

Troubleshooting a transaction-level model

SEEMINGLY MINOR VIOLATIONS OF THE TLM 2.0 STANDARD CAN TURN A SYSTEM-LEVEL MODEL INTO AN AGENT OF EVIL IN YOUR DESIGN FLOW.

The goal of OSCI (Open SystemC Initiative) TLM (transaction-level modeling) 2.0 is to enable high-level component models to simply plug into and play with each other in a system model. The standard identifies modeling styles, several interfaces, a generic payload, and more than 150 rules to define the expected behavior in the simulation. For the system model to work correctly, designers must comply with the standard. Finding errors in compliance can be a major problem, however. Nevertheless, approaches exist for troubleshooting system models to find noncompliant areas.

TLM 2.0's 150 rules specify the expected behavior of a transaction and restrictions to avoid system malfunctions. The OSCI TLM working group defined TLM 2.0 specifications with respect to initiator and target pairs. A typical design has several initiators and targets, which connect through interconnects (Figure 1). The interconnect serves as both a target and an initiator.

The communication path is the path between a communication's starting point and its ending point in a system. A communication path typically contains more than one TLM 2.0 initiator/target pair. In a simple example, there are always two initiator/target pairs in the full communication path. Real systems contain interconnect hierarchies; multiple intercon-

nects that connect bridges; and multiple subsystems, multiple cache hierarchies, or both.

Most of the TLM 2.0 rules apply to each initiator/target pair, not the full communication path. This requirement ensures interoperability between any two components in a system. It is easy to validate that models obey these rules at the starting point of a communication. However, it is more difficult to validate that the models obey the rules in the system because this type of validation forces designers to observe multiple communication paths in parallel. If the models obey all the rules, TLM 2.0 guarantees the interoperability and, thus, the communication between components. The following example illustrates what happens when a model violates a rule.

NONCOMPLIANT COMPONENTS

Violations of rules can create significant interoperability problems in a system. Interoperability issues are often not immediately visible. They might result in system-behavior malfunctions later in a simulation. The problems often occur in a different location in the system or trigger an unexpected corner-case situation. So interoperability issues require constant monitoring of the communication path between the initiator/target pairs and a comparison of the actual with the expected behavior.

One such violation is a generic-payload-access violation.

Several rules limit access to the generic payload to specific components in a model. For efficiency reasons, the generic-payload objects do not go through the high-level system model. Rather, the model uses references from each component in the communication path to one copy of the generic payload to mimic the transmission of data through the system. This technique also keeps the communication between components flexible.

With this reference mechanism, the designer can explore multiple sequential and parallel transmission schemes without changing the model. The disadvantage is that all components in a communication path have access to all the information in the generic payload. Implementing a mechanism in the generic payload that identifies which component accesses which data field is too

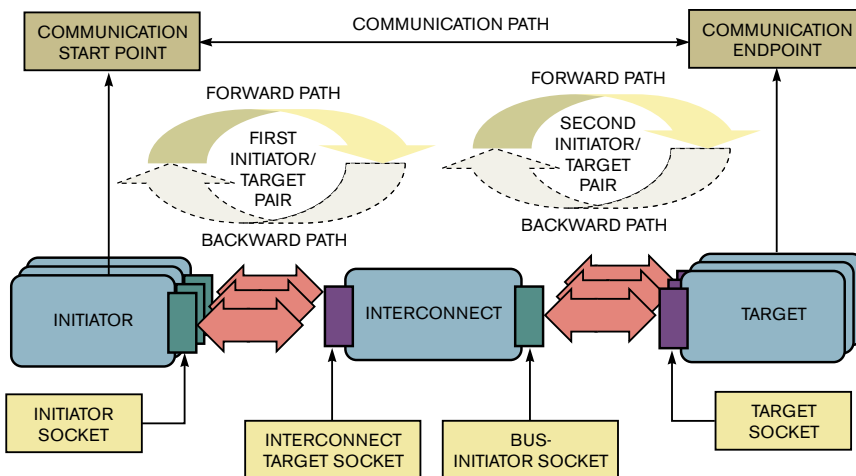


Figure 1 TLM 2.0 provides a strict framework to model the interactions between blocks in a high-level system model.

expensive in simulation performance. Still, some access rules are necessary to guarantee interoperability and avoid a malfunctioning system.

For example, one rule specifies that only the interconnect models can modify address values. Doing so easily mimics virtual-to-real-address translation in a system. In many cases, the interconnect uses address fields in an address-translation table. For that reason, another rule prohibits targets from modifying the address attribute of the generic payload. A target that modifies the address field causes the interconnect's address-translation table to fail. The model either cannot complete the response or sends the response to the wrong initiator.

A second example is that only the targets can enable the attribute DMI (direct-memory interface). So another rule states that the initiator must initialize the DMI-allowed attribute with the value "false." Only targets can change this attribute. Only the target knows whether it can support the DMI. An initiator setting "DMI enable" to "true" with a target that does not support DMI creates side effects in system behavior that are difficult to observe and to debug.

PROTOCOL-STATUS VIOLATIONS

OSCI TLM 2.0 provides a set of rules specifying the usage and the transition of protocol-status information. These rules capture the expected behavior of the protocol. In this way, every user has the same view of the states and state transitions the model allows. TLM 2.0 also restricts the number of allowed "in-flight" transactions between an initiator/target pair before the initiator can issue the next transaction. This restriction protects the targets from memory or register overwrites. The rules do not apply to the full communication path; they apply only to each initiator/target pair. In this way, designers can easily model features such as out-of-order execution and parallel-thread processing without making assumptions about the communication partner.

TLM 2.0 specifies two-phase communication for LT (loosely timed) and four-phase communication for AT (approximately timed) models. It has some restricted ability to pre-empt a communication early by sending a TLM-complete return value. TLM 2.0 predefines the order of the phases and explicitly specifies the allowed transitions between phases. In this way, every component has the same view of all possible states and need not be able to handle any other state transition. This requirement allows the user to implement efficiently tuned models for high-speed simulation. Unfortunately, it also means that malfunctions in models can go undetected.

OSCI TLM 2.0 specifies that you can issue only one transaction with a begin-request command to one target. The target must first respond with an end-request command before the next begin-request command can occur. This rule protects the target from information overwrite by ensuring that only one transaction can be in flight. The target must actively issue an end request before the initiator can start the next transaction.

FOR A USER OUTSIDE THE MODEL-DEVELOPMENT GROUP, THE IMPACT OF INTEROPERABILITY ISSUES INCREASES TO THREE TO FIVE DAYS OR EVEN TO WEEKS.

Ensuring these interoperability restrictions from the starting point of a communication seems trivial, but, because TLM 2.0 models share interconnect systems, multiple transactions can be in flight in the interconnect model. The interconnect model must now ensure that all of its connections of initiator sockets to targets obey the rule. For some systems, this requirement means that 10 to 100 communication paths work in parallel, so 10 to 100 transactions can be in flight at once.

If an initiator accidentally issues more than one begin-request command to one target, depending on the implementation of the target, the second begin-request command will overwrite information from the first transaction before it can process the command. This scenario breaks the communication of the first transaction. The target would respond with only one end-request command, instead of the required two. The initiator of the first transaction waits for the end-request command before it can send the next begin-request command, so a deadlock arises.

When an initiator issues a phase multiple times, so that it issues an additional transport call, the sequence can push the traffic generator out of synchronization. Especially for protocol-status violations, the problem is typically not visible immediately in the simulation but later causes malfunctions in the system in the simulation. Tracing the malfunction back to the original protocol violation is time-consuming.

In general, all of these examples show that rule violations cause interoperability problems. These problems, in turn, create major system-behavior failures. Those system failures may be immediately visible in the form of system crashes, obvious wrong behavior, or deadlock situations. Some others might be visible only to system experts who could spot inconsistencies with the expected behavior or performance. Those interoperability issues are so difficult to find, but designers can use debugging and troubleshooting approaches if they cannot automatically check a model for TLM 2.0 compliance.

FINDING COMPLIANCE VIOLATIONS

All major electronic-system houses and semiconductor companies that are implementing ESL (electronic-system-level) models have an average project delay of three to five hours per issue due to interoperability problems in their models. They confirm that this number is conservative and applies to a highly skilled model developer. The assumption is that the developer can trace the issue to a component in a system or has found it during unit testing.

For a user outside the model-development group, the impact of interoperability issues increases to three to five days or even to weeks. The further the user is from the model developer, the larger the time increase is. For a typical project, these delays lead to 40 to 60% of project time for debugging. To understand why it takes so long to identify interoperability problems, you must understand debugging and identify the steps

necessary for troubleshooting interoperability problems.

Debugging ESL models employs a multi-layer approach involving the application-, software-, component-, and interface-debugging layers. This approach is one way that virtual-platform developers cope with system complexity. Typical virtual platforms contain at least 20 to 40 hardware- and software-component models, all with their own debugging infrastructure. To be able to quickly identify problems, a user traces key parameters in the system. If those key pieces of information are inconsistent with expectations, something is wrong, and debugging starts.

At the application-debugging level, the user looks at application-specific data. This data might be a sequence of video frames, an audio stream, or a wireless trace. This approach allows narrowing down the area in which a problem occurs.

For example, in debugging a video frame, you may look at several video frames. You might identify that some frames are intact, but several are broken. Some lines in a frame are still intact, but the picture fragments shift randomly throughout the frame. This problem is a synchronization issue with the display of the frame. The problem can occur within one frame or between frames. Lines of a picture are recognizable, and some picture segments are intact. This pattern means that the problem probably occurs after decoding and before displaying the frame.

After identifying a frame-synchronization issue, the next step is to look at the software. One of the processors is responsible for synchronizing frame processing. The next step is to identify the piece of software code that is responsible for triggering frame processing. At the software-debugging level, the designer might use software debuggers such as GDB (Gnu debugger). The user sets breakpoints in the source code for frame processing and observes values at register or memory locations. In most debuggers, the user can set “watch points” to specific values. A watch point on the register value triggers the next frame’s processing and stops the simulation at the frame-synchronization point.

These advanced tools help to filter out other behavior and focus on frame synchronization. It still takes time to identify the source code and set the breakpoints and watch points in the correct location and then identify the register values responsible for triggering the frame processing in the hardware accelerators.

In this example, frame synchronization seems to work. The user has set the register values, and frame processing in the hardware component starts. You have thus determined that the software application did not cause the problem and can move to the next debugging step by looking into the hardware models.

Multiple hardware components, including those for encoding, decoding, and frame-buffer processing, are responsible for video processing. Most of the components move pixel information into and

IDENTIFYING AN INTERFACE-SYNCHRONIZATION PROBLEM REQUIRES DETAILED INTERACTIVE SOURCE-CODE DEBUGGING.

out of various local and global memories in parallel. Multiple components share some memories.

To identify the location of problems, developers use monitor components to observe data arrays in memory. In addition, they trace read and write values from processing elements to memory and vice versa. Most ESL-debugging tools are SystemC- and TLM-2.0-aware. This feature allows the tools to automatically and conveniently display

transactions. The ability to set breakpoints and watch points on TLM 2.0 transactions significantly eases debugging. The user still must know the expected values of the transactions throughout the platform, however, to be able to pinpoint issues in data communication.

The user traces transactions from the point at which the data stream is in synchronization with the location in the system where it gets out of synchronization. This example identifies that the transactions are correct until they reach the frame buffer.

The example identifies that the start addresses of the frames are randomly distributed when the frame processing starts. You can expect multiple start addresses because multiple frames are in the buffer. After consulting the documentation, you can identify the expected values for the start addresses of frames, but they won’t match what you saw during debugging.

Now, trace back to where the address is written. The generated addresses are correct. Thus, you can pinpoint the frame buffer as the location of the problem. More accurately, the communication between the processor and the frame buffer is causing the problem. You thus should more closely examine the TLM 2.0 socket interface of those components.

The interface-debugging step is the most detailed one. It requires detailed knowledge about both the synchronization mechanism in the system and its implementation in the model. At the transaction level, multiple ways exist to mimic the behavior of one interface protocol. TLM 2.0 significantly reduces the number of ways of implementing an interface protocol, however. Nevertheless, identifying an interface-synchronization problem requires detailed interactive source-code debugging. This task is especially cumbersome if you have incorrectly implemented the TLM 2.0 interface.

In the case of video, the memory is a shared resource that connects through an interconnect to the processor. Multiple applications can concurrently transmit traffic to this interconnect, meaning, in some cases, hundreds of transactions from various applications are in flight at the same time.

You must filter out only those transactions that are relevant for your video application. You also must predict and confirm the expected values as well as the expected time each value should occur. For TLM 2.0, this requirement means that, for each transmitted value, you must trace two or four transaction calls for LT and AT, respectively, to ensure that they transmit the correct value

➤ Go to www.edn.com/ms4321 and click on Feedback Loop to post a comment on this article.

➤ For more technical articles, go to www.edn.com/features.

at the correct time. It may take hours of staring at transaction traces and comparing expected traces with the simulated traces.

INTEROPERABILITY ISSUES

Eventually, an interoperability problem arises. For example, the processor did not respect the TLM 2.0 rule in some corner-case situations. It updated the register, which triggered the hardware accelerator to start processing the next frame for display, without waiting for the end-request command for the previous frame. The assumption in the system is that data processing in hardware is faster than in software. The model does not implement detailed interrupt processing, which is also not relevant for the initial analysis. So, instead of modeling the full interrupt mechanism, the model uses an end-request command to mimic the interrupt to indicate the end of processing in the display accelerator. This action notifies the processor that it can send the next transaction to trigger processing for the next frame.

The process to wait for the end-request command and send the next transaction is broken for some corner-case situations. The frame view shows that multiple frames had started, but, because the processor triggered another frame too early, it finished transmitting only parts of each frame before the next frame started.

In a project that is supposed to quickly create some initial analysis results, debugging can take several days, so the results may not be available in time. Automatic compliance check-

ing would have instantly detected this problem and made the results available in a reasonable amount of time.

Although the effect of TLM 2.0 protocol violations in high-level models may seem minimal in passing, failing to catch such a problem has deeper implications. Erroneous models with TLM 2.0 violations alter the behavior of the system and cause system malfunctions or biased performance results. Undetected errors can propagate into the RTL (register-transfer level). Especially in the case of biased performance results and corner-case user scenarios, it is nearly impossible to detect these problems at RTL because it is too detailed and simulations are too slow to run the system-level tests. You might be able to identify the problems during FPGA testing, but it is too late in the development cycle and causes project delays. In some cases, you might not detect the problem at all, and that problem would result in misbehaving silicon.[EDN](#)

AUTHOR'S BIOGRAPHY

Andrea Kroll is a vice president of marketing and business development at Jeda Technologies, where she has worked since early 2008. Previously, Kroll worked at Synopsys as an application specialist for CoCentric System Studio and CoWare Processor Designer. In 2005, she took over product marketing for CoWare Processor Designer. By feeding back technical requirements from customers to engineering, she helped double the company's processor-designer business year on year for three years. Kroll has a doctorate in electrical engineering with a concentration in high-level-model validation from the University of Technology (Aachen, Germany).
