CESNET technical report number 17/2003

Verification of COMBO6 VHDL Design [1]

Tomáš Kratochvíla, Vojtěch Řehák, and Pavel Šimeček

November 19, 2003

1 Abstract

This technical report presents current results and experiences of the formal verification of *VHDL* design of *Combo6* hardware accelerator card for packet routing, originating from the *Liberouter* project. The design is quite difficult to prove by conventional methods, therefore model checking as a method of formal verification was employed. We use the symbolic model checker **Cadence SMV**. Information about formal verification itself is enriched by description of transformation from *VHDL* to the **Cadence SMV** specification language. The last part shows the system of assertions established as a compact way of communication with *VHDL* designers.

2 Introduction

The aim of the Liberouter project [LibWWW] is to design and develop the hardware accelerated router. The most important part of this project is a development of the *Combo6* hardware accelerator card [Nov02] allowing to route the most of traffic of a *Gigabit Ethernet* in the hardware.

Combo6 is a PCI card based on FPGA (Field Programmable Gate Array) [FPGA]. The programmable hardware technology represents a new generation of hardware development. FPGA is the class of integrated circuits pioneered by *Xilinx*, in which the logic function is defined by the developer using *Xilinx* development system software. Gate arrays are another type of integrated circuits whose logic is defined during the manufacturing process.

Developers write the design of *Combo6* card in the *VHDL* (*VHSIC* Hardware Description Language) [VHDL]. *VHDL* can be used at many levels of abstraction ranging from algorithmic level to the gate level. On the one hand the wide expressive power is positive for developers, but on the other hand it brings

[1] This work is supported by the FP5 project No. IST-2001-32603, the CESNET project 02/2003, and the GACR grant No. 201/03/0509.

great demands on tools supporting *VHDL*. Hence there are many tools supporting different subsets of *VHDL* and finally, the wide expressibility leads to the incompatibility of tools working with *VHDL*.

The following section briefly justifies the choice of the model checker and describes how to solve problems with incompatibilities between supported languages. The next section is focused on verification itself using the **Cadence SMV** symbolic model checker and there we also mention the system of assertions as a communication interface solving troubles of collaboration with the hardware designing team.

3 Translation from VHDL to *SMV*

VHDL is used by developers of *Combo6* card to write down the design loaded to FPGA. Therefore it is also the input for the formal verification (the model checking technique in our case) [Bar02]. The most suitable model checking tools for hardware verification are the symbolic ones (for more details see Subsection 4.1).

Unfortunately, as *VHDL* is not an input language of any model checking tool available to us at this moment, we have to translate *VHDL*. Currently we do not know about any translator from *VHDL* to any model checker specification language that would be able to translate sufficiently large subset of *VHDL*. Fortunately, **Cadence SMV** model checker installation package [SMV] includes the translator from *Verilog* to *SMV* (the input language of this model checker). The *Verilog HDL* [Ver] is a hardware description language of the same power as *VHDL*. Hence we could form a translator from *VHDL* to *SMV* as a composition of the **Cadence SMV** translator (called **vl2smv**) with a translator from *VHDL* to *Verilog*.

Unfortunately, we have found neither the proper translator from *VHDL* to *Verilog*. We have tried **VHDL to Verilog RTL translator v1.0** (made by *Ocean Logic*) and **VHDL2verilog** (demo version made by *Alternative System Concepts*) but neither one of them satisfies our needs. One of them do not support sufficiently large subset of *VHDL* and the output of the other one was not a proper input for **vl2smv**.

3.1 Translation using a synthesis

We found another solution how to translate to *Verilog* format. The **LeonardoSpectrum** synthesiser [LeoSpec] (currently used by *Combo6* hardware developers) provides ability to write a synthesised code down into *Verilog*. For that reason we decided to verify synthesised code instead of the original one. They

should be functionally equivalent, and hence the verification of such a code is correct (in a sense of preserving validity of temporal formulas). Synthesised *Verilog* code is a correct input for **vl2smv** except for a few syntax entities. We run synthesis directed by the following TCL script:

```
set part 2V3000bf957
set process 4
set modgen_select fastest
load_library xcv2
read -design $DESIGN -technology "xcv2" -format vhdl { $MODULES }
set extract_ram FALSE
set tristate_map TRUE
optimize -ta xcv2 -hierarchy preserve
write -downto PRIMITIVES -format Verilog $NAME.v
```

Where `$DESIGN` contains the name of the top-level design, `$MODULES` stands for the list of all modules of the design, and `$NAME` is the name of the output file (without an extension).

This TCL script is partially adopted from the synthesis scripts currently used by our hardware designers. The changes were done in an optimisation parameters and some special options we use.

3.1.1 **line** `optimize -ta xcv2 -hierarchy preserve`

Citation from **LeonardoSpectrum** User Guide [LSguide]:

> When **LeonardoSpectrum** reads an HDL design, it infers arithmetic and relational operators (e.g. adders) and implements the operators as blackboxes (there is no underlying functionality) in the design. **LeonardoSpectrum** does not implement operators until global area optimization when it replaces these blackboxes with technology-specific netlists (from the *Modgen Library*).
>
> ...
>
> Each blackbox operator uses a naming convention to convey parameter information such as (type, size, sign, carry), for example: `add_16u_16u_0` (16 bit adder, unsigned operands, no carryout)

If there is no optimisation used, the synthesiser infers many black boxes (entities with no description except for their interface) similar to `add_16u_16u_0` and verification is absolutely impossible since the functionality of the most of the hardware design is undefined.

We have also tried to change an effort of optimisation to value `remap` to preserve more from the original structure of *VHDL* code, but this level of optimisation generates black boxes similarly as a synthesis with no optimisation.

The hierarchy preserve option (`-hierarchy preserve`) is necessary for us, because we need to preserve the interface of single parts of a design. When this option is omitted, the synthesiser usually renames the most of signals of used components and changes their behaviour; in that way it gives no sense to ask to check their behaviour in temporal formulas.

3.1.2 **line** `set extract_ram FALSE`

Synthesiser has a capability to recognise (technology specific) blocks of RAM in a *VHDL* code - typically the RAM is inferred from large arrays and vectors. This option disables automatic extraction of RAM from arrays, which are not accessed and used as RAMs. Disabling RAM extraction causes inferring (usually flip-flop) registers instead of RAM blocks. Unfortunately, for the arrays which are inevitably used as RAMs (indexation using a variable) it is inferred a special black box named `CLOCKED_RAM_*` (we write * instead of the suffix of the name, which differs according to parameters of inferred RAM block).

Nevertheless, these black boxes do not worry us too much because we are usually unable to verify the systems with large blocks of memory and we abstract away from them (this is more discussed in Section 4).

3.1.3 **line** `set tristate_map TRUE`

As it is discussed in Subsection 3.3, we have some troubles with modeling high impedance signal `'z'` in *SMV*. Therefore we enable this option in order to remove all tristate values wherever it is possible. However some tristate signals are quite often necessary to preserve functionality of the design.

3.1.4 **line** `write -downto PRIMITIVES -format Verilog $NAME.v`

This line of TCL code makes the final *Verilog* output. The most important option here is the `-downto PRIMITIVES` option forcing synthesiser to write down also the behavioural descriptions of basic entities from the library *Primitives* (e.g. `LATRS`, `DFFRS`, `GND`, etc.).

3.1.5 *Synthesising a signal as a* `dumb` *signal*

If two signals have the same behaviour, the synthesiser can flatten these signals to the only single one. If `hierarchy preserve` option is set, then one such signal is preserved and the second one is tied to `GND` component (ground).

This optimisation can be prohibited by writing attribute `preserve_signal` to the *VHDL* code or to the constraint file. As the occurrence of two signals with the same behaviour is quite rare, we currently solve this situation by using the name of preserved signal in temporal formulas instead of the name of the flattened one.

3.2 Further work with a synthesised code

The synthesised code (in a *Verilog* format) has many advantages, but also some disadvantages. The main (and very important) advantage is that there is used only small fragment of *Verilog* language in a synthesised code. Therefore the portability of such a code is much better then of the original one. In the synthesised code we can also trust that every output signal has assigned a value (`0`,`1`, or some special value like `'bz`) - this claim holds for the entire code except for black boxes and behavioural descriptions of components from included libraries. It is also important that the synthesis encloses the definition of the most of components used in the design (including those from the library *Primitives*).

The main disadvantages are as follows. The code gained from a synthesis is quite large and untransparent. There are preserved only signals in the interfaces of entities (except for the signals with the same behaviour which can be flattened into one signal). Therefore if we want to check the values of inner signals of entities, we have to work around it.

3.2.1 How to preserve inner signals

The `hierarchy preserve` option allows us only to preserve the signals in the interfaces of entities. But we can (mis)use it also for preserving inner signals of entities. Anyway we can define a new entity (called for example `export_signals`) with the only one output signal and one input signal for each signal which should be exported. Architecture of such an entity can be a large logical `AND` of input signals.

3.3 Refining Verilog code

As was mentioned above, the synthesised *Verilog* code is the correct input for **vl2smv** except for logical connectives, assignment using `buf` module, and the name of the top-level entity. The needed translation from a synthesised *Verilog* to the *Verilog* suitable for **vl2smv** is shown in Table 1.

The last row of the table is used to translate `'bz` signal to the other name. It is used to get around a bug in **vl2smv** that translates `'bz` to the meaningless sequence of zeros finished by so called `weak` value, whereas we need it to translate to the single value `weak`.

`xor(a,b,c)`	`assign a = ~(b | c)`
`nand(a,b,c)`	`assign a = ~(b & c)`
`and(a,b,c)`	`assign a = a & c`
`or(a,b,c)`	`assign a = (b | c)`
`buf(a,b)`	`assign a = b`
the first occurrence of `module <name>`	`module main`
`‘bz`	`bz_OUR_ALIAS`

Table 1: Table of translations

Signal `'bz` is usually used to model an access to a bus. We translate it to the `weak` signal. But it is not correct when a signal of a bus is declared only as an output signal of some entity: assigning `weak` value to such signal causes `undefined` value of such signal. We do not know whether it is another bug in **Cadence SMV**, but we have to get around this behaviour so that we replace output signals with input-output signals (it can be simply done by removing keyword `output` in the *SMV* code). There we use the big advantage of synthesised code: value of each output signal in a synthesised code is defined (except for black boxes and behavioural descriptions of components from included libraries, e.g. library *Primitives*). Therefore we do not change the behaviour of the design (except for assigning `weak` values but this change is correct).

3.4 Black boxes

The synthesiser knows the behavioural descriptions of components from the library *Primitives*. But it does not know behavioural descriptions of components from library *xcv2* that is used to synthesise our *VHDL* sources for a Xilinx platform.

There are two cases for components from *xcv2*:

- there exists a behavioural description of a component in the documentation (e.g. entity `SRL16E`)
- there exists no such a description (e.g. `RAMB16_S18_S18` and the rest of block-RAM components)

In the first case we can (manually) copy the description from the documentation of *xcv2* to the *Verilog* source and thus we replace the black box by a fully functional component.

In the second case we have to abstract away from the behaviour of such a component (and take it into account when we create temporal formulas).

3.5 Translation of latch registers

As stated, there is no problem with translation of synthesised code because of its simplicity - synthesised code is composed from a large amount of assignments and many connected copies of components. But when we also include behavioural descriptions of components from libraries *Primitives* and *xcv2* we clutter up the *Verilog* code with a relatively complex codes of included entities.

Therefore there has occurred further (meantime) manual work for us. It is necessary to check out, whether there is a block of code, from which it is inferred the latch register. It can be recognised by the keyword `reg` in a *Verilog* source code. **vl2smv** seems to have no support for a register data type in *Verilog*.

If the identifier is declared as a register (`reg` in *Verilog*, `variable` in *VHDL*) then its value is changed immediately after any assignment and it is preserved until next change (independently on a clock edge). Assignment to these registers is realized in the `always` blocks (in *Verilog*). The keyword `always` is followed by the sensitivity list with neither `posedge` nor `negedge` keyword. Furthermore the code in the `always` block need not to assign these registers (because latch registers should preserve its value). For example the *Verilog* code of the entity `LATRS` is unfortunately translated to the *SMV* code as follows.

Verilog code:

```
module LATRS ( set, reset, in, clk, out );
 input set;
 input reset;
 input in;
 input clk;
 output out;
 reg out;

 always @ (in or clk or reset or set)
  begin
   if (set) out = 1;
   else if (reset) out = 0;
   else if (clk) out = in;
  end
```

SMV code:

```
module LATRS (set, reset, in, clk, out)
```

```
{
 input set : boolean resolve;
 input reset : boolean resolve;
 input in : boolean resolve;
 input clk : boolean resolve;
 output out : boolean resolve;

 out : boolean resolve;
 do
  {
   if (set) out := 1;
   else if (reset) out := 0;
   else if (clk) out := in;
  }
}
```

The declaration of register `out` has turned to redeclaration of a signal `out` (the redeclaration is not the main mistake in this code - actually it is ignored). The main mistake in the resulting code is a transformation of the register into the ordinary signal (that cannot preserve its value).

The only way how to correct this mistake seems to be the simulation of registers by flip-flop registers (in *SMV*). We can always store the last value of a signal to the flip-flop register (`previous` in our example) and during the next tick of clock we can assign its value as a default value of the signal representing the register. Thus the mentioned `LATRS` example is changed to the following one.

```
module LATRS (set, reset, in, clk, out)
{
 input set : boolean resolve;
 input reset : boolean resolve;
 input in : boolean resolve;
 input clk : boolean resolve;

 out : boolean resolve;
 previous : boolean resolve;  -- flip-flop register

 init(previous) := undefined; -- at the first time the notion
                              -- 'previous' has no sense

 do
  {
```

```
   out := previous;            -- default set to previous
   if (set) out := 1;
   else if(reset) out := 0;
   else if(clk) out := in;
  }

 next(previous) := out ;
}
```

After all these translations we can start with a real formal verification. The way we translated the sources have a significant influence to techniques of verification.

4 Verification of design in *SMV* language

This section describes ways how to deal with *SMV* codes obtained from translation described in the previous section. Since the synthesis has been already made, this codes are no longer in the behavioural level. Some problems arise from the synthesis and the others from the state explosion.

4.1 Model checking

The *VHDL* tools allow to simulate the system written in *VHDL* source codes. Each simulation run of the system checks whether the model is correct in this particular run. To ensure that our *VHDL* source code is correct and really works as it is intended, we need to simulate all possible runs. Using *VHDL* simulators one can check only a small test set.

To check all possible runs of the model we use formal methods of verification. We specialise to model checking methods, which allow to automatically prove whether a system satisfies the specification at some level of abstraction (for more details see [Bar02]).

4.1.1 State explosion

The number of states may grow exponentially with respect to the number of storage elements of a system written in *VHDL* source codes, and thus even for small sized examples the state space becomes infeasible. Linear grow up of variables in *SMV* may cause the exponential grow up of states in the system, which has to be checked.

The state explosion problem is inherent to explicit state model checking of asynchronous concurrent systems. The states can be symbolically represented such as representing states using a Binary Decision Diagram (BDD).

In order to handle large designs, we have to abstract away parts of the data path, or the whole entities, or even the components.

Taking into an account that **Cadence SMV** for CTL (branching time logic) formula in the form `AG (subformula)` has to verify whether `subformula` is satisfied, it obviously has to take in notice every possible run of the system. For formula in the form `EF (subformula)` it is again necessary to verify that `subformula` is not satisfied for every possible run of the system, before concluding that formula `EF (subformula)` is not satisfied.

Cadence SMV as a symbolic based model checker has some advantages in comparison with explicit state approach to model checking. It manipulates with the set of states with a specific property and not with every state explicitly.

4.2 Providing assertions in VHDL code

4.2.1 Native assertions in VHDL

VHDL itself contains the following command:

`assert` *condition* `[report` *error_string*`] [severity` *severity_value*`]`

We can verify this assertions better than *VHDL* simulators in the sense that simulator makes only selected runs of the system, whereas we check all possible runs of the system.

4.2.2 Using special assertion comments

We also support several ways to include verification formulas as comments in the *VHDL* design. Briefly we are supporting following forms of comments:

The most simple form is useful when property is very complicated:

`-- assert` *Informal description in English.*

For more complicated properties there could be added a `LEMMA` with an attached email and the *unique_mark* referring to that property:

`-- assert LEMMA` *unique_mark*

The corresponding attached email should have the following form:

`To:` *ipv6-ver@liberouter.org*
`Subject: LEMMA` *file.vhd-unique_mark*
`Content:` *Description of properties of your* VHDL *code.*

Description of properties in the *VHDL* boolean expressions (extended by implication, cycle, branching and more) which should be always true:

`-- assert globally` *extended_VHDL_bool_expression*

From others we mention only one example how signals, ports, variables, or registers, which never falls to have constant value, could be described.

`-- assert alive` *boolean_expression*

This property is known as a liveness.

We have written a *Verification cookbook* [VC], the manual for *VHDL* designers. It contains detailed information how to write assertions. Its purpose is to help them to write the assertions down into the code in some of the forms described above.

4.3 SMV model checker in practice

4.3.1 The automatic tests

There is no need to explicitly write the deadlock free requirement on the modeled system. The **Cadence SMV** automatically checks whether the system has a state with no successors.

4.3.2 Using preconditions and lemmas

Except of a command `assert` the tool **Cadence SMV** supports more sophisticated constructions.

At first all formulas can be uniquely entitled to enable one to refer to it. For example:

```
lemma_RW_mutual_exclusion: assert G (! ( \plx_wr & \plx_rd ) );
```

There is a need for preconditions on formulas. The most of preconditions we obtain from the exact position of formulas in *VHDL* source. The formula have to be valid in the states of system corresponding to the position in *VHDL* source. Rarely it is satisfied even in the other places.

For example `lemma1` in the following *VHDL* code is intended for verification with a precondition.

```
if RESET = '1' then
     WR_REQ      <= '1';
     WR_STATE    <= "1000";
  elsif FSM_RD = '1' then  -- assert LEMMA lemma1
     WR_REQ      <= '0';
```

In *SMV* there can be the proof with precondition expressed as follows.

```
precondition_reset: assert (~RESET & FSM_RD)
using precondition_reset prove lemma1;
```

4.3.3 Disappearing of the inner signals

When we obtain for example this *VHDL* source code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is                       -- configurable counter
        Generic (N : integer := 8);
        Port (CLK     : in std_logic;   -- counted signal
              AS_RESET: in std_logic;   -- asynchronous reset
              RESET   : in std_logic;   -- synchronous reset
              CE      : in std_logic;   -- count enable
              OUTPUT  : out std_logic_vector(N-1 downto 0)
             );                         -- actual counter value
end counter;

architecture Behavioral of counter is
signal VALUE: std_logic_vector(N-1 downto 0); -- internal value
begin

 -- assert active VALUE when CE

        process (CLK,AS_RESET)
        begin
        if AS_RESET='1' then
        VALUE <= (others => '0');
          elsif CLK='1' and CLK'event then
                if RESET='1' then
                        VALUE <= (others => '0');
                elsif CE='1' then
                        VALUE <= VALUE + 1;
           end if;
        end if;
        end process;

        OUTPUT <= VALUE;
end Behavioral;
```

We have to deal with the following problem: Our task is to check whether the formula `active VALUE when CE` is satisfied or not. The problem is that signal `VALUE` is internal only and due to synthesis, it is neither in *Verilog*, nor in *SMV*.

One way is to fetch up the signal `VALUE` to the port list in *VHDL* source. The entity `counter` should contain one of the following lines in the port list:

```
VALUE : out std_logic_vector(N-1 downto 0);
VALUE : inout std_logic_vector(N-1 downto 0);
VALUE : buffer std_logic_vector(N-1 downto 0);
```

The problem is that the line

```
VALUE : out std_logic_vector(N-1 downto 0);
```

results in the following error message:

```
Error, cannot read output: VALUE; use mode buffer or inout.
```

The other possible lines with `inout` or `buffer` works (*VHDL* source can be synthesised) and these lines are correct with respect to our actual formula as it happens. In general this ways of fetching up the signal `VALUE` are incorrect as it changes the model of system in a nonequivalent manner. The solution is described in Part 3.2.1.

The another way is to find out that in *VHDL* behavioural description of `counter` there are two processes:

```
process (CLK,AS_RESET)

OUTPUT <= VALUE;
```

These processes run in parallel and the process `OUTPUT <= VALUE;` causes that signals `OUTPUT` and `VALUE` are connected. This observation could be hard to find. But when it is found it enables to check the formula without intervention to *VHDL* source.

4.3.4 From counterexample to a new precondition

Counterexample gives us a new precondition or a negative result in the following way:

When we obtain a counterexample, we analyse it and as far as this trace could not occur in real hardware we add new preconditions to the formula. We may obtain a counterexample again which often results to many preconditions.

Ordinary counterexample may than have the following effect:
From `using pre1, . . . , prek prove lemma;` makes
`using pre1, . . . , prek, newpre prove lemma;`

Than we proceed as long as we obtain a positive or a negative result.

4.3.5 How to deal with a huge trace table

The counterexample is represented as a trace table. The rows are assigned to variables of *SMV* corresponding to signals, ports, variables, or registers of *VHDL* and each column contains values of these variables in one state. Values are expressed by 0, 1, or - (standing for undefined value).

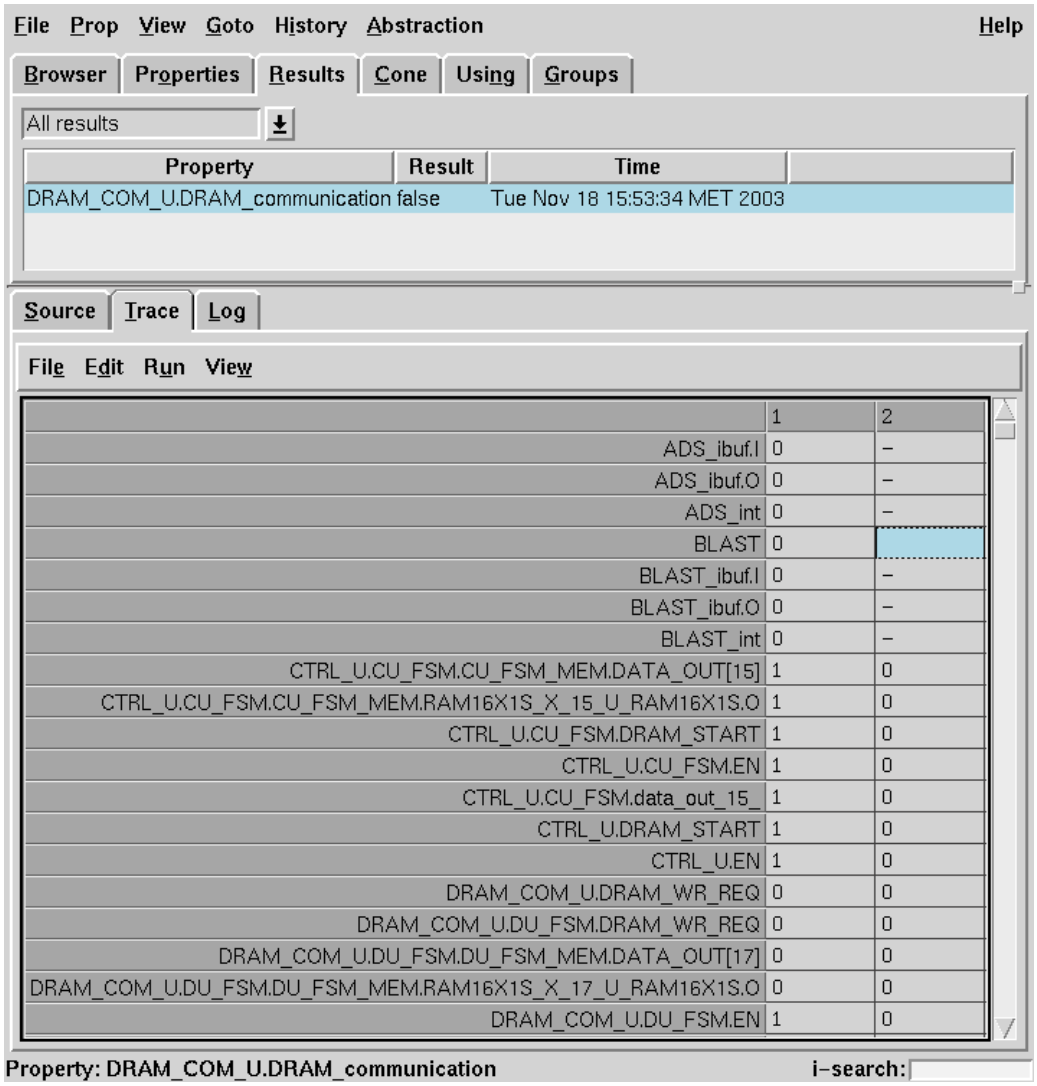

Figure 1: Huge trace in **Cadence SMV**

If one wants to see only changes, then it is useful to choose `view` and then `zoom out` in the trace table but it helps only a little.

We are working with huge traces and **Cadence SMV** shows only a very small part of a trace at once. Therefore we save trace to a file and work on it out of **Cadence SMV**. We compare interesting states of a trace using the program **diff**.

4.4 XML format for reporting results

We need a uniform format for publishing our results. For this purpose we define our own *XML* structure `verification`. For each version of verified design we add a verification report (element `ver`) that contains all important information about performed verification. Every verification report consist of:

- Tree of files in which the design is described and its time of creation.
- List of preconditions, which could be referred by formulas.
- List of formulas, where each formula contains its code, information about duration time, detailed description, possible differences from the original code for verification purposes, and last but not least the result – whether it is valid or not. If formula is not valid, the counterexample is present as well.

We have created a `verification.dtd` and every *verification report* have to be valid against it. The following example of *verification report* should make some details clear:

```
<verification>
  <ver author="Tomas Kratochvila" toplevel="ee">
    <note>Only an example, not real verification.</note>
   <vhdlsourcelist src="liberouter.old/hw/edit_engine/new_cvs/"
                   exportdate="2003-10-31">
      <directory name="edit_engine">
        <directory name="dram_u">
          <file type="vhdl" name="dram_u_fsm_mem.vhd"/>
          <file type="vhdl" name="dram_u_fsm.vhd"/>
          <file type="vhdl" name="dram_u.vhd"/>
        </directory>
        <file type="vhdl" name="send_u/send_u.vhd"/>
        <file type="vhdl" name="alu/alu.vhd"/>
        <file type="vhdl" name="ee.vhd"/>
      </directory>
    </vhdlsourcelist>
    <preconditions>
     <precondition name="simplepre0">G ( F \START)</precondition>
      <precondition name="pre3">
         G ((\START) -> F (\DRAM_ACK))</precondition>
    </preconditions>
    <formulas>
```

```
      <formula name="simplerw" result="true">
        <code>G (! ( \plx_wr & \plx_rd ) )</code>
       <description>Simple RW mutual exclusion.</description>
        <note></note>
        <time>0.1 s</time>
      </formula>
      <formula name="hardcorerw"
                preconditions="simplepre0, pre3"
                result="false">
        <code>G (\START -> (! (\plx_wr & \plx_rd) ))</code>
        <counterexample>not important</counterexample>
       <description>Hard-core RW mutual exclusion.</description>
        <diff></diff>
      </formula>
    </formulas>
  </ver>
</verification>
```

5 Conclusion

We have made the translation from *VHDL* to *SMV* almost automatic and we would like to bring the automatisation further. We have successfully verified various properties of selected parts of a hardware design and we are planning to verify larger pieces of hardware in order to check the interaction between selected components. The possible state explosion problems can be reduced by the symbolic methods of **Cadence SMV**.

The universal system of verification reports is created to easily report results using *XML*.

We provide several ways to write assertions into the *VHDL* source code. These assertions are reformulated to temporal formulas and included into the *SMV* source code obtained from the translation. Although the system of assertions is quite comfortable we have to study the informal descriptions of parts of the design and formulate our own assertions, because hardware designers are often not aware of all presumptions they use to believe that their source codes are correct.

References

[Bar02] Barnat J., Brázdil T., Krčál P., Řehák V., and Šafránek D.: *Model checking in IPv6 Hardware Router Design*

CESNET technical report 8/2002

[FPGA] Xilinx, Inc.: *DS031-1 Virtex-II 1.5V Field Programable Gate Arrays* October 2001

[LeoSpec] LeonardoSpectrum: *LeonardoSpectrum WWW Pages* http://www.mentor.com/leonardospectrum/

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

[LSguide] Mentor Graphics: *LeonardoSpectrum User's Manual* http://www.mentor.com/leonardospectrum/customer/documentation/

[Nov02] Novotný J., Fučík O., Kokotek R.: *Schematics and PCB of COMBO6 card* CESNET technical report 14/2002

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

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

[Ver] Daniel C. Hyde: *Handbook on Verilog HDL* www.eg.bucknell.edu/~cs320/1995-fall/manual.pdf

[VHDL] Ashenden Peter J.: *The VHDL Cookbook* http://tech-www.informatik.uni-hamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf