УДК 004.4'23

Распознавание и интерпретация ошибочного поведения при динамической верификации аппаратуры

А.С. Проценко, М.М. Чупилко

Институт системного программирования РАН, {protsenko,chupilko}@ispras.ru

***Аннотация* — Динамическая верификация аппаратуры предназначена для проверки соответствия поведения HDL-моделей аппаратуры спецификациям в процессе симуляции. Представить спецификации в необходимом виде и обеспечить проверку корректности можно различными способами, но обычно обнаруженное неправильное поведение несет в себе только констатацию факта наличия неправильных данных на выходе HDL-модели. Предложенный в статье подход предназначен не только для обнаружения некорректного поведения HDL-модели, но также и для интерпретации полученных выходных данных с помощью специального алгоритма, использующего совокупность правил их анализа.**

***Ключевые слова* — анализ трасс, динамическая верификация.**

I. Введение

Верификация HDL-описаний требует к себе тщательного внимания, занимая до 80% от всех усилий на разработку цифровой аппаратуры [1]. Снизить количество усилий, затрачиваемых на верификацию, можно при использовании более удобной и значимой диагностической информации. Наиболее сложная проблема, проявляющаяся во всех подходах к верификации – это проблема представления спецификаций в форме, удобной для разработки и полезной для целей верификации. Распространенной практикой является представление спецификаций в явной *исполнимой форме* (*системные симуляторы на языках C/C++*), неявно с помощью *темпоральных утверждений* (в SystemVerilog в общем и в *универсальной методологии верификации* (*Unified Verification Methodology* (*UVM* [2]) в частности), а также, например, с помощью *контрактов*, включающих *пред- и постусловия операций и микроопераций* [3]. При использовании утверждений модель не бывает достаточно полной, так как утверждения покрывают только некоторые свойства аппаратуры и их возможные нарушения показывают только проблему с конкретным невыполненным свойством. Чтобы получить более полную информацию в случае обнаружения расхождения, необходимо иметь несколько более абстрактные спецификации. Неявное задание модели с помощью контрактов позволяет показать, какая именно микрооперация не работает, но такую информацию все еще сложно воспринимать, так как ее интерпретация требует знания низкоуровневых деталей работы модели, отсутствующих в контрактном способе представления поведения.

Исполнимые спецификации могут рассматриваться как наиболее удобные при объяснении ошибок, так как они повторяют структуру верифицируемой HDL-модели на некотором уровне абстракции, описывают логическое разбиение аппаратуры на компоненты. Сравнивая результат работы HDL-модели и исполнимой спецификации, можно автоматизированным образом находить и пытаться объяснять наблюдаемые расхождения. Именно такой механизм, основанный на исполнимых спецификациях, реализован в предлагаемом подходе к разработке тестовых систем.

Статья организована следующим образом. Второй раздел вводит метод специфицирования и разработки тестовых систем. Третий раздел касается работы *тестового оракула*. В четвертом разделе приведена теоретическая информация о механизме анализа ошибок. В пятом разделе рассказывается о реализации подхода в инструменте верификации *C++TESK* [4]. Затем приводится информация о применении подхода на практике, и следуют выводы.

II. Спецификация и архитектура тестовой системы

Обычно тестовая система для динамической верификации *модуля аппаратуры* включает следующие три основные части: *генератор стимулов*, *тестовый оракул* и *верифицируемый компонент* (*design under verification*, *DUV*), соединенный с тестовой системой через специальный *адаптер*. Предлагаемый подход, следуя такой архитектуре, уточняет свойства компонентов тестовой системы. Далее будут кратко рассмотрены составные части тестовой системы, разработанной в соответствии с подходом (см. рис. 1; на данном рисунке закрашенные элементы являются либо внешними, либо автоматически создаваемыми, а незакрашенные создаются на основе средств специальной библиотеки), и затем более детально будет рассмотрена разработка эталонной модели.

Тестовый оракул – ядро тестовой системы. Работа тестового оракула заключается в проверке реакций: он получает *последовательность стимулов* от *генератора стимулов*, получает *последовательность реакций реализации* (*реакций DUV*) и сравнивает их с *модель-*

ными реакциями, создаваемыми *эталонной моделью*. Каждая реакция включает в себя несколько *полей данных*. Сравнивать можно только те реакции, типы данных которых совпадают. Тестовый оракул содержит специальные методы для подключения эталонной модели, которые называются *окружением эталонной модели*. Это окружение состоит из списка *операций* и функциональных зависимостей между данными на *выходных* и *входных интерфейсах*. Описание операций основано на *расширении эталонной модели временными свойствами*.

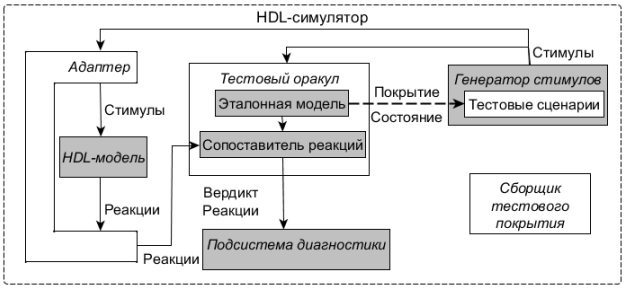

Рис. 1. Архитектура тестовой системы

Сравнение реакций тестовым оракулом осуществляется с помощью *сопоставителя реакций*, который состоит из множества независимых процессов, каждый из которых обрабатывает реакции на одном из *выходных модельных интерфейсов* (соответствующих выходным интерфейсам DUV). Для каждого выходного модельного интерфейса определяется *арбитр реакций*, упорядочивающий модельные реакции. Когда сопоставитель реакций получает модельную реакцию, запускается специальный процесс, ожидающий соответствующую реакцию реализации. Если реакция реализации обнаружена, процесс запрашивает арбитр реакций данного интерфейса, может ли он сопоставить данной реакции одну из модельных реакций. Арбитр реакций содержит список еще не сопоставленных модельных реакций, зарегистрированных на обслуживаемом им интерфейсе. После запроса о возможности сопоставления арбитр проверяет свой список в соответствии с выбранной стратегией (FIFO, LIFO, совпадение данных) и разрешает или запрещает сопоставление. Если сопоставление разрешено, модельная реакция удаляется из списка реакций арбитра, и затем пара модельная реакция - реакция реализации посылается в подсистему диагностики, а данный процесс ожидания реакции заканчивается. Если процессом ожидания превышено некоторое ранее определенное *время ожидания* (время устанавливается для каждого интерфейса по отдельности), модельная реакция отправляется в подсистему диагностики одна, без реакции реализации, с пометкой *пропущенная реакция*. Помимо процессов, ожидающих реакции реализации, запущенными по приходу модельных реакций, тестовая система запускает на каждом интерфейсе специальные процессы, называемые *прослушивателями*. Каждый прослушиватель присоединяется к конкретному выходному интерфейсу и работает следующим образом.

Он содержит бесконечный цикл приема реакций реализации и, работая независимо от процессов, запущенных сопоставителем реакций, создает реакции из всех данных, пришедших от реализации, проверяет, были ли эти реакции сопоставлены модельным реакциям в конце такта обработки данной реакции реализации. Если сопоставление произошло, прослушиватель возвращается в исходное состояние и начинает ожидание следующей реакции реализации. Если же прослушиватель обнаруживает, что реакция реализации не была сопоставлена модельной реакции, он ожидает определенный временной промежуток, поместив реакцию в специальный буфер, пытаясь сопоставить реакцию реализации с вновь создаваемыми модельными реакциями. Если же временной промежуток заканчивается, а сопоставления не произошло, реакция реализации посылается теперь уже в подсистему диагностики одна, без модельной реакции, и с пометкой *неожиданная реакция*.

Если в процессе тестирования было обнаружено расхождение между реакциями реализации и спецификации, тестовый оракул выносит отрицательный вердикт и передает полученные реакции в *подсистему диагностики* для дальнейшей обработки и вывода диагностической информации.

III. Алгоритм работы тестового оракула

Работа тестового оракула может быть описана алгоритмом, явно показывающим его способность отслеживания всех наблюдаемых ошибок DUV. Для описания алгоритма потребуется некоторое введение. Далее последуют два определения, алгоритм и теорема о работе тестового оракула.

Все входные и выходные сигналы (*реализации*) делятся на *входные* и *выходные интерфейсы*. Набор входных и выходных интерфейсов эталонной модели (*спецификации*) соответствует таковому у реализации (*In* и *Out*). Алфавиты стимулов и реакций реализации и спецификации также совпадают (*X* и *Y*). Набор состояний реализации (S_{impl}) и состояний спецификации (S_{spec}) может немного отличаться, за исключением выделенных начальных состояний реализации и спецификации ($s_{impl0} \in S_{impl}$ и $s_{spec0} \in S_{spec}$).

Поданные в процессе тестирования на входной интерфейс $in \in In$ стимулы являются элементами последовательности $\overline{X}^{in} = \langle (x_i, t_i) \rangle_{i=1}^{n}$, где $x_i \in X$ – единичный стимул, $t_i \in N_0$ – временная метка его запуска $(t_i < t_{i+1}, i = \overline{1, n-1})$. Набор последовательностей стимулов, поданных на входные интерфейсы в процессе тестирования, обозначим $\overline{X} = \left\langle \overline{X}_1^{in}, ..., \overline{X}_n^{in} \right\rangle$ и назовем *последовательностью стимулов*. Допустимость последовательности стимулов определяется областью определения $Dom \subseteq \bigcup_{k=0}^{\infty} (X \times N_0)^k$.

Реализация отвечает на последовательность стимулов $\bar{X}$ реакциями $\bar{Y}_{impl}^{out}(X) = \langle (y_i', t_i') \rangle_{i=1}^{m}$, посылая их на выходной интерфейс $out \in Out$, где $y_i' \in Y$ – единичная реакция, $t_i' \in N_0$ – время ее отправки $(t_i' < t_{i+1}', i = \overline{1, m-1})$. Обозначим последовательность реакций, посылаемую реализацией на все интерфейсы, $\bar{Y}_{impl} = \left\langle \bar{Y}_{impl}^{out_1}, ..., \bar{Y}_{impl}^{out_M} \right\rangle$ и назовем ее *последовательностью реакций реализации*.

Спецификация отвечает на последовательность стимулов $\bar{X}$ реакциями $\bar{Y}_{spec}^{out}(X) = \langle (y_i, t_i) \rangle_{i=1}^{k}$, посылая их на выходной интерфейс $out \in Out$, где $y_i \in Y$ – единичная реакция, $t_i \in N_0$ – время ее отправки $(t_i \le t_{i+1}, i = \overline{1, k-1})$. Обозначим последовательность реакций, посылаемую спецификацией на все интерфейсы, как $\bar{Y}_{spec} = \left\langle \bar{Y}_{spec}^{out_1}, ..., \bar{Y}_{spec}^{out_K} \right\rangle$, и назовем ее *последовательностью реакций спецификаций*.

Пусть у каждого выходного интерфейса $out \in Out$ есть временной интервал максимального ожидания реакции реализации $\Delta t^{out} \in N_0$. Пусть также каждая конечная последовательность стимулов приводит к конечной последовательности реакций. Обозначим единичный элемент последовательности реакций $\bar{Y} = \langle (y_i, t_i) \rangle_{i=1}^{k}$ как $\bar{Y}[i] = (y_i, t_i)$. Операция удаления элемента из последовательности реакций $\bar{Y} \setminus (y, t)$ определяется следующим образом: если удаляемый элемент отсутствует в последовательности, результатом удаления является исходная последовательность; если элемент присутствует в последовательности, будет удалено его первое вхождение в последовательность. Длина последовательности обозначается $m = \left| \bar{Y} \right|$.

Определение 1 *Говорят, что реализация соответствует спецификации, если* $\forall out \in Out$ *и* $\forall X \in Dom$ *выполняется* $\left| \bar{Y}_{impl}^{out}(X) \right| = \left| \bar{Y}_{spec}^{out}(X) \right| = m^{out}$ *и существует перестановка* π^{out} *множества* $\{1, ..., m^{out}\}$, *такая что* $\forall i \in \{1, ..., m^{out}\}$ *выполняется* $\left\{ \dfrac{y_i' = y_j}{t_j \le t_i' \le t_j + \Delta t^{out}} \right\}$, *где* $j = \pi^{out}(i)$.

Определение 2 *Говорят, что поведение реализации имеет наблюдаемую ошибку, если реализация не соответствует спецификации, то есть* $\exists X \in Dom$ и $\exists out \in Out$, *такие что либо* $\left| \bar{Y}_{impl}^{out}(X) \right| \ne \left| \bar{Y}_{spec}^{out}(X) \right|$, *либо для любой перестановки* π^{out} *множества* $\{1, ..., m^{out}\}$ $\exists i \in \{1, ..., m^{out}\}$ *для которого выполняется* $\begin{bmatrix} y_i' \ne y_j \\ t_j > t_i' \\ t_i' > t_j + \Delta t^{out} \end{bmatrix}$, *где* $j = \pi^{out}(i)$.

На рис. 2 приводится алгоритм работы тестового оракула, осуществляющий сопоставление реакций в соответствии с данными выше определениями. Ниже приводится и доказываются лемма и теорема о работе такого алгоритма.

Если : *true*

Вход : $\bar{Y}_{impl}$, $\bar{Y}_{spec}$

$\bar{Y}_{spec}^{*} \Leftarrow \bar{Y}_{spec}$

for all $i \in \left| \bar{Y}_{impl} \right|$ **do**

$t^{now} \Leftarrow t'_i$

$Y_{spec}^{now} \Leftarrow \left\{ (y,t) \middle| \exists j \cdot (y,t) = \bar{Y}_{spec}^{*}[j] \wedge t \le t^{now} \right\}$

$Y_{missing}^{now} \Leftarrow \left\{ (y,t) \in Y_{spec}^{now} \middle| t + \Delta t < t^{now} \right\}$

if $Y_{missing}^{now} \ne \varnothing$ **then**

$Y_{defect} \Leftarrow Y_{defect} \cup Y_{missing}^{now}$

endif

$Y_{matched}^{now} \Leftarrow \left\{ (y,t) \in Y_{spec}^{now} \middle| y = y'_i \wedge t \le t'_i \le t + \Delta t \right\}$

if $Y_{matched}^{now} \ne \varnothing$ **then**

$Y_{defect} \Leftarrow Y_{defect} \cup (y'_i, t'_i)$

endif

$(y_{matched}, t_{matched}) \Leftarrow \arg\min_{(y,t) \in Y_{matched}^{now}} t$

$\bar{Y}_{spec}^{*} \Leftarrow \bar{Y}_{spec}^{*} \setminus (y_{matched}, t_{matched})$

endfor

if $\left| \bar{Y}_{spec}^{*} \right| \ne \varnothing$ **then**

$Y_{defect} \Leftarrow Y_{defect} \cup \bar{Y}_{spec}^{*}$

endif

return Y_{defect}

Рис. 2. Алгоритм работы тестового оракула

Лемма 1 *Если последовательности реакций* $\bar{Y}_{impl}$ *и* $\bar{Y}_{spec}$ *конечны и* $|\bar{Y}_{impl}| \ne |\bar{Y}_{spec}|$, *то тестовый оракул возвращает отрицательный вердикт (говорит о наличии ошибки, возвращая ошибочные реакции).*

Доказательство. Предположим, что основной цикл работы алгоритма не обнаруживает ошибку. В этом случае число элементов в последовательности $\bar{Y}^{*}_{spec}$ (которая на первом шаге равна количеству элементов в $\bar{Y}_{spec}$) будет уменьшено на число реакций, которое содержит последовательность $\bar{Y}_{impl}$. Если $|\bar{Y}_{impl}| > |\bar{Y}_{spec}|$, то у тестового оракула нет возможности обнаружить реакцию из последовательности $\bar{Y}_{spec}$, соответствующую текущей обрабатываемой $\bar{Y}_{impl}$. В этом случае тестовый оракул завершает работу с отрицательным вердиктом. Мы полагали, что подобная ситуация не может произойти, поэтому $|\bar{Y}_{impl}| < |\bar{Y}_{spec}|$. В этом случае $|\bar{Y}^{*}_{spec}| = |\bar{Y}_{spec}| - |\bar{Y}_{impl}| > 0$ и тестовый оракул также завершает работу с отрицательным вердиктом по усло-

вию $|\bar{Y}^*_{spec}|\neq 0$ после того, как основной цикл работы алгоритма будет завершен. □

Теорема 1 *Тестовый оракул, работающий в соответствии с предложенным алгоритмом, строит значимые тесты (то есть, тестовый оракул не ошибается, обнаруживая ошибку).*

Доказательство. Случай $|\bar{Y}_{impl}|\neq|\bar{Y}_{spec}|$, когда имеется разное количество реакций реализации и спецификации, считается ошибкой согласно определению 2. Он рассмотрен в лемме: в ней показано, что оракул в этом случае возвращает отрицательный вердикт.

Рассмотрим случай $|\bar{Y}_{impl}|=|\bar{Y}_{spec}|=0$. В этом случае основный цикл работы оракула не выполняется, условие $|\bar{Y}^*_{spec}|\neq 0$ также не выполняется, и оракул возвращает пустое множество ошибочных реакций. Случай пустых последовательностей с точки зрения определения 2 действительно трактуется как отсутствие ошибки. Согласно правилу вывода по индукции, предположим, что для случая $|\bar{Y}_{impl}|=|\bar{Y}_{spec}|=n$ оракул корректно возвращает вердикт. Докажем, что то же самое происходит для длины последовательностей, равной $n+1$. В соответствии с определением 2, ошибка выявляется в том случае, если для любой перестановки π множества $\{1,...,n\}\exists i \in \{1,...,n\}$, для которого выполняется $\begin{bmatrix} y'_i \neq y_j \\ t_j > t'_i \\ t'_i > t_j + \Delta t^{out} \end{bmatrix}$, где $j=\pi(i)$. Удалим последние элементы последовательностей и получим случай длины последовательностей, равной n, для которой оракул работает корректно. Рассмотрим алгоритм работы оракула при наличии двух отброшенных реакций. Отрицательный вердикт может возвращаться только в двух случаях: если $Y^{now}_{missing} \neq \varnothing$ или $Y^{now}_{matched} \neq \varnothing$. Первое условие возникает только в случае $t'_i > t_j + \Delta t^{out}$ из определения 2, а второе – только в случаях $\begin{bmatrix} y'_i \neq y_j \\ t_j > t' \end{bmatrix}$ из определения 2. Следовательно, оракул возвращает отрицательный вердикт только в случае наличия ошибки на любых конечных последовательностях реакций. □

Таким образом, было показано, что тестовая система позволяет находить ошибки, и эти ошибки ей идентифицируются безошибочно.

IV. Подсистема диагностики

Пусть тестовый оракул использует два множества реакций: $R_{spec}=\{r_{spec_i}\}_{i=0}^{N}$ и $R_{impl}=\{r_{impl_j}\}_{j=0}^{M}$. Каждая спецификационная реакция состоит из четырех элементов: r_{spec}=(*data*, *iface*, $time_{min}$, $time_{max}$). Каждая реализационная реакция включает только три элемента: r_{impl}=(*data*, *iface*, *time*). Отметим, что $time_{min}$ и $time_{max}$ показывают интервал, где спецификационная реакция валидна, в то время как *time* соответствует единичной временной метке: создание реализационной реакции всегда происходит в определенный момент времени.

Тестовый оракул уже пробовал сопоставить каждую реакцию из R_{spec} реакции из R_{impl}, создавая *пару реакций*. Если для данной реакции не было найдено либо соответствующей спецификационной, либо реализационной, тестовый оракул создает псевдо-пару реакций только с одной реакцией. Каждой паре реакций присвоен определенный тип ситуации из списка: *нормальная*, *пропущенная*, *неожиданная*, *некорректная*.

Для данных реакций $r_{spec}\in R_{spec}$ и $r_{impl}\in R_{impl}$ эти типы могут быть описаны так, как это сделано в табл. 1. Заметим также, что каждая реакция может находиться только в одной паре реакций. В подсистеме диагностики на типы пар реакций накладываются более общие требования (см. табл. 2).

Таблица 1

Пары реакций тестового оракула

Имя типа	**Пара реакций**	**Описание типа**
Нормальная *NORMAL*	(r_{spec},r_{impl})	$data_{spec}=data_{impl}$ & $iface_{spec}=iface_{impl}$ & $time_{min}<time<time_{max}$
Некорректная *INCORRECT*	(r_{spec},r_{impl})	$data_{spec}\neq data_{impl}$ & $iface_{spec}=iface_{impl}$ & $time_{min}<time<time_{max}$
Пропущенная *MISSING*	$(r_{spec},NULL)$	$\nexists r_{impl}\in R_{impl}\setminus R^{normal,incorrect}_{impl}$: $iface_{spec}=iface_{impl}$ & $time_{min}<time<time_{max}$
Неожиданная *UNEXPECTED*	$(NULL,r_{impl})$	$\nexists r_{spec}\in R_{spec}\setminus R^{normal,\ incorrect}_{spec}$: $iface_{impl}=iface_{spec}$ & $time_{min}<time<time_{max}$

Таблица 2

Пары реакций подсистемы диагностики

Имя типа	**Пара реакций**	**Определение типа**
Нормальная *NORMAL*	(r_{spec},r_{impl})	$data_{spec}=data_{impl}$
Некорректная *INCORRECT*	(r_{spec},r_{impl})	$data_{spec}\neq data_{impl}$
Пропущенная *MISSING*	$(r_{spec},NULL)$	$\nexists r_{impl}\in R_{impl}\setminus R^{normal,\ incorrect}_{impl}$
Неожиданная *UNEXPECTED*	$(NULL,r_{impl})$	$\nexists r_{spec}\in R_{spec}\setminus R^{normal,\ incorrect}_{spec}$

Подсистема диагностики переводит пары реакций, полученные от тестового оракула, в новое представление. Этот процесс может быть описан как $\{(r_{spec},r_{impl},type)_i\}\Rightarrow\{(r_{spec},r_{impl},type^*)_i\}$, где *type* – тип пар реакций в соответствии с табл. 1 и 2. Необходимо заметить, что новые типы могут присваиваться разными способами в зависимости от алгоритма их создания (учет исходного порядка или его отсутствие, исполь-

зуемая метрика). Создав пару реакций, тестовый оракул посылает ее в *подсистему диагностики* для дальнейшей обработки, сигналом к запуску которой является аварийное завершение теста. В основе подсистемы диагностики лежит последовательное применение набора *правил перегруппировки пар реакций*. Перед применением правил проводится оценка близости данных реакций в соответствии с *метрикой близости*.

Метрика 1 *Подсчет количества одинаковых полей в данных реакций.*

Метрика 2 *Подсчет количества полностью совпавших бит в данных реакций (расстояние Хэмминга).*

Метрика 3 *Информационное сравнение данных в реакциях: оценка не только совпадающих данных, но и их порядка, и расположения относительно друг друга.*

Метрика близости между двумя данными реакциями обозначается как $C(r_{spec},r_{impl})$. Общий алгоритм применения правил заключается в том, чтобы на первом этапе объединять и преобразовывать реакции, у которых данные полностью совпадают (метрика близости максимальна). Особенностью этой части алгоритма является то, что тут важен приоритет применения правил. Вторым этапом алгоритма являются операции над всеми оставшимися и вновь полученными парами, где важен уже не приоритет правил, а количественные метрики близости реакций.

Теперь рассмотрим правила, используемые для диагностики. Каждое правило обрабатывает одну или несколько пар реакций. В случаях пропущенной или неожиданной реакций один из элементов пары не определен и обозначается *null*. Левая часть правила показывает исходное состояние, а правая часть (после стрелки) – результат применения правила. Если правило применяется к нескольким парам реакций, они разделяются запятой.

Правило 1. Если есть пара *схлопнутых реакций*, она должна быть удалена из списка пар реакций. $(null,null)\Rightarrow\varnothing$.

Правило 2. Если есть нормальная пара реакций $(a_{spec},a_{impl}):data_{a_{spec}}=data_{a_{impl}}$, она должна быть схлопнута. $(a_{spec},a_{impl})\Rightarrow(null,null)$.

Правило 3. Если есть две пары некорректных реакций со взаимной корреляцией данных, реакции должны быть перегруппированы $\{(a_{spec},b_{impl}),(b_{spec},a_{impl})\}$: $c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl})$ & $c(a_{spec}, a_{impl}) < c(b_{spec}, a_{impl})$ или $c(b_{spec}, b_{impl}) < c(a_{spec}, b_{impl})$ & $c(b_{spec}, b_{impl}) < c(b_{spec}, a_{impl})$ (эта близость наилучшая среди остальных пар и правил): $\{(a_{spec}, b_{impl}),(b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, b_{impl})\}$

Правило 4. Если есть пропущенная реакция и неожиданная реакция со взаимной корреляцией данных, такие реакции должны быть объединены в одну пару реакций: $(a_{spec}, null)$, $(null, a_{impl})$ и $c=(a_{spec},a_{impl})$ – наилучшая среди других пар и правил: $\{(a_{spec}, null), (null, a_{impl})\}\Rightarrow\{(a_{spec}, a_{impl})\}$

Правило 5. Если есть пропущенная реакция и пара некорректных реакций со взаимной корреляцией данных, такие реакции должны быть перегруппированы: $(a_{spec}, null)$, (b_{spec}, a_{impl}) и $c(a_{spec},a_{impl}) < c(b_{spec},a_{impl})$, (эта близость наилучшая среди остальных пар и правил): $\{(a_{spec}, null), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, null)\}$

Правило 6. Если есть неожиданная реакция и пара некорректных реакций, такие реакции должны быть перегруппированы: $(null, a_{impl})$, (a_{spec}, b_{impl}) и $c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl})$ (эта близость наилучшая среди остальных пар и правил): $\{(null, a_{impl}), (a_{spec}, b_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (null, b_{impl})\}$

Если в результате применения правил остаются необработанные реакции, к которым не может быть применено правило 1, они обрабатываются его разновидностью, не требующей полного совпадения данных.

После применения каждого правила обновляется история преобразований, а затем на ее основе можно восстановить предков каждой пары реакций и все правила, которые были для них применены. Подобный анализ истории применения правил порождает диагностическую информацию, обрабатываемую инженерами-тестировщиками и передаваемую инженерам-разработчикам. Описанный способ получения диагностической информации после тестирования значимыми тестовыми наборами позволяет обратить внимание разработчиков на действительно существующие ошибки и сэкономить время на их локализацию.

V. Реализация подхода

Предлагаемый подход к разработке тестовых систем, созданию эталонных моделей, проверке корректности реакций и работе подсистемы диагностики был реализован в инструменте с открытым исходным кодом C++TESK [4], разработанном в ИСП РАН. Инструмент разработан на языке C++ для удобства инженеров-тестировщиков. Библиотека инструмента содержит набор макросов, которые позволяют инженерам разрабатывать все части тестовых систем, у которых предполагается ручная разработка. Некоторые части, такие как реализация алгоритма работы подсистемы диагностики, скрыты внутри инструмента.

Средства создания включают макросы для описания эталонных моделей устройств с произвольной точностью моделирования, адаптеров эталонных моделей, сценариев тестирования (настаивающих встроенные в инструмент генераторы). Упрощенный пример эталонной модели для буфера FIFO с 2 входными интерфейсами и 1 выходным, а также с двумя операциями (прием и выдача данных) выглядит следующим образом:

```
CPPTESK_MODEL(FIFO) {
public:
```

```
    FIFO();
    virtual ~FIFO();
    CPPTESK_DECLARE_INPUT(input_iface1, I1Data);
    CPPTESK_DECLARE_INPUT(input_iface2, I2Data);
    CPPTESK_DECLARE_OUTPUT(output_iface3,O3Data);
    CPPTESK_DECLARE_STIMULUS(push_msg) {
        START_STIMULUS(); // подача данных
        CYCLE();
        STOP_STIMULUS(); }
    CPPTESK_DECLARE_REACTION(get_pop_msg);
};
```

Разработанная тестовая система автоматизированным образом подключается к тестируемому HDL-описанию, и они вместе запускаются в HDL-симуляторе. При нахождении ошибки тестирование останавливается и выводится результат работы подсистемы диагностики. Помимо истории применения правил, результат выглядит как таблицы, содержащие все обнаруженные ошибки, и результаты применения правил: новые пары модельных и реализационных реакций и путь их получения (см. рис. 3, где показывается, что неожиданная реакция, пришедшая на интерфейс iface41, ожидалась на интерфейсе iface38 – результат схлопывания неожиданной и пропущенной реакции).

```
================================================================================
| 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                       |
================================================================================
```

Рис. 3. Консольная диагностическая информация

VI. Использование подхода

Инструмент тестирования C++TESK, включающий подсистему диагностики, был использован в ряде проектов по разработке индустриальных микропроцессоров. Целью подхода является модульная верификация, и здесь он может соревноваться с широко используемой в отрасли методологией UVM, упомянутой во введении. Так, обычно, мы начинали верификацию средствами C++TESK, когда вся система была уже протестирована с помощью методологии UVM или схожими с ней, и находили ряд серьезных ошибок (статистика по выполнению нами некоторых проектов представлена в табл. 3), пропущенных предыдущими подходами; они достаточно быстро исправлялись разработчиками модулей, в том числе благодаря диагностической информации. Несмотря на ее значительные возможности, UVM не включает в себя средства создания направленной тестовой последовательности, которые имеет C++TESK, средства быстрого анализа результатов верификации наподобие подсистемы диагностики. Конечно, результаты применения различных подходов зависят от квалификации инженеров и их знакомства с данным подходом (а сложность освоения C++TESK составляет несколько недель). И здесь надо сказать, что часто наш подход использовался специалистами которые верификацией ранее не занимались, и именно им удалось обнаружить упомянутые ошибки. Теоретических данных о пределе масштабируемости нет, так как алгоритм работы тестового оракула проверяет данные отдельно на каждом интерфейсе на каждом такте, соответственно, тестирование может продолжаться сколь угодно долго, а на практике самой трудной задачей является построение эталонных моделей при отсутствующей документации.

Таблица 3

Результаты верификации

Название модуля	**Стадия разработки**	**Трудозатраты**	**Объемы реализации спецификации**	**Ошибки**
(1) Коммутатор процессор–память	Завершающая	12 чел.-недель	5/9 KLOC	2
(2) Коммутатор процессор-память	Поздняя	16 чел.-недель	7/5 KLOC	23
TLB	Поздняя	48 чел.-недель	8/10 KLOC	25
Буфер команд	Поздняя	32 чел.-недель	10/6 KLOC	6
Контроллер прерываний	Средняя	24 чел.-недель	2/7.5 KLOC	9

VII. Выводы

Предлагаемый подход к динамической верификации аппаратуры решает задачу динамической проверки HDL-описаний на соответствие их спецификациям в том случае, когда на этапе создания тестовой системы присутствует полная документация на устройство, либо оно может быть проверено абстрактной эталонной моделью. Подход включает в себя как средства создания спецификаций, так и подсистему диагностики, объясняющую причины несоответствий с помощью специального механизма, использующего формально представленные спецификации и список правил объяснения. Подход был использован в ряде проектов и показал эффективность в нахождении ошибок и был полезен при их исправлении за счет диагностической информации. Наше дальнейшее исследование касается более удобного представления результатов диагностики средствами временных диаграмм и локализации обнаруженных проблем в исходном HDL-коде.

Литература

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

[2] Accellera: Сайт, посвященный методологии UVM. http://www.uvmworld.org, дата обращения 30.04.14.

[3] Chupilko, M., Kamkin, A. Specification-Driven Testbench Development For Synchronous Parallel-Pipeline Designs // Proceedings of the 27th NORCHIP. 2009. С. 1–4.

[4] ИСПРАН: Сайт, посвященный инструменту C++TESK. http://forge.ispras.ru/projects/cpptesk-toolkit, дата обращения 30.04.14.

[5] Баратов Р. и др. Трудности модульной верификации аппаратуры на примере буфера команд микропроцессора «Эльбрус-2S» // Вопросы радиоэлектроники. Серия ЭВТ. 2013. Вып. 3. С. 84-96.