Model Checking Live Sequence Charts

Jun Sun and Jin Song Dong
School of Computing
National University of Singapore
{sunj, dongjs}@comp.nus.edu.sg

Abstract

Live Sequence Charts (LSCs) are a broad extension to Message Sequence Charts (MSCs) to capture complex inter-object communication rigorously. A tool support for LSCs, named PlayEngine, is developed to interactively "play-in" and "play-out" scenarios. However, PlayEngine cannot automatically expose system design inconsistencies, e.g. conflicts between universal charts and etc. CSP is a formal language to specify sequential behaviors of a process and communication between processes, which has powerful tool supports, e.g. FDR. Semantically, system behaviors specified by LSCs correspond to CSP's traces and failures. This close semantic correspondence makes FDR a potential model checker for LSCs. The challenge is to discover a systematic way of constructing semantic preserving CSP models from LSCs. In this work, we investigate theoretical relations between LSCs and CSP. LSCs are formalized using trace and failure semantics so as to facilitate the semantic transformation from LSCs to CSP. The practical implication is that mature tool supports for CSP can be reused to validate LSCs. In particular, FDR is used to establish the consistency of an LSC model and perform various verification.

Keywords: LSCs, CSP, FDR, Verification

1 Introduction

Message Sequence Charts (MSCs) [14] are widely used to describe scenarios that capture communication between processes or objects. They are used in early stages of system development. They have found their ways into many methodologies [14, 2]. However, MSCs suffer from the rather weak partial-order semantics that makes it incapable of capturing many kinds of behavioral requirements. Live Sequence Charts (LSCs) [7] are introduced by Damm and Harel to overcome the shortcomings of MSCs by adding liveness. LSCs extend MSCs with various constructs to distinguish scenarios that must happen from scenarios that may happen, conditions that must be fulfilled from conditions that may be fulfilled and etc. Together with symbolic objects and various high-level operators like bounded loop, if-then-else, LSCs may well be used to specify complicated inter-objects system requirements. A software package named *PlayEngine* is developed by Damm and Harel to interactively "play-in" and "play-out" scenarios. However, *PlayEngine* does not support automatic verification of LSCs. It is important to expose inconsistencies of system requirements in the early stage of system development. One effective approach to verify LSC models is to use existing mature model checkers, i.e. FDR (Failure Divergence Refinement[20]) of Communicating Sequential Process (CSP) [13].

Semantically, system behaviors specified by LSCs correspond to CSP's traces and failures. This close semantic correspondence makes FDR a potential model checker for LSCs. The challenge is to construct semantic preserving CSP models from LSCs systematically. In this work, we firstly investigate theoretical relations between LSCs and CSP and then develop semantic mappings from LSCs to CSP. The investigation of semantic mappings from LSCs to CSP is more than of theoretical interests. Its practical implication is that various tool supports for CSP can be reused to validate LSCs. In this work, we are particularly interested in using FDR, a well-known CSP model-checker, to verify LSC models. Our approach is automated using JAVA and XML technology.

The semantics of LSCs is briefly discussed in [7] using skeleton automata and program-like pseudo-codes. Only basic charts and pre-charts are covered. To develop a sound transformation from LSCs to CSP, a more complete semantics is needed. We start with formalizing LSCs by completing and extending the skeleton automata approach to give precise semantics to universal charts. Important semantic definitions like traces and failures are defined then. Constructing CSP processes from LSCs by mimicking the states in the skeleton automata is not only impractically expensive, but also results in an unreadable CSP model and creates barriers to relate verification results to original charts.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

We present a systematic way to construct CSP processes instead. The soundness of the construction is shown using the failure semantics. Finally, we show that FDR can be used to verify the constructed CSP processes for various properties so as to establish the consistency and correctness of the LSCs.

As for related works, there are attempts on formalizing LSCs [15, 4]. Bontemps and Heymans [4] use Büchi automata to define the language expressed by a set of LSCs so that standard algorithm for automata can be used to check consistency and refinement and etc. However, as Büchi automata are low-level and not structured, flattening high-level LSCs to automata suffers from the state space explosion problem. Our work preserves the structure of the LSCs and avoids constructing the global state machine. The key point is that our construction allows mature tool supports for CSP readily to validate LSC specifications. Klose and Wittke [15] derive a similar timed Büchi automaton to capture the semantics of an LSC chart in isolation. Our approach handles multiple LSCs and are extensible. Our work is also loosely related to works on formalization and simulation and validation of MSCs/LSCs, e.g. the simulation tool developed by Wang and etc. based on constraint logic programming [21] and theoretical works on MSCs by Thiagarajan [16] and Mauw and Reniers [19].

The remaining of the paper is organized as follows. Section 2 introduces CSP and LSCs. Section 3 presents the semantics of LSC charts using a skeleton automaton and shows how to construct CSP processes from an LSC chart and a set of LSC charts. Section 4 presents an automatic transformation tool and shows the verification of LSCs using examples. Section 5 concludes the paper.

2 Background Review

2.1 CSP

Hoare's CSP [13, 20] is a formal specification language where processes proceed from one state to another by engaging in events. A CSP process is defined by process expressions. The syntactic class of process expression is defined as:

$$P ::= RUN_\Sigma \mid \textsc{Stop} \mid \textsc{Skip} \mid P_1 \lhd b \rhd P_2 \mid e \rightarrow P \mid P_1 \sqcap P_2 \mid P_1 \,\Box\, P_2 \mid P_1;\ P_2 \mid P_1 \,|[\,\Sigma\,]|\, P_2 \mid \Big|\Big|_{k=1}^{n}(P_k, \Sigma_k) \mid P_1 \,\triangledown\, P_2 \mid \cdots$$

CSP defines a rich set of operators to create processes. RUN_Σ is a process always willing to engage any event in Σ. STOP denotes a process that deadlocks and does nothing. A process that terminates immediately is written as SKIP. A process $e \rightarrow P$ is initially willing to engage in event e and behaves as P afterward. CSP allows a hierarchical description of a system by offering various operators to compose processes. The sequential composition, $P_1;\ P_2$, behaves as P_1 until it terminates and then behaves as P_2. A choice between two processes is denoted as $P_1 \mid P_2$. The choice is made either internally ($P_1 \sqcap P_2$) or externally ($P_1 \,\Box\, P_2$). Often, choices are guarded by prefixing or conditionals. A choice that depends on the truth value of a boolean expression b is written as $P_1 \lhd b \rhd P_2$. If b is true, this process proceeds as P_1, otherwise P_2. Parallel composition of two processes is denoted as $P_1 \,|[\,\Sigma\,]|\, P_2$, where events in Σ are synchronized. $\Big|\Big|_{k=1}^{n}(P_k, \Sigma_k)$ is a replicated alphabetized parallel denoting parallel composition of n processes, where each process P_k synchronizes with the rest of the system on events in Σ_k. $P_1 \,\triangledown\, P_2$ behaves as P_1 until the initial event of P_2 engages and P_2 takes control. If P_1 can engage the initial event of P_2.

Three mathematical models for CSP are defined. In the traces model: a process is represented by the set of finite sequences of communications it can perform, denoted as $traces(P)$. In the stable failures model, a process is represented by its traces and also by its failures. A failure is a pair (t, Σ), where t is a finite trace of the process and Σ is a set of events it can refuse after t (refusal). The set of P's failures is denoted as $failures(P)$. In the failures/divergences model [5], a process is represented by its failures, together with its divergences. A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions. Failure/divergence model and stable failure model make no difference for divergence-free systems.

Three main forms of refinements are relevant, corresponding to the three models [20]. Traces refinement is used for proving safety properties. Failures refinement is normally used to prove failures-divergence refinement for divergence-free processes. Failures-divergence refinement is used for proving safety, liveness and combination properties, and also for establishing refinement and equality relations between systems. Two processes P_1, P_2 are equivalent, denoted as $P_1 =_{\mathcal{FD}} P_2$, if and only if $failures(P_1) = failures(P_2)$ and $divergences(P_1) = divergences(P_2)$. Equivalence of processes can be proved or disproved by appeal to algebraic laws. For example, some simple laws relevant to our work are:

$$\begin{array}{ll} [R1] & P \,|[\,\Sigma\,]|\, RUN_\Sigma =_{\mathcal{FD}} P \\ [R2] & P \parallel \textsc{Stop} =_{\mathcal{FD}} \textsc{Stop} \\ [R3] & P \parallel P =_{\mathcal{FD}} P \end{array}$$

FDR, a commercial product of Formal Systems (Europe) Ltd., is a model-checker for CSP. Its method of establishing whether a property holds is to test for the refinement of a CSP process capturing the property by the candidate process. It can also check determinism, deadlock-freeness, livelock-freeness of CSP processes.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 IEEE

2.2 LSCs

There are two kinds of charts in LSCs. Existential charts are mainly used to describe possible scenarios of a system in early stages of system development. In later stage, knowledge becomes available about when a system run has progressed far enough for a specific usage of the system to become relevant. Universal charts are then used to specify behaviors that should always be exhibited. A universal chart may be preceded by a pre-chart, which serves as the activation conditions for executing the main chart. Whenever a communication sequence matches a pre-chart, the system must proceed as specified by the main chart. A universal chart is pre-activated if its pre-chart is activated. A chart is associated with a set of visible events. Only visible events are constrained. A chart typically consists of multiple instances, which is represented as vertical lines. Along with each line, there are a finite number of locations. A location carries the temperature annotation for progress within an instance. A location may be labelled as either cold or hot. A hot location means that the system has to move beyond. Similarly, messages and conditions are also labelled. A hot message must be received and a hot condition must be met.

Typically, a system is described by a set of LSCs, both universal charts and existential charts. We assume that an LSC model consists of a set of universal charts. Existential charts are used to specify test cases. Due to pre-charts, a system run may activate a universal chart more than once and some of the activation might overlap [18]. LSCs support advanced MSC features like co-region, hierarchy and etc. Symbolic instances and messages are adopted to group scenarios effectively. For details on features of LSCs, refer to [11]. LSCs are far more expressive than MSCs, which makes them capable of expressing complicated inter-objects system requirements.

Figure 1 shows an LSC universal chart with a nonempty pre-chart taken from [6] as part of the model of a phone. Four objects participate in this scenario, a user, the cover, the display and the speaker. This scenario describes the behavior of the phone when the cover is closed. Once the cover is closed by the user, the main chart is activated. The display is set to the current time if it is not off. The speaker is turned to silent mode. In this example, all vertical lines in the main chart are dotted, which means that the system may stay forever at some point along the vertical line. Symbolic instances and messages are used in this example as both objects (*Display*) and messages (*SetState*) are parameterized.

3 LSCs as CSPs

In this section, we formalize LSCs by adopting and extending the skeleton automata approach [7]. In [7], the semantics of a basic chart or a pre-chart is explained informally using skeleton automata and program-like pseudo-codes. No indication is given on what the system behaviors specified a set of universal charts are. We present a detailed definition of the skeleton automata for basic charts and pre-charts. The behaviors of a chart is defined using traces, failures. Lastly, we show how to construct CSP processes to capture the behaviors specified by a set of universal charts systematically.

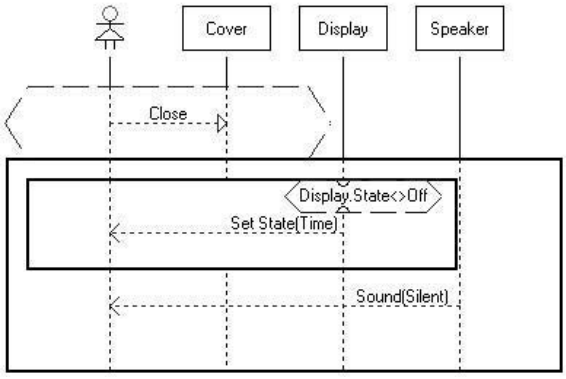

Figure 1. Phone LSC Example: Close Cover

3.1 Semantics of LSCs

In this section, we focus on semantic models of the main features of LSCs in order to demonstrate the essential idea of our approach. For example, in the following semantic model, we assume that all message passings are synchronized and conditions or actions are not shared between instances. It should be clear that other features can be incorporated in our semantic model and proper CSP processes can be constructed systematically. For example, in our transformation tool, asynchronous message passings are handled by explicitly modelling the behavior of the buffer (i.e. FIFO), shared conditions are expressed as two separate conditions on the instances with proper synchronization before and after the conditions. We assume that all locations are cold as CSP only specifies prefix-close languages. Theoretically, communications labelled with hot locations can be captured by the notion of *signals* in Davies's Work on CSP [8], where *singals* are defined as events that must be observed in the future state. Timing events in LSCs are treated as abstract CSP events in the way similar to the treatment of Timed CSP in FDR.

3.1.1 Semantics of Basic Charts

A basic chart is a simplest universal chart that has no hierarchy and preceded by no pre-charts. The semantics of

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

a basic chart m is defined to consist of all runs compatible with the partial order induced by m and its annotations. Let Σ_m represent the set of events visible to m. We assume that there are n instances in the chart. For each instance, there are a finite number of locations. A location is labelled with a finite number of events[1] and at most one condition. The events and condition associated with a location are simultaneously evaluated. We first give a set of axiom definitions to ease the definition of the skeleton automata.

Let $Chart, Instance$ denote the set of all charts and instances, we define function $inst : Chart \rightarrow \mathbb{P}\, Instance$ to return the set of instances in a chart. A location of an instance is uniquely identified by its location index, $Location == Instance \times \mathbb{N}$. A location indexed with 0 is the starting point of the instance in the chart. We define function $max : Instance \rightarrow \mathbb{N}$ to return the maximum location index, i.e. the ending point of the instance. Given a location in the chart, we define function $\mathbf{next} : Location \rightarrow Location$ to compute the next location along the vertical line representing an instance.

$$\forall (i,l) : Location \bullet l < max(i) \Rightarrow \mathbf{next}(i,l) = (i, l+1)$$

An LSC event is either a local action $\mathcal{A}$ or a message passing $\mathcal{M}$. Local actions are mainly assignments or invocations of external functions, which we abstract away here. A message is attached with a flag to indicate sending or receiving.

$$\begin{array}{l} Event == \mathcal{M} \bigcup \mathcal{A} \\ \mathcal{M} == \mathcal{M}\_ID \times \mathcal{M}\_Flag \\ \mathcal{M}\_Flag ::= SND \mid RCV \end{array}$$

Function $label : Location \rightarrow \mathbb{P}\, Event$ labels a location with a finite number of messages and local actions. Function $cond : Location \rightarrow Condition$ labels a location with a condition. If there is no condition associated with the location, it returns $true$. Each condition can be evaluated to either true or false, $eval : Condition \rightarrow (true \mid false)$. Each condition is either labelled as cold or hot, $label : Condition \rightarrow (cold \mid hot)$.

An LSC chart induces a partial order over the events. The partial order is defined as the smallest binary relation $\mathcal{R} : Location \leftrightarrow Location$ satisfying the following axioms and closed under transitivity and reflexivity.

- $\forall l : Location \bullet l \,\underline{\mathcal{R}}\, \mathbf{next}(l)$
- $\forall l_1, l_2 : Location \mid l_1 \neq l_2 \bullet$
  $(\exists m : \mathcal{M}\_ID \bullet (m, SND) \in label(l_1) \wedge$
  $(m, RCV) \in label(l_2)) \Rightarrow l_1 \,\underline{\mathcal{R}}\, l_2$
- $\forall l_1, l_2 : Location \mid l_1 \neq l_2 \bullet$
  $(\exists m : \mathcal{M}\_ID \bullet (m, SND) \in label(l_1) \wedge$
  $(m, RCV) \in label(l_2)) \Rightarrow l_2 \,\underline{\mathcal{R}}\, \mathbf{next}(l_1)$

[1] More than one if it is a co-region.

Basically, the first axiom says that time progresses from top to bottom along each vertical line. The second axiom says message input event must precede the corresponding message output event. The third axiom handles synchronous message passing. An LSC chart m is well-formed if the relation $\mathcal{R}$ is acyclic. In the sequel, we assume that all charts are well-formed.

We define function $\mathbf{preset} : Location \rightarrow \mathbb{P}\, Location$ to return the set of locations that precede a given location in the relation $\mathcal{R}$.

$$\forall l : Location \bullet \mathbf{preset}(l) = \{x : Location \mid x \,\underline{\mathcal{R}}\, l\}$$

One of the basic concepts used for defining the semantics of LSCs is the notion of a **cut**. A cut through the chart represents the progress each instances has made in the scenario. Let **cut** denote the set of all possible cuts, a cut is a set of locations, one for each instance, satisfying the following condition:

$$\begin{array}{l} \forall x : \mathbf{cut}(m) \bullet \#(x) = n \wedge \\ \quad \forall l : x \bullet \nexists\, l' : x \bullet l' \in \mathbf{preset}(l) \end{array}$$

In case of co-regions, a **cut** is not sufficient to represent the progress of each instance in a chart. Thus, we define the notion of a **state**. A state of the chart is defined as a set of pairs of location and set of events, one pair for each instance, satisfying the following condition:

$$\begin{array}{l} \forall s : \mathbf{state}(m) \bullet \#(s) = n \wedge \\ \quad \forall (l, A) : s \bullet A \subseteq label(l) \wedge \\ \quad \forall (l_1, A_1), (l_2, A_2) : s \bullet l_1 \neq l_2 \wedge \\ \quad \exists (l_1, \cdots, l_n) : \mathbf{cut};\ A_1, \cdots, A_n : \mathbb{P}\, Event \mid \\ \qquad \{(l_l, A_1), \cdots, (l_n, A_n)\} = s \end{array}$$

For an instance at location l_k, given a set of events A_k already engaged at the location, we define the set of enabled events as: $\mathbf{enabled}(l_k, A_k) = label(l_k) \setminus A_k$. For a state during the progress of a chart, we compute the set of events that are ready to be engaged as **Enabled**.

$$\begin{array}{l} \forall s : \mathbf{state} \bullet s = \bigcup_{k \in inst(m)} \{(l_k, A_k)\} \Rightarrow \\ \quad \mathbf{Enabled}(s) = \bigcup_{k \in inst(m)} (label(l_k) \setminus A_k) \end{array}$$

We are now ready to define the semantics of a basic chart using a skeleton automaton.

Definition 3.1 *A basic chart m is associated with a skeleton automata* $A_m \mathrel{\widehat{=}} (S_m, S_m^0, \Sigma_m \cup \{\tau\}, T_m)$ *where:*

- $S_m \mathrel{\widehat{=}} \{Aborted, Terminated\} \cup Active$, *where* $Active \mathrel{\widehat{=}} \mathbf{state}(m)$. *A basic chart is either active or aborted or terminated.*
- $S_m^0 \mathrel{\widehat{=}} \bigcup_{k \in inst(m)} \{(0, \varnothing)\}$. *Initially, a chart is active and all instances are at their first location with no events engaged.*

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

- Σ_m *is the set of visible events to the chart. A special event* τ *is added to denote temporal progress along a vertical line.*
- $T_m : S_m \times \Sigma_m \cup \{\tau\} \rightarrow S_m$ *is the smallest transition relation satisfying the following conditions*[2]*:*

  T1: $\forall e : \Sigma_m \bullet (Terminated, e, Terminated) \in T_m$

  T2: $(\bigcup_k \{(max(k), label(max(k)))\}, \tau, Terminated) \in T_m$

  T3: $s \in Active \wedge (l, A) \in s \wedge e \in \mathcal{A} \wedge$
  $e \in \textbf{enabled}(l, A) \Rightarrow (s, e, s \oplus \{l, A \cup \{e\}\}) \in T_m$

  T4: $s \in Active \wedge \{(l_1, A_1), (l_2, A_2)\} \subseteq s \wedge$
  $e \in \mathcal{M}\_ID \wedge (e, SND) \in \textbf{enabled}(l_1, A_1) \wedge$
  $(e, RCV) \in \textbf{enabled}(l_2, A_2) \Rightarrow$
  $(s, e, s \oplus \{(l_1, A_1 \cup \{e\}), (l_2, A_2 \cup \{e\})\}) \in T_m$

  T5: $s \in Active \wedge (l_k, label(l_k)) \in s \wedge$
  $l < max(k) \wedge eval(cond(\textbf{next}(l_k))) = true \wedge$
  $s \setminus \{(l_k, label(l_k))\} \cup \{(\textbf{next}(l_k), \varnothing)\} \in \textbf{state}(m) \Rightarrow$
  $(s, \tau, s \setminus \{(l_k, label(l_k))\} \cup \{(\textbf{next}(l_k), \varnothing)\}) \in T_m$

  T6: $s \in Active \wedge (l_k, label(l_k)) \in s \wedge l < max(k) \wedge$
  $eval(cond(\textbf{next}(l_k))) = false \wedge$
  $label(cond(\textbf{next}(l_k))) = cold$
  $\Rightarrow (s, \tau, Terminated) \in T_m$

  T7: $s \in Active \wedge (l_k, label(l_k)) \in s \wedge l < max(k) \wedge$
  $eval(cond(\textbf{next}(l_k))) = false \wedge$
  $label(cond(\textbf{next}(l_k))) = hot \Rightarrow (s, \tau, Aborted) \in T_m$

Basically, $T1$ says that all behaviors are allowed after a chart is terminated. $T2$ states that a chart is terminated only after all instances have reached their ending points. $T3$ and $T4$ state that a local action or a message passing is always ready to be engaged without breaking the partial ordering. $T5$ states if all events at a location are engaged and the instance hasn't reached its end, the instance proceeds to the next location by taking an internal τ action if the condition labelled with the next location is true. Otherwise, if the condition is labelled *hot*, the chart aborts so that no further behavior is allowed. Whereas if the condition is *cold*, the chart terminates immediately.

LSCs provide operators to compose basic charts hierarchically in various ways. For a chart with hierarchy, we can easily flatten the sub-charts by adding transitions connecting the initial state and *Terminated* state of the sub-chart to the automaton of the upper-level chart. For instance, the if-then-else operator can be flattened by connecting the last state of the upper-level chart to both of the initial states of the two branches. As the flattening is pretty standard, we omit the detail here.

3.1.2 Semantics of Pre-chart

We associate a pre-chart with a skeleton automaton as above. However, due to the nature of pre-chart, the skeleton automaton is defined differently from [7], i.e. unexpected events lead the pre-chart to the *Exited* state, where all behaviors are allowed.

[2] $\oplus$ is the standard function override operator.

Definition 3.2 *A pre-chart p is associated with a skeleton automaton* $A_p \mathrel{\widehat{=}} (S_p, S_p^0, \Sigma_p \cup \{\tau\}, T_p)$ *where:*

- $Active \mathrel{\widehat{=}} \textbf{state}(p)$
  $S_p \mathrel{\widehat{=}} \{Exited, Finished\} \cup Active$
- $S_p^0 \mathrel{\widehat{=}} \bigcup_{k \in inst(m)} \{(0, \varnothing)\}$
- $T_p : S_p \times \Sigma_p \rightarrow S_p$ *is the smallest transition relation satisfying the following conditions in addition to T3, T4, T5:*
  - $\forall e : \Sigma_m \bullet (Exited, e, Exited) \in T_p$
  - $(\bigcup_k \{(max(k), label(max(k)))\}, \tau, Finished) \in T_p$
  - $s \in Active \wedge (l_k, label(l_k)) \in s \wedge l < max(k) \wedge$
    $eval(cond(\textbf{next}(l_k))) = false \Rightarrow (s, \tau, Exited) \in T_p$
  - $s \in Active \wedge e \in Event \wedge e \notin \textbf{Enabled}(s) \Rightarrow$
    $(s, e, Exited) \in T_p$

A pre-chart is *Finished* if all its sequence of events are matched. A *false* condition, either labelled as hot or cold, leads the chart to the *Exited* state. If some events constrained by this chart appear out of order, the pre-chart exits. A universal chart preceded by a pre-chart is defined by connecting the automaton for pre-chart with the one for the main chart.

Definition 3.3 *For a chart preceded by a pre-chart, let* $A_p \mathrel{\widehat{=}} (S_p, S_p^0, \Sigma_p, T_p)$ *and* $A_m \mathrel{\widehat{=}} (S_m, S_m^0, \Sigma_m, T_m)$ *denote the skeleton automata associated with the pre-chart and main chart respectively. The automaton associated with the chart is defined as* $A \mathrel{\widehat{=}} (S, S^0, \Sigma, T)$ *where* $S \mathrel{\widehat{=}} S_p \cup S_m$ *and* $S^0 \mathrel{\widehat{=}} S_p^0$ *and* $\Sigma \mathrel{\widehat{=}} \Sigma_m \cup \Sigma_p$ *and* $T \mathrel{\widehat{=}} T_p \cup T_m \cup \{(Finished, \tau, S_m^0)\}$.

The automaton of a universal chart accepts message sequences that satisfy the constraint for the first pre-activation[3] of the chart. After the first pre-activation, the automaton is expired and all behaviors are allowed. In general, a chart may be pre-activated more than once (in general, may be infinite number of times) during a system run and activation of the same chart may overlap as well. A universal chart should constrain all behaviors of a system at all time. Our solution is to use multiple copies of the CSP process (if necessary) corresponding to the same automaton to constrain the system properly.

Important definitions like traces, failures, can be defined for the skeleton automaton in the standard way [12]. For an automaton $A \mathrel{\widehat{=}} (S, S^0, \Sigma, T)$, we define[4]:

$$
\begin{aligned}
traces(A) = \{t : \operatorname{seq}\Sigma \mid \exists\, & s_1, \cdots, s_k, \cdots, s_n : S \bullet \\
& s_1 = S^0 \wedge \forall k : 1 \mathinner{..} n \bullet (s_k, t(k), s_{k+1}) \in T)\} \\
failures(A) = \{(t, \Sigma) \mid \exists\, & s_1, \cdots, s_k, \cdots, s_n : S \bullet \\
& s_1 = S^0 \wedge \forall k : 1 \mathinner{..} n \bullet (s_k, t(k), s_{k+1}) \in T \wedge \\
& \forall e : \Sigma \bullet \nexists s' : S \bullet (s_n, e, s') \in T\}
\end{aligned}
$$

[3] The main chart may not get activated

[4] The failure definition is only for universal chart. For existential chart, the failures are $\{(s, \varnothing) \mid s \in traces(A)\}$.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

In LSC, a divergence occurs if an empty sub-chart is infinitely looping (or the subchart contains some trivially true conditions, so the trace is an infinite sequences of τ.). In this work, we assume that a chart is divergence-free by requiring that a chart must contain non-empty set of locations and each location (except the starting point and ending point) must contain non-empty set of events. Thus, a universal chart is equivalent to a CSP process if and only if their failures are same.

3.2 LSC Charts as CSP Processes

A CSP process P can be constructed from a skeleton automaton A to capture its behavior by mimicking the states in the skeleton automata $(A.S)$. Initially, P begins in the initial state S^0. Upon entering a new state $s \in S$, the process is ready to engage events that label one of the outgoing transitions. However, the number of states in the skeleton automaton could be huge due to the weak partial ordering. Mimicking the states is impractically expensive as well as it destroys the structure information of the charts. The latter problem is essential in our case as it may create barriers to expose the source of inconsistencies using verification results on constructed CSP processes. The basic idea of our approach is to model LSC instances as CSP processes.

For a basic chart m, we define two processes *Terminated* and *Aborted* to mimic the respective states in the skeleton automaton. At *Terminated* state, all behaviors are allowed. The system halts at *Aborted* state.

$$
\begin{array}{l}
\mathit{Terminated} \mathrel{\widehat{=}} \mathit{RUN}_{\Sigma_m} \\
\mathit{Aborted} \mathrel{\widehat{=}} \textsc{Stop}_{\Sigma_m} \\
P_i^l \mathrel{\widehat{=}} \begin{cases} p_i^l \nleft \mathit{cond}(i,l) \nright (\alpha \rightarrow \textsc{Skip}) \\ \qquad \text{if } \mathit{label}(\mathit{cond}(i,l)) = \mathit{cold}; \\ p_i^l \nleft \mathit{cond}(i,l) \nright \mathit{Aborted} \\ \qquad \text{if } \mathit{label}(\mathit{cond}(i,l)) = \mathit{hot}. \end{cases} \\
p_i^l \mathrel{\widehat{=}} \begin{cases} (\Big|\Big|\Big|_{e \in \mathit{label}(i,l)} (e \rightarrow \textsc{Skip}));\ P_i^{l+1} \\ \qquad \text{if } l \leq \mathit{max}(i); \\ \textsc{Skip} \\ \qquad \text{if } l > \mathit{max}(i). \end{cases} \\
P_i \mathrel{\widehat{=}} P_i^0 \triangledown (\alpha \rightarrow \mathit{SKIP}) \\
P \mathrel{\widehat{=}} ((\,|[\,\mathcal{M}\,]|_{i \in \mathit{inst}(m)} P_i);\ \mathit{Terminated}) \setminus \{\alpha\}
\end{array}
$$

Each instance in the chart is modelled as a CSP process. The CSP process modelling the chart P is composed by processes modelling the instances P_i where $i \in \mathit{inst}(m)$. For an instance i in the chart, each location (i,l) is modelled as a process P_i^l. P_i^l is defined as a choice depending on the condition labelled with the location. If the condition is true, the process proceeds as p_i^l to engage all events labelled with the location and then proceed as the process modelling the next location. Otherwise, if the condition is cold and false, event α is engaged so that all processes modelling the instances are interrupted and the system proceeds as *Terminated*. If the condition is hot and false, the system deadlocks. The process modelling an instance is defined as the process modelling the first location of the instance. Process P is defined as the parallel composition of processes modelling the instances. Message passings are synchronized. From the above construction, we claim that there is an equivalent CSP process for a basic chart. Formally,

Lemma 3.4 *Given a basic chart m associated with skeleton automaton A_m, there exists a CSP process P such that $P =_{\mathcal{FD}} A_m$.*

We remark that LSC hierarchical operators have their exact images in CSP. For instance, the if-then-else construct in LSCs can be easily modelled as an external choice of two guarded processes, the unbounded loop construct can be modelled as an unguarded recursion and etc.

For a universal chart m preceded by a pre-chart, we model the main chart as above. We denote the pre-chart and the chart as CSP processes *Pre* and P respectively:

$$
\begin{array}{l}
\mathit{Exited} \mathrel{\widehat{=}} \mathit{RUN}_{\Sigma} \\
\mathit{pre}_i^l \mathrel{\widehat{=}} \begin{cases} (\Big|\Big|\Big|_e (e \rightarrow \textsc{Skip}));\ \mathit{Pre}_i^{l+1} \\ \qquad \text{if } e \in \mathit{label}(i,l) \wedge l \leq \mathit{max}(i); \\ e \rightarrow \epsilon \rightarrow \textsc{Skip} \\ \qquad \text{if } e \notin \bigcup_i \mathit{label}(i,l) \wedge l \leq \mathit{max}(i); \\ \beta \rightarrow P_i^0 \\ \qquad \text{if } l > \mathit{max}(i). \end{cases} \\
\mathit{Pre}_i^l \mathrel{\widehat{=}} \mathit{pre}_i^l \nleft \mathit{cond}(i,l) \nright (\epsilon \rightarrow \textsc{Skip}) \\
\mathit{Pre}_i \mathrel{\widehat{=}} \mathit{Pre}_i^0 \triangledown (\epsilon \rightarrow \mathit{Exited}) \\
\mathit{Pre} \mathrel{\widehat{=}} |[\,\mathcal{M} \cup \{\beta\}\,]|_{i \in \mathit{inst}(m)} \mathit{Pre}_i \\
P \mathrel{\widehat{=}} ((\mathit{Pre} \setminus \{\epsilon\});\ \mathit{Terminated}) \setminus \{\alpha, \beta\}
\end{array}
$$

When a condition (cold or hot) is violated, the pre-chart exits and puts no further constraints on the system by issuing event ϵ. Moreover, an unexpected events cause the process to behave as *Exited* immediately. Event β is used to synchronize the entering or exiting from a chart (sub-chart). After the pre-chart finishes, the system proceed as the main chart.

Lemma 3.5 *Given a chart m associated with skeleton automaton A_m, there exists a process P such that $P =_{\mathcal{FD}} A_m$.*

A system is typically described by a set of LSC charts, each of which may have its pre-chart. A valid implementation of the system should consist of behaviors that satisfies all the universal chart at all time. A universal chart with no pre-chart specifies constraints that must be met all the time. For a universal chart preceded by a pre-chart, multiple activation of the same chart may overlap. Thus, multiple copies of the process modelling the same chart is needed to constrain the system properly. However, if we assume that activation of the same chart never overlaps, as assumed by

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

Damm and Harel in [10], the CSP model constructed from an LSC model is simply the parallel composition the constructed processes for each chart, with *Exited* replaced by P so that non-overlap multiple activation of the same chart is properly constrained.

Without the assumption, we have to construct the CSP processes carefully so that no violation of any chart at any time is possible. Let $m\mathbf{cut} = \{(i_1, l_1), \cdots, (i_n, l_n)\}$ denote the minimum cut satisfying the following conditions:

$$\begin{array}{l} \exists\, k : 1 \,..\, n \bullet l_k > 0 \wedge \\ \forall\, c : \mathbf{cut} \mid c \neq \bigcup_k \{(i_k, 0)\} \bullet \\ \quad \forall\, k : 1 \,..\, n \bullet (i_k, l'_k) \in c \Rightarrow l'_k \geq l_k \end{array}$$

Basically, $m\mathbf{cut}$ is the first cut where some events are ready to be engaged. Correspondingly, we define $m\mathbf{state}$ as $\bigcup_k \{((i_k, l_k), A_k)\}$ where $\bigcup_k \{(i_k, l_k)\} = m\mathbf{cut}$. The basic idea is to fork a new process to monitor the behavior of the system whenever a universal chart is pre-activated.

Let event e be an event that is enabled at the $m\mathbf{state}$, i.e. $e \in label((i_k, l_k)) \wedge (i_k, l_k) \in m\mathbf{cut}$. Let P denote the CSP process for the chart and $\bigcup_k \{((i_k, l'_k), s'_k)\}$ be the target state of the skeleton automaton after performing e from $m\mathbf{state}$:

$$\begin{array}{l} P_e \mathrel{\widehat{=}} (P' \parallel P) \nleftarrow cond((i_k, l_k)) \nrightarrow P \\ \quad \text{where } P' \mathrel{\widehat{=}} (\,|[\,\mathcal{M}\,]|_{i \in inst(m)} pre_i^{l'_k}) \\ P \mathrel{\widehat{=}} (\Box\, e : \bigcup_k label((i_k, l_k)) \rightarrow P_e) \,\Box \\ \quad (\Box\, e : \Sigma \setminus \bigcup_k label((i_k, l_k)) \rightarrow P) \end{array}$$

Whenever a chart is pre-activated by an event, a new copy of the process is forked to monitor subsequent behaviors. If an activation of the pre-chart results in process *Exited*, either due to a false condition or an unexpected event in the pre-chart, the process modelling the activation can be safely ignored due to $R1$ in section 2.1. If a violation is found, the process would result as process *Aborted*, the system deadlocks due to law $R2$.

Now, we are ready to construct the CSP process that satisfies an LSC model. Assume that an LSC model contains n charts ranging from m_1 to m_n, let P_k denote the CSP process modelling m_k, the CSP process is defined as:

$$M \mathrel{\widehat{=}} (\,|\|_{k=1}^{n} (P_k, \Sigma_{m_k})) \parallel RUN$$

Processes modelling different charts synchronize on events visible to the chart with the rest processes. Thus, only visible events are constrained by a chart. The parallel composition of all processes modelling all charts are further composed with a *RUN* process so that system may behave freely on events not constrained by any chart. *RUN* can be replaced by processes that models assumptions on input patterns from the environment, which is discussed in the next section. It is clear that such a CSP process satisfies all universal charts. Moreover, an inconsistency in the LSC model would result in a deadlock. Thus, if we prove the CSP process is deadlock-free, the LSC model is consistent. Based the above discussion, we claim that[5]:

Theorem 3.6 *For an LSC model consisting a set of universal charts, there is an equivalent CSP process.*

4 Verification

In this section, we show how machine readable CSP processes are constructed automatically from LSCs and feed into FDR for various checking. Using FDR, safety, liveness and combination properties can be verified by showing that there is a refinement relation from the constructed CSP model to the CSP process capturing the properties. As this is the standard usages of FDR, we discuss here checking that are closely coupled with our construction. The construction ensures that inconsistencies between universal charts in an LSC model result in deadlocks. FDR is capable of telling whether a CSP program is deadlock-free. A counterexample is presented if the validation is failed, which gives important clues to refine the system. There are basically two sources of deadlocks, one due to inconsistencies between universal charts and the other due to violation of hot conditions. The former requires re-investigation of the system requirements. The latter may suggest either there is some inconsistency or more system requirements are required so that state variables are sufficiently constrained to satisfy the hot conditions. An existential chart can be validated by constructing the corresponding CSP process and check whether it trace-refines the CSP model constructed from the set of universal charts. Moreover, manual proof on the CSP specification may establish properties of the LSC model expressed with logical expressions over traces or trace-refusal pairs. However, this paper focuses on properties can be verified by FDR.

4.1 Automation

The construction is automated using XML and JAVA technology. There is not yet a standard interchange format for LSCs[6]. Therefore, we start with defining the syntax of LSCs both using BNF grammar and XML schema. The BNF grammar is presented in Appendix A. The XML schema and XML representation of the charts appeared in this paper can be found on the web[7]. The BNF grammar and XML schema are defined to express any LSCs in a structured text document. Together with the XML schema, a

[5] The proof is carried out by showing there is a failure-divergence equivalence between an LSC model and the constructed CSP specification. We skip the proof due to the limit of space.

[6] The XML format used in *PlayEngine* is not intended to communicate LSCs. No schema or DTD definition is developed.

[7] *http://www.comp.nus.edu.sg/~sunj/LSC2CSP.html*

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

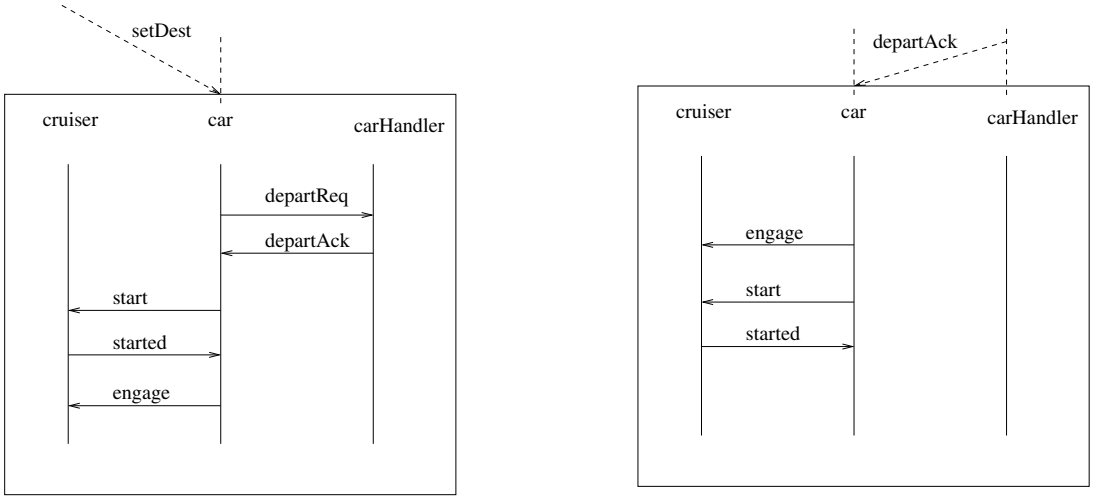

Figure 2. LSC Example: Inconsistency

parser and a transformation module is built using JAVA and existing XML parser [1] to parse XML representations of LSCs and construct CSPs automatically. The output of the program is a machine readable CSP program with a set of assertions, which is readily to be employed and verified in FDR.

4.2 Consistency Check

Due to space limitation, a simple example is presented to demonstrate the construction and verification of CSP models. More complicated examples can be found at above mentioned web page. Figure 2 is presented in [10] as a typical example of inconsistency between universal charts. It is a part of the LSC model for an automatic railway system. A detail description of the system appears in [7]. The objects participating in this scenario are cruiser, car and carHandler. In the upper chart, the message *setDest* sent from the environment to the car activates the chart, which requires that following the *departReq* message, *departAck* is sent from the car handler to the car. This message in turn activates the lower chart, which requires the sending of *engage* from the car to the cruiser before the *start* and *started* messages are sent, while the upper chart requires the opposite ordering. The following program[8] is constructed by feeding the XML representation of both charts to our program.

```
transparent diamond, normalise
channel car_env_setDest
channel car_carHandler_departReq
channel car_carHandler_departAck
channel car_cruiser_start
channel car_cruiser_started
channel car_cruiser_engage
SetDestSigma(0) = {|...|}
SetDestSigma(1) = {|...|}
SetDestSigma(2) = {|...|}
SetDestcar(0) = car_carHandler_departReq ->
    car_carHandler_departAck ->
    car_cruiser_start ->
    car_cruiser_started ->
    car_cruiser_engage -> SKIP
SetDestcruiser(0) = car_cruiser_start ->
    car_cruiser_started ->
    car_cruiser_engage -> SKIP
SetDestcarHandler(0) = car_carHandler_departReq ->
    car_carHandler_departAck -> SKIP
SetDestInst(0) = SetDestcar(0)
SetDestInst(1) = SetDestcruiser(0)
SetDestInst(2) = SetDestcarHandler(0)
Main_SetDest = || x: {0..2}@
    [SetDestSigma(x)] SetDestInst(x)
Prechart_SetDestSigma(0) = {|car_env_setDest|}
Prechart_SetDestcar(0) = car_env_setDest -> SKIP
Prechart_SetDestInst(0) = Prechart_SetDestcar(0)
SetDest = ((Prechart_SetDestInst(0)) ; Main_SetDest)
   [] (car_carHandler_departReq -> SetDest)
   [] (car_carHandler_departAck -> SetDest)
   [] (car_cruiser_start -> SetDest)
   [] (car_cruiser_started -> SetDest)
   [] (car_cruiser_engage -> SetDest)
... Similar for the lower chart ...
Sigma(0) = {|...|}
Sigma(1) = {|...|}
Figure(0) = SetDest
Figure(1) = Depart
SYSTEM = || x: {0..1} @ [Sigma(x)] Figure(x)
assert SYSTEM :[ deadlock free [FD] ]
```

The first part of the program consists of channel definitions for all message passings in the charts. We record the set of events visible of a chart as the set of events that appears in the chart together with the set of forbidden events. The construction follows exactly the discussion in the last section. In this example, we assume that no activation of the same chart overlaps so as to speed up the verification. FDR instantly reports that *SYSTEM* is not dead-lock free. A counter example is found: *car_handler_departAck*, *car_env_setDest*, *car_carHandler_departReq*. The lower chart is firstly activated. Right after that, the upper chart is activated. This is possible as *setDest* is not constrained

[8] Manually simplified so to save space.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

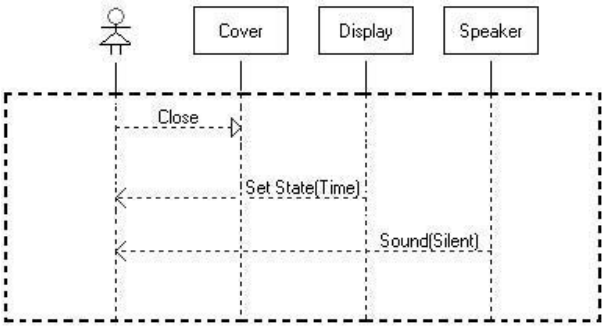

Figure 3. LSC Example: Existential Chart

by the lower chart. After *departReq*, the system deadlocks. However, this deadlock is not what in our mind. An implicit assumption is that *departAck* is engaged only if *setDest* and *departReq* is engaged. We can easily incorporate this assumption using an extra LSC which transforms to the following process $ENV \mathrel{\widehat{=}} car\_env\_setDest \rightarrow car\_carHandler\_departAck \rightarrow ENV$.

Process *SYSTEM* is refined as the parallel composition of the original *SYSTEM* and *ENV*. FDR reports the expected deadlock this time using the following counterexample: *car_env_setDest*, *car_carHandler_departReq*, *car_carHandler_departAck*.

To handle large systems, CSP algebraic laws are used to simplify the constructed processes before feeding them into FDR. Various compression methods in FDR can also be applies, as in the above example (the first line). Our construction is extended to handle symbolic instances and messages, i.e. symbolic instances are modelled as processes with parameters and local definitions, symbolic messages are modelled as typed channels. For instance, the CSP process modelling the symbolic object *Display* in Figure 1 is:

```
datatype DisplayState = On | Off | Time
channel DisplayStatechange: DisplayState
channel DisplayStatequery: DisplayState
DISPLAY(State) = let
    Display(State) =
        DisplayStatequery!State->Display(State) []
        DisplayStatechange?x->Display(x)
    within Display(State)
```

4.3 Existential Charts

A CSP process is constructed from an existential chart in the same way as for a universal chart except that instead of allowing all behaviors after a chart reaches its end, the chart deadlocks. An existential chart is consistent with an LSC model if and only if the existential chart specifies behaviors that are allowed by the set of universal charts. We use FDR to validate an existential chart by checking whether the constructed CSP process is a trace-refinement of the CSP processes constructed for the LSC model. We present here a slightly trivial example. In the phone example, we would like to verify that it is possible that when the cover is close, the display is set to be the current time and the speaker shuts off. This simple scenario is modelled as an existential chart in Figure 3. The CSP process constructed for this chart is:

```
TestInst(0) = User_Cover_Close ->
    User_Display_SetState_Time ->
    User_Speaker_Sound_Silent -> SKIP
TestInst(1) = User_Cover_Close -> SKIP
TestInst(2) = DISPLAYStatechange.Time ->
    User_Display_SetState_Time -> SKIP
TestInst(3) = SPEAKERStatechange.Silent ->
    User_Speaker_Sound_Silent -> SKIP
Test = || x: {0..3}@ TestInst(x)
```

Given *SYSTEM* as the process constructed from the LSCs of the phone system, including the *CLOSECOVER* chart, FDR verifies that *SYSTEM* is trace-refined by *Test*. FDR can also be used to verify whether a property holds by testing for the refinement of a CSP process capturing the property by the candidate process. We can ensure safety requirement by trace refinement and liveness requirement by failure/divergence refinement. Safety and liveness properties may expressed as LSC charts or CSP processes intuitively. For example, we may express a safety property as a universal chart (without pre-chart) containing only a hot condition capturing the property. However, formal derivation of CSP processes or LSCs from temporal specifications is a non-trivial research topic. The former was discussed and evidenced in [3, 17]. The latter is discussed in [9].

5 Conclusion and Future Works

In this work, we study semantic transformations from LSCs to CSP. We formalize main features of LSCs and show that corresponding CSP processes can be constructed effectively and systematically. An automatic tool is developed to automate the transformation. We use FDR to perform various verification on LSC models. Consistency between LSC universal charts is established by proving the constructed CSP processes are deadlock-free. Consistency between existential charts and a set of universal charts is verified by establishing a trace-refinement relation between the constructed processes. One interesting extension to our work is to automatically feed back the verification result from FDR to PlayEngine so as to guide the refinement of the LSC model. Once this is done, the PlayEngine users with little or no knowledge on CSP and FDR may be benefited.

Acknowledgements

We thank Dines Bjørner for his comments on early versions of this paper. We also thank David Harel and Rami Marelly for providing *PlayEngine* and extra documents.

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**

References

[1] XERCES Java Parser v1.4.4. http://xml.apache.org/xerces-j/.

[2] OMG UML v1.3. http://www.uml.org/, June 1999.

[3] R. Berghammer and B. v. Karger. Formal Derivation of CSP Programs From Temporal Specifications. In *Math. of Prog. Cons.*, pages 181–196, 1995.

[4] Y. Bontemps and P. Heymans. Turning High-Level Live Sequence Charts into Automata. In *"Workshop: Scenarios and State-Machines, ICSE'02"*, 2002.

[5] S.D. Brookes and A.W. Roscoe. An Improved Failures Model for Communicating Processes. In *Proceedings of the Pittsburgh seminar on concurrency LNCS 197*, pages 281–305, 1985.

[6] D. Harel and R. Marelly. *Play-Engine User's Guide*, 2003.

[7] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[8] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[9] A. Pnueli Y. Lu H. Kugler, D. Harel and Y. Bontemps. Temporal Logic for Live Sequence Charts. Technical report, The Weizmann Institute of Science Rehovot, Israel, 2000.

[10] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Lecture Notes in Computer Science*, 2088:1–26, 2001.

[11] D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. Technical Report MCS01-15, The Weizmann Institute of Science Rehovot, Israel, 2002.

[12] J. F. He. Process Simulation and Refinement. *Formal Aspect of Computing*, 1(3):229–241, 1989.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[14] ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.

[15] J. Klose and H. Wittke. An Automata Based Interpretation of Live Sequence Charts. In *TACAS*, pages 512–527, 2001.

[16] L. Lavagno, G. Martin, and B. Selic. *UML for Real: Design of Embedded Real-Time Systems*. Kluwer Academic Publishers, 2003.

[17] Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.

[18] R. Marelly and H. Kugler. Multiple Instances and Symbolic variables in Executable Sequence Charts. In *Proc. OOPSLA'02*, pages 83–100, 2002.

[19] S. Mauw and M. A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[20] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[21] T. Wang, A. Roychoudhury, R. H.C. Yap, and S.C. Choudhary. Symbolic Execution of Behavioural Requirements. In *"Proceedings of PADL 2004"*, 2004.

Appendix: An BNF Grammar for LSCs

< *LSCSpec* >::=
    lscspec < *ChartDefList* >< *InstVariList* > **end lscspec**
< *ChartDefList* >::=< *ChartDef* >; < *ChartDefList* >|
< *ChartDef* >::=< *ExtChartDef* >|< *UnvChartDef* >
< *ExtChartDef* >::=
    extchart < *LSCName* >< *InstDefList* > **end extchart**
< *UnvChartDef* >::= **unvchart** < *LSCName* >< *PrechartDef* >
    < *InstDefList* > **end unvchart**
< *PrechartDef* >::= **prechart** < *InstDefList* > **end prechart**
< *InstDefList* >::=< *InstDef* >; < *InstDefList* >|
< *InstDef* >::= **inst** < *InstName* >< *LocationDefList* > **end inst**
< *LocationDefList* >::=< *LocationDef* >; < *LocationDefList* >|
< *LocationDef* >::=
    hotlocation < *LocationDef* > **end hotlocation** |
    coldlocation < *LocationDef* > **end coldlocation** |
    subchart < *SubchartDef* > **end subchart**
< *LocationDef* >::= **event** < *EventDef* > **end event** |
    coregion < *CoregionDef* > **end coregion** |
    hotcondition < *ConditionDef* > **end hotcondition** |
    coldcondition < *ConditionDef* > **end coldcondition**
< *SubchartDef* >::=< *SubchartID* >< *LocationDefList* >
< *CoregionDef* >::=< *EventDefList* >
< *EventDefList* >::=< *EventDef* >; < *EventDefList* >|
< *ConditionDef* >::=< *ConditionID* >< *Condition* >
< *EventDef* >::= **action** < *Action* > **end action** |
    hotmessage < *MessageDef* > **end hotmessage** |
    coldmessage < *MessageDef* > **end coldmessage** |
    timerevent < *TimerEventDef* > **end timerevent**
< *MessageDef* >::= **input** < *InputDef* > **end input** |
    output < *OutputDef* > **end output**
< *TimerEventDef* >::= **set** < *SetTimerDef* > **end set** |
    timeout < *TimeOutDef* > **end timeout** |
    reset < *ResetTimerDef* > **end reset**
< *SetTimerDef* >::=< *Clock* >< *Duration* >
< *TimeOutDef* >::=< *Clock* >
< *ResetTimerDef* >::=< *Clock* >
< *InputDef* >::=< *Message* >< *InstID* >
< *OutputDef* >::=< *Message* >< *InstID* >
< *InstID* >::=< *InstName* >| **env**
< *InstVarList* >::=< *InstVarDef* >; < *InstVarList* >|
< *InstVarDef* >::=< *InstName* >< *VarDefList* >
< *VarDefList* >::=< *VarDef* >; < *VarDefList* >|
< *VarDef* >::=< *Variable* >< *TypeDef* >

Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)
0-7695-2284-X/05 $20.00 © 2005 **IEEE**