Werner Damm and David Harel. LSCs: Breathing life into Message Sequence Charts. In 3rd IFIP Int. Conference on Formal Methods for Open ObjectBased Distributed Systems, (FMOODS'99), pages 293--312. Kluwer Academic Publishers, 1999.

Formal Methods in System Design, 19, 45–80, 2001
© 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

LSCs: Breathing Life into Message Sequence Charts*

WERNER DAMM damm@offis.de
OFFIS, Oldenburg, Germany

DAVID HAREL harel@wisdom.weizmann.ac.il
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 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 makes it possible to specify forbidden scenarios, for example, and enables naturally specified structuring constructs such as subcharts, branching and iteration.

Keywords: formal specification, sequence charts, UML

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 [34] (previously called the CCITT). The standard defines the allowed syntactic constructs rigorously, and is also accompanied by a formal semantics [35] 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 [10, 17, 29], 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

*Early version appeared as Weizmann Institute Tech. Report CS98-09, April 1998. An abridged version appeared in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, Kluwer, 1999, pp. 293–312.

use-cases [5, 25], 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 [35] 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: 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, 20]) 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, despite the fact that there are a number of algorithms for synthesizing state

machines from MSCs [26–28, 30], we believe that this problem should really be contemplated in depth for a far more powerful MSC language.[1]

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 universal (that is, live, or mandatory), thus specifying that messages have to be sent, conditions must become true, etc. By taking the weaker 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 *pre-charts* with an LSC, a live interpretation of the chart becomes more significant; it now means, informally, that whenever the system exhibits the communication behavior of its pre-chart its own behavior *must* conform to that prescribed by the chart. (See figure 4 for an example of a pre-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 [36], which recommends statecharts as well as sequence-charts for modeling behavior, but says little about the precise relationships between the two. The Rhapsody tool from I-Logix is based on the language set for *executable object modeling* (*XOM*) defined in [20], which is really the executable kernel of the UML, and as thus can be regarded as UML's definitive 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 [16, 31–33], used to specify and verify safety-critical requirements for systems modeled using Statemate; see [8, 9, 11, 12, 24].)

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 outlines our approach in defining the semantics of LSCs as the set of runs of a system that is consistent with the chart. A full definition of that semantics appears in the Appendix. Section 5 demonstrates the concepts with an example—the automated rail-car system of [20]. Since, for lack of space, we cannot reproduce that example here, we must assume that the reader has familiarized him/herself somewhat with it. Small figures with variations on this example are used also in earlier sections to illustrate some of the main concepts. In Section 6 we discuss how to use LSCs and supporting tools in the life cycle of system development, and how they relate to use-cases and to the recently proposed play-in scenarios of [19].

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 [20]. 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 [21, 23], instances could correspond to activities, whereas in an object-oriented framework such as the UML [36] or Rhapsody [20], 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.

Consider our railcar system example (again, the basis of this example is taken from [20], with which we must assume familiarity), as described by the charts of figures 1–4 and 8–14.

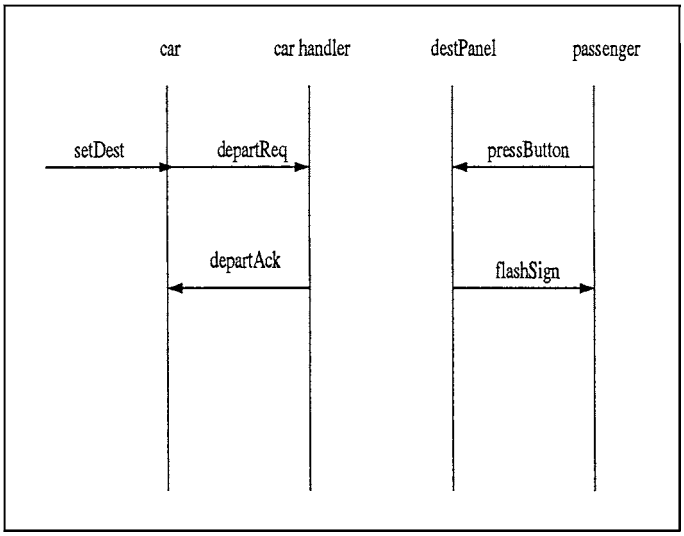

Figure 1. Illustrating visible events.

The instance *car* has the following variables:[2]

$$mode \in \{pass, stop\}$$
$$isEmpty \in \{TRUE, FALSE\}$$
$$state \in \{idle, departure, cruising, arrival\}$$

The *car* object can send events *start*, *stop, engage*, and *disengage* to the *cruiser* and events *departReq* and *arrivReq* to the *carHandler*. The car can receive events *alert*100 and *alertStop* from the *proxSensor*, events *departAck* and *arrivAck* from the *carHandler* and the event *started* from the *cruiser*.

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.

$\langle i, asynch, msgid!j \rangle$	Asynchronous transmission of message *msgid* from instance i to instance j
$\langle i, synch, msgid!j \rangle$	Synchronous transmission of message *msgid* from instance i to instance j
$\langle i, msgid?j \rangle$	Receipt of message *msgid* by instance i from instance j
$\langle i, asynch, msgid!env \rangle$	Asynchronous transmission of message *msgid* from instance i to the environment
$\langle i, synch, msgid!env \rangle$	Synchronous transmission of message *msgid* from instance i to the environment
$\langle i, msgid?env \rangle$	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, but they may contain others too. Thus, the visible events and variables consist not only of the ones that actually show up in the chart, but also of others that we state explicitly as being known to, or visible to, the chart.

M is *compatible* with S (denoted $com(M, S)$) if $vis_events(m) \subseteq events(S)$, and $vis_var(m) \subseteq var(S)$ for all m in M.

Consider the chart of figure 1. The events that can be actually seen in the figure are

$\langle car, setDest?env \rangle$, $\langle car, synch, departReq!carHandler \rangle$,
$\langle carHandler, departReq?car \rangle$, $\langle carHandler, synch, departAck!car \rangle$,
$\langle car, departAck?carHandler \rangle$, $\langle passenger, synch, pressButton!destPanel \rangle$,

⟨*destPanel*, *pressButton?passenger*⟩, ⟨*destPanel*, *synch*, *flashSign!car*⟩, and ⟨*passenger, flashSign?destPanel*⟩.

These can then be enriched with others that are not seen in the chart, but which are also included in *vis_events*(m), such as ⟨*car*, *synch*, *arrivReq!carHandler*⟩, ⟨*carHandler*, *arrivReq?car*⟩, ⟨*carHandler, synch, arrivAck!car*⟩, and ⟨*car, arrivAck?carHandler*⟩.

If the chart is a universal chart (as defined later), none of the events in *vis_events*(m) will be allowed to occur in between the events appearing in the chart itself. In this way, we can specify that the *arrivReq* and *arrivAck* messages should not occur when a car is departing from a terminal since they are relevant for arrival only.

In Section 4 (and in more detail in the Appendix) we 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 [34]. Events and variables not visible in a chart (as defined by *vis_events*(m) and *vis_var*(m)) are not constrained by the chart. However events that explicitly appear in these sets are restricted 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 and not explicitly restricted by m, (i.e., events in the set *events*(S)—*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 will become available about when a system run has progressed far enough for a specific usage of the system to become relevant; in the use-case approach, once a run of the system has reached a point where the use case applies, 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. This condition can be an actual *activation condition*, reflecting some "snapshot" situation, or alternatively, it can be a communication sequence that leads to the activation point; this is expressed in a separate chart called a *pre-chart*.[3]

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-case view calls for the reverse inclusion.

We cater for this distinction by associating with each chart m its mode, where

$$mod(m) \in \{existential, universal\}.$$

Hence, an LSC *specification* for a system S is a tuple

$$LS = \langle M, ac, pch, mod \rangle,$$

where M is a set of LSCs compatible with S and where $ac(m)$ and $pch(m)$ provide the activation condition and pre-chart for each $m \in M$.

A chart $m \in M$ is *satisfied* by a run $r \in runs(S)$ (written $r \models m$) iff the following hold:[4]

- 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 the pre-chart of an existential chart will be weak, possibly degenerating to *true*, and the pre-chart itself will be small or even empty. The reason is that with an existential chart we might have only partial knowledge of what we want in this stage of the system's development. In contrast, for universal charts, a run of the system need never match the activation condition, so that the "body" of a universal 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 the pre-chart, including its activation condition, is satisfied.

Figure 2 describes a car departing from a terminal. The instances participating in this scenario are *cruiser, car*, and *carHandler*. The chart of figure 2 is universal. In it we use a simple kind of activiation condition, which is really an activation message. We denote it by a dashed cold message arrow coming into the chart from the outside. Thus, whenever this message occurs, i.e., the car receives *setDest* from the environment, the sequence of messages in the chart should occur in the following order: the car sends a departure request *departReq* to the car handler, which sends a departure acknowledgment *departAck* back to the car. The car then sends a *start* message to the cruiser in order to activate the engine, and the cruiser responds by sending *started* to the car. Finally, the car sends *engage* to the cruiser and now the car can depart from the terminal. Each send event is followed by the appropriate receive event.

Figure 3 shows an example of an existential chart, depicted by dashed borderlines. The chart describes a possible scenario of a car stopping at a terminal. Since the chart is existential, it need not be satisfied in all runs; we only require that at least one run satisfies it. In an iterative development of LSC specifications, such an existential chart may be considered informal, or underspecified, and can later be transformed into a universal chart specifying the exact activation message or pre-chart that is to determine when it happens.

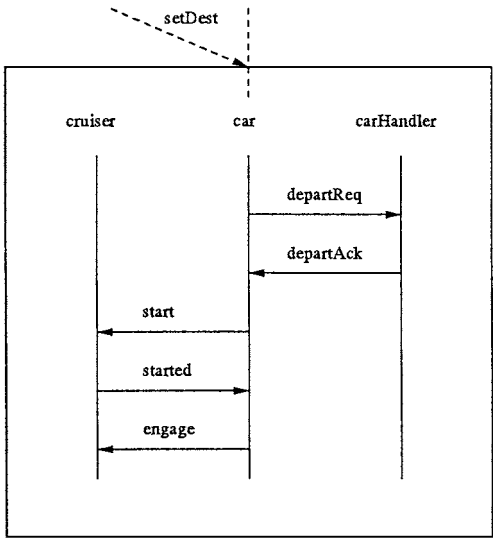

Figure 2. A universal chart.

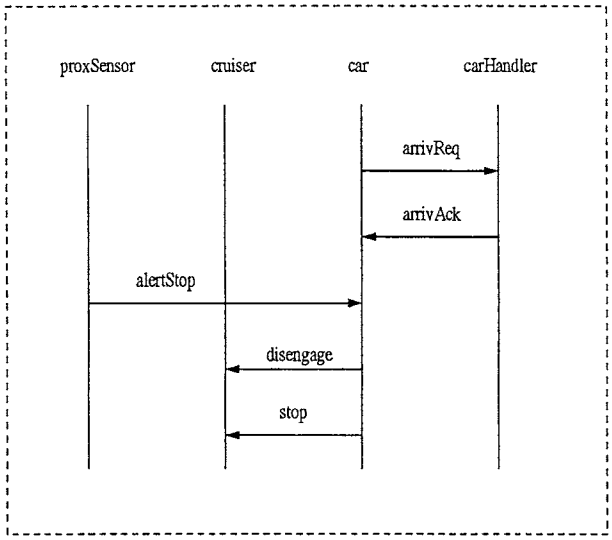

Figure 3. An existential chart.

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. Conditions can thus qualify requirements as assertions over instance variables, and they too come in two flavors, in line with the basic spirit of LSCs: mandatory (hot) conditions, denoted by solid-line condition boxes, and provisional (cold) 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 conditions (that is, hot ones), together with 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 later, 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.

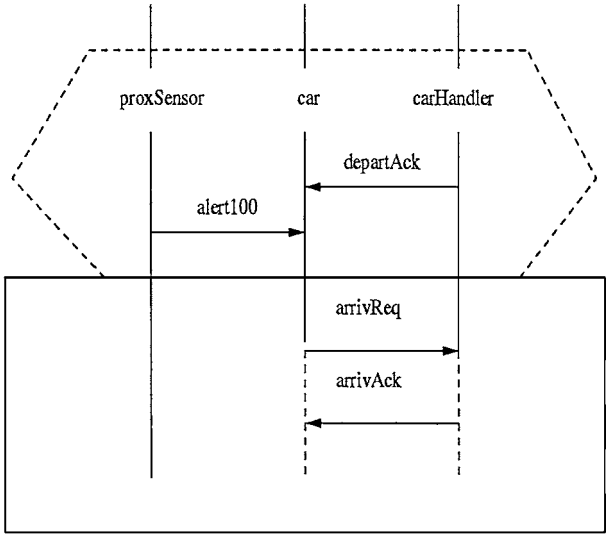

Figure 4. Illustrating pre-charts and cold locations.

Consider figure 4, depicting the *perform approach* scenario. The top part of the figure shows our notation for pre-charts: a dashed frame, like that of a cold condition, surrounding the pre-chart, thus indicating that the scenario is relevant only if the pre-chart has been traversed successfully. The dashed segments in the lower part of the car and carHandler instances specify that it is possible that the message *arrivAck* will not be sent, even in a run in which the pre-chart holds. This might happen in a situation where the terminal is closed or when all the platforms are full.

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

		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 segment. 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 [20]. 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. 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

$$dom(m) = \{\langle i, l\rangle \mid i \in inst(m) \wedge l \in dom(m, i)\}.$$

Locations are labeled with an arbitrary number of *messages* and at most one *condition*. All messages within such a *simultaneous region* are sent/received at the same time and the condition is also evaluated simultaneously. Single messages or conditions are thus special cases of this more general construct. 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 in 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. Each location is then labeled by an element of the set

$$Labels = \wp(Messages) \times Conditions.$$

Intuitively, we can describe a snapshot of a system S monitored by a chart m by picking from each of m's instances the "current" location, indicating which events and conditions of this instance have already been observed.

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

$$label(m) : dom(m) \rightarrow Labels.$$

To enforce progress along an instance line we associate a temperature with *locations*. Temperatures are also used to indicate if a message must be received once sent and if a condition has to be satisfied. This is expressed by the total mapping:

$$temp(m) : (dom(m) \cup Message_Ids \cup Condition_Ids) \rightarrow Temp,$$

with $Temp = \{hot, cold\}$. 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 [34], *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 $order(m)(\langle i, l\rangle) = false$ (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 (an asterisk) and with a cold condition within can be used to specify very naturally *while-do* or *repeat-until* constructs, etc. Thus, cold conditions exit the current subchart and hot conditions abort the entire run.

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 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 transition s 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 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 5 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 idle steps only, i.e., they do not change the state of the transition system.

Readers with no previous exposure to formal semantics may be irritated by the fact that *any* 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 Appendix 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. We first define the set of traces of m to consist of the appropriate (infinite) sequences of valuations of instance variables and events. We then classify the traces into accepted and rejected ones, which is done by inspecting the valuation sequences

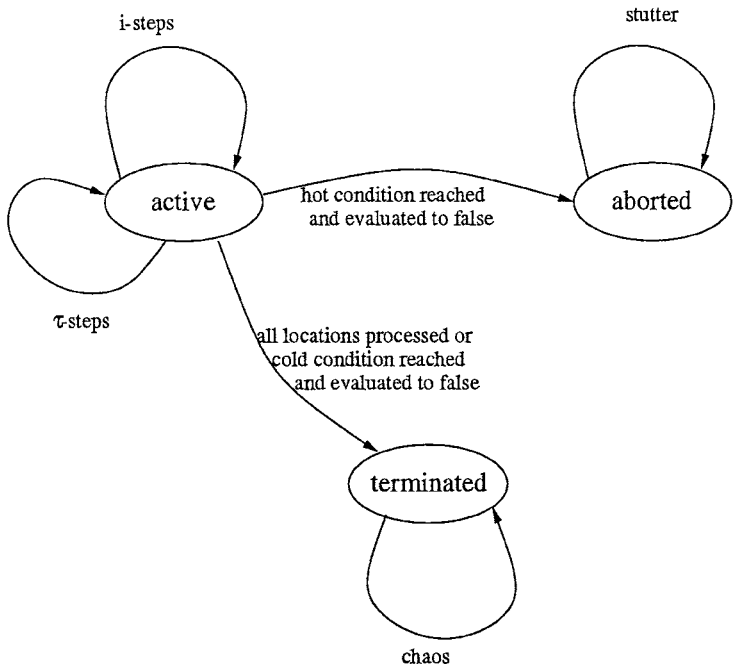

Figure 5. The skeleton automaton of a basic chart.

of the *status* and *promises* variables. We then obtain the runs satisfying the chart m by projecting the accepted traces onto instance variables and events only, hiding things like the system variables *status* and *promises*.

The details of these steps also appear in the Appendix.

We now consider the pre-chart for an LSC. The skeleton automaton for a pre-chart differs from a regular skeleton automaton (cf. figure 6). For the pre-chart we do not allow false cold conditions to be a legal completion of the chart, because a pre-chart describes a *single* prefix to the real LSC. A violated cold condition would provide an extra exit point from the pre-chart, in addition to the one obtained by visiting all locations. We therefore introduce an exit-state into the automaton to reflect this difference.

The semantics of a pre-chart is thus described by the concatenation of the corresponding skeleton automaton with the automaton of the actual LSC (as shown in figure 7). In order to glue the two automata together two things have to be done once the automata have been constructed. First, when the pre-chart has been traversed completely we enter the initial state of the automaton for the actual LSC. Second, an abort from the pre-chart is not an abort from the concatenated automaton, but a legal exit. This abort signifies an illegal prefix, therefore we just do not activate the actual LSC. We analogously exit from the concatenation automaton when we encounter a false cold condition in the pre-chart.

More formally, let $A(p)$ and $A(m)$ denote the skeleton automata associated with the pre-chart p and the LSC m, respectively, and let "$\rightarrow_{pre}$" denote the above concatenation operator on skeleton automata. We then derive the semantics of an LSC m with pre-chart p

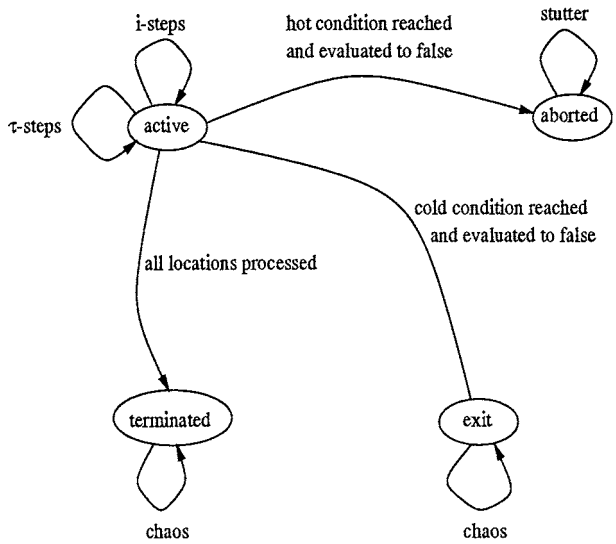

Figure 6. The skeleton automaton of a pre-chart.

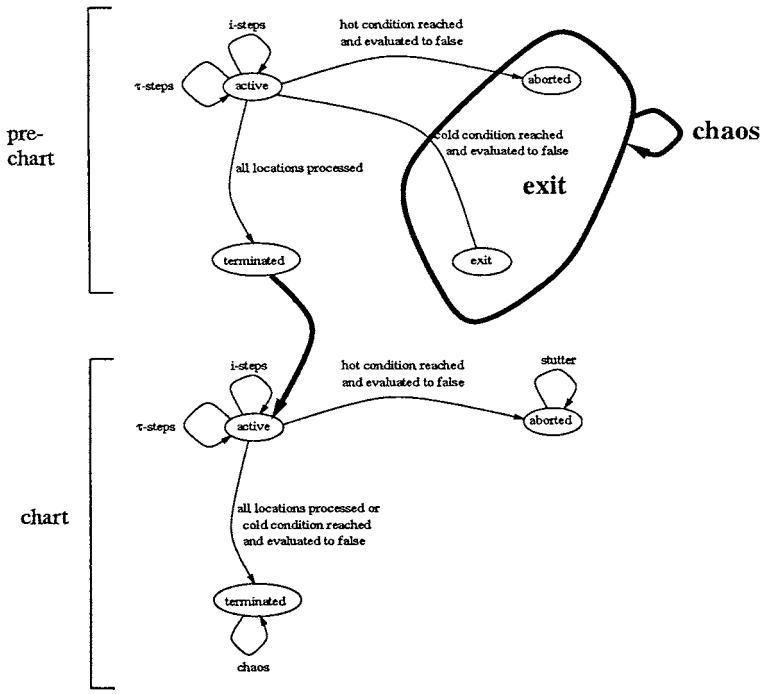

Figure 7. Concatenation of pre-chart and LSC automaton.

and activation condition $ac(m)$ by consistently working with $A(p) \rightarrow_{pre} A(m)$ instead of just with the skeleton automaton $A(m)$ in all formal definitions. Note that this entails that the activation condition of m must be satisfied to even trigger the evaluation of its pre-chart.

5. An example

This section is devoted to illustrating LSCs with an example—parts of the car behavior portion of the rail-car system of [20]. The reader would do well to have [20] handy, since the system itself is described there, as are the relevant scenarios.

In this paper we have not incorporated explicitly specifiable states into LSCs, so we do not provide here a direct mapping between the statecharts of [20] and the LSCs. However, once a state-based system model exists this is rather easy to do. Also, the example has no pre-charts, only activation conditions (denoted by *AC*). In fact, we claim that for the most part the LSCs in the figures are self-explanatory. Figures 8, 9 and 10 provide the *Car Behavior* in increasing levels of detail. That is, each figure is a *refinement* of the previous one. Though we do not provide the formal semantics of refinement in this paper, it can easily be defined using our semantics of LSCs. Figure 11 shows the *Perform approach* subchart on its own.

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

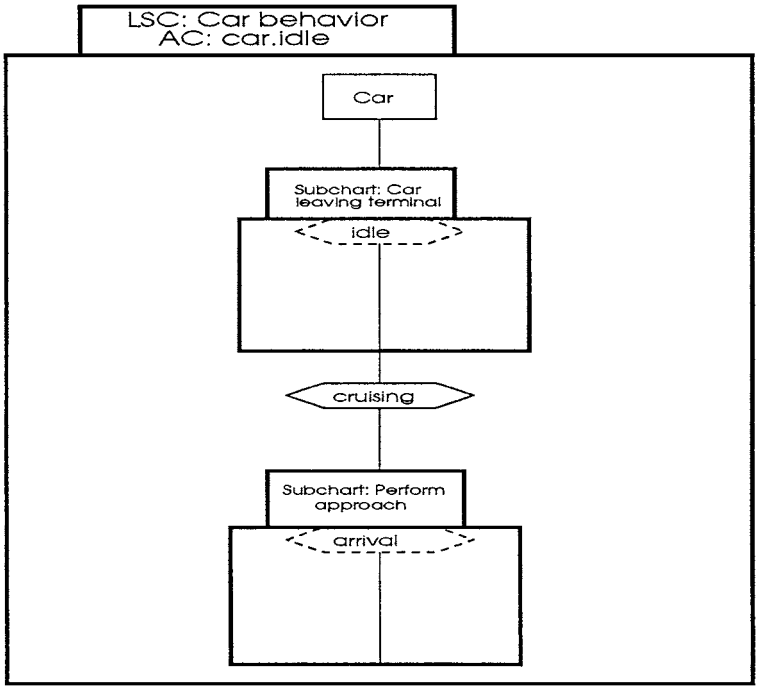

Figure 8. Top level LSC of rail-car.

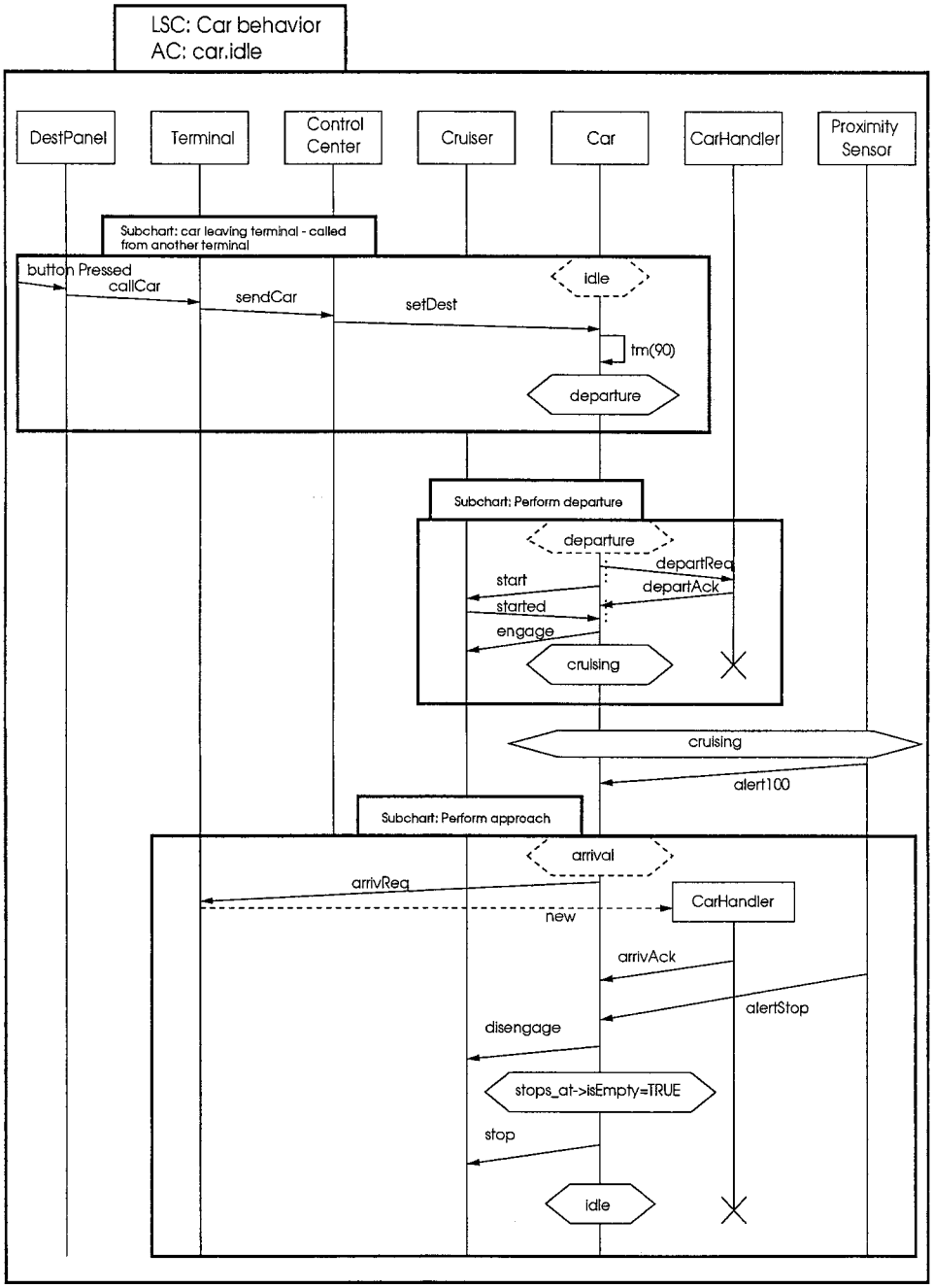

Figure 9. More detailed LSC of rail-car.

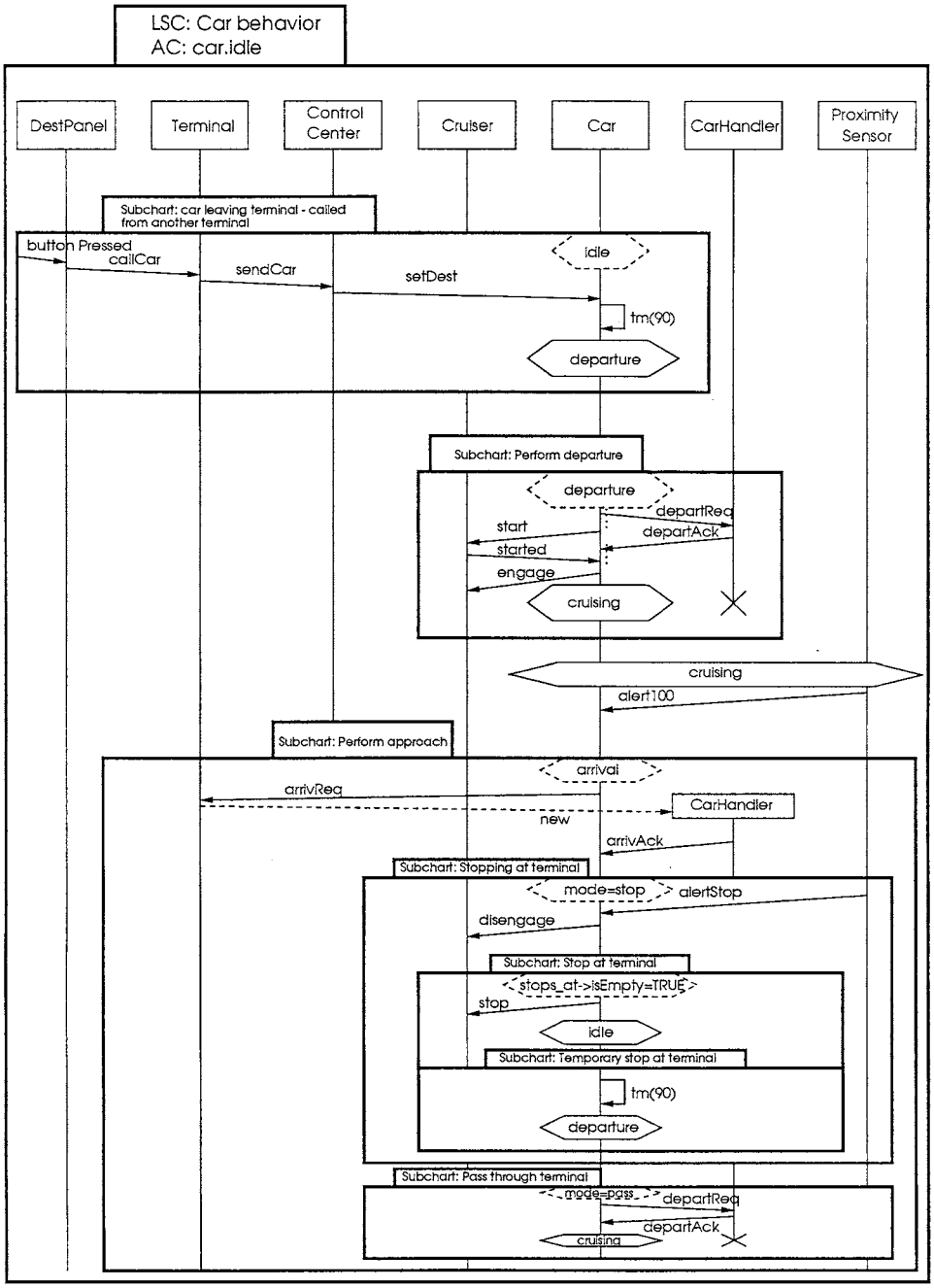

Figure 10. Full LSC of rail-car.

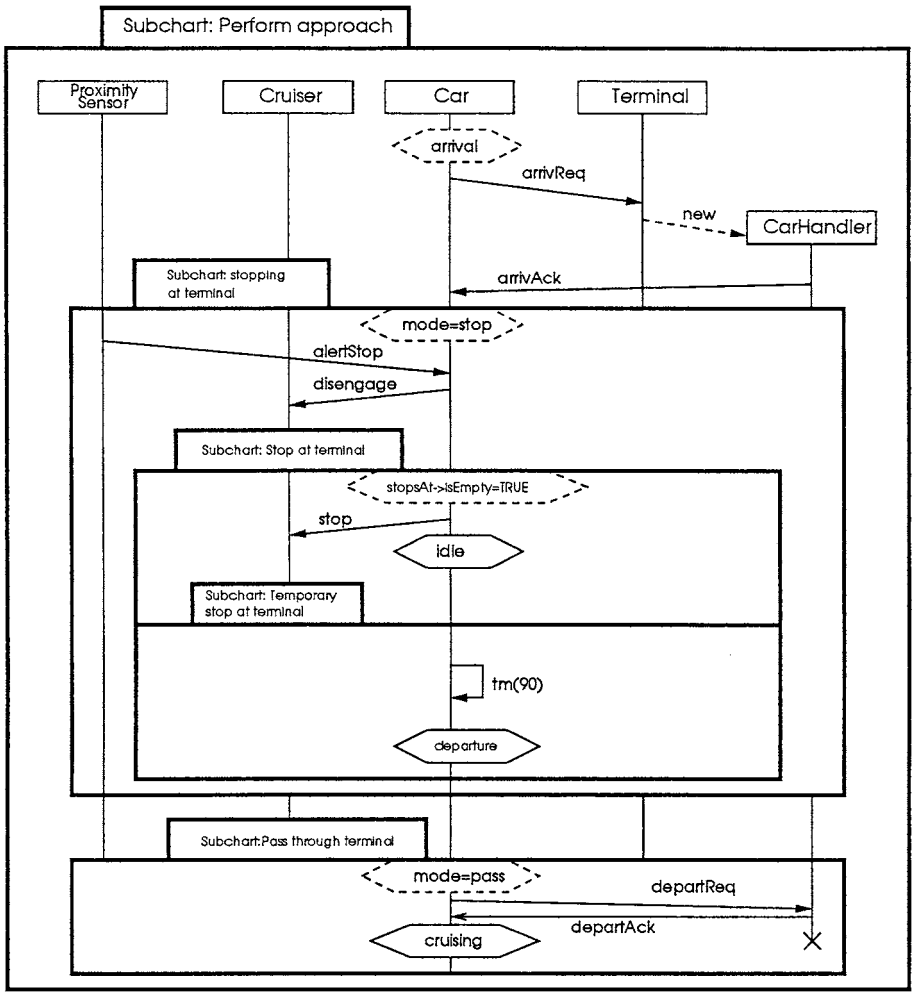

Figure 11. The "Perform approach" subchart.

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 in figure 10; 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 12, 13 and 14 are not subcharts. They are full LSCs, and are presented with dashed borderlines to signify that they are existential. They constitute an alternative way of showing the three possible scenarios of *Perform approach* and hence they do not need to

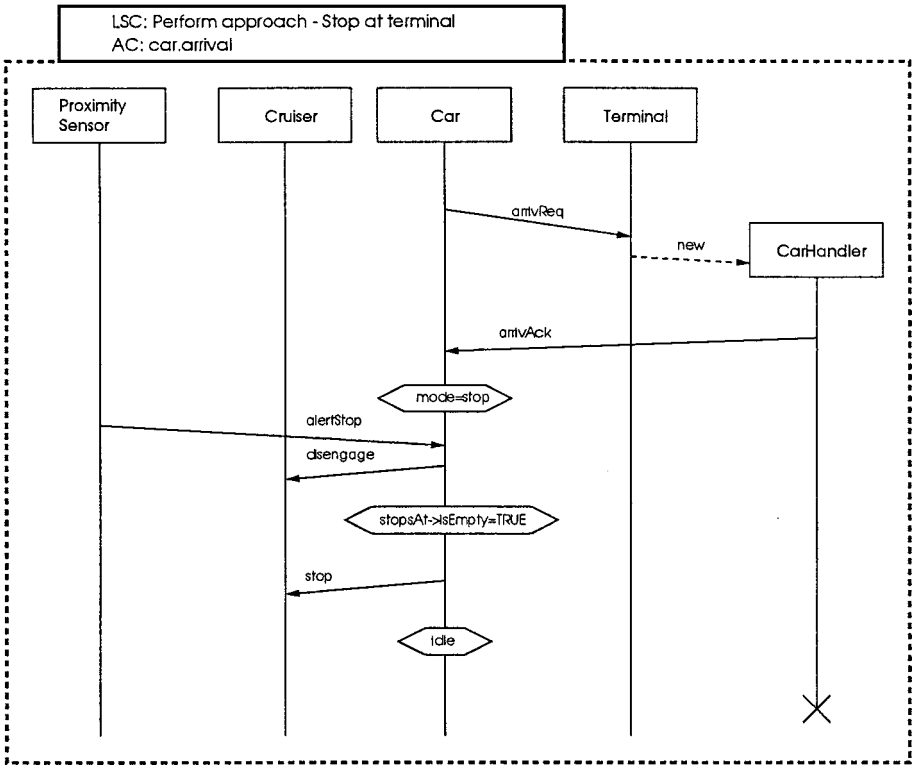

Figure 12. Existential LSC for "Perform approach": Scenario 1.

be satisfied in all runs. In contrast, the main LSC in figures 8, 9 and 10 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 12 to 14, probably early on in the specification process. Later, they would be carefully combined—using the appropriate conditions—into the more informative subchart of figure 11, and then incorporated into the universal parent chart, as in figure 10.

6. Scenarios for applying LSCs

In this section we make an attempt to describe some possible ways (use-cases if you will) of exploiting the expressive power of LSCs in a UML based design process for embedded real-time systems. (One such process is the spiral ROPES one described in [15]). Although we focus here on a UML context, and cast our discussion in terms of a particular case

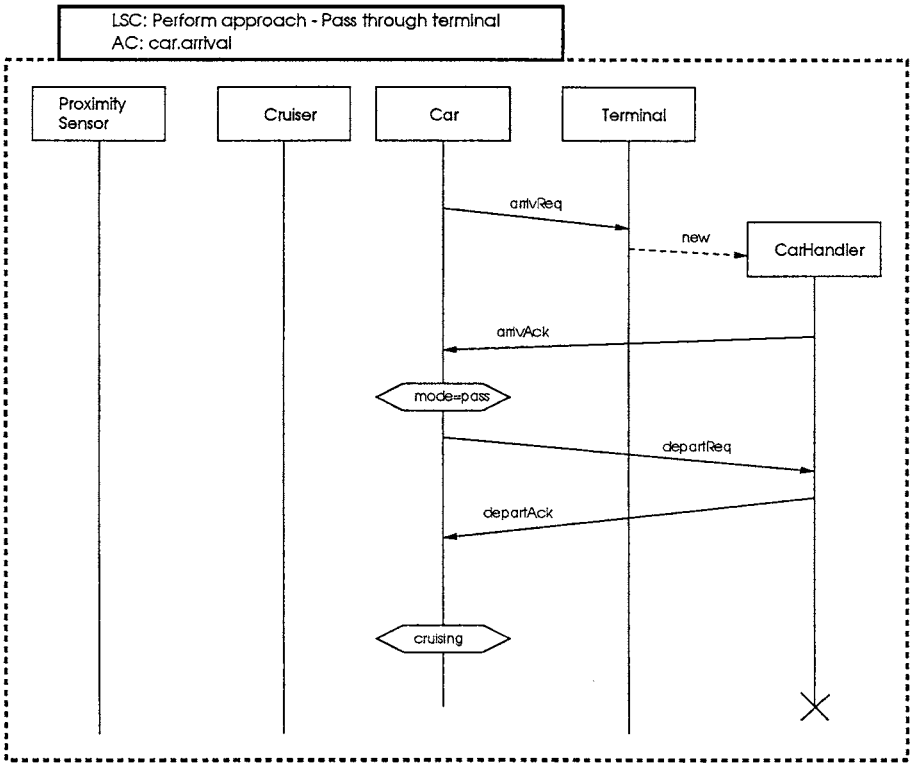

Figure 13. Existential LSC for "Perform approach": Scenario 2.

tool that supports the UML—the Rhapsody tool from I-Logix, Inc. [20]—the methodology described can also be adapted to more traditional design approaches of embedded real-time software, relying on the structural decomposition paradigm (see e.g., [23]). The process described reflects discussions with many actual UML users in application domains such as telecommunication, avionics, automotive and train systems. Nevertheless, here we only demonstrate in basic terms how the expressive power of LSCs can be put to actual use; clearly many variations can be imagined and put to use, depending on the particular application domain and the established design flows of users. In particular, the discussed design flow does not address issues of re-use, although clearly re-use is a must in any real application development.

Any "classical" UML process model, such as ROPES, will propose to give more meat to use-case specifications by elaborating on the dialog between the involved actors using sequence diagrams. While initially designers would see the system as a black box, in the object analysis phase he or she will start to identify supporting objects on a per-use-case basis, hand in hand with elaborating the already captured system-level dialogues into scenarios, capturing the communication patterns between internal objects required to support the given use-case. At this stage, the existential mode of LSCs will be the one typically selected.

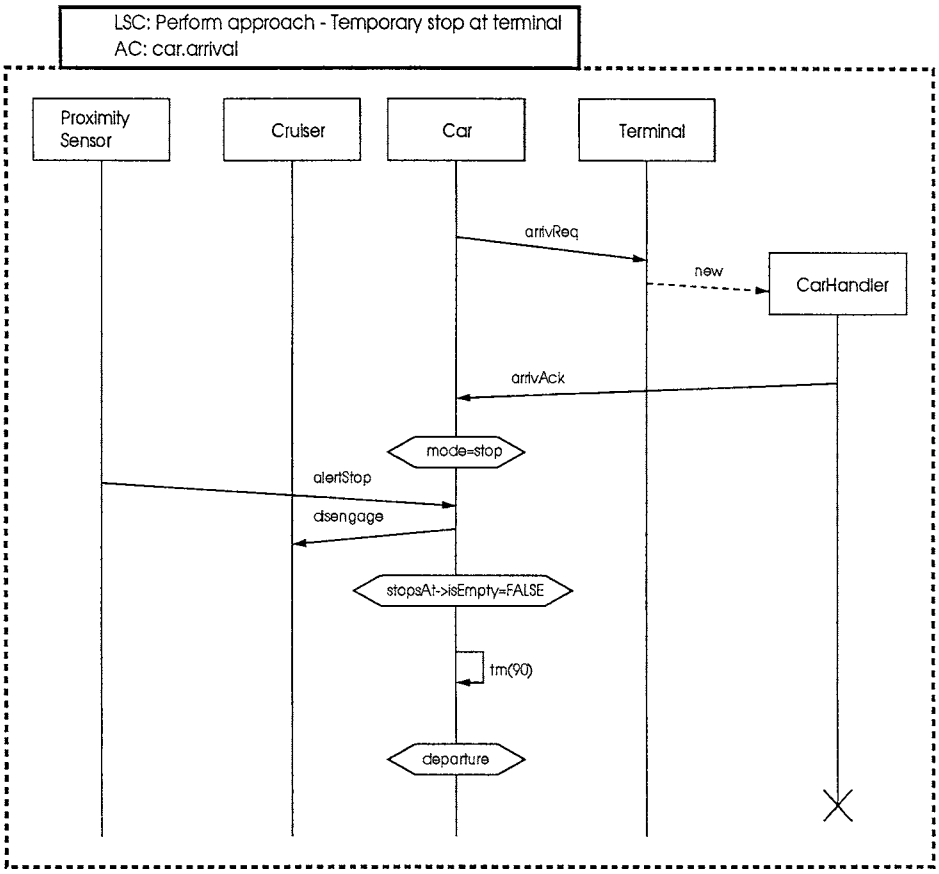

Figure 14. Existential LSC for "Perform approach": Scenario 3.

Activation conditions come into play as a convenient way to capture situations triggering a use-case, such as an event emitted by some actor. As part of the identification of the structure of the object model, methods and events used for inter-object communication are identified, thus gradually building the interface of an LSC. As object behaviour is elaborated, we will be able to enrich LSCs with internal conditions on values of attributes and states of the involved objects, and to refine activation conditions, taking into account expected attribute values and states when entering a given scenario. Internal conditions, i.e., conditions occurring in the chart itself, will be used to both document pre- and post-conditions of method calls, as well as to express local invariants, i.e., conditions on attribute values or states assumed to be true at particular points in time when elaborating the scenario. In both these cases, hot conditions would typically be used, while cold conditions would be employed to express case distinctions.

At this stage, we will have gained sufficient insight into the realization of a use-case to turn most of the charts from existential into universal. This process will be repeated iteratively, until the complete object structure of the essential object model has been defined. We thus

see as a key artifact resulting from the completion of object analysis a set of well defined LSCs documenting all required interobject communication. While UML sequence diagrams would allow one to capture some of the design aspects expressed through LSCs, the key advantage of LSCs here rests in the rigorous definition of the consistency between an LSCs specification and the constructed object model.

We will now capitalize on the semantic rigour of LSCs by describing powerful analysis tools, allowing us to boost the quality of the constructed object model by using increasingly powerful methods of verifying the compliance of the object model against the derived LSC specification.

Current state of the art UML case tools, such as Rhapsody, support validation of the object model by animated simulation and soft panels, providing, in particular, the dynamic creation of scenarios depicting runs of the system. Note that, for lack of a rigorous definition of the link between the object model and the captured scenarios, no compliance check between the generated scenarios and the sequence diagrams documented by the designer can be offered in today's tools: this complex analysis must be carried out on an informal basis by the designer.

The OFFIS group is currently developing a tool suite supporting LSCs, incorporating capabilities for *monitoring*, *test-generation*, and *formal verification*. What is the added value of these in enhancing the quality of the object model? Well, the monitoring tool observes, during simulation, all inter-object communication. Whenever an activation condition of a universal chart is detected, it dynamically creates a watchdog that monitors all traffic between the involved object instances, checking for consistency with all the requirements expressed by its chart. Thus, it will capture all situations in which a visible event or method call was emitted at a point in time violating the ordering constraints. It will also detect if a pre-condition to a method call was violated, it will announce to the user any violation of a local invariant, and will report violation of timing constraints. Once the scenario is exited (through violations of cold conditions) or has been processed completely, the watchdog is killed. All detected violations will be reported, as well as succesfull completion of a chart.

Note, that an activation condition of a chart may become true while a previous incarnation of this chart is still active. This kind of self-activation of charts is an artifact of the formal semantics, but it is highly relevant from an application point of view, where different transactions associated with one and the same use-case may overlap in time. It is thus possible that several watchdogs for one and the same chart are active simultaneously, as required by the formal semantics of live sequence charts, to enforce observation of the requirements of a chart for all its active instances. For existential charts, the tool keeps track of the degree of coverage during a simulation run, depicting which existential charts have been observed so far.

In the above scenario, it is still the user who will need to drive simulation whenever events or methods from actors are involved, or, more generally, when stimuli must be provided from the environment of the currently simulated part of the object model. Using the test-driver tool, such stimuli will be derived automatically from the LSC specification. Similarly to the monitor tool, the test-driver tool detects which scenarios are active and creates objects dedicated to driving the system according to the associated sequence chart. While inheriting all capabilities of the monitor, it will, in addition, detect when environment events or

method calls are enabled and will provide such stimuli to the system based on different heuristics for exploring the space of allowed environment communications. Since stimuli are generated completely automatically, this effectively allows to subject the model to a *stress test*: except for border-case testing, essentially random stimuli are provided, unless environment actions are restricted by the scenarios. We expect this stress test to significantly enhance robustness of the model, since environment stimuli are bound to occur, which are "legal" as far as the LSC specification is concerned, but were never taken into account when specifying object behaviour. Typically, this will induce repair work in the object model, and possibly elaboration of the LSC specification in order to further restrict interactions allowed by the environment.

Finally, adding another level of rigour, we can strengthen the above analysis by applying formal verification techniques. Mathematically, this entails the construction of a formal proof that *all* runs of the object model are compliant with the given scenario specification. From a user's perspective, the effect of carrying out formal verification is essentially the same as running a simulation with the test-driver tool for infinite amount of time: while the test-driver will clealry catch uncovered errors only with a coverage degree that depends on the duration of the simulation, formal verification provides 100% coverage, taking into account *all* legal environment interactions; that is, the only environment interactions not taken into account in formal verification are exactly those prohibited by the scenario definition. We would typically apply formal verification following stress testing. The mathematical theory underlying this verification tool will be discusssed in a follow-up paper currently under preparation.

Taken together, the combination of these techniques will significantly enhance the quality of the essential object model. Having passed a maturity gate defined by the application of monitoring, stress-testing and formal verification will indeed allow us to assert that the functional essence of the application as expressed in the essential object model is correctly realized. We can thus enter the design phase with a high quality essential object model. In particular, this object model can serve as a reference when introducing design related details into the model.

Once we make the transition from a virtual to a real target, where aspects such as real-time scheduling and device drivers come into play, we can still validate conformance of the executable running on the target and the essential object model by test-vectors characterizing the essential object model. The OFFIS group is currently implementing a tool that automatically generates such test-vectors, taking into account both the LSC specification as well as the essential object model. The algorithms underlying automatic test vector generation will drive the object model in a way that all activation conditions of scenarios will be reached. They will also provide coverage with respect to other aspects of the object model, such as states, transitions, guards, etc. Test-vectors not only drive the target code, they also define the expected response of the systems as defined from the essential object model. They can be downloaded onto the target architecture, together with a monitor component, which observes whether the response provided by the target is indeed compliant wih the result expected according to the test vector.

In addition to these methods of using LSCs in an overall design process for systems, we refer the reader to [19], which is orthogonal and complementary to the above discussion. In

it, a high level description is provided of a generic system development process, where LSCs serve as the main language for stating requirements. Besides verification, model execution and debugging, and code generation, two additional techniques are discussed there. One is the automatic *synthesis* of object behavior directly from LSCs (see [22]). This makes it possible to go from requirements to a first-cut implementable model of the system (which can then be refined, the new version verified against the requirements, which themselves can be enriched and refined, etc.).

The second technique is a user-friendly requirement capture method, which is called *play-in scenarios* in [19]. What is proposed there is to play-*in* scenarios, rather than playing them *out*, and this *prior* to building the behavioral model of the system, in order to set up the requirements, perhaps driven by use-cases. This will be done not using visual or other languages, but by working directly opposite such a mock-up of the system's interface, using a highly user-friendly method of 'teaching' your tool about the desired and undesired scenarios. The interactive process will also include means for refining the system's structure as you progress, e.g., forming composite objects and their aggregates and setting up inheriting objects, all reflected in a modified mock-up interface. As the process of playing in the scenario-based requirements continues, the underlying tool will automatically and incrementally generate LSCs (not merely MSCs) that are consistent with these played-in scenarios. Thus, we are automating the construction of rigorous and comprehensive requirements from a friendly, intuitive and user-oriented play-in capability, rather than employing abstract engineer-oriented languages.

The work on play-in scenarios is being carried out at the Weizmann Institute, and will also be described in more detail in a follow-up paper.

Appendix: The semantics of basic charts

The semantics of a basic chart m is defined to consist of all runs compatible with the order induced by m and its annotations. We define these in three stages. Subsection A.1 recalls the definition of the partial-order induced by a basic chart (cf. [35]). Subsection A.2 shows how to construct from m a transition-system called the skeleton-automaton of m. From these we derive the set of runs accepted by a basic LSC in Subsection A.3.

A.1. The partial-order induced by a basic chart

For the sake of completeness we provide here a formal definition of the partial order $\leq_m$ induced by a chart m. As defined in the standard [35], all events along an instance line are ordered, as are any pair of events signaling the sending and receiving of a message (the very same message, of course); the former from top to bottom in the chart, and the latter from sending to receiving. The rest of the ordering is then obtained by taking the transitive closure of these, with the exception that comes from the provision of *coregions* which, as explained in the paper, are captured by the *order* mapping: If $order(m)(\langle i, l\rangle) = false$, then there is *no* ordering relation implied between locations $\langle i, l\rangle$ and $\langle i, l+1\rangle$.

The only non-standard features we have to discuss in this Appendix are extensions of this order to cover *synchronous* messages and *shared conditions*. For synchronous messages, we

want the sender of the message to be blocked until receipt. Hence we include the receive-event location in the preset of the location that is the successor to the send-event. A shared condition induces a synchronization barrier for all instances sharing it: the condition will be evaluated when *all* involved instances have reached the barrier. This decision is tightly coupled with the decision (motivated in Subsection A.2), to allow arbitrary insertion of *local computation steps* between locations visible in the chart, thus relaxing the synchronization between instances as far as local computations are concerned.

A subtle point regards synchronous messages sent to or received from the environment. To keep the definition of the partial-order compile-time computable, we incorporate into our definition the assumption that the environment of a chart will eventually be willing to communicate, and hence infer *no* ordering constraints from such messages. In some situations, e.g., when embedding a subchart into a chart and binding the synchronous message to a matching event in the embedding chart, this assumption may be violated, which can lead to the *blocking* of runs that were allowed when the subchart was considered in isolation. In a follow-up paper we plan to present algorithms that generate *verification conditions* to check for the validity of this (and other) assumptions when embedding subcharts.

We now proceed with the formal definition of semantics for basic LSCs. We first define a binary relation $R(m)$ on $dom(m)$ to be the smallest relation satisfying the following axioms and closed under transitivity and reflexivity:

- order along an instance line:

$$\forall \langle i, l\rangle \in dom(m).\ order(m)(\langle i, l\rangle) = true \Rightarrow \langle i, l\rangle R(m) \langle i, l+1\rangle;$$

- order induced from message sending:

$$\begin{aligned}&\forall m_id \in Message_Id.\ \forall \langle i, l\rangle, \langle i', l'\rangle \in dom(m).\\ &\quad label(m)(\langle i, l\rangle) = \langle _, \langle m_id, _, !\rangle\rangle \wedge\\ &\qquad label(m)(\langle i', l'\rangle) = \langle _, \langle m_id, _, ?\rangle\rangle \Rightarrow \langle i, l\rangle R(m) \langle i', l'\rangle;\end{aligned}$$

- synchronous messages block sender until receipt:

$$\begin{aligned}&\forall m_id \in Message_Id.\ \forall \langle i, l\rangle, \langle i', l'\rangle \in dom(m).\\ &\quad label(m)(\langle i, l\rangle) = \langle _, \langle m_id, synch, !\rangle\rangle\\ &\qquad \wedge\ label(m)(\langle i', l'\rangle) = \langle _, \langle m_id, synch, ?\rangle\rangle \Rightarrow \langle i', l'\rangle R(m)\ \langle i, l+1\rangle;\end{aligned}$$

- shared conditions induce synchronization barrier:

$$\begin{aligned}&\forall c \in Conditions_Id.\ \forall \langle i, l\rangle, \langle i', l'\rangle \in dom(m).\\ &\quad label(m)(\langle i, l\rangle) = \langle _, \langle c, _\rangle\rangle = label(m)(\langle i', l'\rangle) \Rightarrow \langle i, l\rangle R(m) \langle i', l'\rangle.\end{aligned}$$

We say that a chart m is *well-formed* if the relation $R(m)$ is *acyclic*. In the sequel, we assume all charts to be well-formed and use $\leq_m$ to denote the partial order $R(m)$.

We denote the **preset** of a location $\langle i, l\rangle$ containing all elements in the domain of a chart smaller than $\langle i, l\rangle$ by

$$^{\bullet}\langle i, l\rangle = \{\langle i', l'\rangle \in dom(m) \mid \langle i', l'\rangle \leq_m \langle i, l\rangle\}.$$

We denote the partial order induced by the order along an instance line by $\prec_m$, thus $\langle i, l\rangle \prec_m \langle i', l'\rangle$ iff $i = i'$ and $l < l'$.

A **cut** through m is a set c of locations, one for each instance, such that for every location $\langle i, l\rangle$ in c, the preset ${}^\bullet\langle i, l\rangle$ does not contain a location $\langle i', l'\rangle$ such that $\langle j, l_j\rangle \prec_m \langle i', l'\rangle$ for some location $\langle j, l_j\rangle$ in c.

A.2. The skeleton automaton of a basic chart

A skeleton automaton is just a symbolic transition system (STS) in the sense of [14], equipped with a particular state structure induced from the basic LSC. Intuitively, a state in this transition system represents a cut through the partial-order induced by the LSC, annotated by the current valuation of local instance variables and of messages currently sent or received by each instance.

In the definition below, we exploit the property guaranteed by the construction of the partial order, that shared conditions effectively induce "run-time" barriers among the instances sharing the condition. (Recall, however, that local computations are always enabled!) We then evaluate conditions only when all involved instances have reached the barrier. The fact that local computations may arbitrarily interfere with the establishment of shared conditions might look a little unusual. However, one must keep in mind that this "interference" is exactly what is needed in order to "pad" an actual computation as determined by an implementation into the runs accepted by the LSC. In particular, "good" runs that do not include changing of local variables for a sufficient time span are among those accepted by the LSC.

The state space of the STS associated with the basic LSC m is derived from the following (meta)variables, where i is any of the instances referred to in the LSC:

- *i.location* denotes the current location of instance i;
- *i.events* denotes the events currently emitted by i;
  this must be included in $events(i)$, but can also take the value *silent*, representing the situation in which no message is emitted at the current location;
  Note that due to the simultaneous region construct i may emit more than one message;
- $i.v$ the current local value of i's instance variable v;
- *promises* the current set of promises; this variable takes its value in the power-set of $dom(m) \cup \{m_id? \mid m_id \in vis_events(m)\}$;
- *status* can take any of the values *active*, *aborted*, *terminated*.

Initial states of the STS must satisfy the *initialization predicate* $init(m)$, which is the conjunction of the following:

- $i.location = 0$ initially all instances start at location zero;
- $status = active$ initially the system is active;
- $i.events \notin vis_events(m)$ initially no event visible in the chart is present.

Note, that there is *no* restriction on values of variables, nor on the presence or absence of invisible events. The transition relation is partitioned into the following types of moves:

- τ*-steps* perform purely local computations, and are always enabled when the chart is active;
- *i-steps* allow instance i to proceed; this requires the chart to be *active*, and i's next location to be enabled;
- *chaos-steps* may arbitrarily change valuations of variables as well as presence of events; this requires the chart to be in status *terminated*;
- *stutter-steps* perform only "stuttering"; i.e., do not change any of the variables of the STS; this requires the chart to be in status *aborted*.

Our semantics is a pure interleaving one: only a single instance is allowed to proceed at a time, and hence the transition predicate for the global transition relation is just the disjunction of the transition predicates of its partitions.

Figure 5, presented in Section 4, gives a global view of the relation between the status of the basic chart and the allowed transition types. In the case of subcharts, the "terminate" box will include also exiting the subchart with no chaos allowed.

We describe the transition relation of our STS using the concrete syntax suggested in [14], in a self-explanatory imperative style. Following the keyword ***behavior***, the effect of a *single* step of the STS on its (meta-) variables (declared in the section introduced by the keyword ***variables***) is described in an imperative style, making free use of abbreviations (introduced in the section named ***definitions***). Definitions are *evaluated at each step*; in particular, we make heavy use of the fact that non-deterministically chosen values (picked using the ***choose*** expression from some set possibly restricted by a predicate) are drawn anew for each step of the STS.

system *skeleton_automaton*(m) is

variables
$i.location : dom(i)$ ***for all*** $i \in instances(m)$;

the current value of all location pointers of all instances together defines the cut through the LSC: all predecessors up to and including locations in the cut have been visited

$i.events : \wp(events(i)) \cup \{silent\}$ ***for all*** $i \in instances(m)$;
$i.v : type(v)$ ***for all*** $v \in var(i),\ i \in instances(m)$;
$i.blocked :$ ***integer*** ***for all*** $i \in instances(m)$;

if an instance is waiting for the receipt of a synchronous message, it cannot perform local computations; to disallow τ-steps in such an instance, we increment its block-counter whenever a synchronous message is sent the need for a counter arises due to the simultaneous region construct: an instance may send more than one synchronous message and has therefore to keep track of the number of receipts it is awaiting

$promises : \wp(dom(m) \cup \{m_id? \mid m_id \in vis_events(m)\})$;
$status : \{active,\ aborted,\ terminated\}$;

definitions

$enabled(\langle i, l\rangle) = {}^{\bullet}\langle i, l\rangle \subseteq \{\langle i', l'\rangle \in dom(m) \mid l' \leq i'.location\} \wedge l > i.location$
for $\langle i, l\rangle \in dom(m)$;

a location $\langle i, l\rangle$ of instance i is *enabled* iff it has not yet been reached and all its predecessors according to the partial order defined by m have been visited

$active(c) = (\{\langle i, l\rangle \in dom(m) \mid i \in shared(c) \wedge label(\langle i, l\rangle) = \langle -, c\rangle\}$
$= \{\langle i, i.location\rangle \mid i \in shared(c)\})$ ***for*** $c \in Condition_Ids$;

a condition is *active* iff all instances sharing it have reached the location labeled with it; conditions will only be evaluated when they are active

$completed = \forall i \in inst(m).\ i.location = l_max(m, i)$;

a chart has been visited completely once the current cut covers all maximal locations of instances, i.e., the "lowest" locations depicted per instance line; henceforth no further restrictions are implied by the chart

$\langle i, l\rangle$ = ***choose*** $\langle i', l'\rangle \in dom(m)$ ***s.t.*** $enabled(\langle i', l'\rangle)$;

an enabled location is picked, in order to advance the cut by moving to the new enabled location $\langle i, l\rangle$; note that the set of enabled locations may be empty only when the chart is completed

$loc_temp = temp(m)(\langle i, l\rangle)$;

short-hand notation for the temperature of the location to be visited in the next i-step

$transition_type$ = ***choose*** $s_type \in \{\tau\text{-}step, i\text{-}step\}$;

when active, a choice is made between performing local computation steps or i-*steps*; the value of *transition_type* determines the choice for the current step

i_τ = ***choose*** $i \in inst(m)$ ***s.t.*** $i.blocked = 0$;

a candidate instance for a local or terminating computation step is picked, which is not blocked waiting for receipt of synchronous messages

$action$ = ***choose*** $act \in \{update,\ transmission\}$;

a decision is made as to whether the local step will update a variable or involve sending or receiving a message

$\langle msgs, cond\rangle = label(m)(\langle i, l\rangle)$;

$v_\tau =$ ***choose*** $v \in var(i_\tau)$;

the local variable to be updated is picked

$e_\tau =$ ***choose*** $e \in events(i_\tau)$;
$e_local_\tau =$ ***choose*** $e \in events(i_\tau)$ ***s.t.*** $e \in vis_events(m)$;

an event (a *local* event) to be executed in the chaos-step (*local* step) is picked

behavior
case *status* ***of***
active:
if $enabled(\langle i, l\rangle)$ ***and*** *transition_type* = *i-step*
then

an *i-step* is to be performed; the label for each location is of the form $\langle msgs, cond\rangle \in Labels$; first the condition is evaluated

if$(\langle c, b\rangle \in Conditions) \in cond$ ***and*** $active(c)$ ***and not*** b

all instances sharing condition c have reached the location labeled c, hence its condition b (referring to instance variables visible in m and declared in the instances sharing c) is evaluated

then *status* := ***if*** *temp* = *hot* ***then*** *abort* ***else*** *exit* ***fi***

in case of failure the status is updated dependent on the temperature of the condition as explained above

else
$i.location := l$;

if either the condition is not active, or it is active and the condition evaluates to *true*, the cut is advanced by moving the location pointer of instance i to location l; hence, any possible promises of reaching this location are fulfilled, entailing that promise $\langle i, l\rangle$ must be deleted

promises := ***if*** *loc_temp* = *hot*
then $promises \backslash \{\langle i, l\rangle\} \cup \{\langle i, l+1\rangle\}$

promises regarding location $\langle i, l+1\rangle$ must be added if $\langle i, l\rangle$ is *hot*; recall that this implies, that l is not the maximal location of i; hence $\langle i, l+1\rangle$ is indeed a location of i

else $promises \backslash \{\langle i, l\rangle\}$
fi
fi
for all $(\langle m_id, synch_type, dir\rangle \in Messages) \in msgs$ ***do***

evaluate all messages at the current location

```
        if dir =! and temp = hot
            then promises := promises ∪ {⟨m_id, ?⟩} fi
```

> if in addition *m_id* is hot (as indicated by *temp*) and the event represents *sending* a message, the corresponding receive-event is added to the set of promises; note that this is independent of the transmission type *synch_type*.

```
        i.events := i.events ∪ ⟨m_id, dir⟩;
        if synch_type = synch and dir = !
            then i.blocked := i.blocked + 1 fi;
```

> in the case of synchronous message sending, local computations of instance *i* are forbidden until all synchronous messages have been received; note that *i-steps* are forbidden by ordering constraints until the receiver reaches the location labeled with the matching receive event

```
        if synch_type = synch and dir = ?
            then sender(m_id).blocked := sender(m_id).blocked − 1 fi
```

> in the case of synchronous message receipt, the sender's counter is decremented; once it reaches 0 the sender may proceed with local computations

```
    od;
else
    if transition_type = τ-step and i_τ.blocked = 0 then
```

> perform a local action in instance *i_τ* by choosing to emit a local event or to perform an update on an arbitrarily chosen instance variable with an arbitrary matching value; in the case of an update, the instance becomes *silent*

```
        if action = update
            then i_τ.v_τ := choose d ∈ type(v_τ); i_τ.events := silent
            else i_τ.events := e_local_τ fi
            else status := terminated fi
```

> the only remaining alternative is termination of the chart after having reached all maximal instance locations

```
    fi
terminated:
```

> depending on the action, either an arbitrary value is produced for an arbitrarily chosen instance variable or some message is generated for the randomly chosen instance *i_τ*

```
    if action = update
        then i_τ.v_τ := choose d ∈ type(v_τ); i_τ.events := silent
        else i_τ.events := e_τ fi
```

aborted: ***skip***

idle steps are performed, thus maintaining the valuation of all system variables

esac

end system

A.3. Runs of a basic chart

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 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.

We now turn these informal clauses into formal definitions. We assume as given a fixed LSC m and its skeleton automaton $A(m)$, with transition relation $R(m)$ and initialization predicate $init(m)$. We further denote by $events(m)$ (respectively, $var(m)$) the set of all events (respectively, variables) of all instances of m, subsuming its visible events and variables, which together with the system variables $i.location$, $i.blocked$, $status$, and $promises$ constitutes the set $var(A(m))$ of variables of $A(m)$.

Definition

1. A *state* σ of $A(m)$ is a type-preserving valuation of all variables of $A(m)$. We denote the set of all states of $A(m)$ by $\Sigma(A(m))$.
2. A *trace* π of $A(m)$ is an infinite word $\pi = (\pi_n)_{n\in\omega}$ over $\Sigma(A(m))$, s.t. π_0 satisfies $init(m)$. We denote the set of all traces of $A(m)$ by $traces(A(m))$.
3. A trace π is *accepted* (denoted $accepted(\pi)$) *iff*

$$\exists i \in \omega.\, \forall n \geq i.\, ((\pi_n(status) = terminated) \\ \vee (\pi_n(status) = active \wedge \pi_n(promises) = \emptyset)).$$

4. A trace π is *rejected* (denoted $rejected(\pi)$) *iff*

$$\exists i \in \omega.\forall n \geq i.\, ((\pi_n(status) = aborted) \\ \vee (\pi_n(status) = active \wedge \pi_n(promises) \neq \emptyset)).$$

5. Let $V \subseteq var(A(m))$. The projection of a trace $\pi \in traces(A(m))$ on V $(\pi_{|V})$ is obtained from π by simply restricting all states π_n to V.
6. The set of *runs* of an LSC m is defined by

$$runs(m) = \left\{\pi_{|events(m)\cup var(m)} \;\middle|\; \pi \in traces(A(m)) \wedge accepted(\pi)\right\}.$$

Note that the apparent lack of duality in the definition of accepted vs. rejected traces is resolved by the construction of $A(m)$. Indeed, if the status *terminated* is not reached, computation never moves beyond some cut of the LSC, hence only local computation steps may be performed. However, local computation steps cannot resolve promises. Note also that once the LSC has reached status *aborted*, it will never change this status; hence, we could also have phrased this part of the rejection condition by only requiring that *aborted* is reached for some n.

We have thus established the desired connection between a run r produced by some implementation and an LSC m : *r satisfies m iff r is one of the runs accepted by m*.

Definition. Let $V = events(m) \cup var(m)$, let $\Sigma(V)$ be the set of all states over V, and let r be an infinite word over $\Sigma(V)$. We say that r *satisfies* m (denoted $r \models m$) *iff* $r \in runs(m)$.

As mentioned earlier, in this paper we omit the formal semantics of subcharting, as well as that of iteration and branching, and do not deal at all with instance creation and destruction, and timing issues. These we plan to describe in a future paper.

Acknowledgment

We would like to thank Eran Gery for extensive discussions in the initial phases of the work, Jochen Klose and Hartmut Wittke for clarifying and discussing the pre-charts and simultaneous region concepts, and Hillel Kugler for his help in preparing the examples and for comments on an early version. The referees for the *FMOODS'99* conference made several very valuable suggestions.

Notes

1. This problem has recently been addressed for LSCs too; see [22].
2. This paper does not incorporate implementation related concepts, such as states and state-based conditions, into the LSC language. Rather, we stick to message sequencing and use variables extensively.
3. Actually an activation condition can be expressed within a pre-chart by placing a condition at the top of the pre-chart. We nevertheless distinguish activation conditions from pre-charts in order to have an easy to use shorthand notation for the former, which we expect will be the one often used.
4. Here we only consider the activation condition, because we need the skeleton automaton described in Section 4 for a full definition of the pre-charts concept. We postpone the full definition until then. Informally a pre-chart is satisfied if the activation condition is true and the system has shown the communication behavior of the pre-chart. Immediately after completion of the pre-chart the actual LSC is activated.

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, Vol. 1055, Springer-Verlag, 1996, pp. S.35–48.
2. R. Alur, G.J. Holzmann, and D. Peled, "An analyzer for message sequence charts," *Software—Concepts and Tools*, Vol. 17, No. 2, pp. 70–77, 1996.
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*, Vol. 98, No. 2, pp. 142–170, 1992.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill, "Sequential circuit verification using symbolic model checking," in *Proc. 27th ACM/IEEE Design Automation Conference*, 1990, pp. 46–51.
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, 1997.
7. H. Ben-Abdallah and S. Leue, "Timing constraints in message sequence chart specifications," in *Proc. 10th 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, LNCS 1384, pp. 217–231.
9. U. Brockmeyer and G. Wittich, "Real-Time verification of STATEMATE designs," in *Proc. CAV 98*, LNCS 1427, pp. 537–541.
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, B. Josko, H. Hungar, and A. Pnueli, "A compositional real-time semantics for STATEMATE designs," in *Proc. COMPOS'97*. Lecture Notes in Computer Science 1536, pp. 186–238, Springer Verlag, 1998.
13. 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.
14. 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.
15. B.P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison Wesley, Reading, MA, 1999.
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. 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97*. Lecture Notes in Computer Science, Vol. 1231, Springer-Verlag, 1997, pp. 156–168.
17. J. Grabowski, P. Graubmann, and E. Rudolph, "Towards a Petri net based semantics definition for message sequence charts," in O. Ffrgemand and A. Sarma (Eds.), *SDL'93*: *Using Objects, Proc. 6th SDL Forum*, North-Holland, 1993, pp. 179–190.
18. D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, pp. 231–274, 1987.
19. D. Harel, "From play-in scenarios to code: An achievable dream," *Computer*, to appear. Preliminary version in Tom Maibaum (Ed.), *Proc. Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes in Computer Science, Vol. 1783, Springer-Verlag, 2000, pp. 22–34, IEE Computer 34:1, Jan. 2001, pp. 53–60.
20. D. Harel and E. Gery, "Executable object modeling with statecharts," *IEEE Computer*, Vol. 30, No. 7, pp. 31–42, 1997.
21. 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*, Vol. 16, pp. 403–414, 1990.
22. D. Harel and H. Kugler, "Synthesizing object systems from LSC specifications," in *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*. July 2000 Lecture Notes in Computer Science, Springer-Verlag, 2000.
23. D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998.
24. J. Helbig and P. Kelb, "An OBDD representation of statecharts," in *Proc. European Design and Test Conference (EDAC)*, 1994, pp. 142–148.
25. I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.
26. K. Koskimies and E. Makinen, "Automatic synthesis of state machines from trace diagrams," *Software—Practice and Experience*, Vol. 24, No. 7, pp. 643–658, 1994.
27. K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto, "Automated support for modeling OO software," *IEEE Software*, Vol. 15, No. 1, pp. 87–94, 1998.
28. I. Krüger, R. Grosu, P. Scholz, and M. Broy, "From MSCs to Statecharts," in *Proc. DIPES'98*, Kluwer, 1999.
29. P.B. Ladkin and S. Leue, "Interpreting message flow graphs," *Formal Aspects of Computing*, Vol. 7, No. 5, pp. 473–509, 1995.
30. S. Leue, L. Mehrmann, and M. Rezai, "Synthesizing ROOM models from message sequence chart specifications," University of Waterloo Tech. Report 98-06, 1998.
31. R. Schlör, "Symbolic timing diagrams: A visual formalism for model verification," Dissertation, Universität Oldenburg, 1998, March 2001.
32. R. Schlör and W. Damm, "Specification and verification of system level hardware designs using timing diagrams," in *Proc. European Conference on Design Automation*, Paris, France, Feb. 1993, pp. 518–524.

33. 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 1385, Springer-Verlag, 1998, pp. 208–221.
34. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, ITU-TS, Geneva, 1996.
35. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)—Annex B*: *Algebraic Semantics of Message Sequence Charts*, ITU-TS, Geneva, 1995.
36. Various documents on the Unified Modeling Language (UML), available from the Object Management Group (OMG), http://www.omg.org.