Simulation-based Hardware Verification Back-end: Diagnostics

Mikhail Chupilko, Alexander Protsenko
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
{chupilko,protsenko}@ispras.ru

***Abstract*—Hardware development processes include verification as one of the most important part. Verification is very often done in simulation-based way. After comparison of design behavior and its reference model behavior, the verdict about their correspondence appears. It is very useful to have some means of analyzing potential inconsistency of their output data. It is exactly the subject of this work to supply verification engineers with a method and a back-end tool for diagnostics of incorrect behavior using wave diagrams and reaction trace analysis based on recombination of reaction traces.**

I. Introduction

The importance of hardware verification (taking up to 80% of the total development efforts [1]) is raised by difficulties in error correction in already manufactured devices. Many methods address verification, some of them being more formal (*static analysis*), the other ones using simulators (*dynamic verification*). In the first case, the verification is carried out in a strict mathematical way. For example, the approach to verification known as *model checking* [2] means checking satisfability of formally expressed specification and formulae manually created by an engineer. If some error occurs, it is connected with unsatisfiability of the properties set by the engineer and the specification that should be checked and corrected. Dynamic verification implies checking mutual correspondence of output reactions of two models: *design under verification* itself (*DUV*) and the *reference model* (possibly, expressed by a set of *assertions*). The same stimulus sequence is applied to the both models, their reactions are compared and if some problem occurs, incorrect reactions are shown (the reactions can be so due to their time, data, and possibility of their appearing). As the unarmed looking at incorrect reactions is not always enough to understand quickly the problem having occurred, it seems to be very important to show more diagnostics information, including the place of this error on the wave diagram, and the answer to the question why such an error has appeared.

In this paper, we will introduce the way of diagnostics, which should be independent of the reference model organization. It should be also supported by a wide range of test system making technologies, including the one made by our team (*C++TESK Testing ToolKit* [3], [4]) and widely distributed world-known methods of test system development (*Universal Verification Methodology* [5]).

This work evolves the research previously described in [6] and extends it by new understanding of the explanatory rules used in the analysis algorithm and visualization of diagnostics.

The rest of the paper is organized as follows. The second section is devoted to related works on the subject of diagnostics and trace analysis. The third section introduces architecture of test systems for simulation-based verification and the proposed method of *diagnostics subsystem* construction. The fourth section considers method implementation and examples of its work with test system development library C++TESK Testing ToolKit. The fifth section concludes the paper.

II. Related Works

The problem of diagnostics of event-based systems is studied under different angles. Some researchers understand the failt diagnostics as checking of formal properties in formally expressed systems (e.g., [7]). In the other sources the diagnostics is closer to our task where it means construction of such *timed automata*, which can find a bug in behavior of DUV according to some pattern during the simulation (e.g., [8] and [9]).

The processing of reaction traces produced by DUV and the reference model can be also called *trace rewriting* (e.g., [10]). This term is used for describing of *symbolic trace* reducing methods based on some set of rules. There are several types of objects, which can be removed from the trace without losing of the trace expressiveness, including extra data, dependent objects, and all the other objects not influencing the analyzed result. In our case, the reaction trace can be represented as the symbolic trace leading to an error occurred at some moment of time. Having information about parent-child dependences between stimuli and reactions, we can remove unnecessary objects from the trace and provide verification engineer with a meaningful essence for analysis of defect.

The task we formulated for the research is quite close in general sense to trace rewriting but has some technical differences including usage of wave diagrams for visualization of diagnostics results, different set of rules accounting peculiarities of HDL design traces, for example *signal interfaces* (sets of HDL signals) where reactions appear.

III. Diagnostics subsystem

The being developed diagnostics subsystem should be a back-end to common simulation-based test system architecture. To understand the position of the back-end better, let us review quite common architecture with names of objects from C++TESK (see Figure 1). The concrete architecture selection is not specific as the set of objects in mentioned above wide-distributed UVM is very similar to those in C++TESK ([11]).

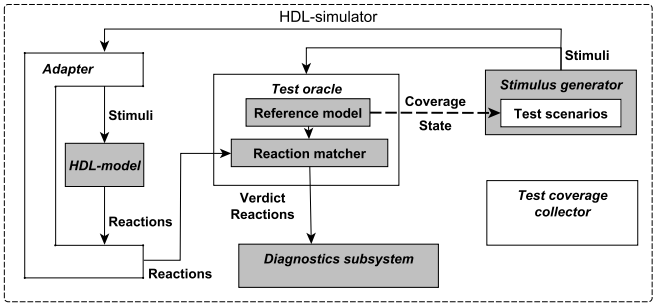

Fig. 1. Common architecture of test system

The typical components of simulation-based test systems are *stimulus generator*, *test oracle* (including *reaction matcher*), *adapter* making the interface between transaction level test system and signal-level DUV, *test coverage collector*. The diagnostics subsystem is also shown in the Figure 1 to clarify its position. After reaction matcher having checked correspondence of two reaction traces, the one from DUV, another from the reference model and produced the false verdict at the current cycle, all this information is given to the diagnostics subsystem, which can work as an extern plug-in for the test system.

The input data for the diagnostics subsystem is a trace including reactions from reference model and DUV, applied stimuli, dependences between them (by some *parent identifier*). All this objects can be provided in XML. The diagnostics subsystem can also process wave diagrams to show the important signals and position of defect reactions. The latter also requires *mapping of DUV signals* to reactions in XML.

Let the reaction checker use two sets of reactions: $R_{spec} = \{r_{spec_i}\}_{i=0}^{N}$ and $R_{impl} = \{r_{impl_j}\}_{j=0}^{M}$. Each specification reaction consists of four elements: $r_{spec} = (data, iface, time_{min}, time_{max})$. Each implementation reaction includes only three elements: $r_{impl} = (data, iface, time)$. Notice that $time_{min}$ and $time_{max}$ show an interval where specification reaction is valid, while $time$ corresponds to a single time mark: generation of implementation reaction always has concrete time.

The reaction checker has already attempted to match each reaction from R_{spec} with a reaction from R_{impl}, having produced *reaction pairs*. If there is no correspondent reaction for either specification or implementation ones, the reaction checker produces some pseudo reaction pair with the only one reaction. Each reaction pair is assigned with a certain type of situation from the list of *normal, missing, unexpected*, and *incorrect*.

For given reactions $r_{spec} \in R_{spec}$ and $r_{impl} \in R_{impl}$, these types can be described as in Table I. Remember that each reaction can be located only in one pair.

The diagnostics subsystem has its own simplified interpretation of reaction pair types (see Table II). In fact, the subsystem translates original reaction pairs received from the reaction checker into the new representation. This process can be described as $M \Rightarrow M^*$, where $M = \{(r_{spec}, r_{impl}, type)_i\}$ is a set of reaction pairs marked with *type* from the list above.

TABLE I. REACTION CHECKER REACTION PAIR TYPES

Type name	Reaction pair	Definition of type
NORMAL	(r_{spec}, r_{impl})	$data_{spec} = data_{impl}$ & $iface_{spec} = iface_{impl}$ & $time_{min} < time < time_{max}$
INCORRECT	(r_{spec}, r_{impl})	$data_{spec} \neq data_{impl}$ & $iface_{spec} = iface_{impl}$ & $time_{min} < time < time_{max}$
MISSING	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R_{impl}^{normal,incorrect} : iface_{spec} = iface_{impl}$ & $time_{min} < time < time_{max}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal,incorrect} : iface_{impl} = iface_{spec}$ & $time_{min} < time < time_{max}$

TABLE II. DIAGNOSTICS SYSTEM REACTION PAIR TYPES

Type name	Reaction pair	Definition of type
NORMAL	(r_{spec}, r_{impl})	$data_{spec} = data_{impl}$
INCORRECT	(r_{spec}, r_{impl})	$data_{spec} \neq data_{impl}$
MISSING	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R_{impl}^{normal,incorrect}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal,incorrect}$

$M^* = \{(r_{spec}, r_{impl}, type^*)_i\}$ is a similar set of reactions pairs but with different label system. It should be noticed that there might be different M^* according to the algorithm of its creation (accounting for original order, strategy of reaction pair selection for recombination, etc.).

To make the translation, the diagnostics subsystem uses a set of reaction trace *transformation rules*. Each of the rules transforms the reaction pairs from the trace but does not change their data. To find the best rule for application, the subsystem uses a *distant function*, showing the closest reactions among the pairs. The distant function can be implemented in three possible ways.

Metric 1: Reaction closeness correlates with the number of equal data fields of two given reactions.

Metric 2: Reaction closeness correlates with the number of equal bits in data fields of two given reactions (the Hamming distance).

Metric 3: Reaction closeness correlates with the number of equal bits in data fields of two given reactions, order of equal and unequal areas, and their mutual disposition.

The measure of closeness between two given reactions is denoted as $c(r_{spec}, r_{impl})$.

Each rule processes one or several reaction pairs. In case of missing reaction or unexpected reaction, one of the pair elements is undefined and denoted as $null$. Each reaction pair is assigned with a signal interface. The left part of the rule shows initial state; the right part (after the arrow) shows result of the rule application. If the rule is applied to several reaction pairs, they are separated with commas.

In general, the algorithm of rule application consists of two stages. At the first stage, reactions with equal data are joined and transformed by the rules. In case of such a transformation, the application order of rules is of importance. The second stage includes processing of the rest reactions and new ones

made at the first stage. Here rule priority is less important than values of the selected distant function.

Now, let us review all the six rules that we found including the first two rules being basic. In description of the rules c means the selected distant function but at the first stage of the algorithm it is always full equivalence of data. At the second stage of the algorithm a rule may be applied only if c value for this rule is the best among c values for the other rules for given reactions.

Rule 1: If there is a pair of *collapsed reactions*, it should be removed from the list of reaction pairs. $(null, null) \Rightarrow \emptyset$.

Rule 2: If there is a normal reaction pair $(a_{spec}, a_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, it should be *collapsed*. $(a_{spec}, a_{impl}) \Rightarrow (null, null)$.

Rule 3: If there are two incorrect reaction pairs with mutual correlation of data, the reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} : c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl})$ & $c(a_{spec}, a_{impl}) < c(b_{spec}, a_{impl})$ or $c(b_{spec}, b_{impl}) < c(a_{spec}, b_{impl})$ & $c(b_{spec}, b_{impl}) < c(b_{spec}, a_{impl})$ (this closeness is the best amoung the other rules), $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, b_{impl})\}$

Rule 4: If there is a missing reaction pair and an unexpected reaction pair with mutual correlation of data, these reaction pairs should be united into one reaction pair. $(a_{spec}, null), (null, a_{impl})$ and $c(a_{spec}, a_{impl})$ is the best among the other rules: $\{(a_{spec}, null), (null, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl})\}$.

Rule 5: If there is a missing reaction pair and an incorrect reaction pair with mutual correlation of data, these reaction pairs should be regrouped. $(a_{spec}, null), (b_{spec}, a_{impl})$ and $c(a_{spec}, a_{impl}) < c(b_{spec}, a_{impl})$ (this closeness is the best amoung the other rules), $\{(a_{spec}, null), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, null)\}$.

Rule 6: If there is an unexpected reaction pair and an incorrect reaction pair with mutual correlation of data, these reaction pairs should be regrouped. $(null, a_{impl}), (a_{spec}, b_{impl})$ and $c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl})$ (this closeness is the best amoung the other rules), $\{(null, a_{impl}), (a_{spec}, b_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (null, b_{impl})\}$.

The first stage of the algorithm is shown in Figures 2 and 4. The first stage having passed, the sets R_{spec} and R_{impl} does not contain any not yet collapsed reactions with identical data. The time of the second stage comes (see Figures 3 and 5). Both stages of the algorithm having passed, the list of reaction pairs may contain some reaction pairs with both specification and implementation parts but not collapsed due to their unequal data. To show diagnostics info for them too, they are collapsed using modified second rule, not requiring equality of data in the reaction pairs.

After the application of each rule, the history of transformation is traced and then it is possible to reconstruct the predecessors of the given reaction pairs and all the rules they were processed by. Such a reconstruction of the rule application trace we understand as the *diagnostics information*.

In the result, verification engineers are provided with a list of problems occurred during verification and with a set of hints making bug localization easier.

```
1: function MATCH1((r1_spec, r1_impl), (r2_spec, r2_impl), rnumb)
2:     RP1 = (r1_spec, r1_impl), RP2 = (r2_spec, r2_impl)
3:     c = total_equivalence
4:     for all rule ∈ |Rules| do
5:         if rule.isApplicable(RP1, RP2, c) then
6:             rnumb ⇐ rule.number
7:             return true
8:         end if
9:     end for
10:    return false
11: end function
```

Fig. 2. Action 1 — match1

```
1: function MATCH2((r1_spec, r1_impl), (r2_spec, r2_impl), rnumb)
2:     RP1 = (r1_spec, r1_impl), RP2 = (r2_spec, r2_impl)
3:     c = selected_metric_function
4:     metric ⇐ 0
5:     for all rule ∈ |Rules| do
6:         metric* ⇐ rule.getMetric(RP1, RP2, c)
7:         if (metric* > metric then
8:             metric ⇐ metric*
9:             rnumb ⇐ rule.number
10:        end if
11:    end for
12:    return metric
13: end function
```

Fig. 3. Action 2 — match2

IV. Implementation of the method

The proposed approach to diagnostics of incorrect output reactions has been implemented as a plugin in C++ and Java languages and attached to C++TESK Testing ToolKit [4].

If the verification process fails, the information provided by the diagnostics subsystem is shown. It looks like tables with all found errors (see Figure 6) and rule application history: new reaction pair sets and the way of their obtaining.

Now let us proceed to the following part of diagnostics subsystem work — the visualization of bugs on wave diagrams.

```
1: function APPLY_STAGE1({(r_spec, r_impl)_i})
2:     for all r ∈ |{(r_spec, r_impl)_i}|&!r.collapsed do
3:         for all p ∈ |{(r_spec, r_impl)_i|}&!p.collapsed do
4:             if match(r, p, rule_number) then
5:                 (r_spec_{i+1}, r_impl_{i+1}), (r_spec_{i+2}, r_impl_{i+2}) ⇐
6:                 Rules[rule_number].apply(r, p)
7:                 r.collapsed ⇐ true
8:                 p.collapsed ⇐ true
9:                 return
10:            end if
11:        end for
12:    end for
13: end function
```

Fig. 4. Action 3 — apply_stage1

function APPLY_STAGE2($\{(r_{spec}, r_{impl})_i\}$)
  for all $r \in |\{(r_{spec}, r_{impl})_i\}| \&!r.collapsed$ **do**
    $metric^* \Leftarrow 0$
    for all $p \in |\{(r_{spec}, r_{impl})_i|\} \&!p.collapsed$ **do**
      $metric = fuzzy\_match(r, p, rule\_number)$
      if $metric > metric^*$ **then**
        $metric^* \Leftarrow metric$
        $rule\_number^* \Leftarrow rule\_number$
        $s_1 \Leftarrow r$
        $s_2 \Leftarrow p$
      end if
    end for
    if $metric^* > 0$ **then**
      $(r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}}) \Leftarrow$
      $Rules[rule\_number^*].apply(s_1, s_2)$
      $s_1.collapsed \Leftarrow true$
      $s_2.collapsed \Leftarrow true$
      return
    end if
  end for
end function

Fig. 5. Action 4 — apply_stage2

Each specification reaction produced during test process keeps its parents — stimuli and other events making this reaction. Therefore, it is possible to reconstruct the whole chain from the very first stimulus up to the reaction with one of the error types. Each reaction contain data that correspond to signals of HDL model. Typically, the HDL signals are grouped into input and output interfaces and correlate with names of data fields in reactions. There should be a map between signals of interfaces and data fields. Such a map is usually created manually before development of test system. Basing on the resulted reaction pairs, a wave diagram produced by simulator (VCD file [12]), and the signal mapping the diagnostics subsystem creates a set of source files for GTKWave [13] to make errors be visual. The diagnostics subsystem creates separated directory with VCD and SAV files for each incorrect reaction pair. According to these files, GTKWave is asked to show the error situation with its history (predecessors), highlighting only those signals which are necessary for understanding the situation. These signals include ones from output interfaces used in reactions and some common signals like clock, reset and so on. It is possible to show the reference values of signals by injecting into VCD files special signals and labeling them as the reference ones for so and so signals. This possibility has not been implemented yet but there is no technological difficulty as the diagnostics subsystem already parses VCD files and creates new files with subset of signals.

Example of visual representation of the error from Figure 6 is shown in Figure 7. The situation described by these figures is as follows. The reaction expected at the 38th interface was received at the 41st interface. First, it resulted in missing and unexpected reactions, and then the diagnostics subsystem joined these reactions to create a normal one. The situation of the reaction appearing at the 41st interface and the reaction absence at the 38th interface is exactly shown in the Figure 7.

```
==============================================================================
| Failure #1 | IFACE VIOLATION? [iface41]:8 | 2 time(s)                       |
|------------|----------------------------------------------------------------|
| OutputData |            data0 |            data1 |            data2 | data4  |
| Received   | b74426de4836da5c | 84c630d7ce01f086 | 1d4cfa86f2955c53 |      0 |
| Expected   | b74426de4836da5c | 84c630d7ce01f086 | 1d4cfa86f2955c53 |      0 |
==============================================================================
| Interface  | expected on [iface38]                                          |
==============================================================================
| Statistics | STIMULI | REACTIONS | NORMAL | INCORRECT | MISSING | UNEXPECTED |
|------------|---------|-----------|--------|-----------|---------|------------|
| 1.12 (r/s) |       8 |         9 |      3 |         0 |     2+2 |          2 |
==============================================================================
| Simulation | 3013 cycle(s) / 1398888886 sec(s) / 0.00 Hz                    |
==============================================================================
```

Fig. 6. Result of the diagnostics subsystem work

V. Conclusion

The proposed means for trace analysis and bug visualization allows in some sense to make the verification easier. It allows to avoid extra information from the reaction trace and to show only meaningful information for verification engineers related to the occurred and examined bug in HDL designs.

Our future research is connected with localization of problems and bugs found in HDL designs using static analysis of source code.

References

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Pub, 2003.

[2] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[3] M. Chupilko and A. Kamkin, "Specification-driven testbench development for synchronous parallel-pipeline designs," in *Proceedings of the 27^{th} NORCHIP*, nov. 2009, pp. 1–4.

[4] C++tesk homepage. [Online]. Available: http://forge.ispras.ru/projects/cpptesk-toolkit/

[5] Unified verification methodology. [Online]. Available: http://www.uvmworld.org

[6] M. Chupilko and A. Protsenko, "Recognition and explanation of incorrect behaviour in simulation-based hardware verification," in *Proceedings of the 7^{th} SYRCoSE*, 2013, pp. 1–4.

[7] S. Jiang and R. Kumar, "Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications," in *IEEE Transactions on Automatic Control*, 2001, pp. 128–133.

[8] S. Tripakis, "Fault diagnosis for timed automata," in *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*, ser. FTRTFT '02. London, UK, UK: Springer-Verlag, 2002, pp. 205–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=646847.707114

[9] P. Bouyer and F. Chevalier, "Fault diagnosis using timed automata," in *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*. Springer-Verlag, 2005, pp. 219–233.

[10] M. Alpuente, D. Ballis, J. Espert, and D. Romero, "Backward trace slicing for rewriting logic theories," in *Automated Deduction CADE-23*, ser. Lecture Notes in Computer Science, vol. 6803. Springer Berlin Heidelberg, 2011, pp. 34–48.

[11] A. S. Kamkin and M. M. Chupilko, "Survey of modern technologies of simulation-based verification of hardware," *Program. Comput. Softw.*, vol. 37, no. 3, pp. 147–152, May 2011. [Online]. Available: http://dx.doi.org/10.1134/S0361768811030017

[12] Value change dump description. [Online]. Available: http://en.wikipedia.org/wiki/Value_change_dump

[13] Gtkwave. [Online]. Available: http://gtkwave.sourceforge.net

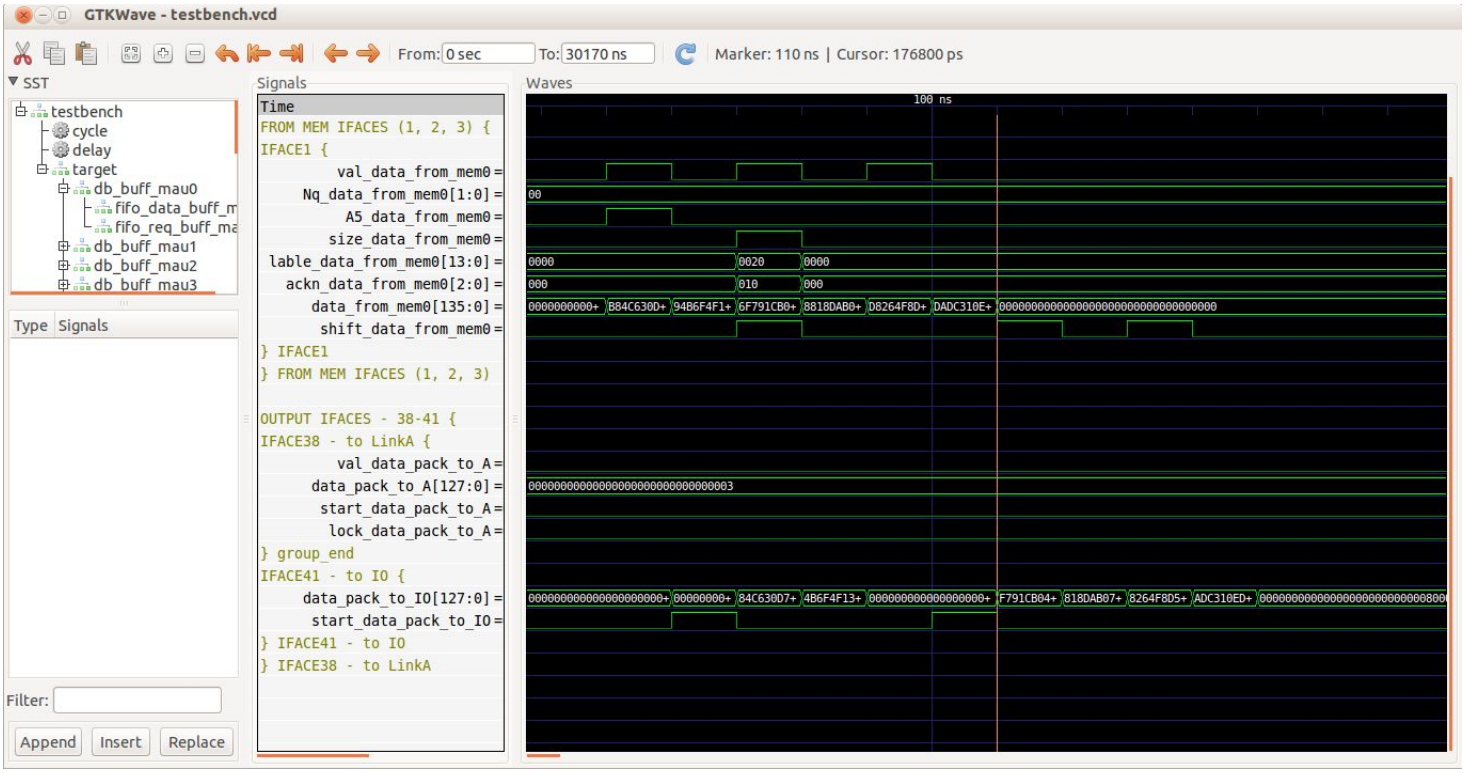

Fig. 7. Visual example of diagnostics