THE NATIONAL UNIVERSITY
of SINGAPORE

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TRA7/03

Symbolic Simulation of Live Sequence Charts

Shishir C. CHOUDHARY, Abhik ROYCHOUDHURY and Roland Hock Chuan YAP

July 2003

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Symbolic Simulation of Live Sequence Charts*

Shishir C. Choudhary Abhik Roychoudhury Roland H.C. Yap

School of Computing, National University of Singapore, Singapore 117543.
{shishirc,abhik,ryap}@comp.nus.edu.sg

Abstract. Message Sequence Charts (MSC) have traditionally been used as a weak form of requirements specification in software design; they denote scenarios which may happen. Live Sequence Charts (LSC) extend Message Sequence Charts by also allowing the designer to specify scenarios which must happen. Live Sequence Chart specifications are executable; their simulation allows the designer to play out potentially aberrant scenarios prior to software construction. In this paper, we develop a simulation engine for Live Sequence Charts using CLP technology. The utility of (constraint) logic programming in this application stems from its ability to execute in presence of variables with non-ground values. This allows us to simulate multiple scenarios at one go. For example, several scenarios which only differ from each other in the value of a variable can be executed as a single scenario where the variable value is left uninstantiated. Similarly, we can simulate scenarios with an unbounded number of processes. We use the power of CLP(R) to also simulate charts with non-trivial timing constraints. Currently work on MSC/LSCs use symbolic variables mainly for ease of specification; they are ground during simulation. Thus, CLP technology advances the state-of-the-art in simulation and testing of MSC/LSC based requirements specifications.

1 Introduction

Message Sequence Charts (MSCs) [12] have traditionally played an important role in software development. MSCs describe scenarios of system behavior. These scenarios are constructed prior to the development of the system, as part of the requirements specification phase. MSCs can be used to depict the interaction between different components (objects) of a system, as well as the interaction of the system to the external environment (if the system is reactive).

Syntactically, a MSC consists of a set of vertical lines, each vertical line denoting a process (or a system component). Computations within a process are shown via internal events, while any communication between processes is denoted by a uni-directional arrow (typically labeled by a message name). Figure 1(a) shows a simple MSC with two processes; $m1$ and $m2$ are messages sent from p to q and a is an internal action.

The main difficulty in using MSCs for requirements specification is that they only denote existential requirements: a scenario which *may* occur. They cannot

* NUS Technical Report TRA7/03

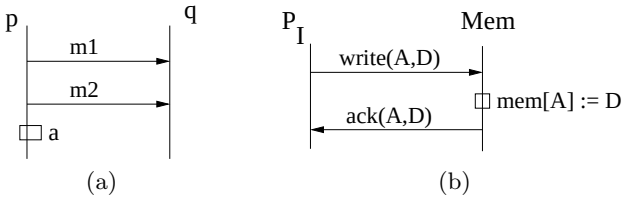


Fig. 1. (a) A simple MSC, and (b) MSC with symbolic variables

capture universal requirements, that is, safety properties which *must* hold in all behaviors of the system. To fill this gap, Damm and Harel recently proposed an important extension of MSCs called Live Sequence Charts (LSCs) [4]. In the LSC formalism, a chart may be marked as existential or universal. Each chart consists of a pre-chart and a body chart. An existential chart is similar to a conventional MSC. It denotes the property: the pre-chart followed by the body chart may execute in some run of the system. A universal chart denotes the following property: *if the pre-chart is executed in any execution trace of the system, then the body chart must execute.* Clearly, this is a safety property.

The LSC formalism serves as an important extension of MSCs; it allows us to describe the set of all allowed behaviors of a reactive system. Note that a universal chart in the LSC formalism represents a safety property that the traces of the system must satisfy. A collection of universal charts can serve as a complete behavioral specification: any trace (over a pre-defined alphabet) which does not violate any of the universal charts is an allowed behavior. Also, LSC based behavioral specifications are *executable*. Roughly speaking, execution of a collection of universal charts proceeds as follows (details appear in the next section). An external event is provided by the user/environment which is "matched" with enabled events. A matching enabled event is chosen and executed; this in turn enables other events one of which is again executed. The execution goes on until no event is enabled at which point it waits for an external stimulus. This search strategy forms the heart of the LSC execution engine proposed by Harel et. al., called the *play engine* [4, 5]. Thus, the LSCs serve as a complete requirements specification of the system behavior. Furthermore, since LSC specifications are executable, they can be simulated via a play engine. The advantage of simulating LSC specifications via a play engine are obvious: it allows the user to visualize/navigate/detect *unintended behaviors which were mistakenly allowed in the requirements.*

The search strategy in a play engine involves choice (between enabled events) and backtracking (due to detected violation of the safety property denoted by a universal chart). In addition, LSCs can be extended with symbolic variables in messages as well as process instances [10]. In order to find enabled events which "match" a given event, unification is used; this allows value passing into the symbolic variables. Figure 1(b) shows a chart with symbolic variables. In this chart, processor I requests main memory to write value D in address A. The main memory performs this task and sends an acknowledgment. This chart

uses two kinds of variables: A and D are symbolic variables appearing in events; I is a symbolic variable appearing in a process instance (the instance id of the processor in question). In this paper, we use normal Prolog variable convention where symbolic variables are capitalized.

Contributions Overall, the search strategy of an LSC play engine involves unification, choices and backtracking. Also note that LSC specifications can involve timing constraints. This allows a natural encoding of the LSC simulation search as a (constraint) logic program. We have developed a prototype LSC play engine using the CLP(R) system [8]. *However, our CLP based play engine is much more than an elegant encoding of an existing search algorithm.* Existing literature on LSCs [4–6, 10] allow symbolic variables to appear in the LSC specification; charts which differ only in the value of a variable can be drawn as a single chart with a symbolic variable. However, even though symbolic variables appear in the specification, they are typically ground during simulation. In our play engine we exploit the CLP search to allow a truly symbolic simulation. Our search is symbolic in at least three ways:

- data variables can be unbound during simulation.
- control variables (such as the instance id I in Figure 1(b)) can also remain non ground during simulation. This allows us to directly simulate a process with unbounded number of instances, which is impossible without allowing symbolic variables and constraints during the search.
- during the simulation of charts with timing constraints, time is maintained as a collection of constraints instead of a fixed value.

This allows for more efficient simulation of finite state systems (multiple scenarios are simulated at one go). Furthermore, it allows us to simulate infinite state systems with data variables from an infinite domain, or systems with unbounded number of processes.

Section Organization The rest of the paper is organized as follows. Section 2 provides an overview of LSCs and play engine. Section 3 describes our simulation based engine through simple examples. Section 4 describes the use of our engine for simulating LSCs with unbounded number of process instances. Section 5 explains the simulation of charts involving timing constraints. Finally, section 6 concludes the paper with discussion and future work.

2 Live Sequence Charts and the Play Engine

Live Sequence Charts (LSCs) [4] is a powerful visual formalism which serves as an enriched property specification language as well as a high-level modeling language. Descriptions in the LSC language are executable, and the execution engine which supports it is called the *Play Engine*. In this section we summarize the existing work on LSCs. We start with MSCs, show how they are extended to LSCs, and then briefly describe existing work on play engine.

2.1 Message Sequence Charts

Message Sequence Charts (MSCs) are written in a visual notation as shown in Figure 1(a). Semantically, a MSC denotes a set of events (message send, message receive and internal events corresponding to computation) and prescribes a partial order over these events. This partial order is the transitive closure of the following ordering relations [1].

- the total order of the events in each process (time flows from top to bottom in each process)
- the ordering imposed by the send-receive of each message (the send event of a message must happen before its receive event)

The events are described using the following notation. A send of message M from process P to process Q is denoted as $\langle P!Q, M\rangle$. A receive event by process Q to a message M sent by process P is denoted as $\langle Q?P, M\rangle$. An internal event A executed by process P is denoted as $\langle P, A\rangle$. As mentioned earlier, the message M as well as the processes P, Q can contain symbolic variables. Symbolic variables transmitted via messages can appear in internal events (like A) as well.

Consider the chart in Figure 1(b). Using the above notation, the total order for process p is $\langle p!q, m1\rangle < \langle p!q, m2\rangle < \langle p, a\rangle$ where $e1 < e2$ denotes that event $e1$ "happens-before" event $e2$. Similarly for process q we have $\langle q?p, m1\rangle < \langle q?p, m2\rangle$. For messages $m1$ and $m2$ we have

$$\langle p!q, m1\rangle < \langle q?p, m1\rangle \text{ and } \langle p!q, m2\rangle < \langle q?p, m2\rangle$$

The transitive closure of these four ordering relations defines the partial order of the chart. Note that it is *not* a total order since from the transitive closure we cannot infer that $\langle p!q, m2\rangle < \langle q?p, m1\rangle$ or $\langle q?p, m1\rangle < \langle p!q, m2\rangle$. Thus, in this example chart, the send of $m2$ and the receive of $m1$ can occur in any order.

2.2 Universal and Existential Charts

In the Live Sequence Chart (LSC) terminology, each chart is a concatenation of a pre-chart followed by a body chart. The notion of concatenation requires some explanation. Consider a chart $Pre \circ Body$ where $\circ$ denotes concatenation. This means that all processes first execute the chart Pre and then they execute the chart $Body$; no event of chart $Body$ takes place before any event of chart Pre. For example, consider the chart in Figure 4. As a notational convention, we always show the pre-chart inside a *dashed hexagonal box*. The process r cannot send the message $m1(X)$ before the pre-chart is finished. Note that this is required even though r does not take part in the pre-chart. This restriction is imposed so that the body chart is executed only when the pre-chart is successfully completed.

As mentioned earlier, charts are classified as existential or universal. An existential chart $Pre \circ Body$ represents the following property ϕ: a system model M satisfies ϕ if there exists a reachable state of M from which an outgoing trace executes (a linearization of) Pre followed by (a linearization of) $Body$. On the

other hand, a system model M satisfies a universal chart $Pre \circ Body$ if : from every reachable state of M if a (linearization of) the pre-chart Pre is executed, then it must be followed by (a linearization of) the body chart $Body$. In other words, for any execution trace of M, whenever Pre is executed, $Body$ *must* be executed. This constitutes a safety property.

Along with universal/existential charts, LSCs also allow locations or events in a chart to be universal or existential in a similar fashion. Indeed our CLP based simulation engine works for the whole LSC language with existential/universal charts as well as existential/universal chart elements (such as location, condition etc). For details on syntax of the LSC visual language, the reader is referred to [4]. Automata based semantics of the language appear in [9].

2.3 The Play Engine

A LSC based system description can serve as a complete behavioral requirements specification. It specifies the desired inter-object relationships in a reactive system before the system (or even an abstract model of it) is actually constructed. It is beneficial to simulate the LSC based behavioral requirements since it detects inconsistencies and under-specification.

LSC based descriptions of reactive systems can be executed by providing an event performed by the user. The LSC simulation engine then computes a "maximal response" to this user-provided event, which is a maximal sequence of events performed by different components of the reactive system (as a result of the user-provided event). Simulation then continues with the user providing another event. In the course of simulation, pre-charts of given universal charts are monitored. If the pre-chart of a universal chart is successfully completed, then we generate a "live copy" – a copy of the body chart. During simulation, there may be several live copies of the same chart active at the same time. Such copies may be "violated" during simulation; this happens when the partial order of the events appearing the body chart is violated. Violation of a universal chart denotes a violation of the safety property denoted by it. To see how this happens, consider the example in Figure 2 consisting of two universal charts. When the user turns on the host, a live copy of both the charts are created. Subsequently the temporal order of events in one of these copies is bound to be violated during simulation since each chart has an opposite event ordering. In other words, simulation detects an inconsistency in the LSC specification.

3 Our Simulation Engine

In this section, we describe how the symbolic execution engine of a CLP system (such as CLP(R) [8]) can be used to simulate Live Sequence Charts. More importantly, we show that the natural support for symbolic variables in a (C)LP system can be exploited to provide a more efficient playout mechanism for LSCs.

Fig. 2. An inconsistent LSC description

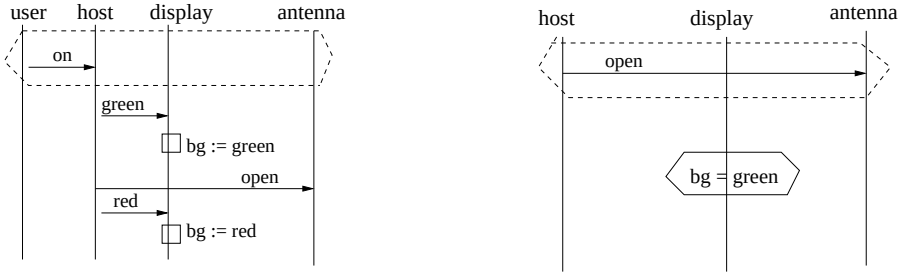


Fig. 3. An under-specified LSC description

3.1 Backtracking Search during Simulation

Prolog's strategy of backtracking through choices can be exploited to detect violations in various execution runs allowed by a LSC description. We showed an example of violation in Figure 2. In general, the visual formalism of LSCs also supports features which are typical of imperative programming languages: local variables for processes, and assignments and conditions for internal events.[1]

For example, consider the charts in Figure 3 which involves four processes: user, host, display and antenna. Suppose both the charts are universal, that is, if the pre-chart executes, the body chart must execute. In this example, the display process has a local variable called *bg*. Also, in the second chart, there is a "**hot condition**" on the value of *display.bg*; if the condition does not hold when it is evaluated this is considered a violation of the chart. In this paper, for the purposes of brevity, we have only described universal and existential charts, and typically assumed that all chart elements (conditions, locations etc.) inside a universal chart are also universal (*must* hold). The LSC specification language is more general; it allows existential conditions (which may hold) within a universal chart. Our LSC simulation engine can also support such specifications.

Let us now study the example of Figure 3. Initially, the user sends an *on* message to the host; once this message is received, the pre-chart is completed and a live copy of the first chart is activated. The simulation engine now executes a sequence of events as a response to the user provided event. Let us suppose that the following events have been executed so far.

[1] Thus the local variables are themselves not logical variables. Unification is used to implement symbolic variables appearing in messages.

$\langle user!host, on\rangle$, $\langle host?user, on\rangle$ (the pre-chart is met)
$\langle host!display, green\rangle$ (send *green* message)
$\langle display?host, green\rangle$ (receive *green* message)
$\langle display, bg := green\rangle$ (update *display.bg* to *green*)
$\langle host!antenna, open\rangle$ (send *open* message)

After these events are executed, the simulation engine has choices for the next event to execute. It could either execute $\langle antenna?host, open\rangle$ (the antenna receives *open* message) or it could execute $\langle host!display, red\rangle$ since both these events are enabled. Depending on which enabled event is selected and the subsequent choices made by the simulation engine it may or may not encounter a violation. In particular, if the antenna receives the *open* message *after* the display receives the *red* message and updates *display.bg* to *red*, then we will have a violation. This is allowed by the partial order of the charts in Figure 2. An execution trace which causes a violation is given by the following subsequent events.

$\langle host!display, red\rangle$, $\langle display?host, red\rangle$
$\langle display, bg := red\rangle$
$\langle antenna?host, open\rangle$ (a live copy of second chart is activated)
$\langle display, bg = green\rangle$ (evaluates to false – **Violation** of second chart)

When a violation is detected, our Prolog based simulation engine reports the violation and introduces a failure. Consequently the engine backtracks to the latest choice point to find a possible execution trace free from violation. For the above example of violation, the engine will backtrack and produce the following execution trace (which is free from violation). This means that the LSC description is possibly under-specified and the partial order of the charts need to be strengthened.

$\langle host!display, red\rangle$, $\langle display?host, red\rangle$
$\langle antenna?host, open\rangle$ (live copy of second chart activated now)
$\langle display, bg = green\rangle$ (evaluates to true)
$\langle display, bg := red\rangle$

On the other hand, consider the LSC description in Figure 2. After the *on* message is sent and received, a live copy of both the charts is activated. At this point the event $\langle host!display, green\rangle$ is enabled in the first chart and the event $\langle host!display, red\rangle$ is enabled in the second chart. Whichever event is chosen by the simulator, a violation of one of the charts must happen. In this case our Prolog based simulator returns a "no" answer (*i.e.* the absence of any execution trace free from violation).

3.2 Symbolic Variables via Prolog Unification

The charts in Figures 2 and 3 do not contain any variables. LSCs can support symbolic data variables in messages [10]. In our CLP based simulation engine, this is naturally achieved by the underlying Prolog unification.

When dealing with symbolic variables, a distinction needs to be made between the LSC specification and the simulation of LSC specifications (also called playout). Even though symbolic variables can appear in the LSC specification, it is possible to develop a simulation mechanism which avoids all symbolic variables. This can be achieved by requiring all symbolic variables to be bound during execution. Thus, the symbolic variables are used for ease of specification. On the other hand, if we use an engine with symbolic variables as the LSC execution engine this can lead to a more powerful operational semantics for LSCs. We have pursued this avenue.

Symbolic data variables correspond to variables appearing in (and transmitted via) messages. Typically, a symbolic data variable appearing in a chart will appear at least twice ; this allows for propagation of data values. For example, in Figure 1(b) the data variables A and D appear multiple times. If the underlying engine of these chart specifications cannot execute with non-ground variables then the first occurrence of each symbolic variable needs to be distinguished. This is the occurrence which binds the variable to a value. This can be problematic since no unique "first occurrence" of a variable may exist in a chart (the events of a chart are only guaranteed to satisfy a partial order). For example, consider the chart in Figure 4 with three processes p, q and r. The two send events $\langle p!q, m(X)\rangle$ and $\langle r!q, m1(X)\rangle$ are incomparable according to the partial order of the chart. If we chose one of them to be the first occurrence then the execution engine can demand that this first occurrence binds X; other occurrences of X simply propagate this binding. To solve this problem, [10] suggests fixing one of the events as the first based on the geometry of the chart.

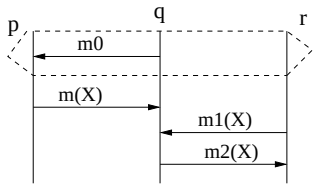


Fig. 4. Non-unique first occurrence of symbolic data variables

For any symbolic data variable, fixing any particular occurrence as the first occurrence constrains the partial order of the chart. In other words, it lets the simulation engine play out only certain behaviors allowed by the chart. A CLP based execution engine will naturally avoid this problem. In our engine, value passing of symbolic variables is supported by Prolog's unification. Given a LSC specification, its playout involves identifying enabled events, executing them and checking for violations of chart specifications. In Figure 4, both $\langle p!q, m(X)\rangle$ and $\langle r!q, m1(X)\rangle$ are initially enabled and our simulation engine can choose to execute either of them. More importantly, if it chooses to execute $\langle p!q, m(X)\rangle$, it does not require X to be bound at all. This constitutes a truly symbolic

simulation, where many execution traces which only differ in the values of the variable X are played out - all at one go.

4 Control Variables and Parameterized Systems

LSCs have been extended to using symbolic process instances [10]. A symbolic process in a chart represents a class (as in Object Oriented Programming) which at run-time produces several instances or objects. The identification number of any process instance is a symbolic control variable. Thus, introducing symbolic control variables to LSCs allows for specification of parameterized systems. An example of such systems is a class which at run-time always produces finitely many instances, but there is no a-priori bound on the number of instances.

As per the LSC language, we allow symbolic variable denoting instance id to be existential or universal. Existentially quantified control variables are handled like symbolic data variables; they may be bound to a particular instance via execution. Universally quantified symbolic control variables form a more compelling case for symbolic playout. *By allowing such symbolic variables during execution, we directly simulate parameterized systems (and not just specify them).*

Consider a process $p(X)$ where the instance id X is universally quantified. Since $p(X)$ in general represents unboundedly many instances, we need to disambiguate messages to and from $p(X)$. We use constraints on X for this purpose. Consider a message from process $p(X)$ to process $q(Y)$ where both X and Y are universally quantified. We require such messages to be of the form $c(X), M, c'(Y)$ where M is the message content and c, c' are constraints on X, Y respectively. The domain of the constraints c and c' depends on the type of X and Y. The variables representing instance ids are integers, and we will consider only unary inequality and equality constraints (*i.e.,* interval constraints).[2] Note that the message content M itself may contain symbolic data variables.

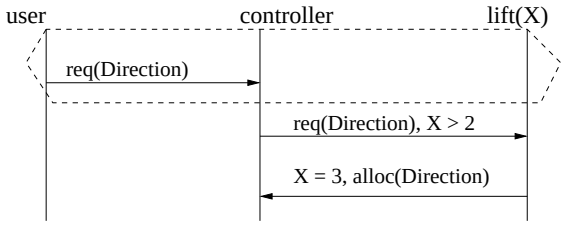


Fig. 5. A chart with universally quantified control variable (X in this case)

As an example, consider the symbolic chart shown in Figure 5. In this chart, $lift(X)$ represents a class of instances with X being a universally quantified

[2] Bigger classes of general constraints can be handled using the underlying CLP engine's constraint solver but even this unary restriction is already quite expressive.

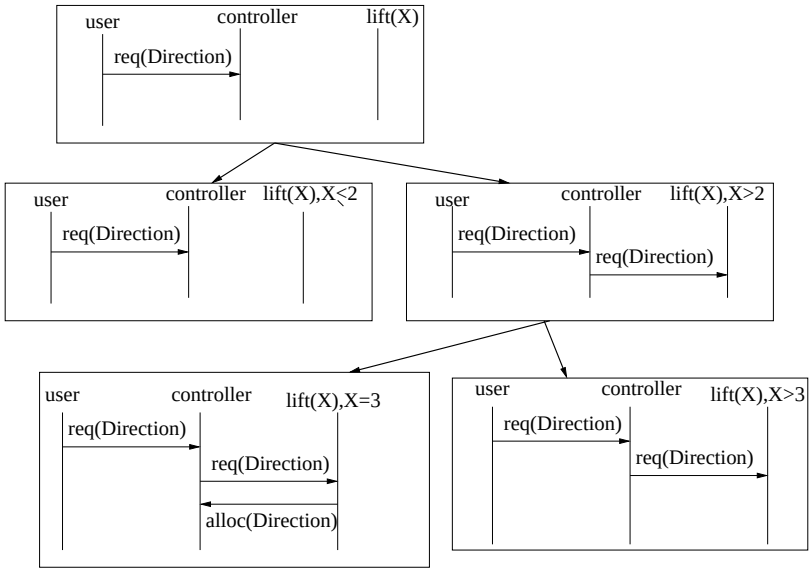


Fig. 6. Playout of Charts with Symbolic Control Variables

control variable. The pre-chart consists of the *user* process requesting movement in a specific direction (up or down). This is captured by the symbolic variable $Direction$. During execution, the user will give a concrete request, say $req(up)$. Hence $Direction$ will be unified to *up*. Now, the user's request is conveyed to the *controller* process which forwards this request to only some of the lifts. In this chart, it forwards the request to all lifts whose id is greater than 2. One of these lifts respond to the controller; which lift responds is captured by the constraint $X = 3$. Now, let us consider the simulation or playout of this chart. Existing works on LSC playout *do not allow* symbolic control variables in the execution. Consequently, whenever a message is sent to several instances of $lift(X)$ the simulation engine creates separate copies of the instances. This is clearly unnecessary. As a trivial example, consider the instances $lift(1)$ and $lift(2)$ in Figure 5. Their behaviors are *indistinguishable* from each other; hence it is meaningless to construct separate copies of such instances during simulation. Furthermore, for processes with unbounded number of instances, it is impossible to create separate copies of each instance.

We illustrate an alternate playout strategy applicable to parameterized systems by playing out the example of Figure 5. Initially there is only one copy of the lift process denoted as $lift(X)$; this represents *all* lifts. Since the pre-chart does not involve the $lift$ process, there is only one copy of the chart after the execution of the pre-chart. Now, when the controller forwards the message $req(Direction)$ it forwards it to only lifts with instance id > 2. Thus, the existing live copy is destroyed and two separate copies are created: one with $lift(X)$

s.t. $X \leq 2$, and the other with $lift(X)$ s.t. $X > 2$. In other words, the two separate copies of *lift(X)* are created in a demand-driven fashion, based on the chart execution. Finally, when the message *alloc(Direction)* is sent, the live copy corresponding to $lift(X)$ s.t. $X > 2$ is discarded to create two fresh live copies. The playout strategy sketched above is truly symbolic. Separate copies of process instances are not created unless required by the messages in the chart.

In general, our symbolic playout strategy works as follows. At any time in execution for a parameterized process $p(X)$, let the domain of X be divided into $k \geq 1$ mutually exclusive partitions so far. Each of the partitions is associated with a constraint on X, which is in fact an interval constraint; let the intervals corresponding to the k partitions be $I_1, \ldots, I_k$. Now, consider a message send from $p(X)$ (or a message receive into $p(X)$) with associated interval constraint $c(X)$. Let I^c be the interval corresponding to $c(X)$. Then, all live copies of the chart involving $p(X), X \in I_j$ such that $I_j \cap I^c \neq \phi$ need to be progressed with the new message send/receive. If $I_j \cap I^c \neq \phi$ and $I_j \subseteq I^c$ then the corresponding live copy is not discarded; it is only progressed with the message send/receive event. However, if $I_j \cap I^c \neq \phi$ and $I_j \not\subseteq I^c$, the live copy needs to be discarded. Instead two new live copies are created corresponding to the intervals $I_j \cap I^c$ and $I_j - (I_j \cap I^c)$. The first of these two live copies is progressed with the message send/receive, while the second one is not. Furthermore since the second live copy cannot be progressed with the message send/receive event, we make sure that for receive events into $p(X)$ the corresponding send event also does not appear.[3]

The reader should note that for any parameterized process $p(X)$, the domain of X (typically integers/natural numbers) is partitioned more and more as simulation progresses; this partitioning reaches a fixed point when all constraints in the chart are encountered. The partitions are always maintained internally by the simulation engine. The behavior of instances within a partition are indistinguishable and hence these instances are not maintained separately.

Finally, note that we could go one step further and represent the constraints over instance ids as variables. For example, in the chart of Figure 5, we could have specified the constraint for the message *req(Direction)* going to *lift(X)* as simply $C(X)$ instead of instantiating it to $X > 2$. Similarly, the constraint for the message *alloc(Direction)* coming from *lift(X)* could be specified as another predicate $C'(X)$. The predicates C and C' are then bound during simulation (this can be achieved by higher-order unification). Our CLP(R) based play engine currently does not support this specification feature.

5 Timing Constraints

LSCs are used as a full-fledged requirements specification language for reactive systems. Reactive systems often involve real-time response to external stimulus; thus a requirements specification of such systems may contain timing constraints. Consequently, the LSC specification language also allows timing constraints to

[3] In MSC/LSC descriptions, all messages sent are *eventually* received.

appear in charts. Primarily this involves the addition of a global variable $Time$ which is visible to all processes. Several other global variables T_i may appear in the chart which capture the time value at a certain snapshot of the chart's execution. For example consider the universal chart in Figure 7 obtained from [6]. This chart specifies that the light must turn on between 1 and 2 time units after the switch is turned on.

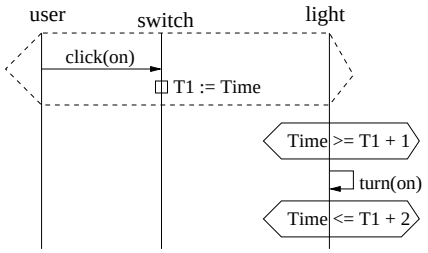


Fig. 7. A LSC with timing constraints

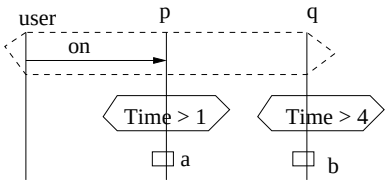
One way to achieve the simulation or playout of LSCs with timing constraints is as follows. We start with $Time = 0$ and wait for external stimulus. Once it arrives, we freeze time and compute a "maximal response" of the system as before (that is, a maximal sequence of events which get enabled after the external stimulus arrives). These events are assumed to take zero time. After the system response, we again allow time to progress. This playout strategy is adopted in the existing play engine by Harel and Marelly [6]. Note that in the presence of timing constraints, certain events in the chart may be stuck which would have otherwise been enabled. For example, in Figure 7, after the pre-chart is completed, the light has to wait for time to progress by at least one time unit.

The above playout strategy is not symbolic in the sense that all constraints on $Time$ are reduced to *tests* when they are evaluated. Furthermore, time is explicitly progressed (by the simulator's own clock or by user specified ticks) so that these tests can be evaluated to true. In contrast, our CLP based simulation engine does not explicitly force progress of time. Each occurrence of the $Time$ variable is captured as a different variable $Time_i$ in our simulation engine. Thus initially, we have $Time_0 = 0$; the next occurrence of $Time$ during the simulation is denoted as the variable $Time_1$ where $Time_1 \geq Time_0$. Since our variables are assign-once Prolog variables, therefore the flow of time in the imperative variable $Time$ is captured by a sequence of variables $Time_0, Time_1, Time_2 \ldots$. At any point during simulation suppose we have introduced variable $Time_0, \ldots, Time_i$. Any event/condition containing the global variable $Time$ introduces a new variable $Time_{i+1} \geq Time_j$ where $j \leq i$ is the greatest index such that the event/condition involving $Time_j$ "happens-before" the event/condition involving $Time_{i+1}$ in the partial order of the chart. Furthermore, we introduce a constraint from the event/condition involving $Time_{i+1}$ by replacing $Time$ with $Time_{i+1}$ in the

event/condition. Timing constraints appearing in LSCs thus translate to constraints on the $Time_i$ variables during simulation; these are constraints and not tests.

Let us revisit the universal LSC of Figure 7. Initially, we set $Time_0 = 0$. The user provides the stimulus $\langle user!switch, click(on)\rangle$. The simulator then executes $\langle switch?user, click(on)\rangle$. The internal action involving the update of $T1$ is now executed. Since this is the first occurrence of $Time$ after $Time_0$, we introduce the constraint $T1 = Time_1 \wedge Time_1 \geq Time_0$. Now, we encounter the "hot condition" `Time >= T1 + 1`. Recall that a hot condition is a condition whose falsehood leads to the violation of the chart; it is analogous to an assertion at a program point. Instead of explicity progressing time, we introduce another $Time_i$ variable which will be able to satisfy this condition and let the simulation proceed. Thus, we introduce the constraint $Time_2 \geq T1 + 1 \wedge Time_2 \geq Time_1$. We then execute the events $\langle light!light, turn(on)\rangle$ and $\langle light?light, turn(on)\rangle$. Finally, we need to evaluate the hot condition `Time <= T1 + 2`. The time at which this hot condition is evaluated refers to a potentially new time, since time might have increased since $Time_2$. So, we introduce a constraint $Time_3 \leq T1 + 2 \wedge Time_3 \geq Time_2$.

Note that when several hot conditions involving $Time$ are blocking simulation, we do not affix any order on the times at which they are evaluated. As a trivial example, consider the following universal chart.



In this chart we will accumulate the following constraints during simulation

$$Time_0 = 0 \wedge Time_1 \geq Time_0 \wedge Time_1 > 1 \wedge Time_2 \geq Time_0 \wedge Time_2 > 4$$

$Time_1$ and $Time_2$ correspond to the time of evaluation of the hot conditions in processes p and q. Note that $Time_1$ and $Time_2$ are incomparable. This is because the chart's partial order does not specify the ordering of these conditions.

Clearly, for LSCs with timing constraints, additional violations are possible during simulation if the timing constraints are inconsistent with the monotonically increasing flow of time. Our simulation engine will detect and report such violations. For example consider the universal chart of Figure 8(a). Initially, we start with $Time_0 = 0$ and execute the pre-chart of the LSC. A live copy of the chart is now created, and we need to satisfy the hot condition `Time >= 2`. This is achieved by adding the constraint $Time_1 \geq 2 \wedge Time_1 \geq Time_0$. The simulator then sends and receives the *green* message and tries to satisfy the hot condition `Time <= 1`. This again introduces a new variable $Time_2$ and constraints on this variable. At this point the constraint store becomes:

$$Time_0 = 0 \wedge Time_1 \geq Time_0 \wedge Time_1 \geq 2 \wedge Time_2 \geq Time_1 \wedge Time_2 \leq 1$$

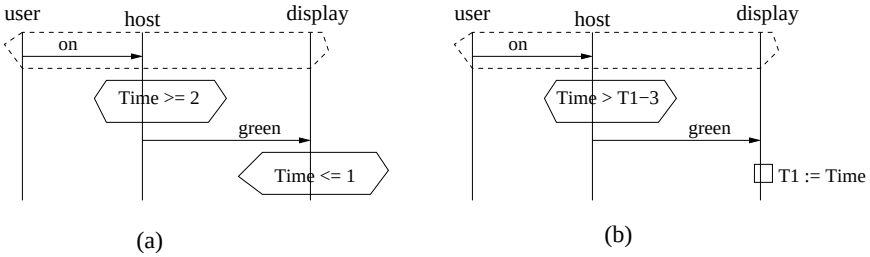


Fig. 8. (a) A LSC with inconsistent timing constraints (b) A LSC requiring symbolic representation of time

The constraint store is now inconsistent giving the timing violation.

Finally, we remark that the symbolic representation of time flow in our simulator (via constraints on $Time_i$ variables) allows us to simulate/playout more LSC descriptions. As a simple example consider the universal chart in Figure 8(b). This is a perfectly valid LSC description. It says that after the host is turned on by the user, the host should send a *green* message to the display; furthermore, the *green* message should be received within 3 time units of being sent. This example LSC description cannot be simulated in the play engine of [6] simply because the hot condition `Time > T1 - 3` refers to a variable `T1` which is uninstantiated. So, the play engine of [6] will get blocked waiting for `T1` to get instantiated. However, `T1` cannot get instantiated unless the *green* message is sent and received; thus the play engine of [6] will be deadlocked forever. On the other hand, our play engine will evaluate the hot condition `Time > T1-3` by adding the constraint $Time_1 > T1 - 3 \wedge Time_1 \geq Time_0$. This will allow the simulation to proceed and constraints on $T1$ will be accumulated subsequently.

6 Discussion

Message Sequence Charts (MSCs) are widely used as a requirements specification of inter-object interactions prior to software development; they constitute one of the behavioral diagram types in the Unified Modeling Language (UML) framework [3]. Live Sequence Charts (LSCs) are an important extension of MSCs; they can be used to describe complete behavioral requirements which are executable. In this paper, we have used a CLP engine to develop a symbolic simulator for Live Sequence Chart descriptions. Our simulator supports symbolic data variables in processes, symbolic control variables (to support many instances of a process) as well as timing constraints. The use of LP and CLP technology in the simulator makes it convenient to support these features. The natural support for backtracking in a (C)LP engine also makes it convenient to perform user-guided simulation of various allowed behaviors in a LSC description. For example, in the play engine of Harel et al. [5], a model checker is used to find a violation-free maximal sequence of events to execute in the universal charts (in response

to a user event); this sequence is called a superstep. However at any point in execution several supersteps may be possible. The play engine of [5] does not backtrack over supersteps and thus the simulation can miss certain valid execution sequences if there is a subsequent violation. Our simulator makes use of backtracking and thus can easily support backtracking (or not) over supersteps.

In a broader perspective, we note that the recent years have seen a spurt of research activity in developing/analyzing complete system specifications based on MSCs – [2, 4, 7, 11] to name a few. LSC is one such visual language with an execution engine. These executable specifications are useful since this allows simulation/analysis of requirements early in the software design cycle. To the best of our knowledge, our work is the first to enable truly symbolic execution of such MSC based requirements specifications. This is particularly useful since early in the design life cycle many variables in the design are left unbounded. Our simulator can directly execute designs with control and data variables of unbounded domain. In future, we plan to apply our ideas in symbolic simulation to other MSC-based models, such as High-level MSCs.

References

1. R. Alur, G.J. Holzmann, and D.A. Peled. An analyzer for message sequence charts. In *Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1055*, 1996.
2. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *International Conference on Concurrency Theory (CONCUR), LNCS 1664*, 1999.
3. G. Booch, I. Jacobsen, and J. Rumbaugh. *Unified Modeling Language for Object-oriented development*. Rational Software Corporation, 1996.
4. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
5. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Intl. Conf. on Formal Methods in Computer Aided Design (FM-CAD)*, 2002.
6. D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *IEEE/ACM Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2002.
7. J.G. Hendriksen, M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Message sequence graphs and finitely generated regular MSC languages. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2000.
8. J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3), 1992.
9. J. Klose and H. Wittke. An automata based interpretation of Live Sequence Charts. In *Intl. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2001.
10. R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Annual. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.
11. A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *IEEE Intl. Conf. on Application of Concurrency in System Design (ACSD)*, 2003.
12. Z.120. Message Sequence Charts (MSC'96), 1996.