Symbolic Execution of Behavioral Requirements

Tao Wang, Abhik Roychoudhury, Roland H.C. Yap, and S.C. Choudhary

School of Computing, National University of Singapore, Singapore 117543.
{wangtao,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 uninstantiated 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 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 data/control 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 checking of MSC based software requirements.

1 Introduction

Message Sequence Charts (MSCs) [16] have traditionally played an important role in software development. MSCs describe scenarios of system behaviors. 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 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

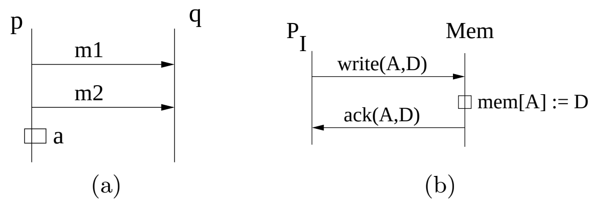

Fig. 1. (a) A simple MSC, and (b) MSC with variables

the MSC. They do not capture universal requirements, that is, temporal 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) [5]. 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 satisfied in any execution trace of the system, then the body chart must be executed.*

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. 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.* The advantage of simulating LSC specifications via a play engine is 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). A full description of the LSC language and the accompanying execution engine (called the play engine) appears in the recent book [10].

Executing behavioral requirements prior to system development however needs to consider the fact that the requirements are often at a higher level than the implementation. Concretely, this may boil down to the behavioral requirements not specifying: (a) data values exchanged between objects, or (b) the number of objects of a process class. Figure 1(b) shows a chart with 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 variables appearing in events; I is a variable representing a process instance (the instance of the processor in question). In fact, if we want the requirements to contain the values of these 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 exchanged 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 during simulation. This does *not* allow symbolic execution, that is, executing several similar scenarios together.

In this paper, we propose the use of Constraint Logic Programming (CLP) for symbolic simulation of LSC based behavioral requirements. 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 (such as inequalities) are used to symbolically represent values of data variables, groups of objects, and timing. Also note that the search strategy of LSC simulation involves resolving non-determinism, since several events may be enabled at a given point during simulation. Thus, a LSC simulator may have to make “choices” among enabled events. These choices need to be *searched* to detect “violations” (*i.e.* inconsistencies) in the specification. A logic programming engine provides natural support for such search via backtracking (this is not fully supported in the existing LSC play engine [8]).

We note that CLP has been shown to be useful for symbolic representation of sets of system states [6]. Use of CLP for symbolic simulation of event based systems has been investigated in [14]. There have also been recent uses of CLP for animation/simulation of formal specifications (*e.g.* [4]).

Contributions We now summarize the main contributions of the paper.

- We develop a methodology and toolkit for symbolic execution 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 remain partially instantiated. Secondly, control variables (such as the instance I in Figure 1(b)) can also remain partially instantiated during simulation. This allows us to directly simulate a process with unboundedly many 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.
- We do not realize concrete objects which are behaviorally indistinguishable. This approach contrasts with the work of [13] which blows up a class of objects in the specification to finitely many concrete objects during execution.
- The search strategy of our tool is derived from a logic programming based search engine; hence it can naturally backtrack over choices. This allows us to search for violations (*i.e.* inconsistencies) in behavioral requirements. Since requirements are specified at a high-level, they are likely to have non-determinism even if the implementation is deterministic.

Our simulation engine for LSCs is implemented on top of the ECL^iPS^e constraint logic programming system [7].

Section Organization The rest of the paper is organized as follows. Section 2 provides an overview of LSCs. Section 3 describes how a CLP engine is suitable for executing LSCs with data variables (of potentially unbounded domain). 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. Section 6 describes the implementation of our engine and experimental results. Finally, section 7 concludes the paper with discussion and future work.

2 Live Sequence Charts

Live Sequence Charts (LSCs) [5] 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* [10]. 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, 16] 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 variables. 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 $\langle p!q, m1\rangle \leq \langle p!q, m2\rangle \leq \langle p, a\rangle$ where $e1 \leq e2$ denotes that event $e1$ "happens-before" event $e2$. Similarly for process q we have $\langle q?p, m1\rangle \leq \langle q?p, m2\rangle$ For the messages we have $\langle p!q, m1\rangle \leq \langle q?p, m1\rangle$ and $\langle p!q, m2\rangle \leq \langle q?p, m2\rangle$. 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 $\langle p!q, m2\rangle \leq \langle q?p, m1\rangle$ or $\langle q?p, m1\rangle \leq \langle p!q, m2\rangle$.

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]. Following the notational convention of LSCs, we always show the pre-chart inside a *dashed hexagon*. The body chart of a universal chart is shown inside a rectangular box. *All examples shown in this paper are universal charts.* Now let us consider the chart in Figure 3. 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. A system model M satisfies an existential chart $Pre \circ Body$ 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*. Thus, 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 [5]. Automata based semantics of the language appear in [12].

2.3 The Play Engine

A LSC based system description can serve as a 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*** (see [10], Chapter 5). 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, or any condition which must be satisfied is evaluated to be false.

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. In other words, simulation detects an inconsistency in the temporal properties denoted by the two universal charts of Figure 2. This is called a ***violation*** in the LSC literature [10]. In the rest of the paper, we discuss how to support symbolic execution of LSCs with variables and/or constraints for the purposes of simulation (finding one violation-free behavior).

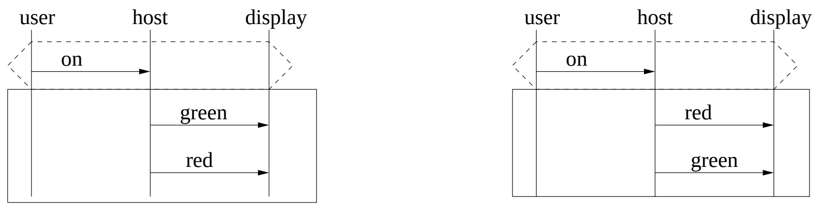

Fig. 2. A LSC specification with "violations"

3 Data Variables

In this section, we describe how the symbolic execution engine of a Constraint Logic Programming (CLP) system (such as ECL^iPS^e [7]) can be used to execute Live Sequence Charts. We start with the handling of data variables (*i.e.* variables appearing in chart events). When dealing with such variables, a distinction needs to be made between the LSC specification and the execution of LSC specifications. Even though variables can appear in the LSC specification, it is possible to develop an execution mechanism which avoids all variables. This can be achieved by requiring all variables to be bound during execution. Thus, the variables are used for ease of specification. On the other hand, if we use a CLP engine as the LSC execution engine this can lead to a symbolic execution mechanism for LSCs. We have pursued this avenue.

Data variables correspond to variables appearing in (and transmitted via) messages. Typically, a 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 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 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.

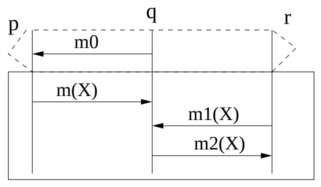

Fig. 3. Non-unique first occurrence of a data variable

For any 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 between variables is supported by CLP's unification. Given a LSC specification, its simulation 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 value of variable X) are simulated together.

4 Control Variables

LSCs have been extended to use symbolic process instances [13]. A symbolic process in a chart represents a parameterized process class which at run-time produces several instances or objects. In other words, these classes always produce finitely many instances at run-time; but there is no a-priori bound on the number of instances. The identification number (or the parameter) of the instances is a control variable. As per the LSC language, we allow such control variables (denoting the instance number) to be existential or universal. Existentially quantified control variables are handled like data variables; they may be bound to a particular instance via execution. Universally quantified control variables however represent many possible process instances, and need to be handled differently.

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]

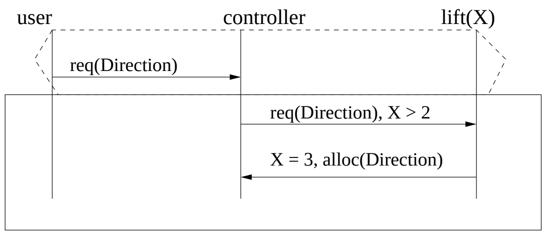

Fig. 4. A chart with universally quantified control variable (X in this case)

An Example We 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 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 responds to the controller; which lift responds is captured by the constraint $X = 3$. In Figure 5, we illustrate a symbolic simulation strategy applicable to parameterized process classes by simulating the example of Figure 4. To notationally distinguish the progress in simulation from the specification of a body chart, we use bold boxes in 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 number greater than 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

[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.

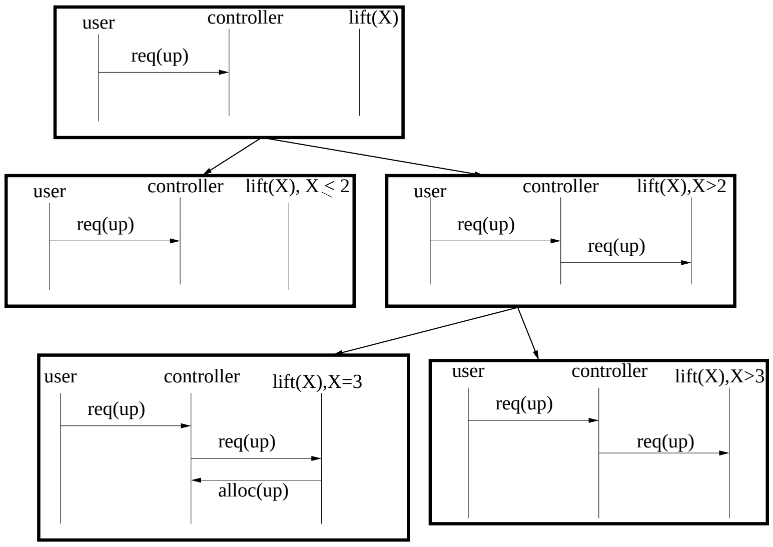

Fig. 5. Simulation of a Chart with Control and Data Variables

simulation strategy sketched above is truly symbolic. Separate copies of process instances are not created unless required by the messages in the chart.

Formal description of our approach 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 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 in our implementation 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.

Key idea in our approach Our simulation strategy can handle LSC descriptions containing parameterized process classes. The key idea of our approach is to *not* maintain all of the concrete instances of a process class explicitly (as is done in the play engine of Harel and Marelly [10, 13]). In other words, their play engine [10] uses universal control variables only for concise specification; these variables are instantiated into all possible values during simulation. Instead, we maintain the values of a control variable symbolically via constraints. In particular, a process class gets split into subclasses during simulation based on *behaviors*. Each subclass is annotated with a constraint, which in general can represent unboundedly many instances (corresponding to unboundedly many control variable values). Instances which are behaviorally indistinguishable are grouped into a subclass. As a simple 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 maintain such instances separately during simulation.

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 $Time$ (representing a global clock) 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(a) obtained from [9, 10]. 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.

The existing play engine of Harel and Marelly [9] simulates LSCs with timing constraints as follows. The simulator starts with $Time = 0$ and waits for external stimulus. Once the stimulus arrives, the simulator freezes time and computes 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

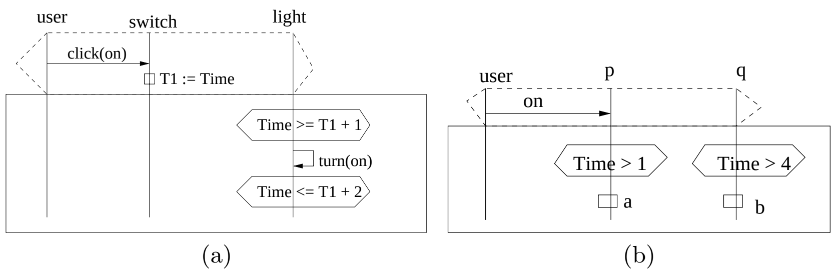

Fig. 6. (a) LSC with timing constraints (b) Choice in evaluating timing constraints

assumed to take zero time. After the system response, the simulator again allows time to progress. 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(a), after the pre-chart is completed, the light has to wait for time to progress by at least one time unit.

The above simulation 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.

Our Approach 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 variables, therefore the flow of time in $Time$ is captured by a sequence of variables

$$Time_0, Time_1, Time_2 \dots$$

Suppose that we have introduced timing variables $Time_0, \dots, Time_i$ at any point during simulation. Any event/condition containing the global variable $Time$ introduces a new variable $Time_{i+1}$. We introduce the constraints

- $Time_{i+1} \geq Time_j$ where $j \leq i$ is any 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. In practice, we only introduce a transitive reduction of such constraints (*e.g.* while introducing the variable $Time_2$, if we introduce $Time_2 \geq Time_1$ and we already have $Time_1 \geq Time_0$, the constraint $Time_2 \geq Time_0$ is redundant).
- 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.

Examples Let us revisit the universal LSC of Figure 6(a). 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`. In LSC terminology [5], 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 6(b). 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 any ordering on the evaluation of these conditions.

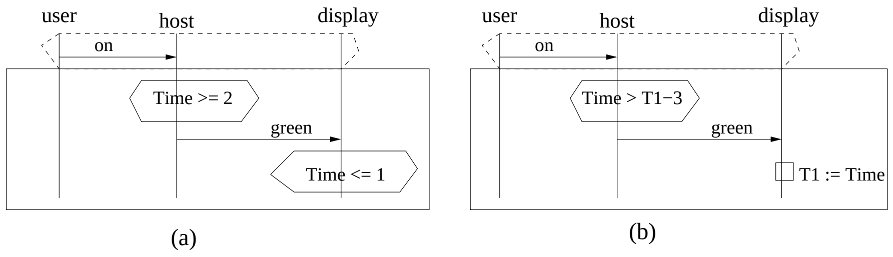

Fig. 7. (a) A LSC with inconsistent timing constraints (b) A LSC requiring symbolic representation of time

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 7(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 more LSC descriptions. Let us consider the universal chart in Figure 7(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 [9] simply because the hot condition `Time > T1 - 3` refers to a variable `T1` which is uninstantiated. So, the play engine of [9] 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 [9] 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 Implementation

We have used the ECL^iPS^e constraint logic programming system to develop a symbolic simulation engine for LSC descriptions. Our engine supports data variables in processes, control variables (to support many instances of a process) as well as timing constraints. The natural support for backtracking in a ECL^iPS^e engine makes it convenient to perform automated simulation of various allowed behaviors in a LSC description. Whenever a violation is detected, the simulator reports the trace of the illegal path, and backtracks to find a violation free path. Our current implementation supports simulation of both existential and universal charts. Existential charts are simulated in a manner somewhat similar to pre-charts. In other words, they are monitored and progressed in each super-step but their violation is not reported.

Examples simulated using our tool We have used our tool for automated simulation of some LSC examples (*including the examples given in this paper and in [5, 13]*). Three of them are non-trivial and are described below. In [5], the authors presented LSC description of an automated rail-car system with several cars operating on a cyclic path with several terminals. We simulate the LSC for the rail-car system using our ECL^iPS^e implementation. This LSC involves 7 classes, 19 messages, 11 conditions, 1 assignment and other control structures like subchart and if-then-else structures. We also simulate portions of a netphone example which has appeared in literature. In particular, [13] describes a model of a telephone network system with LSCs. We have simulated two use cases of this system; these are the only two use cases whose full description is publicly available [10, 13]. One use case specifies the protocol for initial set-up of the identification number of various phones in a telephone network. It contains 4 LSCs, with 4 classes, 7 messages, 3 conditions and other operations. The other user case describes part of the conversation establishment protocol. It consists of 3 LSCs, with 5 classes, 11 messages, 6 conditions and other operations.

Timings and Web-site For all our examples, detection and simulation of one violation free path takes less than 0.1 second. The timings are obtained by running ECL^iPS^e on top of SUN OS 5.8 in a SunFire 4800 machine with 750 MHz Ultra Sparc III CPU (only one processor was used in the experiments).

Existing works on LSCs/play engine present the rail-car and netphone examples, but do not report timings for their simulation. Hence, a performance comparison is problematic. However more important than a performance comparison is to check whether our simulator provides tolerable levels of efficiency to the user. Our timing of less than 0.1 second for simulating non-trivial LSC examples (like the rail-car example of [5]) seems to indicate that our simulator is reasonably efficient. In other words, the symbolic execution mechanism does not compromise efficiency in a significant way. Our prototype simulator toolkit is available from the following web-site (along with description of our examples) `http://www.comp.nus.edu.sg/~wangtao/symbolic_simulator_engine.htm`

Possible Improvements So far, our focus has been in building a constraint based engine for symbolic simulation of LSCs. We have concentrated less on the user-interface issues, such as how a LSC is provided as input to the simulator. Currently, our simulator takes in textual input (describing a LSC), and we lack a full-fledged Graphical User Interface (GUI) to input LSC descriptions. In contrast, the play engine of Harel et. al employs a sophisticated play-in approach [10] which allows the user to input LSCs without fully drawing them. In future we will work on integrating the play-in approach as a front end to our simulator.

7 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]. We note that the recent years have seen a spurt of research activity in developing/analyzing complete system specifications based on MSCs – [2, 5, 11, 15] to name a few. LSC is one such visual language with an execution engine. LSC based executable specifications are useful since they allow simulation/analysis of requirements early in the software design cycle. Our work in this paper is geared towards using CLP technology for symbolic execution of such behavioral requirements. In future, we plan to apply our ideas for simulating descriptions written in other executable specification languages based on MSCs (*e.g.* the Communicating Transaction Processes (CTP) modeling language [15]).

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. M. Butler and M. Leuschel. ProB: A model checker for B. In *Formal Methods in Europe (FME), LNCS 2805*, 2003.
5. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
6. G. Delzanno and A. Podelski. Model checking in CLP. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1999.
7. ECL^iPS^e. The ECL^iPS^e Constraint Logic Programming System, 2003. Available from `http://www-icparc.doc.ic.ac.uk/eclipse/`.
8. 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.
9. 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.
10. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer-Verlag, 2003.
11. 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.
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. S. Narain and R. Chadha. Symbolic discrete event simulation. In *Discrete Event Systems, Manufacturing Systems and Communication Networks*, 1994.
15. A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *IEEE Intl. Conf. on Appl. of Concurrency in System Design (ACSD)*, 2003.
16. Z.120. Message Sequence Charts (MSC'96), 1996.