Symbolic Simulation of Behavioral Requirements

Abhik Roychoudhury Roland H.C. Yap Shishir C. Choudhary
School of Computing
National University of Singapore
{abhik,ryap,shishirc}@comp.nus.edu.sg

Abstract

Message Sequence Charts (MSC) have traditionally been used as a weak form of behavioral requirements 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 propose the use of Constraint Logic Programming (CLP) for symbolic execution of requirements described as Live Sequence Charts. The utility of CLP stems from its ability to execute in the presence of symbolic variables. 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 may 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 to also simulate charts with non-trivial timing constraints. Current works on MSC/LSCs use symbolic variables mainly for ease of specification; they are instantiated to concrete values during simulation. Thus, our work advances the state-of-the-art in simulation and testing of MSC based behavioral requirements.

1 Introduction

Message Sequence Charts (MSCs) [17] 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).

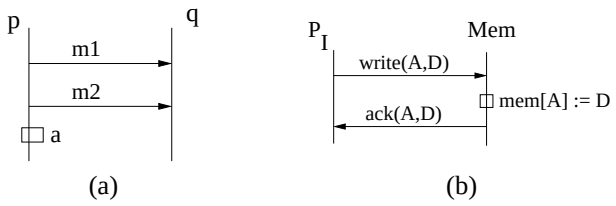

Figure 1. (a) A simple MSC, and (b) MSC with symbolic variables

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 limitation of MSCs is that they only denote a scenario which *may* occur. In other words, an MSC only captures an existential requirement: some execution trace (behavior) of the system contains a linearization of the events in the MSC. They do not capture universal requirements, that is, tenporal 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*.

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 temporal 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. Furthermore, LSC based behavioral specifications are *executable*. A full description of the LSC language and the accompanying execution engine (called the play engine) appears in the recent book [8].

Thus, the LSCs serve as a complete requirements specification of 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*. These unintended behaviors are called "**violations**" since they denote an inconsistency in the overall requirements specification (*i.e.* one requirement "violates" another).

Executing behavioral requirements prior to system development however needs to consider the fact that the requirements are often at a higher level than the system implementation. Concretely, this may boil down to the behavioral requirements not specifying: (a) data values exchanged between objects or (b) number of objects of a class.

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 of the processor in question). In fact, if we want the requirements to represent the details captured by these symbolic variables, this leads to an arbitrarily large (potentially unbounded) number of scenarios. Furthermore, these scenarios are structurally "similar" and can be specified together. To avoid this problem, the authors of the LSC formalism extend the LSC specification language [13]. Each vertical line in a chart now denotes a group of objects rather than a concrete object. Furthermore, data exchange between objects can be depicted by variables (rather than concrete values), thereby enabling value-passing. This allows us to specify several similar scenarios using a single chart, as shown in [13]. However, [13] uses the formal variables mostly for concise specification; they are instantiated to concrete values (denoting concrete data items or concrete objects) during simulation. This does *not* allow symbolic simulation, that is, executing several similar scenarios together. In symbolic simulation, the variables are not forcibly instantiated to concrete values. Instead, their domains get split based on the constraints encountered during LSC execution.

In this paper, we propose the use of Constraint Logic Programming (CLP) for symbolic simulation of LSC based behavioral requirements. In fact, we have built a prototype simulator for LSC specifications on top of the widely used CLP(R) system [11]. A constraint logic programming system is basically a logic programming engine (search based evaluation with backtracking) with enhanced constraint solving capabilities [10]. Equality constraint solving via unification (as in logic programming) is augmented with general purpose constraint solving (such as inequalities over integers/reals). We leverage on the constraint processing capabilities of a CLP engine in several ways. Unification in the CLP engine captures value passing, and is used as a solver for equality constraints. More general constraints are used to symbolically represent values of data variables, groups of objects, and even timing.

The logic programming based search in our simulator is also crucial for the following reason. Note that the search strategy of LSC simulation involves resolving nondeterminism, since several events may be enabled at a given point during simulation. Thus, a LSC simulator may have to make "choices" among enabled events. All these choices need to be *searched* to detect "violations" (*i.e.* inconsistencies) in the specification. A logic programming engine provides support for such search using backtracking (this is not fully supported in the existing LSC play engine [6], refer Section 3.1).

Contributions The contributions of this paper can be summarized as follows.

- We develop a methodology for symbolic simulation of Live Sequence Charts, a visual language for describing behavioral requirements. We exploit three different features of LSCs for symbolic execution. First, data variables (such as D in Figure 1(b)) can be unbound during simulation. Secondly, control variables (such as the instance I in Figure 1(b)) can also remain uninstantiated during simulation. This allows us to directly simulate a process with unbounded number of instances. Thirdly, time is maintained as a collection of constraints instead of a fixed value (for the simulation of charts with timing constraints). By keeping the variables symbolic, we achieve the simulation of many different concrete runs in a single run (thereby speeding up debugging).
- We do not realize concrete objects which are behaviorally indistinguishable. This substantially cuts down memory requirements during simulation. This approach contrasts with the work of Harel et. al. [13] which blows up a class of objects in the specification to finitely many concrete objects during execution.
- The search strategy of our simulator is derived from a logic programming based search engine; hence it can

backtrack over choices. This allows us to search for violations (*i.e.* inconsistencies) in behavioral requirements which allow non-deterministic behavior. Since requirements are specified at a high-level it is common to have non-determinism in such specifications (even if the system implementation is deterministic).

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

Live Sequence Charts (LSCs) [4] is a powerful visual formalism which serves as an enriched requirements specification 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) [1, 17] are written in a visual notation as shown in Figure 1(a). Each vertical line denotes a process which executes events. 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 (a) the total order of the events in each process (time flows from top to bottom in each process) and (b) 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 as well.

Consider the chart in Figure 1(a). 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. In the terminology of Message Sequence Charts, a LSC is a *synchronous* concatenation of its pre-chart and body-chart [2].

For example, consider the chart in Figure 3. 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.

In the LSC language, 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.

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 [12]. Synthesis of finite state machines from LSC specifications is discussed in [5].

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). This maximal response to the user-provided event is called a **super-step**. 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 in the body chart is violated. Violation of a universal chart denotes a violation of the temporal property denoted by it. Detection of violations may involve backtracking in the LSC simulation. We now describe how our CLP based simulator and show how it naturally performs this backtracking.

3 Our Simulation Engine

In this section, we describe how the symbolic execution engine of a CLP system (such as CLP(R) [11]) can be used to simulate Live Sequence Charts.

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.

For example, consider the charts in Figure 2 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. One point needs to be clarified at this point. 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 also allows existential conditions (which may hold) within a universal chart. Furthermore, LSCs can support features which are typical of imperative programming languages: local variables, assignments and conditions within a process. Our LSC simulation engine can also support such specifications.

Let us now study the example of Figure 2. 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.

$\langle user!host, on\rangle$, $\langle host?user, on\rangle$, (the pre-chart is met)
$\langle host!display, green\rangle$, $\langle display?host, green\rangle$,
$\langle display, bg := green\rangle$, (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. The following subsequent events will produce a violation.

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

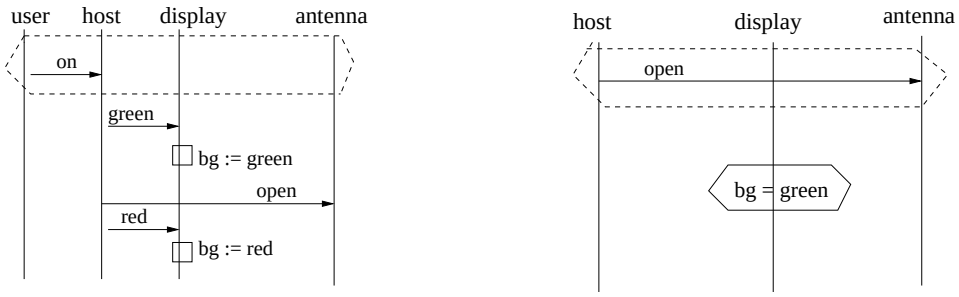

Figure 2. An LSC which produces a violation

The backtracking illustrated in the above example occurs in the computation of a violation-free super-step (a sequence of events in response to the event $\langle user!host, on\rangle$). The LSC play engine of Harel et. al. [6] computes such a violation-free super-step by using a model checking based search procedure. However, their play-engine cannot be used to backtrack over super-steps. Thus the simulation can miss certain valid execution sequences if no violation-free super-step is found due to the choices made in previous super-steps. Our backtracking based search can backtrack over super-steps; it can also backtrack over enabled events in computing a violation free super-step. In other words, our search procedure is more programmable. For example, we can program our simulator to use model checking for computing a single super-step as in [6] while backtracking over super-steps on demand. Efficient model checkers based on model checking already exist, such as the XMC toolset [14].

3.2 Symbolic Data Variables

When dealing with symbolic variables, a distinction needs to be made between the LSC specification and the simulation of LSC specifications. 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 simulation mechanism 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 uninstantiated variables, then the first occurrence of each symbolic variable needs to be distinguished. This is the occurrence which binds the variable

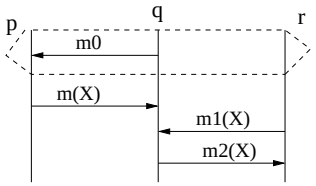

Figure 3. Non-unique first occurrence of symbolic data variables

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 3 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, [13] suggests fixing one of the events as the first based on the geometry of the chart.

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 CLP's unification. Given a LSC specification, its playout involves identifying enabled events, executing them and checking for violations of chart specifications. In Figure 3, 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 charts (which only differ in the values of the variable X) are simulated together.

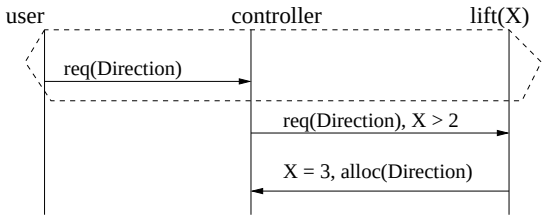

Figure 4. A chart with universally quantified control variable (X in this case)

4 Handling Symbolic Control Variables

LSCs have been extended to use symbolic process instances [13]. 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 a variable denoting the instance number 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 number 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 numbers are integers, and we will consider only unary inequality and equality constraints (*i.e.,* interval constraints).[1]

As an example, consider the universal chart shown in Figure 4. In this chart, $lift(X)$ represents a class of instances with X being a universally quantified 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 instance number is greater than 2. One of these lifts respond to the controller; which lift responds is captured by the constraint $X = 3$.

We illustrate an alternate simulation strategy applicable to parameterized systems by simulating the example of Figure 4. 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 number > 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.

Formal description In general, our simulation 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 $I_1, \ldots, I_k$ where for all $1 \leq j \leq k$ we have $I_j = [l_j, u_j]$ (the lower and upper bounds of the interval). Now, consider a message send from $p(X)$ or a message receive into $p(X)$, with associated interval constraint $c(X)$. Let $I^c = [l^c, u^c]$ be the interval corresponding to $c(X)$. Any live copy I_j (where $1 \leq j \leq k$) satisfies one of the following four cases.

- Case 1 : If $l_j \leq u_j < l^c \leq u^c$ or $l^c \leq u^c < l_j \leq u_j$ (I_j and I^c are disjoint intervals), then the copy for I_j is not progressed with the new message send/receive.
- Case 2: If $l^c \leq l_j \leq u_j \leq u^c$ (I^c contains I_j) then the copy for I_j is progressed with the new message send/receive.
- Case 3: If $l_j \leq l^c \leq u^c \leq u_j$ (I_j contains I^c) we discard the live copy for I_j and replicate it to create three live copies $[l_j, l^c-1]$, $[l^c, u^c]$ and $[u^c+1, u_j]$. The

[1] 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.

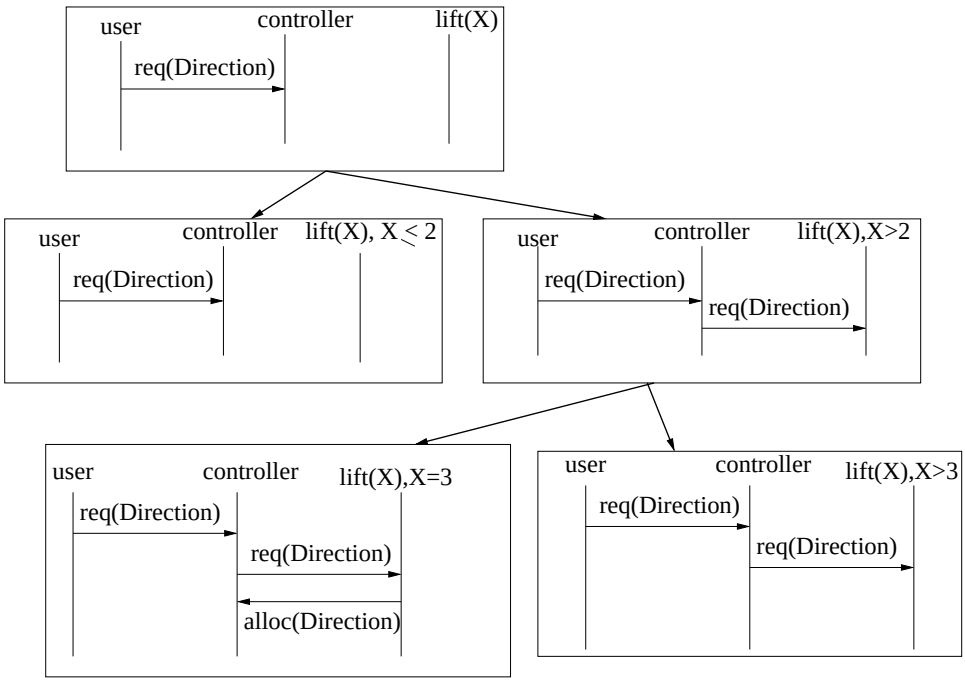

Figure 5. Simulation of a Chart with Symbolic Control Variables

live copy $[l^c, u^c]$ is progressed with the new message send/receive, while the other two are not progressed.

- Case 4: Otherwise, either $l_j < l^c \leq u_j < u^c$ or $l^c < l_j \leq u^c < u_j$ (but not both). We discard the live copy corresponding to I_j and replicate it to create two new live copies: (a) for the portion of I_j common to I^c, which is progressed with the message send/receive and (b) the portion of I_j not common to I^c, which is not progressed. Thus, if $l_j < l^c \leq u_j < u^c$, we create live copies for $[l^c, u_j]$ (common to I^c) and $[l_j, l^c - 1]$. If $l^c < l_j \leq u^c < u_j$ we create live copies for $[l_j, u^c]$ (common to I^c) and $[u^c + 1, u_j]$.

In case 3, the behaviors of the live copies for $[l_j, l^c - 1]$ and $[u^c + 1, u_j]$ are identical; we could maintain a single live copy for them. Indeed we do so by maintaining a set of intervals for each live copy, instead of a single interval. This is a straightforward extension of the above simulation strategy and we omit the details.

The reader should note that for any parameterized process class $p(X)$, the domain of X (typically integers/natural numbers) is partitioned more and more into subclasses as simulation progresses. This partitioning reaches a fixed point when all constraints in the chart are encountered. The partitions (or subclasses) are maintained internally by the simulation engine.

Key idea in our approach Our simulation strategy can handle LSC descriptions containing parameterized process classes. These process classes are split into subclasses during simulation based on *behaviors*. Each subclass is annotated with a constraint, which in general can represent unboundedly many instances. The key idea of our approach is to *not* maintain all of these concrete objects explicitly (as is done in the play engine of Harel and Marelly [13]). Instead we maintain them symbolically via constraints. In fact, since the behavior of the instances within a subclass are indistinguishable, there is no need to maintain them separately. As a trivial example, consider the instances $lift(1)$ and $lift(2)$ in Figure 4. Their behaviors are *indistinguishable* from each other; hence it is not necessary to construct separate copies of such instances during simulation.

A possible extension Finally, note that we could go one step further and represent the constraints over instance numbers as variables. For example, in the chart of Figure 4, 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. Our CLP based play engine currently *does not support* this feature. An extension along these lines will require the LSC descriptions to

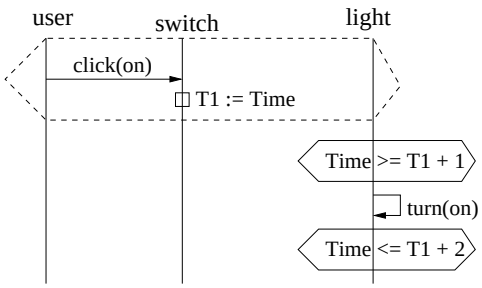

Figure 6. A LSC with timing constraints

be converted into higher-order logic programs (and an engine for evaluating such higher-order programs).

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 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 6 obtained from [7]. This chart specifies that the light must turn on between 1 and 2 time units after the switch is turned on. Note that even though $T1 := Time$ is an internal computation it manipulates global variables.

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 [7]. 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 6, after the prechart 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. We now describe our approach for simulating LSCs with timing constraints.

Formal description We take a different approach in our CLP based simulation engine; we do not force progress of time. Instead, each occurrence of the $Time$ variable (encountered during simulation) 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 $Time_0, Time_1, Time_2 \ldots$.

Suppose that we have introduced timing variables $Time_0, \ldots, Time_i$ at any point during simulation. 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. We also 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 translate to constraints (not tests) on the $Time_i$ variables during simulation.

Example Let us revisit the universal LSC of Figure 6. 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 explicitly 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 universal chart in Figure 7. 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 $Time_1$

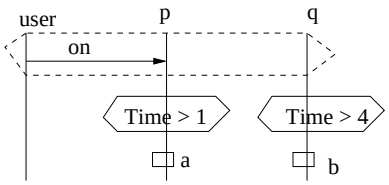

Figure 7. Choices in evaluating timing constraints

and $Time_2$ are incomparable. This is because the chart's partial order does not specify any ordering on the evaluation 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:

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

The constraint store is now inconsistent (since it implies $Time_2 \geq 2 \wedge Time_2 \leq 1$), giving a violation.

Additional power of our approach The symbolic representation of time flow in our simulator allows us to simulate/playout more LSC descriptions. As a simple example, consider the universal chart in Figure 8(b). 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 [7] simply because the hot condition `Time > T1 - 3` refers to a variable `T1` which is uninstantiated. So, the play engine of [7] 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 [7] 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 Summary and Future Work

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.

Summary and Experimental Results In this paper, we have used a Constraint Logic Programming (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 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 automated simulation of various allowed behaviors in a LSC description. Currently, we have used our tool for automated simulation of small LSC examples (*including all the examples given in this paper*). For all the small examples shown in this paper, detection and simulation of a violation free path takes less than 0.05 second. The timings are obtained by running the CLP(R) system on top of Linux in a Pentium IV 1.3 Ghz machine with 1 GB of memory.

Improvements to our Tool The major difficulty in running large LSC descriptions in our simulator is that we do not have a semi-automated mechanism for constructing the input LSC. Currently, our simulator takes in textual input (describing a LSC), and we lack a full-fledged Graphical User Interface (GUI) to input LSC descriptions into our simulator. In contrast, the play engine of Harel et. al employs a sophisticated play-in approach [8] which allows the user to input LSCs without even drawing them. In future we will work on integrating the play-in approach as a front end to our simulator. This will allow us to leverage the power of our symbolic simulation (or playout) on large LSC examples (which are played in).

Overall Perspective 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, 9, 15, 16] to name a few. LSC is one such visual language with an execution engine. Such executable specifications are useful since they allow 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

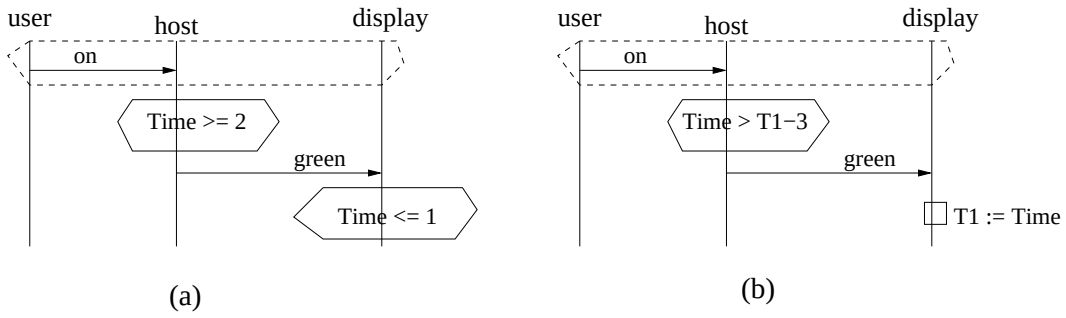

Figure 8. (a) A LSC with inconsistent timing constraints (b) A LSC requiring symbolic representation of time

early in the design cycle many variables in the design are left uninstantiated. 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 executable specification languages based on MSCs (such as the CTP modeling language [15]).

References

[1] R. Alur, G. Holzmann, and D. 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 and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Intl. Jnl. on Foundations of Computer Science (IJFCS)*, 13, 2002.

[6] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart playout of behavioral requirements. In *Intl. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, 2002.

[7] 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.

[8] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[9] J. Hendriksen, M. Mukund, K. Kumar, and P. Thiagarajan. Message sequence graphs and finitely generated regular MSC languages. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2000.

[10] J. Jaffar and M. Maher. *Constraint Logic Programming: a Survey*, volume 5 of *Handbook of Artificial Intelligence and Logic Programming*, chapter 9. Oxford University Press, 1995.

[11] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) language and system. *ACM Trans. Prog. Lang. Syst.*, 14(3), 1992.

[12] 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.

[13] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Intl. Conf. on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2002.

[14] C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, et al. XMC : A logic programming based verification toolset. In *International Conference on Computer Aided Verification (CAV), LNCS 1855*, 2000.

[15] A. Roychoudhury and P. Thiagarajan. Communicating transaction processes. In *IEEE Intl. Conf. on Appl. of Concurrency in System Design (ACSD)*, 2003.

[16] A. Roychoudhury and P. Thiagarajan. An executable specification language based on message sequence charts. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*. Springer Verlag, LNCS 2757, 2003.

[17] Z.120. Message Sequence Charts (MSC'96), 1996.