XML Data in General

We need to read an existing XML report, modify it and write it back. Modifying is not simply a batch processing of the document in this case. It can be pretty complex depending on user actions performed on the report. Hence we need to read the report into a structure which can be easily modified and, on the other hand, preserves the XML structure so that data could be written as an XML report. This is the purpose of Verunka::FullTree module. However, it is not limited to verification reports because the DTD may change in future.

We have created Verunka::FullTree because none of existing modules meets our needs. On the other hand, an XML parser has to be employed to read the document and good XML parsers exist and have Perl interfaces. We have chosen XML::Parser interface to Expat library [PerlExpat], an event-driven parser.

A set of classes reflecting syntactic elements of XML and a style class for XML::Parser have been defined. As a document is being parsed the parser generates events handlers of which are executed. The handlers are provided by the style class, Verunka::FullTree::Style, and they built a tree-like structure each node of which is an instance of an appropriate class. The structure is represented by its root node. Each node provides methods for operations according to its type.

The root node provides a method to write the document back into a file among others. This is a delicate point because users should be still able to read and modify XML data by hand. The user would like to identify his or her modifications even after an automatic processing of the data. On the other hand, an application can add new elements or remove existing ones. From that point of view we have decided a compromise. All formatting blank characters (spaces, newlines, etc.) are ignored. Element tags are written out in a uniform way: indented, one tag per line. The rest of document, namely textual data, is left intact.

6 Conclusion

A need for a verification environment emerged and has been responded by the `verunka.pl` script. This is part of our effort to make the verification process automatic and comfortable as much as possible. Defining Verunka's functionality has also helped us clarify different stages of the process. Main parts of bothering work has been already automated allowing us to concentrate on verification itself.

References

[Formalize] Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, Pavel Šimeček: *How to Formalize a FPGA Hardware Design* Technical report number 4/2004, CESNET, 2004

[Results] Jan Holeček, Tomáš Kratochvíla, Vojtěch Řehák, David Šafránek, Pavel Šimeček: *Verification Results in Liberouter Project* Technical report number 3/2004, CESNET, 2004

[VC] Tomáš Kratochvíla: *Verification cookbook (Liberouter policy WWW Pages)*
`http://www.liberouter.org/policy.php`

[XML] Tomáš Kratochvíla, Vojtěch Řehák, Pavel Šimeček: *Verification of COMBO6 VHDL Design* Technical report number 17/2003, CESNET, 2003

[LibWWW] Liberouter: *Liberouter Project WWW Pages*
`http://www.liberouter.org/`

[SMV] Cadence SMV: *Cadence SMV WWW Pages*
`http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/`

[GTK+] GTK+: *The GIMP Toolkit* `http://www.gtk.org/`

[CursesUI] Marcus Thiesen: *Curses::UI Perl Module*
`http://search.cpan.org/dist/Curses-UI/`

[PerlExpat] Matt Sergeant: *XML::Parser Perl Module*
`http://search.cpan.org/dist/XML-Parser/`