Int J Softw Tools Technol Transfer (2005) 7: 341–350 / Digital Object Identifier (DOI) 10.1007/s10009-004-0145-x

Live sequence charts applied to hardware requirements specification and verification

A VCI bus interface model

Annette Bunker*, **Ganesh Gopalakrishnan, Konrad Slind**

School of Computing, University of Utah, Salt Lake City, Utah 84112, USA
E-mail: {abunker, ganesh, slind}@cs.utah.edu

Published online: 29 October 2004 –

Abstract. Techniques and tools for formally verifying compliance with industry standards are important, especially in System-on-Chip (SoC) designs: a failure to integrate externally developed intellectual property (IP) cores is prohibitively costly. There are three essential components in the practical verification of compliance with a standard. First, an easy-to-read and yet formal specification of the standard is needed; we propose Live Sequence Charts (LSCs) as a high-level visual notation for writing specifications. Second, assertions should be generated directly from the specification; an implementation will be scrutinized, usually by model checking, to check that it satisfies each assertion. Third, a formal link must be made between proofs of assertions and compliance with the original specification. As an example, we take the Virtual Component Interface (VCI) Standard. We compare three efforts in verifying that the same register transfer level code is VCI-compliant. The first two efforts were manual, while the third used a tool, *lscAssert*, to automatically generate assertions in LTL. We discuss the details of the assertion generation algorithm.

Keywords: Formal verification – Protocol compliance – Specification – Live Sequence Charts – Virtual Component Interface Standard

1 Introduction

Despite widespread agreement on the benefits of precise specifications, their development has not yet found wide acceptance in practice. When requirements specifications do exist, they are typically written in English, a particularly ambiguous, and therefore unsuitable, language. One reason for this lack of acceptance could be the difficulty of creating an unambiguous specification. Engineers must undergo training in specialized languages and notations in order for such a specification effort to be successful.

To mitigate this concern, we propose the use of Live Sequence Charts (LSCs) as a specification language that is both formal and easy to use. Examples done by hand suggest that LSCs can ease the property-writing burden on the formal verification engineer [8]; however, specialized training is still required in the input language of the back-end assertion verifier (typically a model checker). Such training may not be available as formal compliance verification becomes widely used in industry, especially as its popularity increases in intellectual property (IP) commerce settings. In order for LSCs to live up to their potential for specifying hardware protocols and aiding in protocol compliance verification, some automatic link between the specification and verification tools must be provided.

As a possible solution to this problem, our tool, *lscAssert*, inputs an LSC specification and outputs temporal properties suitable for use with a model checker. *lscAssert* reads an abstract syntax representation of the LSC, analyzes the relationships between the constituent locations, and generates temporal logic assertions. The model checker can then input the assertions and an implementation model (in Verilog) for use in the formal protocol compliance verification process.

The remainder of Sect. 1 consists of an introduction to LSCs, a discussion of novel features added to LSCs for this work, and a basic example, a handshake protocol. Section 2 summarizes our verification case study, an implementation of the Virtual Component Interface Standard. An overview of *lscAssert* is given in Sect. 2.1. We discuss the major components of the property generation algorithm in Sect. 3 (the complete property generation algo-

* *Present address:* Electrical and Computer Engineering Department, Utah State University, Logan, Utah 84322, USA.
E-mail: bunker@ece.usu.edu

This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406.

rithm resides in the appendix). Section 4 presents related research. Finally, Sect. 5 draws conclusions about this work and points out possibilities for future investigation.

1.1 Live Sequence Charts

Harel and Damm propose LSCs [10] as an extension to Message Sequence Charts (MSCs). Like their predecessors, LSCs consist of sets of processes, each denoted by rectangles containing process identifiers. Process lifelines extend downward from each process; arrows between lifelines represent messages passed between senders and receivers. The time scale of each process is independent of the others except for the preorder imposed on the events occurring in the sender and receiver by a passed message. An unordered set of events along a timeline is a *coregion*, represented by dotted lines near the timeline on which the events occur.

As the name suggests, Live Sequence Charts allow the specifier the ability to require that some or all elements on the chart occur or that a certain event within the chart be reached (liveness). A chart consists of a series of process timelines, each made up of many locations. These locations may represent messages passed or conditions to be checked. Required events, messages, and points along the lifeline are denoted by solid lines, while optional events, messages, and timepoints are denoted by dashed lines. Similarly, required subcharts are outlined by solid lines, while optional subcharts are outlined by dashed lines. Required and optional chart elements are also said to be hot and cold, respectively.

Unlike Message Sequence Charts, LSCs offer facilities for pre- and postconditions, represented by bars with convex ends that cross the lifelines of relevant processes. We require that our LSC specifications begin with a precondition and end in a postcondition stating when the LSC is allowed to fire and in what state the LSC leaves the system after it executes, respectively.

We choose LSCs as our specification language for several reasons, including their intuitive, graphical form, their expressiveness, and the availability of a formal semantics for them. We explore these and other advantages of using LSCs for hardware protocol specifications in previous work [7].

1.2 Adapting Live Sequence Charts for hardware

Two differences separate our work from previous work using LSCs. First, we limit the types of variables we allow, and second, we introduce a clock construct to explicitly represent the passage of system time.

While LSCs allow values of any sort to be assigned to untyped variables, our dialect admits only three operations on counter variables: incrementing, decrementing, and clearing. Some notion of auxiliary variable must be present in order to fully specify hardware protocols [18]. By limiting the possible types and operations on variables in our specifications, we make the property generation burden substantially lighter for our application domain of hardware protocols. Other applications may, of course, require more kinds of variables and operations.

Because we use LSCs to specify hardware protocols at the register transfer level, we must have some notion of a clock to which system behaviors synchronize. This level of abstraction is much lower than that of the software systems LSCs were originally designed to specify. Our approach is to model a clock edge as a horizontal line crossing all lifelines. Since clock pulses are represented as locations along the lifeline, clock events may have multiplicities like other locations. They may not have temperatures, however, as it does not make sense for a clock event to not reach all lifelines, since we assume single clock domain designs for this work. Moreover, clock ticks cannot be assumed to occur with any kind of regularity. The amount of time that passes between clocks i and $i+1$ may be completely unrelated to the amount that passes between $i+1$ and $i+2$.

Recent enhancements to the LSC language by its originators [14] support a notion of current system time that can be stored in *timestamp* variables. The values of these variables can be compared with the current time in order to enforce timing constraints. For instance, to specify that an acknowledgment must be received within three time units of when the request was sent, the specifier stores the current time in a timestamp variable, t_0, when the request is sent. When the acknowledgment is received, a condition compares the new current system time with t_0+3. Hardware protocols – at least those we have experience with – seem not to need this feature; indeed, it seems that using timestamp variables to model a hardware clock would require extensive use of conditions.

Our adaptations can be expressed in the full LSC syntax, and our semantics are compatible [6]. The full syntax of LSCs provides the specifier with great expressive powers; however, our limited syntax provides sufficient specification power for hardware protocols without making property generation intractable.

1.3 Example: a handshake protocol

Figure 1 shows an LSC specification of a handshake protocol. Although simple, it is emblematic of some of the difficulties in adapting LSCs as requirement specifications for hardware protocols.

There are two processes, `initiator` and `target`. A precondition bar requires that the interface be in a state in which `valid` and `ack` be deasserted before any actions of the protocol begin. Since this precondition is cold, or optional, as denoted by the dashed bar, if the specified interactions attempt to execute when the interface is not in this state, then the attempted interactions should simply exit or cease. Once `valid` and `ack` both deassert, however, the interface may begin a new transaction by waiting zero or more clock edges as noted by the hot, starred, clock tick.

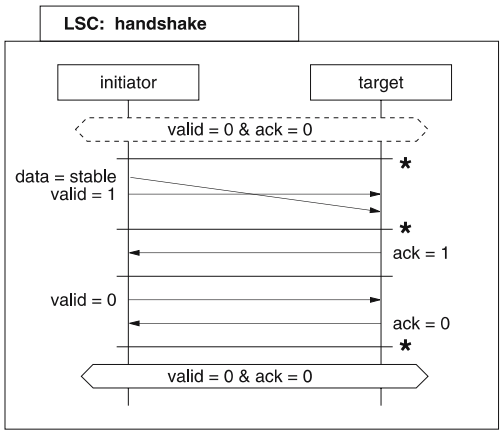

Fig. 1. Handshake protocol in the graphical LSC representation

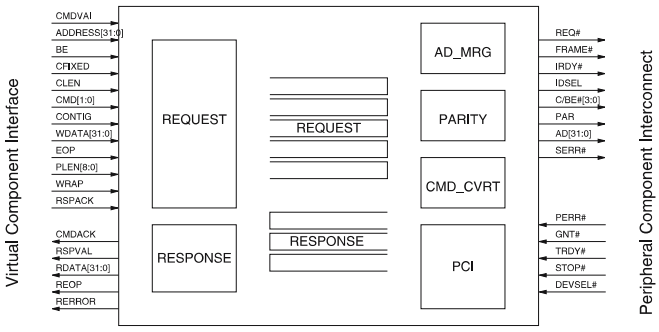

Fig. 2. Block diagram of PCI bus wrapper implementation

At this time, the initiator sets the `data` line and then asserts the `valid` signal. Once the target samples the `valid` signal asserted, it is free to sample the value on the `data` line. As noted by the crossing messages, however, it is not allowed to sample the data value prior to seeing `valid` asserted.

After zero or more clock ticks, the target then asserts `ack` to acknowledge receipt of the data value. At this point, a single clock tick must occur before the `initiator` can deassert `valid` and the `target` can deassert `ack` since the data transfer officially occurs on the clock edge on which both `valid` and `ack` are asserted. Again, zero or more clock ticks may occur before the transaction exits by leaving the interface in a state in which `valid` and `ack` are both deasserted. The hot postcondition indicates that the design must be considered flawed if the interface ever exits a transaction without ensuring that the condition holds.

2 Case study

In this section we compare three verifications of a PCI bus wrapper model (the first two efforts were first reported in detail elsewhere [8]):

1. A completely manual verification process: ad hoc specification created from English standards documents, manual property writing, and trial-and-error interaction with the model checker;
2. A more systematic approach, though still manual: properties generated by traversing an LSC specification and systematically writing down relevant assertions; and
3. An automatic process in which *lscAssert* generates all properties.

Figure 2 depicts the structure of the Peripheral Component Interchange (PCI) bus wrapper model. The model derives its name from the usage it would most likely receive in a System-on-Chip environment. An initiator module communicates via the Virtual Component Interface (VCI) to the wrapper, which translates and communicates requests via the system bus to a target module. In our example, the system bus is a PCI bus. Two state machines, `REQUEST` and `RESPONSE`, interface with the VCI initiator, while one, `PCI`, handles all communication with the PCI bus.

The Live Sequence Chart specification of the VCI Standard is shown in Fig. 3. It not only serves as the base-

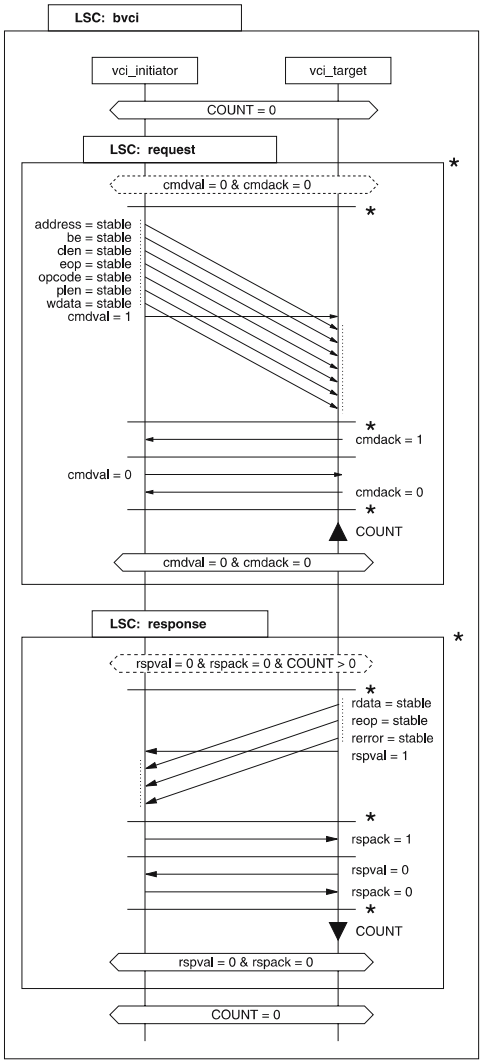

Fig. 3. Visual representation of the VCI specification

line for the second verification effort, but it provides the input specification for *lscAssert* for the third verification as well. As in the specification of the handshake protocol, the VCI Standard specification begins by requiring that `COUNT` be 0, then proceeds to process either a request or a response, depending on the truth value of the (cold) preconditions associated with each.

To make a request, the `initiator` first asserts data on the data lines `address`, `be`, `clen`, `eop`, `opcode`, `plen`, and `wdata`; then it asserts `cmdval` to signal that the request data is valid. After zero or more clock edges, the `target` acknowledges. Finally, `cmdval` and `cmdack` must deassert within one clock cycle, the counter increments to denote that a request is outstanding, and request postconditions are enforced.

To respond, the `target` puts the response data on `rdata`, `reop`, and `rerror` and signals that a valid response is available. After `initiator` acknowledges, `rspval` and `rspack` must both deassert within a clock cycle and `COUNT` decrements to denote that an outstanding request has seen a response.

2.1 Verification environment overview

Figure 4 shows how protocol compliance verification proceeds, using Live Sequence Charts, *lscAssert*, and a model-checking tool. As shown at the top, a translation between the visual LSC specification and the abstract syntax LSC specification occurs, which, while manual at present, could be automated. From the abstract syntax LSC representation, the automatic translation performed by *lscAssert* produces a list of assertions for the model checker. Following the design and verification path around to the left, a design team creates the implementation via its chosen design flow.

The model checker (we used FormalCheck [3]) checks each assertion against the implementation, producing waveforms when it finds a violation. The properties generated are in LTL, which is relatively straightforward to interpret in the model checker's property language.

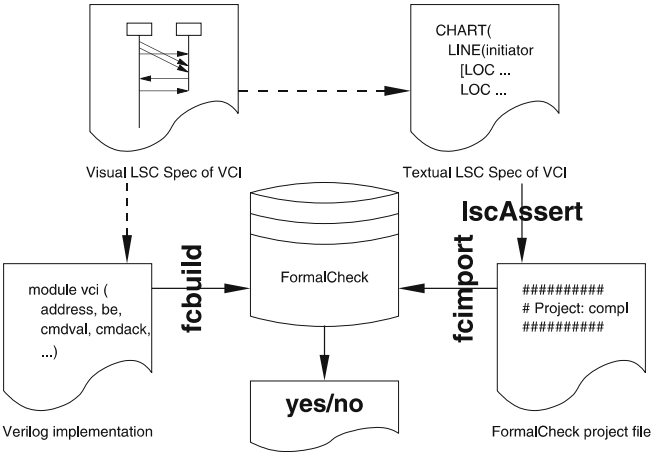

Fig. 4. Compliance verification system

One important aspect of the verification environment is the mapping between variable names in the specification and signals used in the implementation. At present, this *signal mapping* needs to be set up manually.

2.1.1 Model reductions and constraints

Our implementation of VCI is too large for model checking; therefore we reduce the Verilog model to allow the verification to proceed. The reduction shortens transaction queues, narrows bus widths, and constrains counter depths. All three verifications use the same reduced model. Done manually, this is a tedious and error-prone process, although ameliorated by use of Verilog parameters and symbolic constants. Happily, recent versions of FormalCheck can perform such reductions automatically.

Another consideration is *constraints*. Constraints represent the environment in which the implementation is assumed to execute. The constraint library for the earlier two verification projects contains ten constraints: four assert the liveness of the environment, four enforce timing rules on wrapper inputs, and two represent the system clock and reset. The *lscAssert* verification, however, requires that extra constraints be added to this library. Why? Because the properties generated by *lscAssert* probe more deeply into the design than those created manually, thus more of the circuit must be stimulated with correct timing in order to run this verification.

For instance, for the earlier projects, it is enough to require that the environment return responses to the translator, but the *lscAssert* verification also requires that the environment return valid response signal timing from the PCI interface. The final constraint library for this project contains 35 constraints, but it is never the case that all 35 are required for a single property to pass. In fact, no more than 15 constraints are ever placed on a single property.

2.1.2 Properties

lscAssert generates 34 properties where the manual processes generates only 28 properties, as the automatic generation considers only a small window of the LSC at once where properties in the previous verification projects address larger sections of the LSC. *lscAssert* also generates properties that check aspects of the specification, such as whether or not postconditions hold, which the properties of the previous projects did not examine.

Table 1 shows the FormalCheck statistics for nine representative properties proved in the three separate verifications. The first six show how properties most similar among the three verifications compare. Variations in characteristics can be ascribed to the differences in the exact expression of the properties among the three verifications and the progressively refined environmental model.

Table 1. Selected statistics, automatic LSC-based verification

	Verification 1			Verification 2			Verification 3		
Property	SVars	Time(s)	Mem(mb)	SVars	Time(s)	Mem(mb)	SVars	Time(s)	Mem(mb)
Live 1	67	82	202	71	249	481	78	250	475
Live 2	70	444	238	72	346	488	82	199	476
Live 3	69	1355	303	71	451	497	77	233	482
Safe 1	44	86	205	67	220	473	51	182	471
Safe 2	45	83	205	66	195	471	55	168	471
Safe 3	64	85	205	67	195	471	71	175	471
Res 1	–	–	–	143	dnf	dnf	150	2763	680
Res 2	–	–	–	143	dnf	dnf	66	5638	723
Res 3	–	–	–	143	dnf	dnf	157	5867	898

The three remaining properties all involve response queues, which eliminates much of the state space reduction FormalCheck can do internally. As can be seen, checking of these properties did not finish in the second verification. Even though the runtime characteristics of these properties are less desirable than others in the third verification, they at least complete; hence, *lscAssert* represents an improvement in verification abilities.

Of the 25 properties not represented in the table, all but 2 verify in less than 5 min and consume less than 480 MB of memory. These 2 remaining properties verify in 6 min and 30 min, respectively, and both take less than 585 MB of memory.

2.1.3 New bugs found

The first two verifications find eight behavioral anomalies in the PCI bus wrapper model. The first project produces these results in about 4 engineering weeks while the second produces similar results in roughly 2 weeks.

The *lscAssert* verification, however, finds all the known issues as well as four previously unreported issues in the bus wrapper. All four newly found issues represent functional incorrectness and relate to returning a response to the VCI initiator after the PCI network has processed the request. In one case, duplicate responses can be placed in the response queues. In another case, requests that should be retried on the PCI network are instead inserted into the response queues for transmission to the VCI initiator. The final two issues are closely related and involve incorrect timing of the assertion of response validation signal and the response data buses that must be stable while it is asserted. This verification effort requires approximately 2 engineering weeks.

3 Property generation

For each property generated, at least two and possibly as many as four consecutive locations along the timeline must be considered. A location l_i, sometimes called the *trigger* location because it forms the trigger for the resulting property, marks the beginning of the lifeline segment considered by a particular property. If the following location, l_{i+1}, is a clock location, then the timing information extracted from it must attach to the property trigger. If, however, l_{i+1} is any other type of location, it (or l_{i+2}, if l_{i+1} is a clock) forms the verification condition, attaching timing information from l_{i+2} (or l_{i+3}).

Property generation occurs in two passes. In the first pass, each timeline is traversed without consideration of interaction with the others. It walks timelines top-to-bottom, recursing into subcharts as it progresses. Three of the four possible formula generation cases may be encountered in this pass as discussed in Sects. 3.1, 3.2, and 3.3 below. Responsibility for the fourth case rests in the second pass, in which lifelines are traversed pairwise simultaneously to allow for this relationship analysis, which is described in Sect. 3.4. Finally, more advanced features, such as subcharts, temperatures, and multiplicities, are considered in Sect. 3.5.

3.1 Eventually/never pairs

In the LSC fragment shown in Fig. 5, two messages travel parallel routes between the sending process and receiving process with no required clock ticks following either message. This fragment, taken directly from our handshake protocol LSC, demonstrates the simplest situation for which *lscAssert* generates properties.

As it is hot, the location containing the `valid = 0` message forces execution to eventually reach the following location, containing `ack = 0`. Put differently, after the `valid` bit is deasserted, eventually the `ack` bit must be deasserted.

In linear temporal logic (LTL), the eventually property reads

$$(valid = 0) \Rightarrow \Diamond(ack = 0).$$

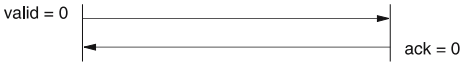

Fig. 5. LSC construct resulting in eventually/never pairs

Because time flows forward down the lifeline of a process, events later in the listing of the timeline must occur after events earlier in the listing. The never[1] property generated for the message pair of Fig. 5 reflects these semantics. In this case, the `ack` signal may never be invalidated unless the `valid` line has been previously deasserted as well. In LTL, the never property is

$$\neg(ack = 0)\mathbf{U}(valid = 0).$$

Locations containing multiple messages (coregions) generate one eventually property and one never property for each message, whether the coregion is in the trigger location or the verification location.

An interesting subcase of the above pattern that should be noted here is that of the precondition being the first location in a lifeline. Because preconditions should always be cold, they are never required to hold by the specification. However, if they do hold, then ensuing actions specified by the LSC must be observed. As the first location of the timeline, the precondition would need special consideration if it were ever possible for it to be the verification location. Since it does not make sense, though, to verify that a cold condition holds on the implementation, the precondition need never be considered as a candidate verification condition. This fact allows the algorithm to begin by placing the precondition in the trigger condition slot and ensuing locations in timing-related slots or the verification condition slot.

3.2 Eventually/never pairs with timing information

Figure 6 shows one variation of the second type of property that may be generated by *lscAssert*. This case is similar to the one discussed in Sect. 3.1 except that it contains information regarding timing. In this example, also taken from the handshake protocol, a clock edge occurs after `ack = 1`. This means that in order for `valid = 0` to be allowed to occur, `ack = 1` must be observed on the rising edge of the clock. The eventually property describing this case, then, says

$$(ack = 1) \wedge (clock = rising) \Rightarrow \Diamond(valid = 0).$$

Similar adjustments are made to the never property. Note that the timing information always accompanies the location preceding it when added to the property generated. Adding relevant timing information to this never property generates

$$\neg(valid = 0)\mathbf{U}((ack = 1) \wedge (clock = rising)).$$

The example shown here requires that we consider three of the possible four locations in order to generate our property pair. Another possibility that requires us to examine three locations simultaneously is that the trigger location is followed by the verification location, which is then followed by a clock location. Finally, all four locations must be taken together to generate properties for instances in which both trigger locations and verification locations are followed by required clock ticks. For instance, a lifeline segment that receives signal `a`, which must be valid on the clock tick, and then sends signal `b`, which must also be valid on the next clock edge, would require *lscAssert* to consider four locations simultaneously, producing the following two properties:

$$(a = 1) \wedge (clock = rising) \Rightarrow \Diamond(b = 1) \wedge (clock = rising)$$

and

$$\neg(b = 1) \wedge (clock = rising)\mathbf{U}((a = 1) \wedge (clock = rising)).$$

3.3 Postconditions

The process of generating properties for postconditions deviates slightly from the property generation scheme presented above, as postconditions generate only one property. As the handshake protocol does not illustrate the reason for this behavior very well, in its stead, imagine a protocol identical to the handshake protocol, with one exception. The postcondition for this hypothetical protocol requires some variable `X` to be set to 0 upon exiting the chart, rather than determining the values of `valid` and `ack`. This LSC does not prohibit `X` from being set to 0 at some point in the transaction prior to the occurrence of the last event, hence the property $\neg(X = 0)\mathbf{U}(ack = 0)$ may or may not hold. The LSC merely requires that the transaction set `X` to 0 before exiting. As a result, only the following property would be generated for our imaginary protocol:

$$(ack = 0) \Rightarrow \Diamond(X = 0).$$

Similarly, for the handshake protocol, *lscAssert* produces the following, based on the handshake LSC:

$$(ack = 0) \Rightarrow \Diamond((valid = 0) \wedge (ack = 0)).$$

3.4 Stable data properties

Checking that data lines remain stable during the entire time interval over which they are valid requires a separate

[1] We use the term *never* as shorthand for *not-until*.

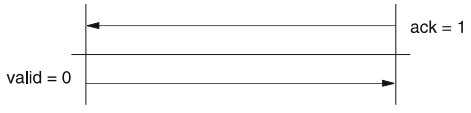

Fig. 6. LSC construct resulting in eventually/never pairs with timing

traversal of the chart because the relationships analyzed in the first pass are different from those analyzed in the second. In the first pass, contiguous locations in a single timeline must be considered together. However, timelines are traversed in pairs so that message crossings such as the one depicted in Fig. 7 and used in the Virtual Component Interface Standard [23] are detected in the second pass.

The algorithm begins by inspecting the first location of the current two timelines. If the locations are equal, as in the case of the handshake protocol in which the same precondition is required of both timelines, or if the locations contain events that are sends and receives of the same messages, the algorithm moves to the next pair of locations. If the locations are not equal and do not represent matching sends and receives, then auxiliary lists are checked to see if the matching send or receive has already been seen on the opposite timeline. If matches are not found, then the current locations are inserted into the lists to await future matches.

If matches are found, on the other hand, the messages must have crossed. In our experience, the only reason for crossing message paths of this sort in hardware protocol specification is the specification of the data/flag idiom. Once crossing messages are found, *lscAssert* detects which message is the data and which is the flag. The message that is sent second and received first must be the validation or flag signal. It cannot be asserted until after the data signal has had time to stabilize or else it violates precisely those guarantees for which it is responsible.

The only information still needed to output the property is the point at which the flag releases its guarantee on the data line. To do this, the algorithm searches forward on the receiving timeline until it finds the next location at which the flag signal changes values. It then uses that message as the discharge condition for the property, resulting in the LTL property

$$(valid = 1) \Rightarrow ((data = stable)\mathbf{U}(valid = 0)) ,$$

which states that the assertion of `valid` requires that the `data` bus remain stable until `valid` is deasserted.

The extraction of stable data properties is relatively complex, since the property must be inferred from interactions between multiple timelines. Very recent work on the semantics of LSCs [5] addresses this problem by augmenting LSCs with explicit syntax for data stability.

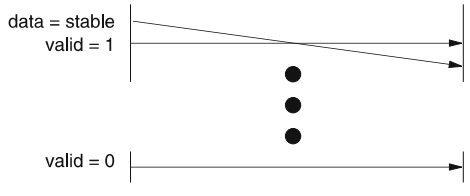

Fig. 7. LSC construct resulting in a stable data property

3.5 Advanced features

Even though it is a simple specification, the handshake protocol specification illustrates the process of generating properties for most of the constructs provided by LSCs, as shown in the previous section. The purpose of this section is to consider those features omitted by the previous section, which include cold locations, multiplicities, subcharts, and metavariables.

3.5.1 Subcharts and cold locations

Subcharts, as used in the Live Sequence Chart language, act like parentheses to group constructs together. As such, by themselves subcharts provide no new information regarding the requirements of the specification. Only when they are used in conjunction with other constructs do they specify behavior required of the implementation. For this reason, the presence of a subchart in the LSC makes *lscAssert* merely recurse on the structure of the charts, thereby continuing property generation. Only when subcharts have multiplicities attached to them are the generated properties changed in an interesting way. Full discussion of this situation takes place in Sect. 3.5.2 along with that of multiplicities.

The presence of a cold location indicates that execution of the process need not move past the location in question. In other words, previous liveness guarantees are released at a cold location. Hence, *lscAssert* does not generate a property stating that events at a later location will eventually occur for trigger messages inside cold locations. However, because the ordering of events on a lifeline imposes a compatible order on the occurrence of events in the implementation process, the events following a location may only happen if that location has been passed. This ordering relation is imposed on locations whether they are hot or cold. For this reason, cold locations generate only the never half of an eventually/never pair.

3.5.2 Multiplicities

Property generation for multiplicities is an interesting problem because *lscAssert* must create two different types of properties, depending on the construct to which the multiplicity is attached. This section considers the two cases of property generation applied to multiplicities.

Four LSC constructs may have multiplicities attached to them: clock locations, send messages, receive messages, and charts (subcharts). Figure 8 shows the first three cases, sends with multiplicities, receives with multiplicities, and clocks with multiplicities, all of which invoke the same behavior in *lscAssert*. For each, no property is generated because the LSC construct specifies no guarantee. By requiring that a message be sent or received zero or more times, the LSC essentially requires that either the event happens or it does not. Likewise, requiring that a clock tick zero or more times places no functional requirement on its behavior.

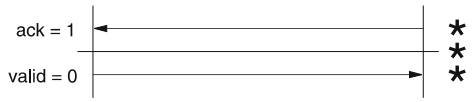

Fig. 8. Multiplicities resulting in no property generation

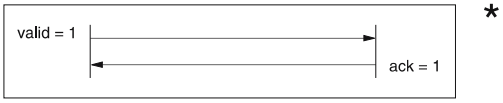

Fig. 9. Multiplicities resulting in property generation

Generating properties for multiplicities attached to subcharts, however, is slightly more interesting. By attaching a multiplicity to a subchart, the specifier requires that the contents of the subchart be placed in a loop, which may execute zero or more times. For example, the LSC in Fig. 9 shows two messages exchanged between two processes within a subchart that, in turn, has a multiplicity attached to it. Notice that if the `valid = 1` message is sent, then execution has entered the loop and all ensuing events must take place.

Because *lscAssert* cannot know from the specification how many times the subchart executes, it cannot generate properties describing the loop itself. It can, however, generate properties that reflect the above observation: that if the first event within the loop executes, then all others (subject to the temperatures of the locations in which they reside) must execute as well. As a result, the LTL formula describing this LSC is

$$\Box((valid = 1) \Rightarrow \Diamond(ack = 1)),$$

which requires that every time `valid` is asserted, `ack` be observed asserted at some point in the future.

4 Related research

Work related to our research is generally found in three categories: integration of specification and verification techniques into industrial design flows, case studies involving specifying and verifying standards at the register transfer abstraction level, and investigations surrounding the applicability of Message Sequence Charts and Live Sequence Charts to various problem domains.

The importance of high-level, formal interface specification and verification methods in industrial processes has increased greatly in recent years. The complexity of recently proposed standards implies that their adoption depends on the development of reliable SOC cores that implement these standards [12]. This, in turn, relies on effective ways of specifying the standards and verifying RTL designs for compliance. A recent expert panel points out that the integration and verification of IP will depend not only on the individual tools chosen but also on the design process adopted [20]. Two suggested key ingredients of such a process are the use of high-level abstractions and interface monitors [1].

We aimed to create an automatable verification process that bridges the gap between high-level graphical specifications of standards, RTL level implementations, and industrial-scale model checkers. Of course, *lscAssert* is merely a prototype, but it has helped close the gap, as our experiments show.

Most of the work previously done in the area of standards specification and verification involves the PCI standard. Shimizu, et al. [26] specify the PCI standard using monitors written in HDLs. Monitor-based specifications can be checked for consistency and for receptivity, and they can be used to generate testbenches, correctness checkers, and functional coverage metrics [25]. However, they seem to lack conceptual cohesion and we expect them to be difficult to understand and to communicate to colleagues. Other work demonstrates the verification of the PCI specification at roughly the same level of abstraction as the one reported here [9, 27] and at the transaction level [15].

More research than we can adequately summarize here focuses on using Message Sequence Charts as a specification language. Challenges in using MSCs include conflicting language interpretations [17], unwanted scenario introduction [2], lack of expressiveness [13], and poor mechanical analysis characteristics [21]. In an attempt to resolve some of these issues, Harel and Damm present Live Sequence Charts, specifying a railcar control system and providing a formal semantics for the language [10]. Later work by Klose and Wittke [16] attempts to close gaps apparent in the formal semantics by defining LSCs in terms of Büchi Automata. Following up on the former aspect of the work, Bohn et al. discuss the specification and verification of radio-based train signaling systems [4], Damm and Klose consider the integration of LSCs into the Statemate verification tool [11], and Harel and Marelly introduce time into LSCs [14]. Bontemps and Heymans [5] provide an interpretation of a slightly altered version of LSCs (in which data stability is directly expressible) into automata.

5 Conclusions and future work

Live Sequence Charts show promise as a language for graphical and formal specifications of hardware protocols. Their intuitive syntax overcomes some of the obstacles inhibiting the wide use of formal specifications. The addition of constructs for describing timing requirements makes them expressive enough to specify common hardware protocol communication idioms. The formal foundations on which LSCs develop make them suitable for specifying standards as well as providing requirements specifications for formal verification.

This paper presents our approach to providing the critical link between a readable formal specification and the input language of a commercial model checker. A tool called *lscAssert* plays a central role in this approach: it traverses the LSC specification and produces a series of temporal logic formulas in a format appropriate for input to an industrial model checker. The algorithm underlying the tool consists of two passes, one of which produces properties that check the ordering of events. The other, however, produces properties checking whether or not output stability guarantees are met.

Our case study shows the (expected) benefits of mechanization over working by hand, e.g., fewer mistakes were made. However, we were somewhat surprised that more assertion verifications were able to finish with *lscAssert*. We believe that this is due to the fact that the granularity of generated properties is smaller and probe deeper into the design, thus providing more "well-focused" verification tasks.

In order to complete the link between the specification and the results produced by the model checker, however, protocol compliance must be formally defined. Since the formulas generated from each timeline are checked with respect to the formulas generated for other timelines, we believe Assume-Guarantee reasoning [19, 22, 24] can provide a formal basis for the judgments returned by our verification environment.

In particular, suppose we are given an implementation I comprising subcomponent implementations I_1, I_2, ..., I_n that communicate with one another via some protocol specified by an LSC S containing timelines $T_1, T_2, \ldots, T_n$. For each timeline T_i, *lscAssert* generates an LTL formula F_i; thus, $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ is an LTL formula describing the entire LSC S. Suppose the following formulas VC_1, VC_2, ..., VC_n all hold:

$$
\begin{array}{ll}
(VC_1) & F_2 \wedge F_3 \wedge \ldots \wedge F_n \wedge I_1 \Rightarrow F_1 \\
(VC_2) & F_1 \wedge F_3 \wedge \ldots \wedge F_n \wedge I_2 \Rightarrow F_2 \\
 & \ldots \\
(VC_n) & F_1 \wedge F_2 \wedge \ldots \wedge F_{n-1} \wedge I_n \Rightarrow F_n \,.
\end{array}
$$

Then we conclude, via an application of Assume-Guarantee reasoning, that $I \Rightarrow S$ holds, i.e., that the implementation of the entire system satisfies its specification. This reasoning step is currently an informal one; making it formal is important future work.

Acknowledgements. The authors thank Michael D. Jones for comments on drafts of this paper.

References

1. Albin K (2001) Nuts and bolts of core and SoC verification. In: Proceedings of the 2001 conference on design automation, pp 249–252
2. Alur R, Etessami K, Yannakakis M (2000) Inference of Message Sequence Charts. In: Proceedings of the 22nd international conference on software engineering, pp 304–313
3. Bell Labs Design Automation and Lucent Technologies (1998) FormalCheck User's Guide, v2.1 edn
4. Bohn J, Damm W, Wittke H, Klose J, Moik A (2002) Modeling and validating train system applications using statemate and Live Sequence Charts. In: Ehrig H, Krämer BJ, Ertas A (eds) Proceedings of the 6th biennial world conference on integrated design and process technology, June 2002. Society for Design and Process Science, p 34
5. Bontemps Y, Heymans P (2003) Turning high-level live sequence charts into automata. Technical report, Computer Science Department, University of Namur
6. Bunker A (2003) Applying a visual specification language to hardware protocol verification. PhD thesis, University of Utah, August
7. Bunker A, Gopalakrishnan G (2001) Using Live Sequence Charts for hardware protocol specification and compliance verification. In: Proceedings of the IEEE international workshop high level design validation and test workshop, November 2001. IEEE Press, pp 95–100
8. Bunker A, Gopalakrishnan G (2002) Verifying a VCI bus interface model using an LSC-based specification. In: Ehrig H, Krämer BJ, Ertas A (eds) Proceedings of the 6th biennial world conference on integrated design and process technology, June 2002. Society of Design and Process Science, p 48
9. Chauhan P, Clarke EM, Lu Y, Wang D (1999) Verifying IP-Core based System-On-Chip designs. In: Proceedings of the IEEE international ASIC/SOC conference, September 1999, pp 27–31
10. Damm W, Harel D (2001) LSCs: Breathing life into Message Sequence Charts. Formal Methods Sys Des 19(1):45–80
11. Damm W, Klose J (2001) Verification of a radio-based signaling system using the statemate verification environment. Formal Methods Sys Des 19:121–141
12. Grahm T, Clark B (2001) SoC integration of reusable baseband bluetooth IP. In: Proceedings of the 2001 conference on design automation, pp 256–261
13. Gunter EL, Muscholl A, Peled DA (2001) Compositional message sequence charts. In: Margaria T, Yi W (eds) Proceedings of the conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York, pp 496–511
14. Harel D, Marelly R (2002) Playing with time: on the specification and execution of time-enriched LSCs. In: Proceedings of the 10th IEEE/ACM international symposium on modeling, analysis and simulation of computer and telecommunication systems, October 2002, pp 193–202
15. Jones MD (2001) Formal verification of parameterized protocols on branching networks. PhD thesis, University of Utah
16. Klose J, Wittke H (2001) An automata based interpretation of live sequence charts. In: Margaria T, Yi W (eds) Proceedings of the conference on tools and algorithms for the construction and analysis of systems. Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York, pp 512–527
17. Krüger I, Grosu R, Scholz P, Broy M (1999) From MSCs to statecharts. In: Distributed and parallel embedded systems. Kluwer, Dordrecht
18. Martin AJ (1993) Synthesis of asynchronous VLSI circuits. Technical report Caltech-CS-TR-93-28, California Institute of Technology
19. McMillan KL (1999) Circular compositional reasoning about liveness. In: Pierre L, Kropf T (eds) Correct hardware design and verifiation methods. Lecture notes in computer science, vol 1703. Springer, Berlin Heidelberg New York, pp 342–345
20. Moretti G (2001) Your core – my problem? Integration and verification of IP. In: Proceedings of the 2001 conference on design automation, pp 170–171
21. Muscholl A, Peled D (2000) Analyzing message sequence charts. In: Proceedings of SDL and MSC'00, June 2000
22. Namjoshi KS, Trefler RJ (2000) On the completeness of compositional reasoning. In: Proceedings of the conference on computer aided verification. Lecture notes in computer science, vol 1855. Springer, Berlin Heidelberg New York, pp 139–153
23. OCB Design Working Group (2000) VSI Alliance Virtual Component Interface Standard. Virtual Socket Interface Alliance, November 2000

24. Rushby J (2001) Formal verification of McMillan's compositional assume-guarantee rule. Technical report, Computer Science Laboratory, SRI International, September 2001
25. Shimizu K, Dill DL (2002) Deriving a simulation input generator and a coverage metric from a formal specification. In: Proceedings of the 39th conference on design automation. Association for Computing Machinery, pp 801–806
26. Shimizu K, Dill DL, Hu AJ (2000) Monitor-based formal specification of PCI. In: Hunt WA Jr, Johnson SD (eds) Proceedings of the conference on formal methods in computer-aided design, November 2000. Lecture notes in computer science, vol 1954. Springer, Berlin Heidelberg New York, pp 335–352
27. Wang D (1999) Formal verification of the PCI local bus: a step towards IP Core based System-On-Chip design verification. Master's thesis, Carnegie Mellon University, Pittsburgh, May 1999

Appendix: Property generation algorithm

This section summarizes the entire property generation algorithm employed by *lscAssert*. As noted previously, the tool makes two passes over an LSC, each looking at different types of relationships between locations as shown in the following ML-style pseudocode.

Pass 1
foreach $t \in$ timelines, $\mathbf{F}(t)$
where
$\mathbf{E}(l_i, l_{i+1}, l_{i+2}, l_{i+3}) =$
  if is_clock_loc (l_{i+1}) **then**
    if is_clock_loc (l_{i+3})
    then output $(l_i \wedge l_{i+1}) \Rightarrow \Diamond\, (l_{i+2} \wedge l_{i+3})$
    else output $(l_i \wedge l_{i+1}) \Rightarrow \Diamond\, l_{i+2}$
  else if is_clock_loc (l_{i+2}) **then output** $l_i \Rightarrow \Diamond(l_{i+1} \wedge l_{i+2})$
    else output $l_i \Rightarrow \Diamond\, l_{i+1}$
and
$\mathbf{N}(l_i, l_{i+1}, l_{i+2}, l_{i+3}) =$
  if is_clock_loc (l_{i+1}) **then**
    if is_clock_loc (l_{i+3})
    then output $\neg\, (l_{i+2} \wedge l_{i+3})$ **U** $(l_i \wedge l_{i+1})$
    else output $\neg\, (l_{i+2} \wedge l_{i+3})$ **U** l_i
  else if is_clock_loc (l_{i+2})
    then output $\neg\, (l_{i+1} \wedge l_{i+2})$ **U** l_i
    else output $\neg\, l_{i+1}$ **U** l_i
and
$\mathbf{F}(l_i{::}l_{i+1}{::}l_{i+2}{::}l_{i+3}{::}\text{rst}) = \mathbf{E}\ (l_i, l_{i+1}, l_{i+2}, l_{i+3});$
  $\mathbf{N}(l_i, l_{i+1}, l_{i+2}, l_{i+3});$
  $\mathbf{F}(l_{i+1}{::}l_{i+2}{::}l_{i+3}{::}\text{rst})$

Pass 2
foreach $t_i, t_j \in$ timelines, $\mathbf{G}(t_i, \emptyset)\ (t_j, \emptyset)$
where
$\mathbf{G}(l_{i,k}{::}rst_i, aux_i)\ (l_{j,k}{::}rst_j, aux_j) =$
  if event_match$(l_{i,k}, l_{j,k})$
  then $\mathbf{G}(rst_i, \{l_{i,k}\} \cup aux_i)\ (rst_j, \{l_{j,k}\} \cup aux_j)$
  else let (data, flag, flagline) = df_detect$(l_{i,k}, l_{j,k})$
      f = find_next_change(flag, flagline)
    in write_cross_props(flag, data, f);
      $\mathbf{G}(rst_i, aux_i \setminus \{l_{i,k}\})\ (rst_j, aux_j \setminus \{l_{j,k}\})$

Pass 1 matches on the types of locations found consecutively. Once the location pattern and the presence of relevant modifiers are determined, then the property is written out. Pass 2, on the other hand, begins by creating two auxiliary variables that track the list of locations already seen that do not match up across lifelines while it traverses the chart. Taking up the first locations in each of two lifelines, it compares them to see if they match. A match is defined as either an identical location such as a condition that touches all lifelines or the send and receive of the same message. If the two events do not match, then *lscAssert* searches the auxiliary lists to see if their matches have been seen along the opposite lifeline previously. If this is not the case, then the locations are added to the previously seen lists and traversal continues.

However, if matches to the current messages have been seen, then crossing messages have been detected, in which case *lscAssert* must determine which of the two locations under consideration is the data and which is the flag; `df_detect` performs this task. *lscAssert* then searches forward in the lifeline to find the next time the flag signal changes values. Finally, *lscAssert* emits the property and the location matches are removed from the auxiliary lists so that future sightings of the signal names do not cause premature property generation.