A Proposal for Transaction-Level Verification with Component Wrapper Language

Koji Ara, Kei Suzuki
Central Research Laboratory, Hitachi, Ltd., Tokyo, Japan
{k-ara, ksuzuki}@crl.hitachi.co.jp

Abstract
We propose a new approach to accelerate transaction level verification by raising the productivity of the verification suites including test patterns, protocol checker, and simulation-coverage analyzer. This approach combines the conventional transaction level language such as C and the signal level language based on our previously developed Component Wrapper Language (CWL).

This approach is based on two concepts. The first one is a complete separation between transaction-level verification and signal-level verification for generating suitable verification suites in each design phase. The second one is the quick generation of signal-level verification suites from the original specification written in CWL. Experimental results show that our approach should yield much shorter verification periods versus conventional methods.

1 Introduction
System-on-Chips are used in fields where changes are frequent, such as digital controllers for home-electrical appliances and portable equipment, so that short-periods of development are required. However, the scales of the designs are increasing, and it is difficult to forecast design periods in man-hours. The main reason is the correspondingly larger scale of logical verification tasks. Verification takes up a large part of the total design period. In one case known to the authors, verification took up one third of a 19-month design period. It is difficult to set fixed periods for verification, and verification makes a big contribution to overruns of scheduled timing in design. In the above example, two months were scheduled for verification. In fact, this was extended three times, initially to four months, then to five months, and finally to six months. This represented about half of the total overrun of scheduled timing.

A various tools for use in verification have been placed on the market. In particular, transaction-level methods of verification are coming to play a major role[1][2]. Transaction-level verification raises the level of verification, from cycle-by-cycle checking of the correctness of individual signals to the handling of sequences of signals as transactions. However, the lack of fixed procedures and criteria for verification is a big problem. The work of verification is divided into three areas: making test patterns, validation of the test, and verification-coverage analysis. In conventional transaction-level verification, these works must be covered at two levels: the transaction-level and the signal-level. This creates large numbers of individual tasks. This can lead to problems such as insufficient time for verification to reach the transaction level, or poor quality for the final design. Additionally, limitations in the range of tools used in the various areas may mean that verification misses some points.

One solution is to write the required verification items (called verification suites) to cover all six tasks in a single language such as transaction-level language (TestBuilder[3] and SystemC[4], etc.) or a verification-specific language (e and VERA[5], etc.). However, there are still problems:
• effectiveness of transaction-level verification can decrease if items in the verification suites are mixed in the description
• huge numbers of man-hours required for making the verification environment

We propose new concepts to solve these problems.
(1) Level-specific verification: separating transaction-level from signal-level verification by creating verification suites using languages specialized for each level. We believe that this approach raises the capability of verification in each level.
(2) Specification-based verification: generating verification suites required to support tasks (e.g. making of test patterns) from a single description of the specification. This reduces the man-hours required in making the verification suites, and raises the quality of verification. We use previously developed Component Wrapper Language (CWL)[6] at the signal-level verification because CWL is suitable for the level.

In this paper, we evaluate the above concepts by implementing several items and applying them to the verification of actual design. In Section 2, we describe current problems with verification. In section 3, we briefly describe the implementation of specification-based verification at the signal-level with CWL, including generation of test-pattern definitions,
simulation checker, and coverage analyzer. In section 4, we estimate the effectiveness of our concepts.

2 Current problems with verification

Figure 1 shows the typical flow of verification that we assume in this paper. The works of verification in this figure is divided into three parts[7]: making of test patterns, validation of the test, and analysis of verification coverage.

Verification tasks are shown in Figure 2. Two or more tools on the market are generally available for each task. However, there is no consensus on which tools are the best for the respective tasks. Tools thus have to be selected for each design facility. The choice may lead to various inconveniences. For example, conversion of descriptions for use with tools for different purposes may take a lot of time. Complete verification may not be possible because of the capabilities of a tool.

One solution is to write the verification suites to cover all six areas in a single language (TestBuilder, SystemC, e, and VERA, etc.). However, this solution can cause the following problem. Although these languages have the ability to describe the six items in a verification suite separately, if the user is not careful, these items can be mixed. In this case, the effectiveness of transaction-level verification can decrease because verification is done at the signal-level.

Another problem concerns numbers of man-hours. Even if a single language is used, the user must describe the individual items for all six areas according to the purpose of the verification. This takes huge numbers of man-hours.

3 Signal-level verification with CWL

We propose the following new concepts to solve current problems.

(1) Level-specific verification: separating transaction-level from signal-level. We believe that this approach raises the efficiency of the transaction level verification.

(2) Specification-based verification: generating verification suites required to support verification tasks from a single description of the specification. This reduces the man-hours required in making the many verification suites, and raises the quality of verification.

In this section, we describe the means we have provided to support tasks in three areas, that is, in the making of test patterns, validation of the tests, and verification-coverage analysis. The following tasks for CWL correspond to items in the three areas.

- Generation of definitions of test patterns
- Generation of the simulation checker
- Generation of the simulation coverage analyzer

By using these methods, we can verify the following steps (shown in Figure 3):

(1) Building up a scenario by calling test pattern definitions (with expected values)

(2) Generating a signal-test pattern from the scenario

(3) Generating a simulation checker from CWL

(4) Analysis of the simulation coverage with CWL

3.1 The interface-specification language CWL

We write statements in our interface specification description language CWL (Component Wrapper Language)[6] to embody the signal-level specification. We have originally developed an interface specification language called OwL (Object Wrapper Language)[8] and design-support tools to improve the efficiency of the IP-based design. CWL is based on OwL and includes several enhancements.

CWL is used to define changes in the signals on the terminal of a logical module (IP) in its respective functional states. Changes in signals can be represented by a timing chart (Figure 4).

In CWL, a regular-expression-based syntax is used to represent signal sequences. Thompson’s algorithm is...
used to convert regular expressions into automata. The use of automata makes various tools possible. For example, a timing diagram can be obtained by tracing automata’s states from a first state to the corresponding final state. Aside from tracing the automata’s response to signal sequences, whether or not a given sequence is defined in a CWL description can be checked.

Figure 4: Concept of the CWL

3.2 Generating test-pattern definitions

The way definitions of test patterns are generated by CWL is described below. As was described in section 3, a signal pattern can be generated by tracing the automata’s states. The user can easily make the signal pattern from a pattern generated as a task definition in the HDL. For instance, a description of the timing charts of Figure 5(a) in CWL might take the form given in (b). Task definition (c) is then automatically generated from this CWL description. As a result, the user can use forms like those shown in (d) to describe the pattern, rather than considering the signal-level definition (c).

The simple algorithm for generating test pattern definitions is shown in Figure 6. When the pattern definitions are generated by tracing the automaton, the important point is how to decide the route at the branch. First of all, the branches are classified into two types.

(1) The route is uniquely decided by output value from target module.

(2) The route is decided independent from the output value from target module (called non-deterministic).

For type (1), it is necessary to synchronize with the output signal from the target module during simulation. In the proposed algorithm, it is possible to synchronize only in the case of the loop of a single alphabet.

For type (2), two kinds of approaches exist to decide the route of the task.

(a) Generate a task by deciding a route uniquely

(b) Generate a task so that the route is decided during simulation.

Additionally, two ways for deciding the route exist.

(i) Decide a route randomly.

(ii) Decide a route by specifying parameters (number of loop repetitions or the alphabet sequence) when generating the task or when calling the task (specified by the arguments and parameters).

In this paper, we implement approach (a), and we can decide a route randomly and by specifying it when generating. The following limitations for demand on the pattern generation work exist and need improvement.

- The synchronization of output signals is limited; the algorithm’s operation does not correspond to synchronization in one cycle.
- Patterns which may be generated correspond to the range of description. So, if only legal state have been specified, only legal pattern can be generated. In general, illegal input test patterns must be covered in verification, but this is not possible with a description of the above type.
We need to generate various patterns by hand if we raise the verification-coverage. To raise the effectiveness of the pattern generation, the following enhancements can be applied.

(1) Random generation in a simulation
(2) Generate patterns as a result of coverage analysis. For example, decide a route by searching the edge in the automaton that has not occurred.

**Figure 6: Algorithm for the generation of pattern definitions**

3.3 Generating simulation checkers

Simulation checkers are generated from CWLs in the way described below. As was described in the previous subsection 3.1, whether or not a signal sequence has been defined in a CWL can be judged by tracing the automaton generated from the description. We implement this method as a checker-generation program which generates checker modules in the HDL. By connecting such modules to the systems and then simulating the system, errors in the module operation and external input values can be detected.

For instance, Figure 7 shows the checker generated for the CWL description of Figure 5(b). The checker has a state machine that represents the word section of the description. The state machine is traced according to the values on the ports. If tracing is not possible for some input signals, a protocol error to have occurred is considered, and the checker outputs an error message.

This checker generator adds the following functions to the automaton, which is generated by Thompson’s algorithm, and generates the automaton as HDL code.

- **On the rising edge of the clock signal, compare the current signal value with the value in the alphabet which connects the current and next states.**
  - If the values match, make the edge transition, and continue the process.
  - Otherwise, an error has occurred: output an error message, and break the process.

The following limitations apply to the checker function in its current form.

When a given transition does not occur in the automaton, there is clearly some kind of error. However, there is no way to find what kind of specification has been violated. The language specification is being enhanced to allow the direct writing of restrictions on specifications.

For example, in a req-ack handshake protocol, the following specification would apply after the first req: req can’t be asserted before ack is asserted. Req must thus remain negated for at least 1 cycle after ack has been asserted. If the simulation violates such a specification, that specification should be indicated. However, when the checker has been made from regular expressions, the reasons for errors can’t be found. The only indication is something like “right value is 0.”

**Figure 7: The HDL checker generated from the CWL description of figure 5(b)**

3.4 Generating verification-coverage analyzers

The use of CWL to generate simulation coverage analyzers is now described.

In general, verification coverage is measured objectively by using various metrics. With an automaton, coverage metrics can be defined in terms of the structure of the automaton and the possible values in the contexts (such as address or data). For example, degrees of coverage of states or coverage of state-transitions can be defined. Coverage can be obtained by adding flags to indicate particular states and transitions for inclusion in the simulation model. This approach to coverage allows us to widen the scope of simulation.

Coverage metrics can be defined as follows:

(1) **Structure-oriented**: Coverage metrics that are defined on the basis of the automaton’s structure. In the software for verification, we can define coverage of all states (called C0 coverage) and coverage of all transitions (C1 coverage). In hardware verification, we wish to examine result of the occurrence of all possible transactions and all possible combinations of control signals in each transaction. Additionally, if this set of coverage metrics is expanded to include sequences of transitions (“path coverage”), we will be able to get results for the occurrence of sequences of transactions.

(2) **Context-oriented**: Coverage metrics in terms of
Coverage can be obtained after simulation from a log output during the simulation. In this paper, we implement four simple coverage metrics:
- single transaction (C0 coverage of transactions)
- transition between two transactions (C1 coverage of transactions)
- single state in transaction (C0 coverage within transactions)
- transition between two states within a transaction (C1 coverage within transactions)

4 Experimental analysis
4.1 Methodology

We have applied the tools described above to the design that is used in a commercial microprocessor and evaluated the following results.

(1) The effectiveness of separating transaction-level from signal-level (Level-specific verification): We have verified the target module by three method.
   (a) Using patterns with signal-level stimuli.
   (b) Using signal-level specification in CWL, generated pattern definitions from it, and hand coding scenarios by calling these definitions.
   (c) Using signal-level specification in CWL, generated pattern definitions from it, and automatically generated scenarios by transaction-level language (C language).

By comparing (a) with (b), the effect of separation of description in terms of making test patterns can be obtained. We also could obtain the effect of using transaction-level by comparing (b) with (c).

(2) The effect of generating verification suites from a single description of the specification (Specification-based verification): We evaluate verification suites that are generated by our algorithms in terms of quality and man-hours required to generate them. We have verified the target module as follows:
   (a) We describe the signal-level specification in CWL (This description is the same one as (1) )
   (b) We generate test pattern definition
   (c) We generate simulation checker
   (d) Since we are still constructing a coverage tool, we estimated the effectiveness of our method on this point by calculating the percentages of bugs that it is capable of detecting.

The target module is a controller for data transfer (about 8k gate size). The module firstly fetches transfer commands from memory in response to a request signal. The module then transfer data on the basis of the some factors shown in Table 1.

If we verify this module entirely by simulation, the numbers of required combinations (more than 100k) of test patterns are huge. When time is limited, all patterns can’t be tried. The key to verification is thus to simulate as many patterns as possible in a short time.

<table>
<thead>
<tr>
<th>Parameters of the transfer</th>
<th>Size: byte/word/long word</th>
<th>Endian: little/big</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address size: 3 byte/4 byte</td>
<td>Mode: single/repeat (from 1 to 2^16)</td>
<td></td>
</tr>
<tr>
<td>Next transfer: same address/+1/-1, etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Factors of the transfer

4.2 Experimental result

Experimental results are as follows:

(1) Effectiveness of Level-specific verification in the making of test patterns are given in Table 2.
   • By separating the signal-level from transaction-level, effectiveness was raised by 1.8 times. If we neglect the initial 2 days for the CWL preparation, effectiveness was raised by a factor of 3.6.
   • By using transaction-level language, the effectiveness is raised by more than 2000 times neglecting the 5 days for preparation of the CWL and transaction-level specification.

We expect good results for CWL and transaction-level language working together, including higher levels of efficiency for transaction-level verification.

(2) Effectiveness of Specification-based verification are as follows:
   (a) It took 2 days to describe the CWL description. This result means that it only required 2 days to prepare the all verification suites: the pattern definition, checker, and the coverage analyzer. We expect that it take 2-3 days with a verification-specific language to prepare each verification item. The number of man-hours is expected to be reduced to 1/3.
   (b) As was shown in Table 2, our algorithm generates test patterns more efficiently than conventional way.
   (c) Table 3 shows that by using our algorithm for generating simulation checkers, the effective time required for validation per test can be reduced to 1/50.
   (d) The effectiveness of coverage-analyzer is shown in Table 4. In this experiment, the design included nine bugs. Verification analyzer can detect seven of the nine. Two bugs still remained. The reason is that the designer misunderstood the specification and introduced similar errors in both the design
and CWL descriptions. Verification suites were then generated incorrectly. This is a general problem in verification independent from the language used. We avoid them by requiring that the design and verification suites be made by different persons. The results showed the patterns and checkers and coverage-analyzer produced by CWL have sufficient capabilities.

Table 2: Effectiveness in the pattern-making

<table>
<thead>
<tr>
<th></th>
<th>Conventional</th>
<th>With CWL</th>
<th>With CWL and C</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Patterns</td>
<td>250</td>
<td>88</td>
<td>80000</td>
</tr>
<tr>
<td>No. of Days</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal-level Days</td>
<td>20</td>
<td>2 (1)*</td>
<td>2 (1)*</td>
</tr>
<tr>
<td>Transaction-level</td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>No. of Patterns per a day (normarized)</td>
<td>12.5</td>
<td>22 (1,8)</td>
<td>16000 (1280)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.5</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1)</td>
<td>(3.6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2133)</td>
</tr>
</tbody>
</table>

(*1): We regard them as a preparation for making test

Table 3: Effectiveness in the validation of simulation result

<table>
<thead>
<tr>
<th></th>
<th>Conventional</th>
<th>With CWL, with CWL and C</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of patterns</td>
<td>50</td>
<td>6000</td>
</tr>
<tr>
<td>Time taken for validation</td>
<td>3 minutes</td>
<td>60 minutes</td>
</tr>
<tr>
<td>Relative effectiveness</td>
<td>1</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 4: Estimation of effectiveness with our coverage-analyzer

<table>
<thead>
<tr>
<th></th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detectable</td>
<td>One of the bugs was only identifiable by the conventional method after 640 days of verification.</td>
</tr>
<tr>
<td>Non-detectable</td>
<td>Designer misunderstood the specification and made design and CWL that were incorrect in the same way.</td>
</tr>
</tbody>
</table>

5 Summary

Verification is one of the major time-consuming components in SoC design. We have described a way of constructing a verification environment at the signal-level and the transaction-level in a short time by using the interface specification description language CWL. We obtained the following conclusions.

1) By using the transaction-level language with CWL, we were able to quickly generate huge amounts of test pattern. We believe that this approach will lead to greater efficiency in transaction-level verification.

2) The verification suites at the signal-level (pattern definition, simulation checker and coverage analyzer) can be obtained in a short time, no longer than the time taken in pattern-generation alone with the conventional method.

5.1 Future work

In this paper, we have generated patterns, etc. in HDL description because HDL has the advantage of being independent of the design environment. We will try generating other formats such as transaction-level language or verification-specific language as a step towards better verification environments.

We want to increase reusability in verification environments. If a language is applicable in multiple areas (pattern generation, checking, coverage analysis), the environment is simplified. On the other hand, various functions might be mixed together in single bodies of descriptive code, and this might make reuse difficult.

6 Acknowledgments

The authors would like to thank Yuji Ogata, Takahiro Matsumoto, and Masashi Suenaga of Hitachi for making experimental data available, Tsuneo Nakata, Hiroaki Iwashita of Fujitsu Laboratories, and Satoshi Kowatari of Fujitsu for useful discussions.

7 References