HIGH LEVEL HARDWARE VALIDATION USING HIERARCHICAL MESSAGE SEQUENCE CHARTS

Praveen K. Murthy, Fujitsu Labs of America (FLA), Sunnyvale CA 94085
Sreeranga P. Rajan, FLA, Sunnyvale CA, 94085
Koichiro Takayama, Fujitsu Labs Ltd., Kawasaki, Japan

Abstract

We describe a methodology for designing, testing, and verifying hardware designs at a high level of abstraction, using a visual formalism based on hierarchical message sequence charts. We develop a method for generating behaviors and monitors automatically from this high level description, and using it to validate actual hardware implementations developed by design teams. We apply our methodology to the design of a PCI-Express switch, and show that the methodology is useful in finding many design errors. We develop an enhanced hMSC language that can be much better suited for describing complex standards and protocols like the PCI-Express.

Introduction and motivation

It is being universally recognized that the level of abstraction at which designs are initiated needs to be raised for a variety of reasons, including the serious need to handle the validation bottleneck that increasingly consumes a greater and greater percentage of the design cycle. One of the many advantages of raising the level of abstraction is that we can develop better and more systematic testing and verification methods since the functional aspects of the design are captured in a high-level “golden” model. To that extent, we have been investigating testing and verification methods using hierarchical message sequence charts (hMSCs) [1] for certain protocol and control dominated applications. Not only is the hMSC a useful model of computation for this application domain, but it is also closely related to UML sequence diagrams [7]. Since UML is gaining in popularity, the use of UML-related models in the design and validation cycle has practical benefits in adoption of ideas and methodologies by design groups.

Our investigation has been motivated by a particular type of design process that is quite common in industrial designs of network switches and other such custom hardware. Typically, this design process starts by acquiring and understanding some standard such as the IEEE 802.1 Gigabit Ethernet standard or the PCI-Express standard that we use as the application driver in our study. These standards are English documents that are hundreds of pages long and are described using an ad-hoc combination of textual description, simple state machines that do not have well-defined or agreed-upon semantics, timing charts, flow diagrams, again with “intuitive”, or “back-of-the-envelope” type of semantics that are not precisely defined but understood only in an approximate, intuitive sense. The design team then proceeds to form some sort of architectural map of the system they want to implement, and different designers than start writing different modules in Verilog or VHDL directly. These modules are supposed to interpret and implement the standard correctly. After everything has been written, the design enters the testing and verification phase, where test engineers develop test benches in order to ensure that the design is “correct”. There is no well-defined, universally accepted formal methodology in place currently for performing verification and test. The use of formal techniques such as model checking is sparse and ad-hoc. Typically, the methodology used is to develop the test suites by using the guidelines and checklists provided by the standard itself. For example, the IEEE 802.1 standard for gigabit ethernet has a large appendix called the “PICS” (protocol implementation conformance statement) that specifies a number of questions about the implementation that should be asked and verified. The test bench might be developed to test these set of questions and verify correct behavior. However, there is no guarantee that this set of questions completely “covers” the standard and specification, and one is merely hopeful that the standards committee has developed this set of questions with some such confidence.

Our strategy is to raise the level of abstraction at which the design specification is expressed and produced. This has the advantage of not having all of the irrelevant details, and instead allows the design to contain only the functional aspects of the specification. It also has the advantage of re-targetability: since the design exists at a level above implementation and platform-specific details, the testing strategy does not have to be modified if those details change. A third advantage is that by grounding the design language in an abstract model of computation with well defined semantics, formulation of strategies where properties can be extracted automatically and tested in a systematic manner is feasible. Finally, problems of complexity are finessed by having smaller design inputs because the design input does not have unnecessary details. As the design is refined, equivalence checkers can be used to ensure that each refinement includes the behaviors in the original; this type of check is easier to perform in many cases.

We have focused on hierarchical message sequence charts (hMSC) [1] as an appropriate model of computation for our particular domain of application. These diagrams depict concurrent processes and the messages they exchange. The diagram only displays the partial ordering relationship between the various abstract messages exchanged, and is thus well-suited for specifying communication protocols. An additional advantage of hMSCs is that it is closely related to UML sequence diagrams, and thus our methodology can eventually use subsets of UML as a starting point. How-
ever, in this paper we will only describe the methodology using hMSCs.

**Hierarchical Message Sequence Charts**

Fig 1. A bMSC and a Hasse diagram showing the partial order between the events.

Fig 2. hMSC graph.

Figure 1 shows a basic message sequence chart (bMSC). As can be seen, the diagram has several components. The parallel lines represent processes, for instance, P, Q, R, S, and are parallel to each other to denote their concurrent behavior. Arrows between the processes represent messages exchanged. For each message m exchanged, we associate a pair of send/receive events m_S and m_R corresponding to the event where one process sends a message and the event where the other process receives the message. There is an abstract concept of time that is used for ordering these events. However, note that the visual order only applies to the events, and this is depicted by the Hasse diagram in figure 1. This diagram shows that, for instance, the event a_R takes place after the event a_S since c_R occurs after (visually) a_S on the process line P. Thus, even though it appears that visually b_S occurs after a_S, there is in fact no ordering relationship between them.

A hierarchical message sequence chart (hMSC) (Figure 2) consists of a directed graph where each node in the graph is either an hMSC, or a bMSC. Directed edges between the nodes represent the control flow for how messages exchange proceeds. Each branch represents a “scenario”, and in the simplest flavor of hMSCs, the branches are considered non-deterministic. In figure 2, a path from M1 to M3 can go through arbitrarily many executions of M2 since the choice to proceed to M3 or go back to M2 when in M2 is non-deterministic. Each bMSC node has the same set of processes. An execution of an hMSC is simply a path through the hMSC graph that results in the processes exchanging the messages described in each node along the path. Different paths in the hMSC graph represent different scenarios. For instance, for the path M1 → M4, process P sends message c after sending message a in the execution of M1, whereas along path M1 → M2, process Q sends message d after receiving message b in M1.

Events that occur in each bMSC along a path in the hMSC graph can be either concatenated synchronously, where all events in a bMSC have to complete before any events occur in a successor bMSC, or asynchronously, where each process continues independently, and can move to another bMSC even before other processes have finished their events [2]. This means that in the example of figure 2, when process P sends message c in the path M1 → M4, under synchronous concatenation, message b should have been sent and received before P sends message c, whereas under asynchronous concatenation, message b need not have been sent, as there is no ordering relationship between the sending of c with the reception of b. It has been shown in [2] that the asynchronous model is fundamentally more powerful than the synchronous model, and in fact, can encode Turing machines. Thus many decision problems become undecidable in the asynchronous model. Even though we do not use formal decision procedures in our methodology, we use the synchronous concatenation model largely because we have found that the application driver (described in a later section) can be modeled using the synchronous model. Applying our methodology using the asynchronous model is an interesting area for future work.

**Application Driver — PCI-Express**

We concentrate on a particular subsection of the PCI-Express specification (a next generation interconnect technology [5]) for this study. We apply our methodology to the physical layer section of the specification, in particular, the link training status state machine (LTSSM). This protocol specifies how a PCI-Express port communicates with another such port. The protocol specifies things like the manner in which link widths and data rates are negotiated, how a port detects the presence of another port, how a port goes into various power saving modes and comes out of it, how errors are handled and so on. The standards documents describe each of the states, and depict them hierarchically as many substates using a combination of text and state diagrams.

The basic abstraction used to model the LTSSM in hMSC is to model the LTSSM as consisting of two concurrent processes: a transmit module and a receive module. These modules respectively communicate with a receive and transmit module that represent another LTSSM device (figure 3). This level of abstraction allows the sending and reception of messages to happen concurrently, and does not fix any unspecified ordering in the behavioral description.

In order to model the LTSSM functional behavior accurately using hMSCs, we have to make certain obvious semantic and syntactic extensions to the hMSC language that has been described so far. These extensions are required to model simultaneous signals,
timeouts, and explicit ordering (synchronization) relationships between sends and receives, three things that arise frequently in the functional behavior. Some or all of these extensions have been proposed by others; for instance, in [4]. We describe these extensions here briefly.

Simultaneous events

Often times, we need to be able to specify that two or more messages are received or sent simultaneously, by the same process. Recall that for any given process, all of its events are totally ordered in the visual order they are depicted along the process line. However, what if we want to model the situation where some of these events occur concurrently, or simultaneously? We use boxes drawn on process lines to group messages to indicate that the corresponding events occur simultaneously, and do not have any ordering relationship between them. We attach a label “simul” to these boxes that group simultaneous message events together. “Simultaneous” in this context does not necessarily mean “at the same time”; all it means is that in the granularity that is used for determining the instances when the system reacts to events, these “simultaneous” events will be considered to all cause a reaction in the same instance. This instance could be a clock tick, for a tightly synchronized hardware description, or it could be some time period in which a real-time operating system schedules a process for execution before that process returns control. The syntax we use is similar to the co-region syntax used in live sequence charts [4].

Similarly, we also have the “simul-OR” box that specifies that the messages originating or terminating inside the box have an “OR” relationship rather than the “AND” relationship of the “simul” box. For received messages, the meaning is that the system will react to any subset of these messages that are received in that instance, rather than reacting only if all messages are received. For messages that are specified to be sent from a “simul-OR” box, the meaning is that any subset of these messages should be sent. This means that the execution model needs to extract all subsets of the messages in the “simul-OR” box and create branches in the hMSC so that all scenarios can be explored.

Timeouts

We use boxes labelled with numbers to represent timeouts. Some MSC languages have a syntax with a 3-sided arrow that begins and ends on the process line to represent timers; our syntax is similar. The semantic meaning of the timeout is that when the process encounters that box, if the hMSC is being implemented using a timed execution model such as a hardware description model, then the process continues only after that amount of time has elapsed. If the hMSC were being implemented using a non-timed execution model, the timeout box would be a no-operation.

Timeouts and simultaneous message events can be combined by drawing messages into or out of the timeout box. What this means is that the message events in the message box occur simultaneously upon completion of the timeout period.

Synchronization edges

Recall that events get an ordering relationship only when they have a process in common (or via transitive relationships involving common processes). In the LTSSM model, we have transmit and receive modules that do not communicate with themselves ever. Instead, all of the LTSSM’s communication behavior is with the outside LTSSM. This allows us to be fully general in specifying that incoming messages are concurrent with outgoing ones, and no ordering relationship is imposed. For example, in figure 3, even though the send message \( a \), sent by the TX process is visually depicted as occurring before the received message \( b \), received by the RX process, there is no ordering relationship between these messages, and they can occur in any order. However, there are times when we may wish to specify a particular order; in that case, we use synchronization messages between the TX and RX processes. These messages are labelled “synch”, and are used only for establishing an ordering relationship through transitive deduction. For example, if we drew a “synch” message from the TX process to the RX process such that the “synch” message on the TX process came after message \( a \) and it ended on the RX process before message \( b \), then the ordering relationships on the process lines would lead to a transitive ordering relationship between messages \( a \) and \( b \) where the sending of \( a \) would occur before the reception of \( b \).

Figure 4 shows all of the syntactic enhancements. Messages \( ms_1 \) and \( ms_2 \) are sent simultaneously, and messages \( mr_1 \) and \( mr_2 \) are received simultaneously, but the actual order in which the send and receive occur is not specified and they can occur in any order. The “synch” edge however forces the other sends \((ms_3, ms_4)\), and the timeout of 20, to occur after the receives \((mr_1, mr_2)\) on the RX process. The “synch” edge does not enforce that the other receives on RX, for example receive \( mr_7 \), should occur after the message \( ms_2 \) has been sent. In order to enforce that ordering, a “synch” edge should be directed from the TX process to the RX process. The receive \( mr_7 \) occurs simultaneously with the expiration of a timer of duration 6, and then messages \( mr_3, mr_4, mr_5 \) are received simultaneously. After that, \( mr_6 \) is received. On the TX process, \( ms_3, ms_4 \) are sent simulta-

![Fig 4. Syntactic enhancements to MSCs.](image-url)
neously, and the TX process completes on the expiration of a timer of duration 20.

We will refer to the LTSSM bMSC to mean a bMSC with the four processes mentioned, and with the TX process sending messages and RX process receiving messages, and with synchronization edges between TX and RX processes that can go in either direction.

**Validation methodology**

The class of systems we target are those whose behavior can be validated purely through observing input-output behavior. For example, reactive systems whose behavior is defined by the manner in which they interact with an environment, can be validated purely through their input-output behavior. Of course, the reason for targeting this class is simple: hMSCs are best at describing message exchanges, and at the most abstract level, we describe the message exchanges between the environment and the system. The PCI-Express switch falls into this class naturally, as the LTSSM portion is a specification of how the switch interacts with another such device. This specification is captured in the hMSC graph that naturally depicts different scenarios of message exchanges, and the functional behavior of the device is completely transparent from the manner in which it responds to stimuli from the environment.

![Fig 5. Validation methodology using automatically generated interactors.](image)

Thus, the basic idea behind our hMSC validation methodology for input-output systems is to automatically generate test scenarios in a hardware description language, and simulate these scenarios against the designed RTL. In other words, each scenario mimics another such device that initiates a particular sequence of messages possible in the specification. If, at any point during this exchange, the device under test (DUT) responds with an output that does not match what is expected according to the specification in that particular scenario, an error is flagged, and we can report a defect in the functional behavior of the DUT. Figure 5 shows the simple setup, with the generated scenario from the hMSC referred to as the interactor. The signal interface is needed to map signal names used by the hMSC model to those used by the implementation in the DUT. In our project, we used the same signals and abstractions as used by the DUT in the hMSC model obviating the need for a signal interface, but this might not be possible in general.

**Scenario generation and coverage**

Each bMSC can be transformed into a finite state machine (FSM) using a standard procedure, described in [2]. We have made appropriate extensions to the standard procedure for FSM generation to allow for the additional syntax and semantics we have introduced into the hMSC model. These extensions are too lengthy to describe in this paper, and are thus omitted. Now, since the hMSC models the behavior of the LTSSM, we have to generate the interactor by reversing the roles of the TX and RX processes. In other words, if the behavior calls for the reception of a message, then the interactor has to send this message. If the behavior calls for sending a message, then the interactor should be waiting for this message. Assuming that the message signals have been suitably abstracted, we implement these as signals directly, asserting them when they are sent, and awaiting them when they are expected.

Recall that a scenario is simply a path in the hMSC graph. To generate the code for an interactor for a scenario, we simply generate the code for each bMSC FSM along the path, and add the necessary glue to ensure that the entire path works as a module. We also add all of the signal and variable declarations, and can generate the interactor in Verilog to the extent that it can be compiled and simulated against the DUT with no manual intervention at all.

In order to systematically cover the entire specification via proper tests, we have to define what it means to cover the entire specification. One simple metric is node coverage: to find a set of paths that cover all of the hMSC nodes. However, this is not as strong as edge coverage, where we seek a set of paths that cover all of the edges in the hMSC graph. Note that edge coverage implies node coverage. An even stronger metric is path coverage where we test all possible paths in the graph. The total number of paths in the graph can be exponential in the size of the graph. Finally, we can have metrics such as covering each adjacent sequence of $n$ nodes, or covering certain nodes at least/at most a given number of times in each sequence in the test suite [8]. The correct metric to use will probably depend on the design being tested. The LTSSM, for instance, does not have too much other state information not already reflected in the state reached via message exchanges. It might be considered “Markovian” in the sense that once we are in a particular state, it does not matter how we got there. Hence, the design should not have behaviors dependent on whether or not some other node was in the path; if it is in state $C$, then it does not matter whether that state was reached from state $A$ or state $B$. Of course, this point arises since the DUT has many more internal states than modeled by the hMSC. Therefore, it may not be necessary to use coverage criteria that require adjacent sequences of nodes, or all paths. We use the simple edge coverage criteria in this project based on this intuition, but we have no way of asserting that stronger coverage criteria would not have found defects not found in the edge-covered test suite, since we have not implemented scenario generators for all of these coverage criteria. Edge coverage has been used in software conformance testing, for example, in [11] at Microsoft based on abstract state machines as the modeling language, and in [8].

Given the criterion of edge coverage, we can generate scenarios by solving the Chinese Postman problem [10] on the (flattened) hMSC graph. The Chinese Postman algorithm takes a strongly connected graph, and returns the lowest cost tour that visits every edge at least once, where the cost of tour is the sum of the weights on the edges. The algorithm is efficient and runs in polynomial time ($O(n^3)$, where $n$ is the number of nodes in the hMSC graph). If each edge has a weight of one, then the lowest cost tour is simply the shortest tour that visits every edge at least once. We use the algorithm and implementation published by H. Thimbleby [3]. Of course, for a non-strongly connected graph, we run it on the strongly connected components (SCC) and use a simple algorithm to cover the edges in the acyclic graph connecting the SCCs.
Fig 6. Screen dumps of the Scenery tool. These dumps show the tool being used to generate scenarios based on all paths between two hMSC nodes.

**Scenery**

Our hMSC capture and validation tool is based on the LTSA tool developed at Imperial College in London [6]. We have made several enhancements to the syntax and semantics, and added the code generation engine and the scenario generation based on the Chinese Postman algorithm. Figure 6 shows a screen dump of the tool and the generated code windows.

We captured the entire LTSSM specification in Scenery; the resulting hMSC graph has 125 nodes and 195 transitions. The equivalent RTL implementation consists of about a 1000 lines of Verilog code. Since the hMSC model of the LTSSM is strongly connected, the Chinese Postman algorithm returns an optimal tour from one initial node of the hMSC graph, and the Chinese Postman tour is defined as the path, say, “ABCADFBAHG”, then we “chop” the tour on node A, and break it up into three sections: “ABC”, “ADF”, and “AGH”. In the LTSSM example, the chopping process results in 21 smaller sections, each averaging about 40 nodes in length. Interactors are generated for each section and simulated against the implementation. An added convenience of the tour being split based on the initial node is that it is feasible to test each section independently since the RTL is started most easily from the initial state. If the chopping were done on some other node, it may not be possible to get the RTL into the equivalent of that state to start the scenario from that state. In that case, we would have to add a path to whichever node we wished to start that scenario on.

The validation methodology, as described, is efficient and scalable. The main analysis engine, which is the scenario generator based on edge coverage, runs in polynomial time. Besides, the model itself is at a sufficiently high level that there is not any state space explosion since no enumeration or deduction of states is being performed. The model is only as complex as the specification. If other coverage criteria are desired, such as all paths between all nodes, then obviously there is exponential blowup and loss of efficiency. However, we believe that for the class of systems targeted, namely reactive, protocol type of systems whose behavior is almost entirely modeled by the input-output behavior, stronger criteria than edge coverage may not be required.

Based on the simulation of the scenarios generated by the Chinese postman tour, we were able to catch 15 inconsistencies and defects in the implementation. Some of the inconsistencies were due to ambiguities in the specification itself, and differing interpretations between us and the design team. Other defects were genuine and acknowledged as such. A typical type of defect was:

The specification requires that in the G state, the state will change to H for the Transmitter if directed to that state regardless of the presence of a directive signal to states B and/or C. For the state to change to H for the Receiver, the specification requires that an EOS signal be received and that directive signals to states B or C not be present. But there is no such specific requirement (that directive signals to B or C be not present) for the state to change to H for the Transmitter. So if the directive signals to go to either B or C state, are asserted with the directive signal to transition to H for the Transmitter, then the RTL state machine drives to some other state instead of H for the transmitter.

Another type of defect found was:

In the CI state, the specification requires LU to be 1, implying that the RTL is supposed to assert the equivalent signal. This signal is not asserted in this state.

The exercise proved that the methodology and tool can be used profitably with large industrial designs. Further refinements to the tool and its capabilities should enhance its validation ability even more. Code coverage analysis showed that we were achieving 100% line and state coverage for those parts of the RTL that deal with the specification; parts of the RTL written for debugging or other extraneous features not in the specification, and hence not in the hMSC model, are not covered obviously. This gives confidence that the methodology is precise and complete.

Note that our methodology makes no claims on finding fundamental defects in the specification itself. For instance, we do not attempt to find concurrency related bugs such as deadlock or livelock that may result for some particular set of (valued) message exchanges. These types of bugs are better found through model checking techniques, where we can use the hMSC specification to derive a labeled transition system and then apply well known techniques. There have also been techniques proposed that work directly on the bMSC model to prove certain types of properties, such as “some sub-sequence of send/receive events in the bMSC always occurs at the beginning”, using “property templates” [9]. Extensions of these ideas to hMSCs would be interesting and could be used as a first step before full blown model checking. In that context, for proving certain properties such as asserting that something should never happen, or something should always happen, a more general and powerful model like live sequence charts (LSCs) might be used [4]. Applying our methodology using LSCs, and for instance, their capability of specifying forbidden behaviors, is again an interesting area for future work. However, we should note that in the LTSSM example, the specification itself does not delineate forbidden behaviors in the protocol. Hence, the
modeler has to deduce this in some other non-trivial way, and this may become an exercise as impractical as finding “good properties”, the bane of most formal verification techniques to date. The strength of our methodology is the fact that it is rather straightforward for a designer to translate the natural language specification into an hMSC using an intuitive visual programming environment, and start generating test suites, without needing a doctorate in computer science, or advanced familiarity with mathematical logic. In fact, the entire model of the LTSSM was developed by a contractor (with suitable direction) who did not have a background in computer science, and was experienced only in electronic hardware design with a traditional electrical engineering background.

**Conclusion**

We have described a validation methodology using hierarchical message sequence charts (hMSC) that is effective in making the validation process more formal and systematic by leveraging the mathematical precision of the hMSC. The visual nature of the modeling language and formalism makes it particularly attractive and easy-to-use. We have developed a visual programming environment Scenery that we have used to validate part of the PCI-Express physical layer specification, and shown that the methodology is far more efficient and precise in generating test suites than traditional ad-hoc methods that are currently standard industrial practice.

**Acknowledgements**

Hemang Lotlikar helped us with modeling the LTSSM in Scenery and running the detailed simulations against the designed RTL. Atul Sharma helped us with the coverage analysis.

**References**

1. ITU Recommendation Z.120, Message Sequence Charts (MSC 96), Telecommunication Standardization Sector, Geneva, 1996.