Recognition and Explanation of Incorrect Behavior in Simulation-based Hardware Verification

Mikhail Chupilko*, Alexander Protsenko*†
* Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
† National Research University Higher School of Economics (NRU HSE)
{chupilko,protsenko}@ispras.ru

***Abstract*—Simulation-based unit-level hardware verification is intended for dynamical checking of hardware designs against their specifications. There are different ways of the specification development and design correctness checking but it is still difficult to diagnose something more than incorrect data on some or other design outputs. The proposed approach is not only to find erroneous design behavior but also to make an explanation of incorrectness on the base of resulted reactions based on special mechanism using a list of explanatory rules.**

I. Introduction

Taking up to 80% of the total verification efforts [1], verification of HDL designs remains being very important. We expect the verification labor costs to be decreased by means of more convenient and substantial diagnostic information. The most complicated problem that underlies in all the approaches to hardware verification is how to represent the specification in machine-readable form that can be both convenient for development and useful for verification purposes. Typically, the specifications can be represented by means of temporal assertions (like in SystemVerilog in general and in Unified verification methodology [2] in particular), or using implicit contracts in form of pre- and post-conditions applied for each operation and micro-operation [3], or by means of executable models. The way of assertion usage lacks of certain incompleteness as assertions covers some of other quality and their possible violation shows only the quality without any guesses why it has happened. To guess something in this case, we should have had a bit higher representation of specifications. The way of implicit specification by means of contracts allows showing which micro operation does not work, but it is still difficult to interpret such information as such interpretation requires lower specification representation.

The executable specification can be considered as being the most useful in the error explanation. To be the most appropriate, the specification should imitate the logic architecture of HDL designs, and the test system is to have mechanisms of explanations the results of simulation. Exactly such mechanisms based on executable specifications are implemented in the proposed approach to test system development as it will be shown later.

The rest of the paper is organized as follows. The following chapter introduces the method of specification and test system development. The third chapter tells about reaction checker work. The fourth chapter reveals the theory underlying the explanatory mechanism. The fifth chapter says a few words about implementation of the approach in C++ library named C++TESK Testing ToolKit [4].

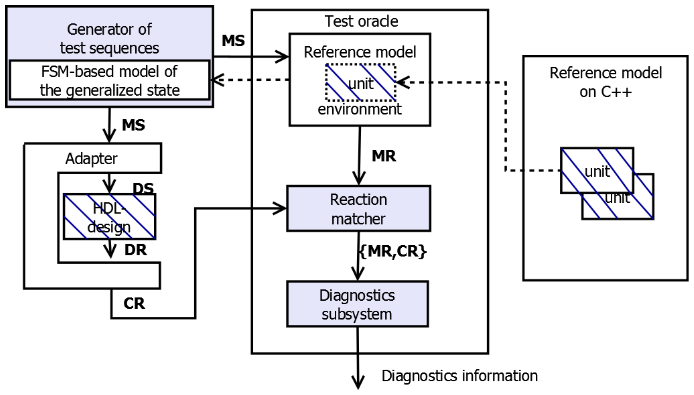

Fig. 1. Common architecture of test system

Then a few words about the approach application are given. The seventh chapter concludes the paper.

II. Specification and Test System Architecture

The typical test system for unit-level simulation-based hardware verification includes the following three parts: generator of stimuli, reaction checker, and design under verification (DUV) connected to the test system via special adapter. The proposed approach follows the same tradition but formulates properties of test system components more strictly. Let us shortly consider all the parts of the ordinary test system developed according to the approach (see Figure 1) and then review reference model development more thoroughly.

It should be noticed that fully colored elements in Figure 1 are derived from the supporting library (C++TESK), half-colored elements are developed for each DUV manually on the base of the supporting library, and white-boxed elements are developed fully manually.

Test oracle is the test system core. In fact, the test oracle works as a typical reaction checker; it receives stimuli flow from stimuli generator, receives implementation reactions (DUV reactions) enveloped into messages, and compares them with model reactions produced by reference model. Each message consists of a number of fields carrying data. Only messages with the fields of the same data types are comparable. The test oracle includes a replacement for the reference model which is called reference model environment. The environment consists of a list of operations and functional dependencies between data on output and input interfaces. The operation description is based on extension of external reference model with timing properties.

The other parts of the test oracle are reaction matcher, and diagnostics subsystem. The reaction sequence made by the reference model is processed by the reaction matcher. It consists of processes each of which processes reactions on one particular reference model interface. As each reference model reaction is bound to a particular output interface, so that all the reactions are subdivided into a set of model interfaces. A reaction arbiter is defined for each output model interface. This component orders model reaction as follows.

When the model reaction is received by the reaction matcher, special process waiting for correspondent implementation reaction is started. If the implementation reaction is found, the process asks the reaction arbiter of the interface whether it can catch the reaction. The reaction arbiter contains a list of model reactions, registered at the interface where the arbiter is defined and not yet matched to the implementation reactions. The match process asking the arbiter about possibility of catching, the arbiter checks the list and according with a strategy of reaction selection (i.e. FIFO, LIFO, data matching) permits or forbids the matching process to catch the implementation reaction.

It is the way of reaction arbitration on each output interface. If the catching is allowed, the model reaction is deleted from the arbiter‘s reaction list, and the couple of model and implementation reaction is sent to the diagnostics subsystem. If the catching is forbidden, the matching process returns to the state of looking for the next implementation reaction. If the waiting for implementation reaction timeout is reached (the timeout can be set up to each interface separately), the reaction is sent to the diagnostics subsystem alone without implementation reaction marked as *missing reaction*. Besides processes looking for implementation reactions launched by model reactions, special processes named *listeners* are launched by test system for each interface. Each listener is bound to a particular interface and works as follows.

It contains an infinite loop of receiving implementation reaction, shaping the message with the reaction data, checking whether the reaction is matched to the correspondent model reaction at the next cycle after the implementation is completely received by test system. If the matching has happened, the listener returns to its first state and starts looking for the next implementation reaction. If the listener finds out that the implementation reaction has not been taken by any model reactions, it has been waiting for a certain implementation reaction timeout, having placed the implementation reaction into special buffer, offering next model reaction to match with the given implementation reaction. If the implementation reaction timeout is reached, the reaction is sent to the diagnostics subsystem alone without model reaction marked as *unexpected reaction*.

III. Reaction Checker Algorithm

The reaction checker work can be described by means of an algorithm showing clearly its possibility of catching all visible DUV defects. To provide the algorithm, some introduction might be useful. There are two definitions, the algorithm and a theorem about the reaction checker work.

All the input and output signals of DUV (*implementation*) are subdivided into *input* and *output interfaces*. The set of input and output interfaces of the reference model (*specification*) matches the one of the implementation (In and Out). Alphabets of stimuli and reactions of the implementation and specification also match each other (X and Y). Set of implementation state (S_{impl}) and specification states (S_{spec}) speaking generally might differ but initial states of implementation and specification are marked out ($s_{impl_0} \in S_{impl}$ and $s_{spec_0} \in S_{spec}$).

Applied during testing to input interface $in \in In$ stimuli are elements of the sequence $\bar{X}^{in} = \langle (x_i, t_i) \rangle_{i=1}^{n}$, where $x_i \in X$ is a single stimulus, $t_i \in \mathbb{N}_0$ is the time mark of its application ($t_i < t_{i+1}, i = \overline{1, n-1}$). The set of stimuli sequences applied during testing to input interfaces will be denoted as $\bar{X} =< \bar{X}^{in_1}, \ldots, \bar{X}^{in_n} >$ and called *stimuli sequence*. Stimuli sequence admissibility is defined by definitional domain $Dom \subseteq \bigcup_{k=0}^{\infty} (X \times \mathbb{N}_0)^k$.

Implementation answering the stimuli sequence $\bar{X}$ produces reactions $\bar{Y}_{impl}^{out}(\bar{X}) = \langle (y'_i, t'_i) \rangle_{i=1}^{m}$ and sends them to the output interface $out \in Out$, where $y'_i \in Y$ is a single reaction, $t'_i \in \mathbb{N}_0$ is time of its sending ($t'_i < t'_{i+1}, i = \overline{1, m-1}$). Let the set of reaction sequences emitting by the implementation to all interfaces be denoted as $\bar{Y}_{impl} =< \bar{Y}_{impl}^{out_1}, \ldots, \bar{Y}_{impl}^{out_M} >$ and called *implementation reaction sequence*.

Specification answering the stimuli sequence $\bar{X}$ produces reactions $\bar{Y}_{spec}^{out}(\bar{X}) = \langle (y_i, t_i) \rangle_{i=1}^{k}$ and sends them to the output interface $out \in Out$, where $y_i \in Y$ is a single reaction, $t_i \in \mathbb{N}_0$ is time of its sending ($t_i \leq t_{i+1}, i = \overline{1, k-1}$). Let the set of reaction sequences emitting by the implementation to all interfaces be denoted as $\bar{Y}_{spec} =< \bar{Y}_{spec}^{out_1}, \ldots, \bar{Y}_{spec}^{out_K} >$ and called *specification reaction sequence*.

Let each output interface $out \in Out$ to be equipped with reaction production timeout $\Delta t^{out} \in \mathbb{N}_0$. Let also each finite stimuli sequence results in a finite reaction sequence. Let us denote single element of reaction sequence $\bar{Y} = \langle (y_i, t_i) \rangle_{i=1}^{k}$ as $\bar{Y}[i] = (y_i, t_i)$. The operation of element removing from the reaction sequence $\bar{Y} \backslash (y, t)$ is defined as follows: if the element being removed is absent in the sequence, the result consists of the former sequence; if the element is in the sequence, its first entrance in the sequence will be removed. The sequence length is denoted as $m = |\bar{Y}|$.

Definition 1: The implementation is said to correspond to the specification if $\forall out \in Out$ and $\forall \bar{X} \in Dom$ $|\bar{Y}_{impl}^{out}(\bar{X})| = |\bar{Y}_{spec}^{out}(\bar{X})| = m^{out}$ is satisfied and there is a rearrangement π^{out} of the set $\{1, , m^{out}\}$ so that $\forall i \in \{1, \ldots, m^{out}\}$ $\left\{ \begin{matrix} y'_i = y_j \\ t_j \leq t'_i \leq t_j + \Delta t^{out} \end{matrix} \right\}$ is satisfied, where $j = \pi^{out}(i)$.

Definition 2: The implementation behavior is said to have an observable failure if the implementation does not correspond to the specification or $\exists \bar{X} \in Dom$ and $\exists out \in Out$ so that either $|\bar{Y}_{impl}^{out}(\bar{X})| \neq |\bar{Y}_{spec}^{out}(\bar{X})|$, or for each rearrangement π^{out} of the set $\{1, \ldots, m^{out}\}$ $\exists i \in \{1, \ldots, m^{out}\}$ for which $\left[\begin{matrix} y'_i \neq y_j \\ t_j > t'_i \\ t'_i > t_j + \Delta t^{out} \end{matrix} \right]$ is satisfied, where $j = \pi^{out}(i)$.

Action 1 $reactionMatcher[\bar{Y}_{impl}, \bar{Y}_{spec}]$

```
Guard: $true$
Input: $\bar{Y}_{impl}, \bar{Y}_{spec}$
  $\bar{Y}^*_{spec} \Leftarrow \bar{Y}_{spec}$
  for all $i \in |\bar{Y}_{impl}|$ do
      $t^{now} \Leftarrow t'_i$
      $Y^{now}_{spec} \Leftarrow \{(y,t) | \exists j \cdot (y,t) = \bar{Y}^*_{spec}[j] \wedge t \leq t^{now}\}$
      $Y^{now}_{missing} \Leftarrow \{(y,t) \in Y^{now}_{spec} | t + \Delta t < t^{now}\}$
      if $Y^{now}_{missing} \neq \emptyset$ then
          $Y_{defect} \Leftarrow Y_{defect} \cap Y^{now}_{missing}$
      end if
      $Y^{now}_{matched} \Leftarrow \{(y,t) \in Y^{now}_{spec} | y = y'_i \wedge t \leq t'_i \leq t + \Delta t\}$
      if $Y^{now}_{matched} = \emptyset$ then
          $Y_{defect} \Leftarrow Y_{defect} \cap (y'_i, t'_i)$
      end if
      $(y_{matched}, t_{matched}) \Leftarrow argmin_{(y,t) \in Y^{now}_{matched}} t$
      $\bar{Y}^*_{spec} \Leftarrow \bar{Y}^*_{spec} \setminus (y_{matched}, t_{matched})$
  end for
  if $|\bar{Y}^*_{spec}| \neq \emptyset$ then
      $Y_{defect} \Leftarrow Y_{defect} \cap \bar{Y}^*_{spec}$
  end if
  return $Y_{defect}$
```

Lemma 1: If reaction sequence $\bar{Y}_{impl}$ and $\bar{Y}_{spec}$ are finite, and $|\bar{Y}_{impl}| \neq |\bar{Y}_{spec}|$ then test oracle returns negative verdict.

Proof: Suppose the main cycle of the algorithm not to find a failure. In this case the number of elements in sequence $\bar{Y}^*_{spec}$ (which at the first step was equal to the number of elements in $\bar{Y}_{spec}$) will be decreased to the number, which the sequence $\bar{Y}_{impl}$ contains. If $|\bar{Y}_{impl}| > |\bar{Y}_{spec}|$, then there is no step of the test oracle algorithm to find reaction from sequence $\bar{Y}_{spec}$ correspondent to current being worked under reaction from sequence $\bar{Y}_{impl}$. In this case test oracle finishes its work with negative verdict. We had supposed that such a situation cant occur, so that $|\bar{Y}_{impl}| < |\bar{Y}_{spec}|$. In this case $|\bar{Y}^*_{spec}| = |\bar{Y}_{spec}| - |\bar{Y}_{impl}| > 0$ and the oracle finishes its work with negative verdict in due to condition *if* $|\bar{Y}^*_{spec}| \neq \emptyset$ *then return (false)* after the main cycle having finished. ■

Theorem 1: Test oracle working according to the proposed algorithm allows constructing significant tests (it means that oracle is not mistaken having found certain defect).

Proof: The case when $|\bar{Y}_{impl}| \neq |\bar{Y}_{spec}|$ meaning that there are different numbers of implementation and specification reactions is considered to be erroneous according to the definition 2. It was considered in the lemma and shown that the test oracle in this case does return negative verdict.

Let us consider the case $|\bar{Y}_{impl}| = |\bar{Y}_{spec}| = 0$. Here the main cycle of test oracle work is not executed, the condition *if* $|\bar{Y}^*_{spec}| \neq \emptyset$ *then return (false)* is not satisfied too and the test oracle returns positive verdict (true). The case of empty sequences is understood as correct according to the definition 2. According to the induction rule of inference, let us suppose that for the case $|\bar{Y}_{impl}| = |\bar{Y}_{spec}| = n$ test oracle returns verdict correctly. Let us prove that the same situation takes place if the numbers of elements in sequences are equal to $n+1$. According to the definition 2, defect can be found if for each rearrangement π of set $1, \dots, n \exists i \in 1, \dots, n$, when

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	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R^{normal,incorrect}_{impl} : iface_{spec} = iface_{impl}$ & $time_{min} < time < time_{max}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R^{normal,incorrect}_{spec} : iface_{impl} = iface_{spec}$ & $time_{min} < time < time_{max}$

TABLE I. REACTION CHECKER REACTION PAIR TYPES

$\begin{bmatrix} y'_i \neq y_j \\ t_j > t'_i \\ t'_i > t_j + \Delta t^{out} \end{bmatrix}$ is satisfied, where $j = \pi(i)$.

Let us remove last elements of sequences and make sequences where the numbers of elements are equal to n and to which test oracle works correctly. Let us consider the case of the following two removed reactions. Negative verdict can be returned only in two cases: the first one is *if* $Y^{now}_{missing} \neq \emptyset$ *then return (false)*, the second one is *if* $Y^{now}_{matched} = \emptyset$ *then return (false)*. The first case occurs only when $t'_i > t_j + \Delta t^{out}$ according to the definition 2, and the second case takes place only in when $\begin{bmatrix} y'_i \neq y_j \\ t_j > t'_i \end{bmatrix}$ according to the definition 2. Therefore, test oracle returns negative verdict only when there is erroneous reaction in any finite reaction sequences. ■

IV. DIAGNOSTICS SUBSYSTEM

Let 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 timemark: generation of implementation reaction always has concrete time mark.

The reaction checker has already attempted to match each reaction from R_{spec} with a reaction from R_{impl}, making a *reaction pair*. 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 *normal, missing, unexpected, 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 simultaneously be located only in one pair.

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

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	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \backslash R_{impl}^{normal,incorrect}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal,incorrect}$

TABLE II. DIAGNOSTICS SYSTEM REACTION PAIR TYPES

set of reaction pairs marked with *type* from the list above. $M^* = \{(r_{spec}, r_{impl}, type^*)_i\}$ is a similar set of reactions pairs but with different label system. It should be noticed that these might be different M^* dependent on the algorithm of its creation (accounting for original order, strategy of reaction pair selection for recombination, etc). This question will be discussed after so called *transformation rules* are presented.

Having made reaction pair set, reaction matcher sends it to the *diagnostics subsystem* to process them providing verification engineers with explanation of problems having occurred in the verification process. The diagnostics subsystem is underlain with a special algorithm, consisting of consequent application of the set of so called *rules*, each of which transforms the reaction pairs. Some rules decrease the number of pairs, having found pairs with correspondent implementation and specification reactions, collapse them and write diagnostics information into log-file. Other rules make it possible to recombine reaction pairs for better application of rules from the first type. The third part of rules uses special technique to find similar reactions according to the *distant function* to recombine the reaction pairs for better readability but do. The distant function can be implemented in three possible ways. To begin with, it may account the number of equal data fields in two given messages. Second, Hamming distance may be used as one can compare not only the fields but the bits of data carried by the fields. The measure of closeness between two given reactions is denoted as $C(r_{spec}, r_{impl})$.

Each rule consists of one or several pairs of reactions. In cases of missing of unexpected reactions, one of the pair elements is undefined and called $null$. Each pair of reaction is assigned with model interface. Left part of the rule shows initial state and right part (after the arrow) shows result of the rule application. If the rule is applied to several reaction pairs, they are separated with comma. Now, let us review all these twelve rules that we found.

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 $(a_{spec}, b_{impl}), (b_{spec}, a_{impl})$: $data_{a_{spec}} = data_{a_{impl}}$ & $data_{b_{spec}} = data_{b_{impl}}$, these reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, b_{impl})\}$.

Rule 4: If there is a missing reaction pair and an unexpected reaction pair $(a_{spec}, null), (null, a_{impl})$: $data_{a_{spec}} = data_{a_{impl}}$, they should be united into one reaction pair. $\{(a_{spec}, null), (null, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl})\}$.

Rule 5: If there is a missing reaction pair and an incorrect reaction pair $(a_{spec}, null), (b_{spec}, a_{impl})$: $data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be regrouped. $\{(a_{spec}, null), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, null)\}$.

Rule 6: If there is an unexpected reaction pair and an incorrect reaction pair $(null, a_{impl}), (a_{spec}, b_{impl})$: $data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be regrouped. $\{(null, a_{impl}), (a_{spec}, b_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (null, b_{impl})\}$.

Rule 7: If there are two incorrect reaction pairs $(a_{spec}, b_{impl}), (c_{spec}, a_{impl})$: $data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (c_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (c_{spec}, b_{impl})\}$.

The rules 1-7 allow finding the closest reaction pairs. The algorithm of their implementation is shown in 2 and 4 algorithms.

Action 2 $match[(r1_{spec}, r1_{impl}), (r2_{spec}, r2_{impl})]$

```
Input: $RP_1 = (r1_{spec}, r1_{impl}), RP_2 = (r2_{spec}, r2_{impl})$
  for all $rule\_number \in |NormalRules|$ do
      if $rules[rule\_number].isApplicable(RP_1, RP_2)$ then
          return $rule\_number$
      end if
  end for
  return 0
```

Action 3 $fuzzy\_match[(r1_{spec}, r1_{impl}), (r2_{spec}, r2_{impl}), rule]$

```
Input: $RP_1 = (r1_{spec}, r1_{impl}), RP_2 = (r2_{spec}, r2_{impl})$
  $proximity\_metric \Leftarrow 0$
  for all $rule\_number \in |FuzzyRules|$ do
      if $(metric^* = rules[rule\_number].metric(RP_1, RP_2)) > proximity\_metric$ then
          $proximity\_metric \Leftarrow metric^*$
          $rule \Leftarrow rule\_number$
      end if
  end for
  return $proximity\_metric$
```

Action 4 $apply\_normal\_rules[\{(r_{spec}, r_{impl})_i\}]$

```
Input: $\{(r_{spec}, r_{impl})_i\}$
  for all $r \in |\{(r_{spec}, r_{impl})_i\}|$ do
      if $!r.collapsed$ then
          for all $p \in |\{(r_{spec}, r_{impl})_i|\}$ do
              if $!r.collapsed\&!p.collapsed$ then
                  if $rule\_number = match(r, p)$ then
                      $(r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}}) \Leftarrow rules[rule\_number].apply\_rule(r, p)$
                      $r.collapsed \Leftarrow true$
                      $p.collapsed \Leftarrow true$
                      return
                  end if
              end if
          end for
      end if
  end for
```

Action 5 $apply\_fuzzy\_rules[\{(r_{spec}, r_{impl})_i\}]$

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

When the rules from the list of normal rules have been applied, the sets R_{spec} and R_{impl} does not contain any not yet collapsed reactions with identical data. In this part of diagnostics subsystem work the stage of *fuzzy rules* (See 3 and 5 algorithms) comes.

Rule 8: If there are two reaction pairs $\{(a_{spec}, b_{impl}), (b'_{spec}, a'_{impl})\}$: $c(a_{spec}, a'_{impl}) < c(a_{spec}, b_{impl})$ & $c(a_{spec}, a'_{impl}) < c(b'_{spec}, a'_{impl})$ or $c(b'_{spec}, b_{impl}) < c(a_{spec}, b_{impl})$ & $c(b'_{spec}, b_{impl}) < c(b'_{spec}, a'_{impl})$, where c is the selected distance function and the value of c is the best amoung other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (b'_{spec}, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl}), (b'_{spec}, b_{impl})\}$

Rule 9: If there are two reaction pairs $\{(a_{spec}, null), (null, a'_{impl})\}$ and the value of the selected distant function $c = (a_{spec}, a'_{impl})$ is the best amoung other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, null), (null, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl})\}$

Rule 10: If there are two reaction pairs $\{(a_{spec}, null), (b_{spec}, a'_{impl})\}$: $c(a_{spec}, a'_{impl}) < c(b_{spec}, a'_{impl})$, where c is the selected distance function and the value of c is the best amoung other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, null), (b_{spec}, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl}), (b_{spec}, null)\}$

Rule 11: If there are two reaction pairs $\{(null, a_{impl}), (a'_{spec}, b_{impl})\}$: $c(a'_{spec}, a_{impl}) < c(a'_{spec}, b_{impl})$, where c is the selected distance function and the value of c is the best amoung other fuzzy rules, these reaction pairs should be regrouped. $\{(null, a_{impl}, null), (a'_{spec}, b_{impl})\} \Rightarrow \{(a'_{spec}, a_{impl}), (null, b_{impl})\}$

When all the metrics of fuzzy rules have been measured and all the most suitable rules have been applied, the time of the last rule comes.

Rule 12: If there is a reaction pair (a_{spec}, a'_{impl}) with both specification and implementation parts, it should be collapsed. $(a_{spec}, a'_{impl}) \Rightarrow (null, null)$.

The last rule allows transforming all the incorrect reaction pairs to show the diagnostics for the whole list of reaction pairs. Typically, after the application of each rule, the history of transformation is traced and then it is possible to reconstruct the parents of the given reaction pairs and all the rules they are undergone. Such a reconstruction of the rule application trace we understand as the *diagnostics information*.

V. Implementation

The proposed approach to development of test systems, reference model construction, reaction correctness checking, and diagnostics subsystem has been implemented in the open source library C++TESK Testing ToolKit [4] developed by ISPRAS. The library is developed in C++ language to be convenient for verification engineers. It contains macros enabling the engineers to develop all the parts of the test systems which should be done by hands. Some parts, like diagnostics subsystem algorithm, are hidden inside of the tool.

Results of diagnostics work are shown each time after the verification is over. Now they look like tables with all found errors and results of rule application: new reaction pair sets and the way of their obtaining.

VI. Results

The C++TESK testing toolkit including diagnostics subsystem has been used in the number of projects of industrial microprocessor development in Russia. The aim of the all approach is unit-level verification and on this level it can be a competitor to widely used UVM mentioned in the introduction.

It might be shown by the following fact. Typically, we started verification by means of C++TESK starts when the whole system had been already verified by UVM-like approaches. In spite of power of UVM, it does not include means to direct test sequence generation, which C++TESK does, means of quick analysis of verification results as diagnostics subsystem etc.

Results of application of different approaches depend on the qualification of the engineers and their familiarity with the approach. And on this point, we should say that our toolkit was used by people now being close to its development kitchen and despite it, they exactly managed to find those bugs we have already mentioned.

VII. Conclusion

The proposed approach to simulation-based unit-level hardware verification solves in some sense the task of dynamical checking of hardware designs against their specifications. It includes both means of specification development and diagnostics subsystem producing an explanation of incorrectness on the base of special mechanism using formally represented specifications and a list of explanatory rules.

The approach has been used in the number of projects and shown its possibility to find defects and help verification engineers to correct them by means of diagnostics information.

Our future research is connected with more convenient representation of diagnostics results by means of wave-diagrams, localization of found problems in source-code.

References

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

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

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

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