LSC'S: BREATHING LIFE INTO MESSAGE SEQUENCE CHARTS

Werner Damm
OFFIS, Oldenburg, Germany

David Harel
The Weizmann Institute of Science, Rehovot, Israel

Abstract: While message sequence charts (MSCs) are widely used in industry to document the interworking of processes or objects, they are expressively quite weak, being based on the modest semantic notion of a partial ordering of events as defined, e.g., in the ITU standard. A highly expressive and rigorously defined MSC language is a must for serious, semantically meaningful tool support for use-cases and scenarios. It is also a prerequisite to addressing what we regard as one of the central problems in behavioral specification of systems: relating scenario-based inter-object specification to state-machine intra-object specification. This paper proposes an extension of MSCs, which we call *live sequence charts* (or *LSCs*), since our main extension deals with specifying "*liveness*", i.e., things that must occur. In fact, LSCs allow the distinction between possible and necessary behavior both globally, on the level of an entire chart and locally, when specifying events, conditions and progress over time within a chart. This also makes it possible to specify forbidden scenarios, and strengthens structuring constructs like as subcharts, branching and iteration.

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

1. INTRODUCTION

Message sequence charts (MSCs) are a popular visual medium for the description of scenarios that capture the typical interworking of processes or objects. They are particularly useful in the early stages of system development. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU [29] (previously called the CCITT). The standard defines the allowed syntactic constructs rigorously, and is also accompanied by a formal semantics [30] that provides unambiguous meaning to basic MSCs in a process algebraic style. Other efforts at defining a rigorous syntax and semantics for MSCs have been made [17, 24, 10], and some tools supporting their analysis are available [1, 2, 6].

Surprisingly, despite the widespread use of the charts themselves and the more rigorous foundational efforts cited above, several fundamental issues have been left unaddressed. One of the most basic of these is, quoting [7]: "What does an MSC specification mean: does it describe all behaviors of a system, or does it describe a set of sample behaviors of a system?". While typically MSCs are used to capture sample scenarios corresponding to use-cases [23, 5], as the system model becomes refined and conditions characterizing use-cases evolve, the intended interpretation often undergoes a metamorphosis from an *existential* to a *universal* view: earlier one wants to say that a condition *can* become true and that when true the scenario *can* happen, but later on one wants to say that *if* the condition characterizing the use-case indeed becomes true the system must adhere to the scenario described in the chart. Thus, we want to be able to specify *liveness* in our scenarios, that is, *mandatory* behavior, and not only provisional behavior.

In fact, the confusion between necessity and possibility arises even within a basic MSC itself: should edges of an MSC prescribe only (partial) ordering constraints, or should they entail causality? While the standard [30] views the semantics of MSCs as merely imposing restrictions on the ordering of events, designers are often interested in shifting the intended meaning depending on the current design level. And this, again, means preferring initially a provisional interpretation, but transforming these into mandatory interpretations as design details are added, thus enforcing messages to be sent and received, progress to be made, etc. We feel that the lack of variety in the semantic support of conditions in the ITU standard may well have contributed to its inability to distinguish between possibility and necessity.

Hence, we feel the dire need for a highly expressive MSC language with a clear and usable syntax and a fully worked out formal semantics. Such a language is needed in order to construct semantically meaningful computerized tools for describing and analyzing use-cases and scenarios. It is also a prerequisite to a thorough investigation of what we consider to be one of the central problems in the behavioral specification of systems, and, we feel, *the* problem in object-oriented specification: relating *inter*-object specification to *intra*-object specification. The former is what engineers will typically do in the early stages of behavioral modeling; namely, they come up with use-cases and

the scenarios that capture them, specifying the inter-relationships between the processes and object instances in a linear or quasi-linear fashion in terms of temporal progress. That is, they come up with the description of the scenarios, or "stories" that the system will support, each one involving all the relevant instances. An MSC language is best used for this. The latter, on the other hand, is what we would like the final stages of behavioral modeling to end up with; namely, a full behavioral specification of each of the processes or object instances. That is, we want a complete description of the behavior of each of the instances under all possible conditions and in all possible "stories". For this, most methodologists agree that a state-machine language (such as statecharts [18, 19]) is most useful. The reason we want something like a state-machine intra-object model as an output from the design stage is for implementation purposes: ultimately, the final software will consist of code for each process or object. These pieces of code, one for each process or object instance, must — together — support the scenarios as specified in the MSCs. Thus the "all relevant parts of stories for one object" descriptions must implement the "one story for all relevant objects" descriptions.

Investigating the two-way relationship between these dual views of behavioral description is an ultimate goal of our work. How to address this *grand dichotomy of reactive behavior*, as we like to call it, is a major problem. For example, how can we synthesize a good first approximation of the statecharts from the MSCs? Finding good ways to do this would constitute a significant advance in the automation and reliability of system development. However, it is not really worth contemplating this problem in any depth without a far more powerful MSC language.

In this paper we propose a language for scenarios, termed *Live Sequence Charts*, or *LSCs* for short. LSCs constitute a smooth extension of the ITU standard for MSCs, along several fronts. We allow the user to selectively designate parts of a chart, or even the whole chart itself, as live, or mandatory, thus forcing messages to be sent, conditions to become true, etc. By taking the existential interpretation as a default, the designer may incrementally add liveness annotations as knowledge about the system evolves. Hand in hand with this extension comes the need to support conditions as first-class citizens: we assume availability of interface definitions for instances, containing events that can be sent and received, and also variables that may be referred to when defining (first-order) conditions. By associating *activation conditions* with an LSC, a live interpretation of the chart becomes more significant; it now means, informally, that whenever the system satisfies the chart's activation condition its behavior *must* conform to that prescribed by the chart. As we shall see, live elements (we call them *hot*) also make it possible to define forbidden scenarios, i.e., ones that are not allowed to happen — a very important need for the engineer at the early stages of behavioral modeling.

Another use of LSCs, indeed one of our motivations for the present work, comes from the UML standard [25], which recommends statecharts as well as sequence-charts for modeling behavior, but says little about the precise rela-

tionships between the two. The Rhapsody tool from i-Logix is based on the language set for *executable object modeling* (*XOM*) defined in [19]. This set is really the executable kernel of the UML, and as thus can be regarded as UML's definitive rigorous core. It consists of the constructive languages of object-model diagrams and statecharts, and allows a variant of MSCs, but as a descriptive language only. The work presented in this paper provides the semantical basis for rigorous and complete consistency checks between the descriptive view of the system by sequence charts and the constructive one. Such checks could eventually be made using formal verification techniques like model-checking [3, 4]. (Some of the ideas of this paper were indeed inspired by the symbolic timing diagrams of [26, 27, 16, 28], used to specify and verify safety-critical requirements for systems modeled using Statemate; see [11, 13, 8, 9, 22].)

The paper is organized as follows. Section 2 defines the way we link LSC specifications to a system, assuming the semantics of basic charts as given. Section 3 presents and motivates our basic extensions to message sequence charts and outlines their semantics informally. We assume a linear time semantics of systems, where each system is associated with a set of (possibly infinite) runs. Section 4 highlights our approach in defining the semantics of LSCs, as the set of runs of a system that is consistent with the chart. Section 5 demonstrates the concepts with an example.

2. RELATING CHARTS TO SYSTEMS

In this section we show how a set of LSCs is related to a conventional behavioral description of the system given in some operational specification language, such as statecharts [18] or an object-oriented version thereof [19]. Usually, this description will be of the intra-object species, but for the purposes of the present paper the precise form it takes is unimportant; as we shall see, all we need is a behavioral description that defines the runs of the system. To avoid confusion, we refer to the language of such descriptions as the *implementation language*, reserving the term *specification* for our LSCs.

We should remark that we have attempted to define LSCs with a minimal amount of commitment to the particulars of the implementation language, so as to preserve as much flexibility as possible. Thus, the reader will detect a certain amount of abstractness in our requirements from the languages and models surrounding the LSCs.

For LSCs to make sense as a specification language, the implementation language must contain explicit ways of creating instances of the modeled system. For example, in a structured analysis framework, such as that of STATEMATE [20, 21], instances could correspond to activities, whereas in an object-oriented framework such as Rhapsody [19], they would correspond to instances of objects. Moreover, the implementation language will associate with each instance its data-space as induced by variable declarations, and its possible events; the latter might contain the sending or receiving of messages, timeouts, and the creation and destruction of instances. We refer to the variables of an instance i by $var(i)$ and to its events by $events(i)$. Variables may be local to an instance

or globally known. All we require is that $var(i)$ contain all variables known to i.

The following table shows the events discussed in this paper. To help keep the present paper focussed on the key aspects of our approach, we have decided to omit from it instance creation and destruction, as well as real-time features such as the setting and expiration of timers.

$< i,\ asynch,\ msgid!j >$	asynchronous transmission of message *msgid* from instance i to instance j
$< i,\ synch,\ msgid!j >$	synchronous transmission of message *msgid* from instance i to instance j
$< i,\ msgid?j >$	receipt of message *msgid* by instance i from instance j
$< i,\ asynch,\ msgid!env >$	asynchronous transmission of message *msgid* from instance i to the environment
$< i,\ synch, msgid!env >$	synchronous transmission of message *msgid* from instance i to the environment
$< i,\ msgid?env >$	receipt of message *msgid* by instance i from the environment

A *snapshot* s of a system S shows all current events and gives a valuation to all variables. In particular, if c is a condition involving events in $events(S)$ (i.e., the collection of events of the system's instances) and variables in $var(S)$ (i.e., the variables of all its instances), then $s \models c$ denotes the fact that c is satisfied in snapshot s.

As mentioned, we assume a *linear time* semantics of our implementation language. For a system S, a *run* of S is an infinite sequence of snapshots. We typically use r to denote a run, $r(i)$ for its i-th snapshot, and r/i for the infinite sequence obtained from r by chopping its prefix of length $i-1$. The set of all runs of S is denoted $runs(S)$.

We now start talking about our chart language. Let M be a set of LSCs. With each LSC m in M, we require as given the set of events and variables visible to m, and denote them by $vis\_events(m)$ and $vis\_var(m)$, respectively. These include all events explicitly shown in m as well as all variables occurring in conditions of m. M is *compatible* with S (denoted $com(M,S)$) if $vis\_ \times events(m) \subseteq events(S)$, and $vis\_var(m) \subseteq var(S)$ for all m in M.

Section 4 will define the concept of satisfaction of a single chart m by a run r of S, denoted by $r \models m$, as a conservative extension of the semantics proposed in the ITU standard [29]. Events and variables not visible in a chart (as defined by $vis\_events(m)$ and $vis\_var(m)$) are not constrained by the chart. In particular, if $r \models m$, and r_jitter is obtained from r by inserting an arbitrary number of events of S which are invisible in m (i.e., events in

the set $events(S) \setminus vis\_events(m)$), then $r\_jitter \models m$. Similarly, inserting an arbitrary but finite number of changes of local variables (not occurring in $vis\_var(m)$) will not impact validity of m.

To a large extent, the ITU standard leaves open the interrelation between a set of MSCs and an independent system description. However, this is a key issue to be resolved for any tool-development exploiting the existence of the two complementary views of system behavior (i.e., inter- and intra-object). The problem to be solved in addressing these issues is the unification of two seemingly contradicting views of the usage of LSCs:

- In early stages in the design process, LSCs will most often be used to describe *possible scenarios* of a system; in doing so, designers stipulate that the system should at least be able to exhibit the behavior shown in the charts. In particular, for each chart drawn, at least one run in the system should satisfy the chart.
- In later stages in the design, knowledge about enabling conditions characterizing different usages of the system to be developed will become available; in the use-case approach, once a run of the system has reached a point where the conditions characterizing the use case apply, designers expect that from now on, regardless of possible ways the system may continue its run, the behavior specified in the chart should *always* be exhibited.

At a logical level, the distinction between the two views is that between an *existential* and a *universal* quantification over the runs of the system: while the scenario view requires the *existence* of a run, the use-case view requires *all* runs of the system to exhibit the specified behavior once the initial condition characterizing the use-case is met. In terms of inclusion of behaviors, the scenario view calls for the legal runs of an LSC specification M of S to be contained in those of S, while the use-cause view calls for the reverse inclusion.

We cater for this distinction by associating with each chart m its mode, with $mod(m) \in \{existential, universal\}$. Hence, an LSC *specification* for a system S is a triple $LS = < M, ac, mod >$, where M is a set of LSCs compatible with S, and $ac(m)$ provides for each $m \in M$ its *activation condition*.

A chart $m \in M$ is *satisfied* by a run $r \in runs(S)$ (written $r \models m$) iff the following hold:

- if m is existential, then $\exists i.\ (r(i) \models ac(m) \ \wedge \ r/i \models m)$;
- if m is universal, then $\forall i.\ (r(i) \models ac(m) \ \Rightarrow \ r/i \models m)$.

The system S *satisfies* the specification LS (written $S \models LS$) iff the following hold:

- for all existential charts $m \in M$, $\exists r \in runs(S).\ r \models m$;
- for all universal charts $m \in M$, $\forall r \in runs(S).\ r \models m$.

Typically, the activation condition of an existential chart will be weak, possibly degenerating to *true*, since we might have only partial knowledge at this state of the system's development. Dually, note that a run of the system need never match the activation condition of a universal chart, so that the "body" of such a chart might become vacuous, imposing *no* restrictions on the system at all. Good tool support for LSCs should offer "healthiness" checks for universal charts, guaranteeing that at least one run eventually reaches a point where its activation condition is true.

In addition to the distinction between existential and universal charts, we may wish to say at some stage that we are done, namely, that the specification $LS = < M, ac, mod >$ completely characterizes the system. We term this *closing LS with respect to S*, and take it to mean that for each run r of S, there is at least one LSC in M satisfied by r. Thus, LS is closed with respect to S iff $\forall r \in runs(S).\ \exists m \in M.\ r \models m$.

3. BREATHING LIFE INTO BASIC CHARTS

As pointed out in the Introduction, the question of which parts of behavior are provisional and which are mandatory is not only an issue when an entire chart is considered. It arises in full force already within a single LSC. Should a message arc linking instances i and i' entail that the communication *will indeed* take place, or just that it *can* take place? Does an instance have to carry out all events indicated along its instance line or can it stop at some point, without continuing? What is the fate of false conditions? Are they mandatory; that is, does the run abort if a false condition is reached? Or are they provisional, meaning that there is some escape route that is taken in such a case?

These are fundamental questions, and one of the main features of our LSC language, which turns it into a true enrichment of MSCs, is the ability to answer them in any of the two ways in each individual case. This is done by adding liveness to the individual parts of the charts, via the ability to specify mandatory, and not only provisional, behavior. Thus, we allow local parts of the chart to be labeled as mandatory or provisional, and this labeling is carried out graphically. We refer to the distinction regarding an internal chart element as the element's *temperature*; mandatory elements are *hot* and provisional elements are *cold*. We have attempted to make the graphical notation simple and clear, trying to remain as close as possible to the visual appeal of the ITU standard for MSCs. Here, now, are the extensions themselves.

Along the horizontal dimension of a chart we not only distinguish between asynchronous and synchronous message-passing by two kinds of arrow-heads (solid for synchronous and open-ended for asynchronous), but the arrows themselves now come in two variants: a dashed arrow depicts provisional behavior — the communication *may* indeed complete — and a solid one depicts mandatory behavior — the communication *must* complete. Along the vertical dimension we use dashed line segments to depict provisional progress of the instance — the run *may* continue downward along the line — while solid lines indicate mandatory progress — the run *must* continue.

As far as conditions go, in order to help in capturing assertions that characterize use-cases, we turn conditions into first-class citizens, allowing not only qualifying requirements as assertions over instance variables, but also properties of the state-space assumed to be true. Our conditions thus also come in the two flavors, mandatory ones denoted by solid-line condition boxes and provisional ones denoted by dashed-line boxes. If a system run encounters a false *mandatory* condition, an error situation arises and the run aborts abnormally. In contrast, a false *provisional* condition induces a normal exit from the enclosing subchart (or the chart itself, if it is on the top-level).

This two-type interpretation of conditions is quite powerful. Mandatory (hot) conditions, together with the other hot elements, make it possible to specify *forbidden* scenarios, i.e., ones that the system is not allowed to exhibit. This is extremely important and allows the behavioral specifier to say early on which are the "yes-stories" that the system adheres to and which are the "no-stories" that it must not adhere to. Also, as we shall see in Section 3, provisional (cold) conditions provide the ability to specify conventional flow of control, such as conditional behavior and various forms of iteration.

Along the vertical time axis, we associate with each instance a set of *locations*, which carry the temperature annotation for progress within an instance. As explained, provisional progress between locations is represented by dashed lines and mandatory progress by solid lines.

The following table summarizes the dual mandatory/provisional notions supported in LSCs, with their informal meaning:

element		**mandatory**	**provisional**
chart	mode	universal	existential
	semantics	all runs of the system satisfy the chart	at least one run of the system satisfies the chart
location	temperature	hot	cold
	semantics	instance run must move beyond location	instance run need not move beyond location
message	temperature	hot	cold
	semantics	if message is sent it will be received	receipt of message is not guaranteed
condition	temperature	hot	cold
	semantics	condition must be met; otherwise abort	if condition not met exit current (sub)chart

One notational comment is in order. While we feel that the consistent use of dashed lines and boxes for provisional elements is important, it raises a problem

with the graphical notation used in the standard (and elsewhere) to denote co-regions — a dashed vertical instance line segments. To avoid this confusion, we denote co-regions by *dotted* line segments running in parallel to the main instance axis.

We have not included figures describing each of the graphical features alone, and prefer to show fuller examples. Thus, Section 5 contains LSCs for parts of the rail-car example of [19]. They illustrate the expressibility of some of the newly introduced concepts in LSCs.

We now define the abstract syntax of the basic charts of our language (the semantics being described briefly in Section 4 and in more detail in the Appendix to the full version of the paper). Let $inst(m)$ be the set of all instance-identifiers referred to in the chart m. With each instance i we associate a finite number of "abstract" discrete locations l from the set $dom(m,i) \subseteq \{0,\ldots,l\_max(m,i)\}$, to which we refer to in the sequel as i's *locations*. We collect *all* locations of m in the set

$$dom(m) = \{ < i,l > | \, i \in inst(m) \wedge l \in dom(m,i) \} \ .$$

Locations are labeled with *conditions* or *messages*. Both messages and conditions are assumed to have unique names. Messages with no defined partner as indicated by a matching message label are assumed to be sent or received from the environment. A *shared* condition by definition reappears as the label of locations in all instances sharing the condition. Formally, the sets of messages and conditions are defined by:

$$Messages = Message\_Ids \times \{synch,\ asynch\} \times \{!,?\}$$

$$Conditions = Condition\_Ids \times Bexp(vis\_var(m))$$

where $Bexp(V)$ denotes the set of boolean expressions involving only variables in the set V. Intuitively, we can describe a snapshot of a system S monitored by a chart m by picking from each of i's instances the "current" location, indicating which events and conditions of this instance have already been observed.

For an MSC m, the association between locations and events or conditions is given by a partial *labeling function*:

$$label(m):\ dom(m) \rightarrow Temp \times (Messages \cup Conditions) \ ,$$

where the first component of the label in the set $Temp = \{hot, cold\}$ defines the temperature of the associated event or condition. To enforce progress along an instance line we associate a temperature with *locations* too, by the total mapping:

$$temp(m):\ dom(m) \rightarrow Temp.$$

As outlined above, labeling a location with the temperature *hot* entails that the chart *must* progress beyond the location, along the subsequent (vertical) segment of the instance line. We add the one restriction that *maximal* locations must be *cold*; this is consistent with the graphical representation depicting a

hot location by a solid line segment *originating* from the hot location: by convention of the ITU standard [29], *no* time-line originates from the endpoint of an instance line, which is its maximal location.

To capture ordering information that will make it possible to associate locations with coregions, we assume a total mapping:

$$order(m) : dom(m) \rightarrow \{true, false\}$$

A coregion is then defined as a maximal unordered set of locations within a given instance; i.e., a maximal connected set L of locations of i all satisfying $order(m)(< i, l >) = false$ (where $l \in L$).

Our LSCs are also endowed with hierarchy and the ability to specify simple flow of control. This is done by allowing a straightforward subchart construction, similar to the one present in the ITU standard, together with multiplicity elements for specifying subchart iteration (both limited iteration — using constants or numeric variables — and unlimited iteration — denoted by an asterisk), and a special notation for conditional branching, also similar to that of the standard. The subcharts are themselves LSCs, specified over a set of instances that may contain some of the instances of the parent chart and some new ones.

While these extensions (the formal definitions of which we omit here) are not in themselves truly novel, when coupled with the dual notions of hot and cold elements in the charts (mainly conditions) their power is significantly enhanced. Whereas hot conditions serve in general to specify critical constraints that must be met to avoid aborting the entire run, in the presence of subcharts cold conditions become of special interest. For example, they can be used to control the flow of the run, by exploiting the fact that our semantics causes a false cold condition to trigger an exit from the current (sub)chart. For example, a subchart with a cold condition at its start is really an *if-then* branching construct, and a subchart annotated with an unbounded multiplicity element and with a cold condition within can be used to specify *while-do* or *repeat-until* constructs, etc.

Thus, cold conditions exit the current subchart and hot conditions abort, providing a clean way to exit iterative and alternative constructs.

4. SEMANTICS OF BASIC CHARTS

A key topic in the formalization of sequence charts is the proper level of abstraction chosen to capture computations on variables. MSCs, and therefore LSCs too, are suitable for capturing the inter-workings of processes and objects, but are not intended to specify how the valuations of variables change during the runs of a system. For this there is a rich variety of specification formalisms. However, as mentioned earlier, we are interested in capturing the conditions that qualify use-cases, and to do so our semantic model must include knowledge about instance variables.

Our approach to reconciling these seemingly contradictory facets of sequence charts is to provide sufficiently loose constraints on variable valuations. We thus

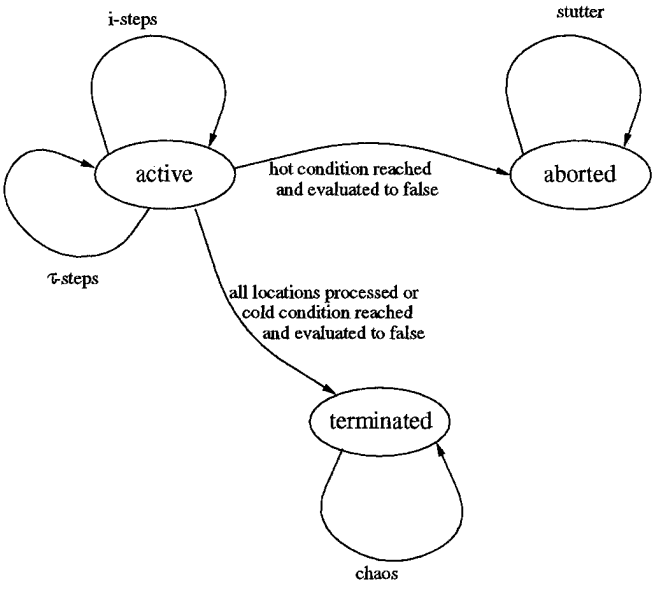

Figure 1 The skeleton automaton of a basic chart

allow runs accepted by an LSC to include any implementation choice in updating instance variables, as long as the constraints expressed by conditions are satisfied. Technically, this can be achieved by allowing a potentially infinite number of local computation steps to occur anywhere between transitions visible in the LSC; such local computation steps hence do not advance the current cut in the partial order, but may arbitrarily change the values of local variables. Note that annotating locations as hot will ensure that local computations do not get stuck in some instance line-segment. Local computation steps may in fact also generate messages, as long as they are not visible in the chart.

Progress requirements induced by hot locations introduce an additional component in the states of the transition-system associated with an LSC: whenever a hot location is reached, its local successor must be reached too. Technically, we achieve this kind of requirement by a list of *promises* we maintain, which will include the successor that has to be reached. For a run to be accepted by the LSC, all promises must be eventually kept, by traversing the LSC at least up to the promised locations. Once thus reached, the promised locations are removed from the list. Similarly, when a run reaches the sending of a hot message, its reception is added to the list of promises, and is removed when the message arrives.

Our definition of the semantics takes a two stage approach. We first associate with an LSC m a transition system $A(m)$ called the *skeleton automaton* of m. Since standard message sequence charts are expressible in our language by always picking the provisional interpretation, the semantics will also be a

conservative extension of that provided by the ITU standard. The semantics of the standard builds on the partial order induced by an LSC m , which we denote by $\leq_m$. The states of $A(m)$ correspond to cuts in $\leq_m$, augmented by the current valuation of visible variables, the currently emitted events of all instances, the set of promises, and finally the *status* of m, in which we record whether the chart is *active*, or *terminated*, or *aborted* due to encountering a hot condition in a state where it evaluates to *false*. A chart may become terminated either after a complete successful run, or upon encountering a cold condition in a state where it evaluates to *false*.

Figure 1 shows the transitions allowed in a particular status. The *τ-steps* perform purely local computations and are always enabled when the chart is active. The *i-steps* allow instance i to proceed; this requires the chart to be *active*, and i's next location to be enabled according to the partial order $\leq_m$. We allow *chaos-steps* to arbitrarily change valuations of variables as well as the presence of events. Also, *stutter-steps* perform only stuttering, i.e., they do not change the state of the transition systm.

Readers with no previous exposure to formal semantics may be irritated by the fact that chaotic behavior is allowed, once the chart has terminated. To understand why chaos is in this case desired, in fact *required*, recall that we have to be able to pad runs of the implementation into behaviors accepted by the LSC. Chaotic behavior hence represents the most liberal restriction possible: all runs that have successfully passed all ordering and liveness constraints causing the chart to achieve status *terminated*, may now behave *ad libitum*.

The full version of this paper [12] contains a complete definition of the transition system $A(m)$.

Given the skeleton automaton $A(m)$ we derive the set of runs accepted by the LSC m in the following steps.

1. We view $A(m)$ as a symbolic transition system, thus obtaining the set $traces(A(m))$ of all infinite sequences π of valuations of instance variables and events, such that the first valuation satisfies the initialization predicate of $A(m)$, and consecutive elements are related by $A(m)$s transition relation.

2. We classify $A(m)$s traces into *accepted* and *rejected* runs, by analyzing the valuation-sequences of the system variables *status* and *promises*:

   - π is *accepted* if one of the following holds:
     (i) it reaches status *terminated* (and maintains this status forever); in this case, either the complete LSC has been matched or a cold condition was not satisfied, causing exit from the chart;
     (ii) it stays forever in status *active*, having, however, fulfilled all promises (thus from some point in time onward, $promises = \emptyset$ continuously); in this case, the LSC has been traversed only partially, with the frontier not progressing beyond some cut through the LSC. Such a computation is perfectly legal, as long as no progress annotations have been given by the designer to force the LSC to move

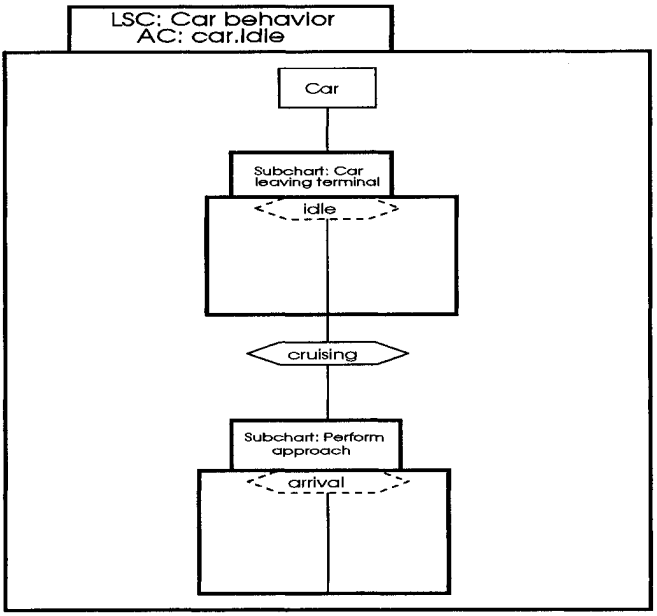

Figure 2 Top level LSC of rail-car

beyond the cut; in particular, this is the case if the LSC is restricted to the notations supported by the current standard.

- π is *rejected* if one of the following holds:

  (i) it reaches status *aborted* (and maintains this status forever); in this case, some hot condition has not been matched, causing abortion of the chart;

  (ii) it stays forever in status *active*, but fails to fulfill its promises, entailing that the set of promises remains non-empty forever; in this case, again the evaluation of the LSC gets stuck at some intermediate cut, performing local computations, but the promises accumulated up to and including this cut have still to be met.

3. We obtain a run of the LSC by projecting an accepted trace onto valuations of instance variables and events only, hiding the system variables *status* and *promises*, as well as *i.blocked* and *i.location*, for all instances i of m.

4. We can now derive the satisfaction relation between a run r produced by some implementation and an LSC m. We say that m is *satisfied* by r, denoted $r \models m$, *iff* r is one of the runs of m according to clause 3 above.

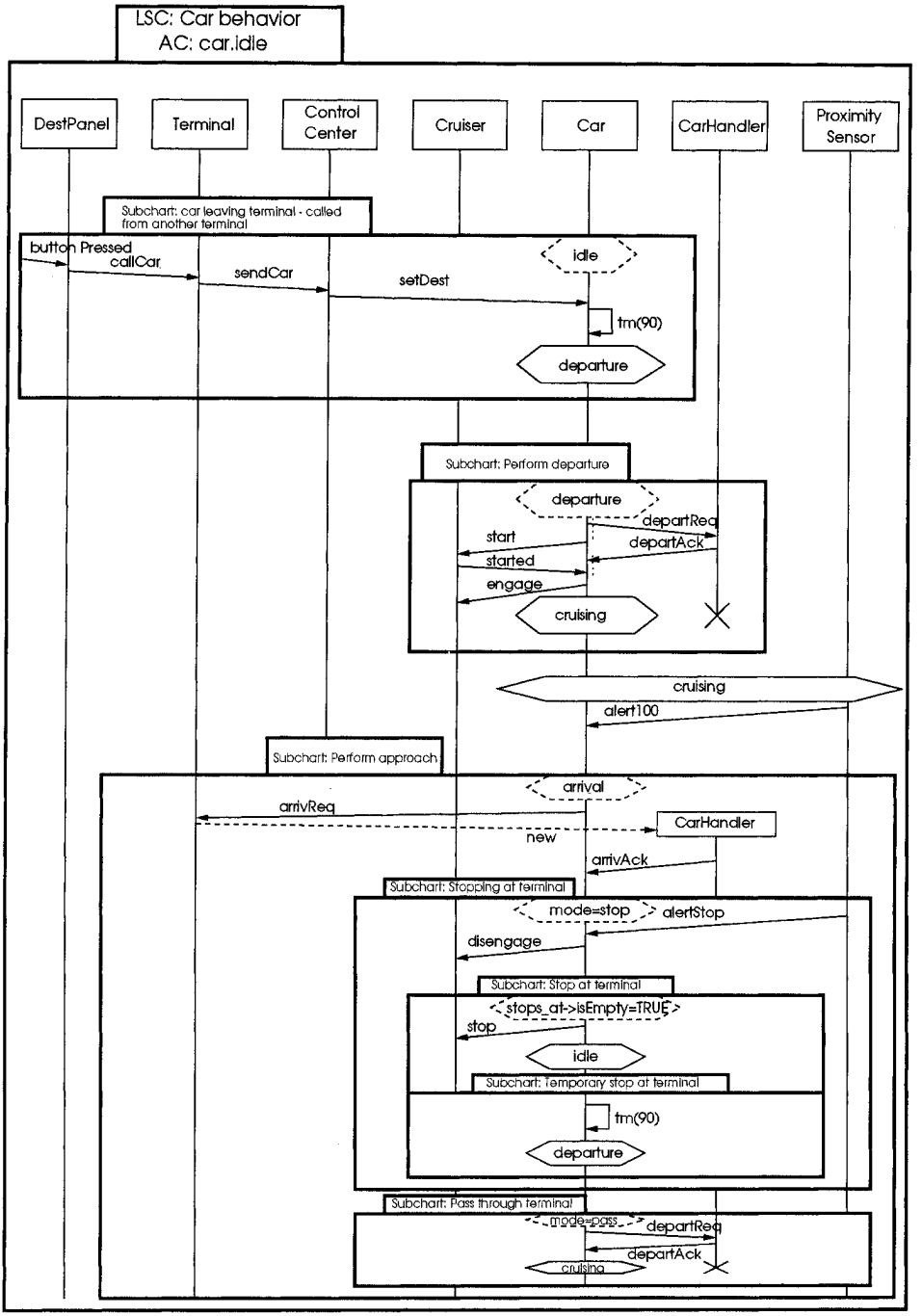

Figure 3 Full LSC of rail-car

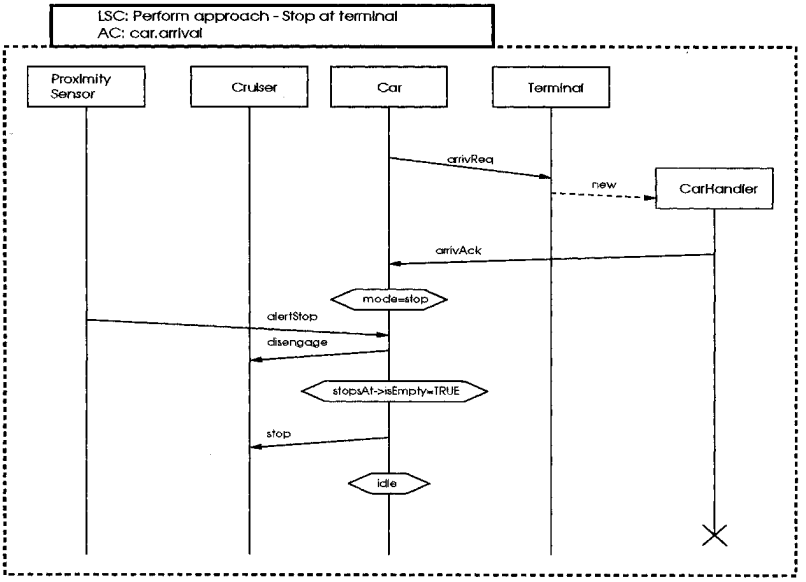

Figure 4 Existential LSC for "Perform approach": Scenario 1

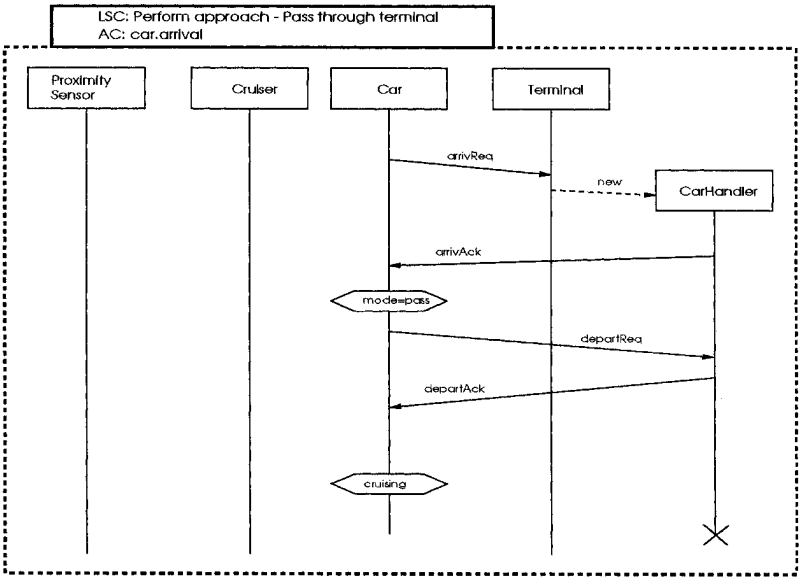

Figure 5 Existential LSC for "Perform approach": Scenario 2

5. AN EXAMPLE

This section is devoted to illustrating LSCs with an example — the car behavior portion of the rail-car system of [19]. The reader would do well to have [19] handy, since the system itself is described there, as are the relevant scenarios. Also, for lack of space, we show here only a few of the relevant charts. More of them appear in the full version of the paper [12].

We have not yet incorporated states into LSCs, so we do not provide a direct mapping between the statecharts of [19] and the LSCs below, but we feel that the connection is quite clear. In fact, we claim that for the most part the LSCs are self-explanatory. Figure 2 shows a very high-level LSC for *Car behavior*, and Figure 3 provides the full LSC for it.

A few things are worth noting: the way we denote a full chart by "LSC: name" and a subchart by "Subchart: name"; the way a top-level condition "activating" a subchart drawn within a parent chart is attached from within to the top of the subchart borderline; the fact that the only instance lines shown passing through a subchart are the ones relevant to it, and that the others become transparent to it; the *cruising* condition that is joint to the *Car* and *Proximity Sensor*; the if-then-else construct within the *Stopping at terminal* subchart; the termination of the *CarHandler* instance, and the two small coregions with their dotted lines, inside the *Perform departure* subchart. Note also that we are using the standard timeout notation from statecharts, although we do not deal with timing issues in this paper.

Figures 4 and 5 are not subcharts. They are full LSCs, and are presented with dashed borderlines to signify that they are existential. They show two of the three alternative scenarios of *Perform approach* (we omit the third for lack of space in this version of the paper), and hence they do not need to be satisfied in all runs. In contrast, the main LSC in Figures 2 and 3 is universal, so that it has to be satisfied in all runs, but its activation condition *car.idle* makes sure that only runs satisfying the *car.idle* condition need be considered, as prescribed by the semantics of universal LSCs.

The contrast between the two ways of presenting the possible scenarios of *Perform approach* (by existential charts or by an appropriately guarded subchart) illustrates our comments in the Introduction about the different stages of behavioral specification. Typically, the scenarios would first be specified existentially, as in Figures 4 and 5, probably early on in the specification process. Later, they would be carefully combined — using the appropriate conditions — into the more informative subchart that appears within Figure 3.

Acknowledgments

We would like to thank Eran Gery for extensive discussions in the initial phases of the work, and Hillel Kugler for his help in preparing the examples and for comments on an early version. The referees made several very valuable suggestions.

References

[1] R. Alur, G.J. Holzmann and D. Peled. An analyzer for message sequence charts. In T. Margaria and B. Steffen (eds.), *Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*'96), Lecture Notes in Computer Science 1055, S. 35–48, Springer-Verlag, 1996.

[2] R. Alur, G.J. Holzmann and D. Peled. An analyzer for message sequence charts. *Software — Concepts and Tools* **17**(2), (1996) 70–77.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation* **98**(2) (1992) 142–170.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Proc.* 27^{th} *ACM/IEEE Design Automation Conference*, pp. 46–51, 1990.

[5] G. Booch, I. Jacobson and J. Rumbaugh. *Unified Modeling Language for Object-Oriented Development.* Rational Software Corporation, 1996.

[6] H. Ben-Abdallah and S. Leue. *Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications.* Technical Report 97–04, Department of Electrical and Computer Engineering, University of Waterloo, April 1997.

[7] H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. In *Proc.* 10^{th} *International Conference on Formal Description Techniques FORTE/PSTV*'97, Chapman and Hall, 1997.

[8] U. Brockmeyer and G. Wittich. Tamagotchis need not die — verification of Statemate designs. In *Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*'98), 1998 (to appear).

[9] U. Brockmeyer and G. Wittich. Real-Time Verification of STATEMATE Designs. *Proc. CAV 98*, to appear.

[10] M. Broy, C. Hofmann, I. Kröger and M. Schmidt. A Graphical Description Technique for Communication in Software Architectures. In *Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC'97/ICSC'97)*, 1997.

[11] W. Damm, M. Eckrich, U. Brockmeyer, H.-J. Holberg and G. Wittich. Einsatz formaler Methoden zur Erhöhung der Sicherheit eingebetteter Systeme im Kfz. In 17. *VDI/VW-Gemeinschaftstagung System-Engineering in der Kfz-Entwicklung*, VDI-Tagungsbericht, 1997.

[12] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Technical Report CS98-09, The Weizmann Institute of Science, Rehovot, Israel, April 1998.

[13] W. Damm, B. Josko, H. Hungar and A. Pnueli. A compositional real-time semantics for STATEMATE designs. In *Proc. COMPOS'97*, Lecture Notes in Computer Science, Springer Verlag, 1998.

[14] W. Damm, B. Josko and R. Schlör. Specification and verification of VHDL-based system-level hardware designs. In E. Börger (ed.), *Specification and Validation Methods*. Oxford University Press, 1995, pp. 331–410.

[15] W. Damm and A. Pnueli. Verifying out-of-order execution. In D.K. Probst (ed.), *Advances in Hardware Design and Verification: IFIP WG* 10.5 *International Conference on Correct Hardware Design and Verification Methods* (*CHARME*), Montreal, Canada, Chapman and Hall, 1997, pp. 23–47.

[16] K. Feyerabend and B. Josko. A visual formalism for real time requirement specifications. In M. Bertran and T. Rus (eds.), *Transformation-Based Reactive Systems Development, Proc.* 4^{th} *International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97*, Lecture Notes in Computer Science 1231, pp. 156–168, Springer-Verlag, 1997.

[17] J. Grabowski, P. Graubmann and E. Rudolph. Towards a Petri net based semantics definition for message sequence charts. In O. Frgemand and A. Sarma (eds.), *SDL'93: Using Objects, Proc.* 6^{th} *SDL Forum*, pp. 179–190, North-Holland, 1993.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* **8** (1987) 231–274.

[19] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, pp. 31–42, July 1997.

[20] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Software Engineering* **16** (1990) 403–414.

[21] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.

[22] J. Helbig and P. Kelb. An OBDD representation of statecharts. In *Proc. European Design and Test Conference (EDAC)*, pp. 142–148, 1994.

[23] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Reading, MA, 1992.

[24] P.B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing* **7**(5) (1995) 473–509.

[25] Rational Corp. Documents on UML (the Unified Modeling Language), http://www.rational.com/uml/resources.html, 1997.

[26] R. Schlör. *Symbolic Timing Diagrams: A Visual Formalism for Specification and Verification of System-Level Hardware Designs.* Dissertation, Universität Oldenburg, 1998 (to appear).

[27] R. Schlör and W. Damm. Specification and verification of system level hardware designs using timing diagrams. In *Proc. European Conference on Design Automation*, pp. 518–524, Paris, France, February 1993.

[28] R. Schlör, B. Josko, and D. Werth. Using a visual formalism for design verification in industrial environments. In *Proc. Workshop on Visualization Issues for Formal Methods, VISUAL'98*, Lecture Notes in Computer Science, Springer-Verlag, 1998 (to appear).

[29] *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, 1996.

[30] *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC) — Annex B: Algebraic Semantics of Message Sequence Charts.* ITU-TS, Geneva, 1995.