Assertion Based Verification in TLM

AmirAli Ghofrani, Fatemeh Javaheri, Zainalabedin Navabi
University of Tehran
{ghofrani, n-javaheri, navabi}@cad.ut.ac.ir

Abstract

Design verification is an important matter in digital design process. There are many different ways to verify designs. One of these methods is based on Assertions. In this method, the properties of the design are defined in terms of assertions which are fired in case of any contradiction to those properties. On the other hand, the emergence of higher abstraction levels to describe the digital circuits necessitates new and novel verification methods for the new abstraction levels.

In this paper, we propose a method of assertion based verification for Transaction Level Modeling (TLM) by completing SystemC assertions and manipulating the TLM1.0 library. This general purpose set of assertions can be used to efficiently debug and verify circuits described in TLM.

1. Introduction

In recent years, designers faced many difficulties in describing their large and complex designs. That was partly due to the ever-growing complexity of digital circuits and lack of good verification techniques. This increasing complexity led the designers to heighten the abstraction level from the gate level to register transfer level (RTL) and finally resulted in the emergence of TLM as a dominant methodology to implement large digital systems. Figure 1 shows hardware design evolution [6].

The acceptance of this new methodology was due to the fact that by using it, the designer was no more concerned about the details of the design. Furthermore, the designer could have an overview of the complete system, since most modules were described at a high level of abstraction. Therefore, the designer could easily decide on matters like module partitioning or test-bench development in a more systematic way. Moreover, Due to its higher simulation and prototyping speed, which is caused by heightening the level of abstraction, TLM has a better performance compared to RTL methods. OSCI [1] and OCP/IP [2] are two organizations working with design and CAD tool developers and develop transaction level models and libraries.

In TLM, the intra-module communications are separated from their interior computation. The communication is modeled by some high level channels. The goal is to keep the designer from the cumbersome details of communication and provide him or her with easy to use means of transferring data between modules.

New abstraction levels may require new description languages to implement the design, like the advent and extension of VHDL to describe RTL implementations. In today's digital designs, the IEEE standard SystemC language and its TLM (1 and 2) derivatives are the most commonly used design languages for electronic system level (ESL) designs. Unlike VHDL, SystemC and TLM are really not defined as stand-alone design languages and are basically a set of libraries in C++. As a result the compiling and debugging environments are the same as those used for C++ language.

In spite of the fact that the same environments are used, there is a substantial difference between the problems which are faced in C++ programming as compared to those in designing a digital circuit in TLM. This obliges a special meaning of debugging in digital circuit design using SystemC.

In the absence of proper debugging tools, which would help designers cope with common but hard to detect design and programming mistakes, the necessity of special verification methods for TLM becomes apparent. However, unlike RTL that has standard methods of verification, TLM verification is neither well developed nor well defined. Consequently,
Introducing methods to improve design verification and code debugging will enable designers to take a better advantage of the high simulation speed in TLM design methodology.

In this paper, a good solution to code debugging and TLM design verification is introduced based on assertions. Two different kinds of assertions are provided for this purpose. The first kind which is RTL SystemC OVL assertions are the SystemC OVL-like assertions that are modified to be used in the computational part of TLM modules which is described by SystemC language. It can also be utilized for the verification of mixed SystemC RTL-TLM designs. The second kind of assertions is provided by manipulating the TLM core libraries and could be used to verify the intra-module communications.

The rest of this paper is organized as follows: Section 2 shows the related works done on the TLM verification subject. Sections 3 and 4 describe the two components of the presented work which are SystemC RTL and TLM assertions. Section 5 shows the usage of proposed assertions in a real case and next comes the conclusion.

2. Related Works

Transaction Level Modeling using SystemC has been introduced by EDA industry to implement system-on-chip designs. Verification of these systems is crucial. Therefore, some methods have been proposed to verify TLM designs. A methodology proposed in [3] is based on Aspect-Oriented-Programming to record transaction without modifying the existing TLM. In addition, each transaction is mapped into Boolean signals. In this method, System Verilog Assertions have been used to verify SystemC TLM designs.

In [4], to describe the functionality of TLM, an abstract model called Message Sequence Model (MSM) has been introduced. In addition, based on given properties from Linear Time Logics (LTL) some formal specifications have been created. It can be verified using symbolic simulation whether or not a property holds for the MSM.

The method proposed in [5] separates communication and computation parts. A testbench exists which can be used both for TLM and RTL by defining unified interfaces. RTL and TLM designs are tested using this testbench and the results are compared to verify the functionality of a system. Tests have been done in cycle accurate level.

However, none of these methods were easy to use and easy to understand, and they have a limited domain of utilization. This urged us to think of some methods that can be applied to a wider range of applications and can be used even by novice programmers.

3. SystemC Assertions

Due to the higher level of abstraction, describing TLM modules and their communication need a more abstract language. For this reason, SystemC language was used. SystemC is an extension of C++ and is capable of simulating concurrency that is the most crucial requirement of hardware implementation.

Despite the theoretical differences between RTL design methodology and TLM, designing the computational parts of TLM modules using SystemC is not that much different from their relative RTL SystemC implementations. On account of this fact, same assertions could be used to verify the design. At the RT Level, Open Verification Library (OVL) has been used to define proper assertions to verify RTL HDL codes. However, these assertions are not modified to be used in the SystemC language.

In the first component, OVL assertions are ported to the SystemC language so that they can be used to verify the SystemC definition of RTL designs. In addition to the usage of these assertions to verify the computational parts of TLM modules, these assertions are suitable for the cases in which a TLM module is substituted by a SystemC RTL module, or perhaps for SystemC codes developed as parts of a design to interact with the other design components in TLM or SystemC. Figure 2 shows an instance of such mixed design and the usage of these assertions. In cases of mixed TLM and RTL SystemC designs, such SystemC RTL assertions, could verify both the functionality and the interfaces of such SystemC RTL code.

Figure 2. RTL and TLM mixed designs

All assertions implemented by OVL for Verilog-95, are also implemented in this part. Coverage calculations are also included in this implementation.
As an example, the SystemC implementation of "assert_decrement" is shown in Figure 3.

```c
#include "assert.h"

template <int severity_level=0, int width=1,
          int val=1, int coverage_level = 0,
          class Msg = MsgViolate>
class assert_decrement :
    public sc_assertion<coverage_level> {
public:
    sc_in<long> test_exp;
    #ifdef OVL_COVER_ON
    sc_lv<width> prev_test_exp;
    #endif
    void do_assert(void){
    long last_test_exp = 0;
    sc_logic rst_n1, rst_n2;
    long tmexp, temp_lstestexp, temp_expr1 = 0;
    while(1){
        wait(clk.posedge_event());
        tmexp = test_exp;
        temp_lstestexp = last_test_exp;
        if (ASSERT_RESET_SIGNAL.read()){
            #ifdef OVL_COVER_ON
                if (coverage_level != OVL_COVER_NONE) {
                    if (OVL_COVER_BASIC_ON) {
                        if (test_exp.read() != prev_test_exp) {
                            coverMessage("covered");
                            prev_test_exp = test_exp.read();
                        }
                    }
                }
            rst_n1 = ASSERT_RESET_SIGNAL;
            rst_n2 = rst_n1;
            if (rst_n1 && rst_n2 &&
                (last_test_exp != test_exp.read())){
                temp_expr1 = temp_lstestexp - tmexp;
                if (temp_expr1 != val) {
                    printMsg(Msg::message());
                    sc_stop();
                }
            } else {
                rst_n1 = false;
                rst_n2 = false;
                last_test_exp = test_exp;
            }
        } else {
            rst_n1 = false;
            rst_n2 = false;
            last_test_exp = test_exp;
        }
    }
    SC_CTOR(assert_decrement){
    SC_THREAD(do_assert);
    sensitive_pos(clk);
    }
};
```

Figure 3. SystemC implementation of assert_decrement

"Assert_decrement" ensures that the value of an expression, test_exp, changes only by the specified decrement value, val. This assertion is implemented as a thread. In that thread, there is an infinite loop which is evaluated at the positive edges of the clk. The #ifdef primitive inside the while loop implements the coverage calculation. The reset signal is registered by means of rst n1 and rst n2 since the assertion is only active when there is no reset signal issued in the current or the previous clock cycle. If the assertion is active, temp_lstestexp and tmexp which contain the value of test_exp in two consecutive clock cycles will be subtracted to be checked against the margin.

4. TLM Assertions

So far, verification of systems modeled in TLM has been only possible by RTL HDL test benches or creating abstract models. Also, designers are facing many problems debugging TLM. For many of them debugging equals to lots of changes to the original code. In our proposed method, the system is verified directly in TLM making the debugging process easy. To do so, some assertions are defined and added into TLM1.0 core library. These assertions make the TLM channels self-verifying which can be verified easily by defining the properties of that desired channel and then specifying the appropriate assertion for that property. These assertions verify the functionality of TLM1.0 channels and communication between processing elements. These assertions monitor channels’ events and check for any violation of properties specified using assertions. More than 40 assertions were defined which can cover almost all the properties that a channel can have. Some of the implemented TLM channels assertions and their usage are listed in Table 1.

The most important advantage of these TLM assertions is the ease of use and the separation of verification and design codes. Each assertion has been implemented as a SystemC module containing a thread named "assert_". These threads have been implemented in TLM1.0. The use of these assertions is very similar to RTL OVL assertions. All one needs is to select the appropriate assertion, specify the arguments, and call the assertion for the desired channel in the TLM code, wherever the channel is accessible.

<table>
<thead>
<tr>
<th>TLM assertions and their usage</th>
<th>Fire if</th>
<th>Fire if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assert_full()</td>
<td>fifo channel is full</td>
<td>fifo channel is empty</td>
</tr>
<tr>
<td>Assert_empty()</td>
<td>fifo channel is empty</td>
<td>fifo channel is full</td>
</tr>
<tr>
<td>Assert_overflow(int num)</td>
<td>number of data in fifo channel becomes greater than &quot;num&quot;</td>
<td></td>
</tr>
<tr>
<td>Assert_underflow(int num)</td>
<td>number of data in fifo channel becomes less than &quot;num&quot;</td>
<td></td>
</tr>
<tr>
<td>Assert_get()</td>
<td>&quot;get&quot; interface calls (the same assertion is available for other interfaces)</td>
<td></td>
</tr>
<tr>
<td>Assert_blocking()</td>
<td>transaction is blocking (the same assertion is available for nonblocking)</td>
<td></td>
</tr>
<tr>
<td>Assert_num_get_time(sTime, eTime, limit)</td>
<td>number of get interface calls becomes greater than &quot;limit&quot; in the period between &quot;sTime&quot; and &quot;eTime&quot;</td>
<td></td>
</tr>
<tr>
<td>Assert_no_same_data_put(0)</td>
<td>data put into the channel differs from the desired data</td>
<td></td>
</tr>
</tbody>
</table>
TLM assertions can be divided to three groups according to the input argument:

- Some of the assertions have no arguments and check the properties of a channel such as "assert_blocking".
- Some of them have arguments that specify a limiting factor. The assertion fires if the desired property of the channel violates that limit. "assert_num_get_time" is a prominent example of this kind.
- The last and more complicated assertions are those firing according to the values of a signal during the program execution. To implement these assertions, an interface has been placed between the design and TLM library. Users should define the input argument of the assertion and bind the defined port to the appropriate port of the interface before calling the assertion. "assert_same_data_put" is one such assertion.

Figure 4 shows an example of using an assertion to verify the correctness of the data put into a channel.

```cpp
// bind input data to its correspondent port
tt_fifo.instance1.wanted_di_put0(data);
// fires if the data put into fifo is not equal to "data"
// fires if the fifo becomes full
// fires if fifo becomes empty
// fires if put interface is called
// fires if more than 4 nbget calls issue in 10-40 ns period
// fires if fifo contains 5+ data
// some asserts have ports to be compared with user data
// ports should be specified according to the readme file
// code segment to verify the channel
// end of verifying code segment
```

Figure 4. Using an assertion in a TLM channel

5. Case Study

In this part a simple example shows the usage of SystemC and TLM assertions in verifying a simple TLM design. Consider a system with a reader and writer connecting by a TLM FIFO channel. The writer is a serial to parallel converter and the reader is a parallel to serial converter. The processing elements are verified using SystemC assertions mentioned in Section 3 of this paper.

Verifying of the computation parts has been done by SystemC assertions discussed in section 3.

Figure 5 shows the testbench header file of a serial to parallel convertor containing some additional code for verifying properties of this unit. "assert_even_parity" has been used to verify the correctness of transmitted data. Sometimes, verifying the designs using SystemC assertions requires some auxiliary logics. As an instance, to verify a gray code counter using "assert_onehot", two consequent data have to be XORed. Or in some other cases, like serial to parallel convertor, some type changing are needed which are done by the auxiliary logic implanted in the cpp file of the design.

```cpp
int myChannel(){
    sc_signal<bool> test;
    st_signal<int> data;
    write_fifo1 wr1("WR1"), wr2("WR2");
    read_fifo1 rs1("RD1";
    initialize init("INIT");
    tim_fifo<int> tt_fifo(10);
    wr1.port_out(tt_fifo); wr2.port_out(tt_fifo);
    rd1.port_in(tt_fifo); rd1.test(test);
    tt_fifo.assert_put(); //fires if put interface is called
    tt_fifo.assert_nblocking();//fires if nb interface is called
    // fires if more than 4 nbget calls issue in 10-40 ns period
    tt_fifo.assert_num_nbget_time(10,40,4);
    tt_fifo.assert_overflow(5);//fires if fifo contains 5+ data
    //some asserts have ports to be compared with user data
    // ports should be specified according to the readme file
    portArray = (1,3);
    tt_fifo.set_interface_port(portArray);
    // end of verifying code segment
    return 0;
}
```

Figure 6. A sample code of verifying a FIFO channel using TLM assertions

The communication part of the design has been verified using TLM assertions. Figure 6 shows the code used to verify the channel.
As illustrated in Figure 6, to verify the channel, only a segment of code has been added to SystemC TLM code and the main design has remained unchanged. The easiness of this method is clearly observable.

6. Conclusion and Future Works

In this paper, a novel method of verifying both TLM computational and communicational parts was introduced. Verification of computational parts was based on RTL OVL assertions and those assertions were transformed to SystemC assertions. On the other hand, for the verification of TLM channels which are the means of communication in TLM, TLM library core was revised and many assertions were defined inside the library. This way, assertions could be used for every instanced channel.

However, ESL industry keeps moving forward, thus TLM 2.0 was introduced with the new concepts of sockets and payloads. The next step is to work on TLM 2.0 and make verifiable sockets.

Acknowledgement

The authors would like to thank the former member of the CAD group, Anahita Naghilou who provided the original SystemC codes of the assertions. Her work was presented at the University Booth at DATE 2007 in Nice.

7. References


