Message Sequence Charts

David Harel
Faculty of Mathematics and Computer Science
The Weizmann Institute of Science
Email: `dharel@weizmann.ac.il`

P.S. Thiagarajan
School of Computing
National University of Singapore
Email: `thiagu@comp.nus.edu.sg`

April 8, 2003

Abstract

Message sequence charts (MSCs) constitute an attractive visual formalism that is widely used to capture system requirements during the early design stages in domains such as telecommunication software. A version of MSCs called *sequence diagrams* is one of the behavioral diagram types adopted in the UML. In this chapter we survey MSCs and their extensions. In particular, we discuss *high level* MSCs, which allow MSCs to be combined in various regular ways, and the more recent mechanism of *communicating transaction processes*, which can be used to structure sequence charts to capture system behaviors more directly. We also discuss in some detail *live sequence charts* (LSCs), a multi-modal extension of MSCs with considerably richer expressive power, and the *play-in/out* method that makes it possible to use LSCs directly as an executable specification.

1 Introduction

The language of *message sequence charts* (MSCs) is a popular mechanism for specifying scenarios that describe patterns of interactions between processes or objects. MSCs are particularly useful in the early stages of system development. The language has found its way into many design methodologies, and a variant of it has been made part of the UML notational framework, where it is called *sequence diagrams*. There is also a standard syntax for MSCs that appears as a recommendation of the ITU (previously called the CCITT) [31].

In many object-oriented system development methodologies, the user first specifies the system's use cases and some specific instantiations of each use case are then described using sequence diagrams (MSCs). In a later modeling step, the behavior of a class is described by a state diagram (usually a statechart [9]) that prescribes a behavior for each of the instances of the class. Finally, the objects are implemented as code in a specific programming language. Parts of this design flow can be automated, such as the generation of code from object model diagrams and statecharts, as exemplified in ObjecTime [4] and Rhapsody [11, 26].

In such design flows, the main role of MSCs is to capture system requirements in the form of "good" scenarios that the implemented system should exhibit. Sometimes an MSC is prepared for a "bad" scenario that the implementation should not allow. System requirements captured in this intuitive fashion can serve as a useful interface between the end-users of the system and the system designer. They can also serve as a test bench for validating some aspects of the implementation. A substantial portion of research on MSCs has been driven by this way of using MSCs, with the focus on mechanisms for describing collections of scenarios, techniques for analyzing such collections and relating them to a state-based executable specification.

However, there are several disadvantages to this way of using MSCs in a design methodology. First, MSCs possess a rather weak partial order semantics that makes it impossible to capture interesting behavioral requirements. For instance, one cannot say "if P sends the message M to Q then Q *must* pass on this message to R". In this sense, MSCs are far weaker than formalisms such as temporal logics for capturing requirements and constraints. A second disadvantage is that it is not obvious what the relationship between the MSC requirements and the executable specification (modeling the implementation) should be. Often, it is also very problematic to verify that the desired relationship between requirements and executable specification exists.

To address these concerns, a broad extension to MSCs called *live sequence charts* (LSCs) was proposed in [7]. LSCs can be viewed as a multi-modal version of MSCs, with various means for distinguishing between possible, necessary and forbidden behavior. The expressive power of LSCs is comparable to that of temporal logic and statecharts (except that for pragmatic reasons, LSCs have been limited in the allowed depth of nested alternation of modalities). Using LSCs, one can start to look more seriously at the relationships and possible automated transitions between requirements, as captured by use cases and LSCs, and executable specifications, as captured by, say, statecharts. In addition, very recent *play-in/out* method and the corresponding Play-Engine tool, described in detail in [15], makes it possible to view the LSCs themselves as executable specifications.

In this chapter we present a brief survey of MSCs, covering both these themes. In Section 2, we focus on research related to the use of MSCs and their high-level extensions, HMSCs, to capture system requirements and test cases. As stated already, the main concerns here are the mechanisms for capturing a collection of MSCs and relating such collections to executable specifications. In Section 3, we turn to the use of LSCs to capture behavioral requirements, and in Section 4 we describe the play-in/play-out method and the Play-Engine. In Section 5 we present a related approach, in which a restricted kind of LSCs called *transaction schemes* are used, in conjunction with conventional control flow notations like Petri nets, to formulate executable specifications. Section 6 discusses some ideas about needed extensions.

2 MSCs and HMSCs

Here we will focus on the most basic kind of MSCs; those that model communication through message-passing via reliable FIFOs. In what follows, we will often refer to MSCs as charts. We shall define MSCs as certain kinds of labeled partial orders. We shall use the visual representation shown in the MSC of Figure 1.

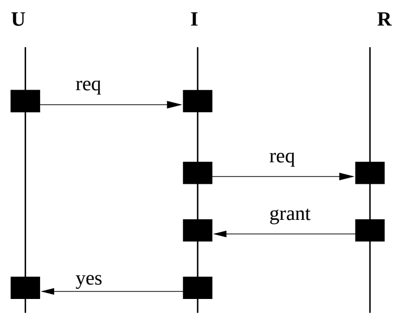

Figure 1: A simple MSC

This chart captures a scenario in which a user (U) send a request to an interface (I) to gain access to a resource R. The interface in turn sends a request to the resource, and receives “grant” as a response, after which it sends “yes” to U. The vertical lines represent the life-lines of the processes taking part in the scenario . As usual, time is assumed to flow downwards along each life-line. The directed arrows going across the life-lines represent the causal link from a send event (the source of the arrow) to the corresponding receive event (the target of the arrow), with the label on the arrow denoting the message being transmitted.

2.1 Basic MSCs

Throughout this section we assume a finite set of processes (agents, objects) $\mathcal{P}$, a finite message alphabet M and a finite set of actions Act. We let p and q range over $\mathcal{P}$, m and m' range over M, and a and b range over Act. The processes in $\mathcal{P}$ communicate with each other by sending and receiving messages taken from M via point-to-point, reliable FIFOs. The processes can also perform internal actions taken from Act, representing computational steps performed by the agents. We shall use Σ_p to denote the set of actions executed by the process p. These are actions of the form $\langle p!q, m\rangle$, $\langle p?q, m \rangle$ and $\langle p, a \rangle$. The communication action $\langle p!q, m \rangle$ stands for p sending the message m to q and $\langle p?q, m \rangle$ stands for p receiving the message m from q. On the other hand, $\langle p, a \rangle$ is an internal action of p, with a being the member of Act being executed. We set $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$.

Turning now to the definition of MSCs, we first define a Σ-labeled poset (partially ordered set) to be a structure $Ch = (E, \leq, \lambda)$, where $(E, \leq)$ is a partially ordered set and $\lambda : E \rightarrow \Sigma$ is a labeling function. For $X \subseteq E$ we define $\downarrow X = \{e' \mid e' \leq e \text{ for some } e \in X\}$. When $X = \{e\}$ is a singleton we shall write $\downarrow(e)$ instead of $\downarrow(\{e\})$. We say that X is *downclosed*, whenever $X = \downarrow(X)$.

For $p \in \mathcal{P}$ and $a \in \Sigma$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$. These are the events in which p takes part. Further,

$$E_{p!q} = \{e \mid e \in E_p \text{ and } \lambda(e) = \langle p!q, m\rangle \text{ for some m in M}\}$$

Similarly,

$$E_{p?q} = \{e \mid e \in E_p \text{ and } \lambda(e) = \langle p?q, m\rangle \text{ for some m in M}\}$$

Finally, for each channel $c = (p, q)$, we define the communication relation R_c via $(e, e') \in R_c$ iff $\mid \downarrow(e) \cap E_{p!q} \mid = \mid \downarrow(e') \cap E_{q?p} \mid$ and $\lambda(e) = \langle p!q, m\rangle$ and $\lambda(e') = \langle q?p, m\rangle$ for some message m.

An MSC over $(\mathcal{P}, M, Act)$ is a Σ-labeled poset $Ch = (E, \leq, \lambda)$ that satisfies:

(1) $\leq_p$ is a linear order for each p, where $\leq_p$ is $\leq$ restricted to $E_p \times E_p$.

(2) Suppose $\lambda(e) = \langle p?q, m\rangle$. Then $\mid \downarrow(e) \cap E_{p?q} \mid=\mid \downarrow(e) \cap E_{q!p} \mid$ and there exists $e' \in \downarrow(e)$ such that $\lambda(e') = \langle q!p, m\rangle$ and $\mid \downarrow(e) \cap E_{q!p} \mid = \mid \downarrow(e') \cap E_{q!p} \mid$.

(3) For every p, q with $p \neq q$, $\mid E_{p?q} \mid=\mid E_{q!p} \mid$.

(4) $\leq= (\leq_{\mathcal{P}} \cup R_{\mathcal{P}})^{\star}$, where $\leq_{\mathcal{P}}= \bigcup_{p\in\mathcal{P}} \leq_p$ and $R_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} R_p$.

The first condition says that all the events that a process takes part in are linearly ordered; each process is a sequential agent. The second condition says that messages must be sent before they can be received. The third condition says that there are no dangling communication edges in an MSC; all sent messages have also been received. The final condition says that the causality relation between the events in an MSC is completely determined by the order in which the events occur within each process and communication relation relating a send-receive pairs. (Many other variants of MSCs with added features can be defined along similar lines.)

In the graphical presentation of the chart $Ch = (E, \leq, \lambda)$, the elements of E_p are arranged along a life-line with the earlier elements appearing above the later elements. Further, a directed arrow labeled with m is drawn from $e \in E_p$ to $e' \in E_q$ provided $\lambda(e) =< p!q, m >$ and $\lambda(e') =< q?p, m >$ and $\mid \downarrow(e) \cap E_{p!q} \mid = \mid \downarrow(e') \cap E_{q?p} \mid$.

2.2 Regular collections of MSCs

A collection of charts constitutes the requirements that an implementation should meet. One key issue involves the specific kinds of chart collections that should be considered as requirement sets, the point being that the requirements themselves could be subjected to analysis and thus help detect design errors at an early stage. Temporal logics, on which model checking is based, suggest that *regular* collections of behavioral objects are ideal candidates for logical analysis [29]. Hence a fruitful notion to look for is that of a regular collection of charts [17], in analogy with the standard notion of regular collections of strings (i.e., regular languages).

Let $Ch = (E, \leq, \lambda)$ be a chart. Each linearization of this Σ-labeled poset is a run of the scenario depicted by the chart and it is a member of Σ^*. Thus the runs of this chart are a subset of Σ^*, which we shall denote by $lin(Ch)$. For instance, the sequence $\langle U!I, req \rangle$ $\langle I?U, req \rangle$ $\langle I!R, req \rangle$ $\langle R?I, req \rangle$ $\langle R!I, grant \rangle$ $\langle I?R, grant \rangle$ $\langle R!U, yes \rangle$ $\langle U?I, yes \rangle$ is in $lin(Ch_1)$ for the chart Ch_1 shown in Figure 1. In fact, it is its only member.

Now let $\mathcal{L}$ be a collection of charts over $(\mathcal{P}, M, Act)$. Then $\mathcal{L}$ is said to be a *regular collection* if $lin(\mathcal{L})$ is a regular subset of Σ^*, where, as might be expected, $lin(\mathcal{L}) = \bigcup\{lin(Ch) \mid Ch \in \mathcal{L}\}$. From now on we will use the term "language" instead of "collection".

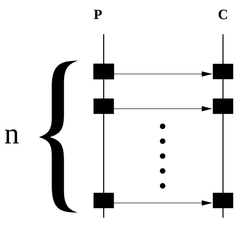

Figure 2: The producer-consumer example

Here now is a simple example of an MSC language that is *not* regular. Assume $\mathcal{P} = \{P, C\}$, $M = \{O\}$ and $Act = \emptyset$. Then the language $\{Ch_1, Ch_2, \ldots, Ch_n \ldots\}$ is not regular, where for each n, the chart Ch_n is as shown in Figure 2. Intuitively the non-regularity follows from the fact that the FIFO size is not bounded in the set of scenarios described by this collection.

In the setting of strings and labeled trees there is a close connection between finite state automata, regular languages and Monadic Second Order (MSO) logics (which naturally contain temporal logics) [29]. A similar relationship exists also for regular chart languages. Here the corresponding automata are called *message passing automata* (MPAs), and they consist of a network of finite state sequential automata, one for each process, that communicate with each other through reliable FIFOs that are *bounded* in length. The component automata can perform three types of transitions; internal, send and receive. In general, these automata will need to use a "broader" alphabet than M. Thus, there will be a finite message alphabet Θ associated with each automaton. Consequently, messages that are sent and received by the component automata will be of the form (m, θ), with $m \in M$ and θ in Θ.

Some of the global control states are designated to be initial and some are designated to be final ones. An initial configuration consists of an initial global control state and empty FIFOs. Similarly, a final configuration consists of a final control state and empty FIFOs. The notion of an accepting run over a string in Σ^* is defined in the usual way, and it is easy to show that any string accepted by an MPA is in $lin(Ch)$ for some chart Ch. Moreover, if one member of $lin(Ch)$ is accepted by the automaton, then all members of $lin(Ch)$ are accepted by it too. In this sense, an MPA accepts a language of charts.

The main result here is that a chart language is regular iff there is an MPA that accepts it. This result, as well as a number of other results concerning regular chart languages, including a logical characterization, can be found in [17].

2.3 High-level MSCs and message sequence graphs

Though MPAs characterize regular MSC languages, they are not an appropriate mechanism for presenting a collection of MSCs. Indeed, the visual appeal of MSCs is lost, especially in terms of capturing system requirements. Rather, it is the role of MPAs and related mechanisms to describe the implementation of a set of requirements. One popular and standard mechanism for directly presenting a collection of MSCs is called *high-level MSCs*, or HMSCs; see [31].

An HMSC is basically a finite state automaton whose states are labeled by MSCs. Consequently one can write out finite specifications involving choice, concatenation and iteration operations over a finite set of seed MSCs. In general, the specification can be hierarchical, in that a state of the automaton can be labeled by an HMSC instead of an MSC. In what follows we shall ignore this feature and instead consider flattened HMSCs, which will be called *message sequence graphs* (MSGs). An example of an MSG is shown in Figure 3.

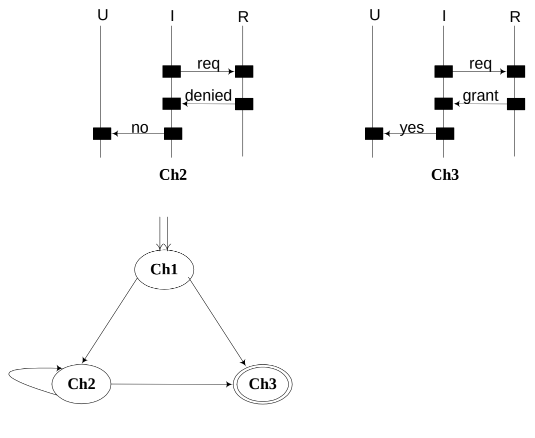

Figure 3: A simple MSG

Intuitively, this chart captures a family of scenarios consisting of a user (U) sending a request to an interface (I) to access a resource (R). The interface queries the resource, and if it gets the response "denied", it sends a "no" to the user and tries again. It keeps trying until it gets the response "granted", at which point it send "yes" to U and the transaction ends.

The edges in an MSG represent the natural operation of chart concatenation. The collection of charts represented by an MSG consists of all those charts obtained by tracing a path in the MSG from an initial control state to a terminal control state and concatenating the MSCs that are encountered along the path. There are two intuitive ways of concatenating charts. In synchronous concatenation, $Ch.Ch'$ denotes a scenario in which *all* the events in Ch must finish before *any* event in Ch' can occur. This is the method that we will encounter in dealing with conditions and precharts in the setting of live sequence charts (LSCs) presented in the next section. The synchronous composition requires a protocol for all the concerned life-lines to synchronize. It rules out the parallelism that could arise had we let the second chart start its operation before the predecessor chart completely finished.

The second way of concatenating charts — which is the one we will consider in this section — is the asynchronous composition. Here the concatenation is carried out at the level of life-lines. More precisely, let $Ch^1 = (E^1, \leq^1, \lambda^1)$ and $Ch^2 = (E^2, \leq^2, \lambda^2)$ be a pair of MSCs. Assume that E^1 and E^2 are disjoint sets. Then $Ch^1 \circ Ch^2$ is the MSC $Ch = (E, \leq, \lambda)$, where:

- $E = E^1 \cup E^2$
- $\lambda(e) = \lambda^1(e)$ $(\lambda^2(e))$ if e is in E^1 (E^2).
- $\leq$ is the least partial ordering relation over E that contains $\leq^1$ and $\leq^2$ and satisfies: If $e \in E^1_p$ and $e' \in E^2_p$ for some p, then $e \leq e'$.

Clearly the asynchronous concatenation of two charts is also a chart. In contrast, the synchronous concatenation of two charts may not result in a chart. For instance, suppose the chart $PC0$ consists of two life-lines P and C, with P having a single send event $\langle P!C, m\rangle$ and C having a single receiving event $\langle C?P, m\rangle$. Consider the synchronous concatenation $PC0.PC1$, where $PC1$ is an isomorphic copy of $PC0$. In the resulting labeled poset, the first receive event of C will be earlier than the second send event of P with nothing in between. As a result, it is not an MSC.

Returning to MSGs, one traces a path from an initial state to a final state in the graph and concatenates *asynchronously* the sequence of MSCs encountered on the way. The resulting MSC is in the language defined by the MSG. Thus for the MSG shown in Figure 3, the chart $Ch1 \circ Ch2 \circ Ch2 \circ Ch3$ is in the language defined by the MSG, while $Ch1 \circ Ch2$ is not.

It turns out that MSGs have a surprising amount of expressive power in terms of the MSC languages they can define. For instance, they can easily define languages that are *not* regular. Both MSGs shown in Figure 4 define languages that are not regular. In these MSGs the one control state is both initial and final.

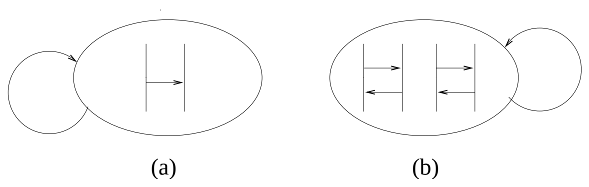

Figure 4: MSGs Generating non-regular MSC languages

An interesting issue here is to figure out when an MSG specifies a regular chart language. In general, this is an undecidable problem [16]. However, there is a sufficient (though not necessary) condition for regularity, that can be checked syntactically, called com-boundedness [3, 16]. To define this property, we need the following notion.

Let $Ch = (E, \leq, \lambda)$ be a chart. The *communication graph* of Ch, denoted by G_{Ch}, is the directed graph with node set $\mathcal{P}$, in which there is an edge from p to q iff there is an event of the form $\langle p!q, m \rangle$ in the chart. We say that Ch is *com-bounded* if G_{Ch} consists of precisely one strongly-connected component and possibly additional isolated nodes. An MSG is com-bounded if the chart obtained by concatenating all the charts encountered while traversing any one of its elementary circuits is com-bounded. For example, the MSG shown in Figure 4(a) is not com-bounded because the communication graph of the chart generated by the only elementary circuit does not have a strongly connected component. As for the MSG of Figure 4(b), it is not com-bounded because the chart generated by its only elementary circuit consists of *two* strongly connected components. A detailed characterization of the regular MSC languages definable by MSGs can be found in [16].

2.4 Other work on MSCs

A number of studies are available that are concerned with individual MSCs in terms of their semantics and properties [27, 25]. Muscholl, Peled, and Su [1] investigate various *matching problems* for MSCs and MSGs, where matching denotes embedding one partial order in another. In [5] Ben-Abdallah and Leue identify and characterize two properties of MSCs that are intuitively undesirable from an implementation point of view and give algorithms to detect such anomalies. The first of them is that of *process divergence*, signifying that the specification allows some process to have an unbounded number of unreceived messages in its buffer. Divergence-freeness is implied by regularity, but does not coincide with it. Figure 4(b) provides a simple counter-example, because the language of this MSG is divergence-free but not regular. The second under-specification detected by Ben-Abdallah and Leue is that of *nonlocal choice*. This denotes the existence of branching choices where different processes have the possibility of taking conflicting routes in the MSG specification.

Alur and Yannakakis [3] consider model checking problems (in a simple linear time framework) for systems modeled as MSGs with respect to various semantics. They show that for synchronous concatenation of MSCs on the paths of the MSG, the problem is co-NP-complete, while it is undecidable in general for asynchronous concatenation. They also define the notion of com-boundedness, which they term boundedness. They then show that the set of linearizations of the MSC language defined by a com-bounded MSG is a regular string language. Thus, in our terminology, com-bounded MSGs define regular chart languages. This notion has also been termed *locally synchronous* by Muscholl and Peled [22], who show that even for com-bounded MSGs two behavioral properties, race condition and confluence, are undecidable. Alur, Etiessami and Yannakaki [2] have also shown that one can not effectively determine whether the MSC language specified by a bounded MSG is weakly realizable. Actually, weak realizability is a closure property; an MSC language L is deemed to be weakly realizable if for each MSC Ch, the following holds: Suppose for each p, there exists an MSC Ch_p in L such that both Ch and Ch_p have identical p-projections. Then Ch itself must be in L. The point is, if the chart languages specified by a bounded MSG has this property then it can be realized as message passing automata in a straightforward fashion whereas the corresponding problem for bounded MSGs in general is much more involved.

A detailed study about realizing HMSCs as Petri nets under different notions of approximation can be found in [6].

3 Live Sequence Charts

MSCs are excellent candidates for specifying *inter-object* behavior in the form of scenarios. However, the implementation of the system will have to be in terms of *intra-object* behavior, which describes — say, in the form of a statechart or an FSM — how an individual object behaves under all possible circumstances. A vital issue here is the relationship between the inter-object specifications given in terms of charts, which we have been referring to as *requirements* and the intra-object specifications that are state-based, and which we shall hence refer to as *executable specifications*. A number of possibilities are:

(1) The scenarios generated by the executable specifications through all the

possible runs should *be included* in the the set of scenarios constituting the requirements.

(2) The scenarios generated by the executable specifications through all the possible runs should *include* the set of scenarios constituting the requirements.

(3) The scenarios generated by the executable specifications through all the possible runs should *equal* the set of scenarios constituting the requirements.

Regardless of the choice made here, requirements given in MSCs still place only a weak *existential* constraint on the executable specifications. In possibility (1), for example, we require that for every scenario Ch in the requirements there *exists* a run of the executable specification that corresponds to Ch. Possibility (2) requires that if Ch is a scenario generated by the executable specification along *some* run, then Ch is in the set of requirements. Possibility (3) is just a conjunction of (1) and (2). Thus, there are no means for requiring that under certain conditions, a particular scenario *must* evolve, or that certain specific scenarios should *never* occur. The language of *live sequence charts* (LSCs; see [7]) has been designed to address this deficiency, yielding a far more powerful formalism for scenario-based behavioral specification. This formalism also helps to establish a tighter and more fruitful relationship between requirements and executable specifications. Indeed, later on we sketch a way to view the LSC requirements themselves as the executable specification.

3.1 The duality of possible and necessary

The basic feature of LSCs is its catering for the dual notions of possible and necessary, or existential and universal. This is done both on the level of an entire chart and on the level of its elements. There are thus two types of charts, *universal* and *existential*. The universal charts are used to specify requirements that *all* the possible system runs must satisfy. A universal chart typically contains a *prechart* followed by a main chart, to capture the requirement that if along any run the scenario depicted in the prechart occurs, the system *must* also execute the main chart. Existential charts specify sample interactions — typically between the system components and the environment — that at least one system run must satisfy. Existential charts can be used to specify system tests and illustrate typical unrestricted runs, whereas universal charts give the “hard” behavioral requirements of the system. So much for an entire chart.

As to elements in the charts, most of these can be *cold* or *hot*, corresponding to possible and necessary, respectively. Thus, LSCs have *cold* and *hot* conditions, which are provisional and mandatory guards. If a cold condition holds during an execution, control passes to the location immediately after the cold condition, and if it is false, the chart context in which this condition occurs is exited and execution may continue. A hot condition, on the other hand, must always be true. If an execution reaches a hot condition that evaluates to false this is a violation of the requirements, and the system should abort. For example, if we form an LSC from a prechart Ch and a main chart consisting of a single *false* hot condition, the semantics is that Ch can never occur. In other words, it is forbidden, an *anti-scenario*. On the other hand cold conditions can be used to program branches and if-then-else constructs. The LSC framework also allows bounded and unbounded loops which, when combined with cold conditions, can be used to construct *while* loops and *repeat-until* loops.

Other elements in an LSC can be cold or hot, such as events and locations along the life-lines, and we shall not dwell on the different semantics of these here. See [7] for details.

Figure 5 shows an example of an LSC that captures the following requirement:

> *Whenever the user dials '2' and then clicks the 'call' button on Phone1, the phone sends the message 'call(2)' to Chan1. If Chan1 is out of order the scenario ends. Otherwise it forwards the message to the switch, which then sends a message 'call(1)' to Chan2. If Chan2 is in order it forwards the message to Phone2. Otherwise, it sends an error message to Phone1.*

In the diagram, the if-like prechart (the top dashed hexagon) is placed at the beginning of the universal chart, and while the main chart contains the consequential part of the requirement. Cold elements are denoted by dashed lines and hot elements by solid lines.

The LSC shown in Figure 5 illustrates many features offered by the LSC formalism. It will be convenient to break these features down into simple units and present them individually.

A *basic universal LSC* (over $(\mathcal{P}, M, Act)$) *with a prechart* is a structure $S = (E, \leq, Pch, \lambda)$ where :

(1) $Pch = (E_{Pch}, \leq_{Pch}, \lambda_{Pch})$ is a chart with $E_{Pch} \cap E = \emptyset$.

(2) $(E, \leq, \lambda)$ is a labeled partial order with $\lambda : E \to \Sigma \cup \{Pch\}$, where Σ is as defined in the previous section w.r.t. $(\mathcal{P}, M, Act)$.

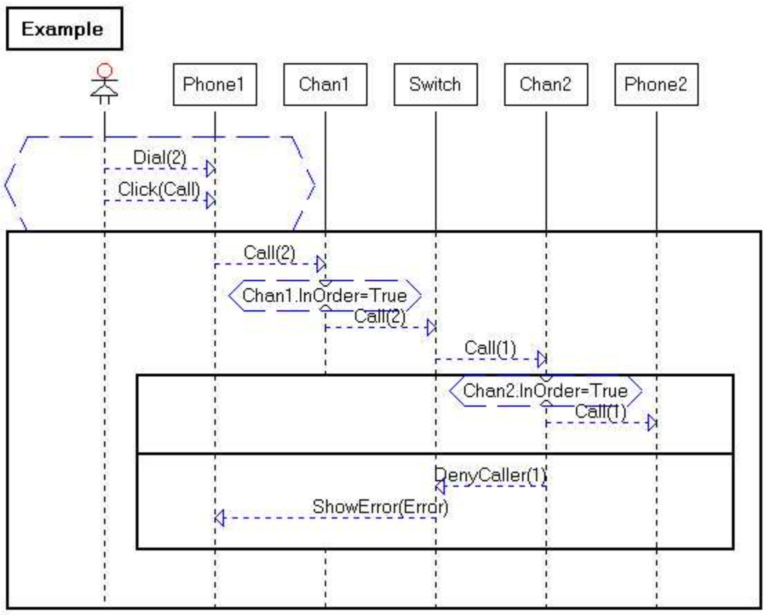

Figure 5: A sample LSC

(3) There is a unique event e_0 which is the least under $\leq$ and $\lambda^{-1}(Pch) = \{e_0\}$.

(4) $Ch = (E', \leq', \lambda')$ is a chart called the *main chart*, where $E' = E - \{e_0\}$ and $\leq'$ is $\leq$ restricted to $E' \times E'$ and λ' is λ restricted E'.

Pch is the prechart serving as the guard of the main chart. In this presentation, the prechart is specified as a refinement of the least event e_0, to capture the idea that the prechart must execute before the main chart can begin. The semantics of this basic LSC is that in any execution of the system, whenever Pch is executed, it must be followed by an execution of the main chart.

We can also use conditions as guards. The conditions are predicates concerning the local states of the processes. In a basic setting, which is the one we will consider here, these predicates will be just propositional formulas constructed out of boolean assertions concerning the local states. We shall accordingly assume a family of pairwise-disjoint sets of atomic propositions $\{AP_p\}_{p \in \mathcal{P}}$, and set $AP = \bigcup_{p \in \mathcal{P}} AP_p$. We let A and B range over AP. Suppose φ is a propositional formula built out of AP. Then $loc(\varphi)$ is the set of processes whose atomic propositions appear in φ. More precisely, $loc(A) = p$ if $A \in AP_p$, $loc(\sim\varphi) = loc(\varphi)$ and $loc(\varphi_1 \vee \varphi_2) = loc(\varphi_1) \cup loc(\varphi_2)$.

A *basic universal LSC* (over $(\mathcal{P}, M, Act)$) *with a pre-condition* is a structure $S = (E, \leq, \varphi, \lambda)$, where :

(1) φ is a boolean formula over AP.

(2) $(E, \leq, \lambda)$ is a labeled partial order, with $\lambda : E \to \Sigma \cup \{\varphi\}$, where Σ is as before.

(3) There is a unique event e_0, which is least under $\leq$ and $\lambda^{-1}(\varphi) = \{e_0\}$.

(4) $Ch = (E', \leq', \lambda')$ is a chart called the *main chart*, where $E' = E - \{e_0\}$ and $\leq'$ is $\leq$ restricted to $E' \times E'$ and λ' is λ restricted E'.

Here again, the idea is that all the processes in $loc(\varphi)$ first synchronize and φ is evaluated. If true, the main chart is executed and if false it is skipped. The semantics of this basic LSC is that, along any execution, if φ holds it must be followed by an execution of the main chart, otherwise there is no constraint.

There are also corresponding versions of basic existential LSCs. For the version with the prechart, the semantics is that there exists an execution along which the prechart is executed followed by an execution of the main chart. This does not add much to the expressive power, except for the requirement that the prechart must finish executing before the main chart can execute. For the version with the pre-condition, the semantics is that there exists an execution along which the guard condition holds and this is then followed by an execution of the main chart.

Finally, we may define LSCs with post-conditions in a similar way. Here the event e_0 labeled with the condition is required to be the *greatest* event under the partial order associated with the LSC. The intended semantics is that along any execution, whenever the main chart of the LSC executes then at the end the post-condition must hold. This gives a different way to specify anti-scenarios: Ch would be taken to be the main chart and the post-condition would be taken to be false. One can also define basic existential LSCs with post-conditions with the obvious semantics.

Note that a "stand-alone" cold condition φ can be obtained by a basic universal LSC whose pre-condition is φ and whose main chart is empty. Similarly, a hot condition φ can be obtained by a universal LSC whose post-condition is φ and whose main chart is empty.

3.2 Control constructs

We can now compose the basic LSCs and the derived cold and hot conditions to construct more complex LSCs. One basic idea would be to allow the main chart of an LSC itself to be an LSC. The second idea would be to asynchronously concatenate LSCs. For instance, in Figure 5, at the outermost level, we have a universal LSC with a prechart. The prechart consists of the scenario where the user dials “2” and clicks “call”. The main chart can be viewed as consisting of a simple chart concatenated with a universal chart with a pre-condition. This pre-condition consists of ‘Chan1-in-Order’ being true. The main chart of this universal chart can be viewed as consisting of a simple chart (Chan1 forwards the call to the switch which in turn forwards the call to Chan2) concatenated with a universal chart, etc. In [7] these combinatory operations were limited, so as not to overwhelm real-world users of LSCs. Thus, only one level of universal/existential alternation was allowed when nesting charts: you may have existential subcharts inside the main chart, but you cannot have universal charts inside those.

Yet another feature of LSCs is the looping construct, which comes in three flavors: The *unbounded* loop, in which a subchart is annotated with the $\star$ symbol, indicating that that the number of times it is executed is not known a priori. Typically, such a loop will contain a cold condition which is used to exit the loop upon becoming false. A *fixed* loop is annotated by a number or an integer variable name, and is executed a fixed number of times, according to the number or the variable’s value. A third kind of loop, the *dynamic* loop, is annotated with a “?”, and its semantics makes sense only in the context of play-out, described later, since the user determines the number of iterations at run time. In Figure 6, we show an LSC containing an unbounded loop.

4 The Play-in/Play-out Approach

Given the expressive power of LSCs, it is tempting to consider using them to capture a *complete* set of requirements. Hence one could start looking seriously at the issue of going from the inter-object style specification, in terms of LSCs, to the intra-object style of description, say, along the lines of statecharts, that is needed for implementation. This is a difficult synthesis problem with a very bad worst case complexity, but efforts are under way to address it in ways that might be come useful in practice; see, e.g., [12]. In this section we discuss an alternative approach, which can be very useful for the design process, and for certain kinds of applications can serve as the final implementation itself. A full-fledged treatment of this approach (which also contains the syntax and operational semantics of LSCs, with several extensions), appears in [15].

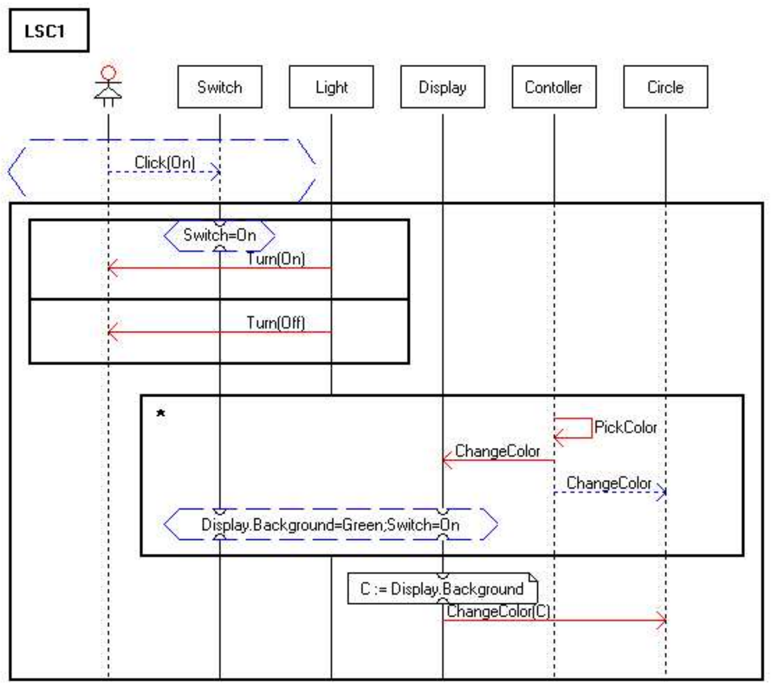

Figure 6: An LSC with a loop

4.1 Playing in Behavior

First we consider the following question: How should the LSC requirements themselves be specified in a practical context? Synthesizing LSCs or similar specifications from use cases is impractical since use cases are informal and abstract. Manually constructing LSCs is possible, but the user would still have to learn how to work with a formal (albeit visual) language, like LSCs, with its syntax and semantics. To make this task easier, the *play-in* approach has been developed [10, 15].

What “play-in” means is that the system developer first builds the GUI of the system with *no* behavior built into it. In systems in which hidden objects plays a role (e.g., a board of an electrical system), the user may build graphical representations of these objects as well. Often one can just use object model diagrams for representing the hidden objects. The user, usually the system developer, “plays” the GUI by clicking on buttons, rotating knobs and sending messages to hidden objects in an intuitive drag-and-drop manner. With the object model diagrams, the developer plays by manipulating the (GUI) of the objects, methods and parameters.

By playing with the GUI, the developer also describes the desired reactions of the system and the conditions that may or must hold. As this is being done, the Play-Engine (the underlying tool built to support the play-in approach; see [15]) continuously constructs LSCs. It queries the application GUI built by the developer for its structure and interacts with it. In the process, it automatically builds and exhibits the appropriate LSCs. An example of a Play-Engine development environment is shown in Figure 7.

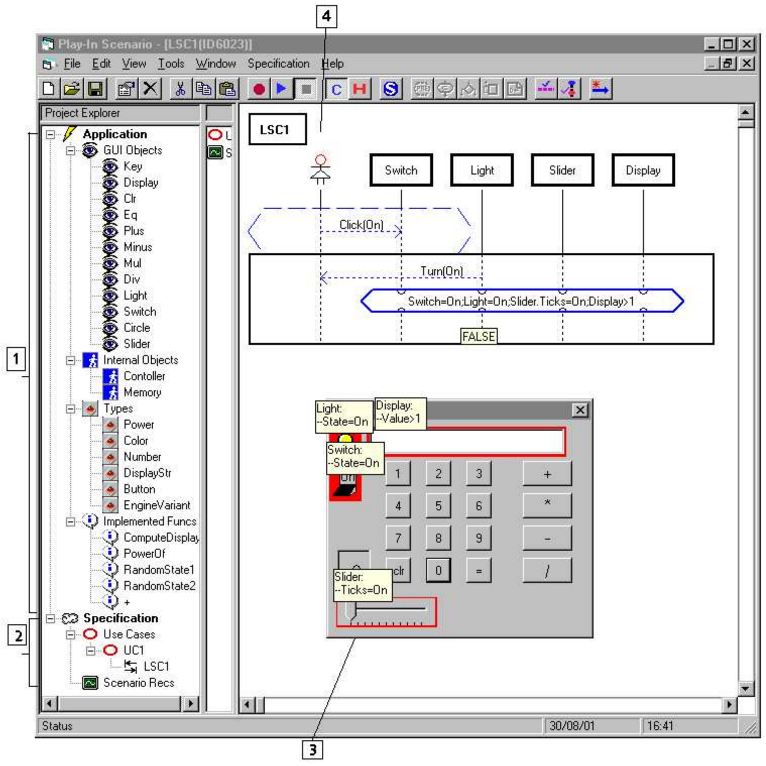

Figure 7: The Play-Engine environment

The main elements of of the environment are numbered in the diagram:

(1) **The Application Section**
This section contains the items defined in the GUI application. The Play-Engine queries the application for this information and displays it in a tree format. The information consists of:

- The GUI objects defined in the application, such as Key, Display, Switch etc. Properties such as value, color, state may be defined for each object.

- The internal objects defined in the application, such as Controller and Memory.
- The types used for capturing the properties of the objects (Colors, Number, etc.).
- The functions implemented by the application that are external to the Engine and are used in the play-in process (e.g., ComputeDisplay updates the value of the display in response to a digit being clicked).

(2) **The Specification Section**
This is the main part of the requirements specification and it consists of the use cases defined by the developer and the LSCs implementing them, which are constructed by the Play-Engine as the behavior is played in.

(3) **The GUI Application**
The GUI application (for example, the calculator) created by the user. It can be constructed by any means as long as it supports the interface required by the Play-Engine. (The Engine actually comes with a GUIEdit package that helps with this.)

(4) **The LSCs**
The LSC that is currently being constructed by the Play-Engine as a part of the play-in process, or the LSC that is currently being executed as a part of the play-out process, as discussed below.

4.2 Play-out

After playing in the specification (or a part of it), it is natural to try and verify that it reflects the user's intention. One way to achieve this would be to construct a prototype implementation and to test it. Instead, in the play-out approach, the power of the interface-based play-in methodology is extended to also validate the requirements. In play-out, the developer simply plays the GUI as he/she would have done when executing the system model but limiting himself/herself to the "end-user" and environment actions only. During this process, the Play-Engine keeps track of the actions and causes other events and actions to occur as dictated by the universal charts in the specification. To achieve this, the Play-Engine interacts with the GUI application and uses it to reflect the system state at any given moment.

This process of the developer operating the GUI application and the Play-Engine reacting according to the specification has the effect of working with a conventional executable model. No code needs to be written in order to play out the requirements, nor is any code generated. One does not have to prepare any kind of intra-object model, as is required in most system development methodologies, like statecharts or some other artifacts that describe the full behavior of each object. It is important to note that the behavior played out is up to the user and need not resemble the behavior as it was played in. Thus the user is not merely tracing the scenarios capturing the requirements. Rather, he/she is freely executing the system that incorporates all the requirements that have been captured. This minimalist system works like a perfect citizen, who does everything by the book; indeed does nothing unless it is called for by the grand "book of rules", but provided it does not contradict anything else written in the book. It is up to the requirements engineers to build the desired liveness properties into the requirements.

Play-out is an iterative process (see [15] for detailed explanations) where each step taken by the user is followed by the Play-Engine computing a sequence of steps called a *superstep*. This superstep is then carried out by the system as the system's response to the step input by the user/developer. The computation of the superstep is nontrivial for several reasons, one of which is that fact that in general more than one superstep may be possible. This is due to the fact at the requirement stage, each scenario just partially orders the events of the scenario. Moreover, the way different charts compose could also be under-specified. In the current implementation, the "naive" play-out process simply picks one way to go, with no backtracking, but a more powerful process has been worked out too, called *smart play-out* (see [13]). This approach, which has been implemented in the Play-Engine for a subset of the LSCs, uses heavy-duty model checking methods to compute a "correct" super step or to declare that none exists (and hence there is a requirements violation).

We conclude this section by pointing to some related pieces of work. First, Magee et. al. [20, 30] present a methodology supported by a tool called LTSA for specifying and analyzing labeled transition systems (LTSs). This tool works with a framework called SceneBeans, yielding a nicely animated executable model. An interesting idea would be to use SceneBeans as an animation engine to describe the behavior of internal (non-GUI) objects in the Play-Engine. Second, Lettrai and Klose [18] present a methodology supported by a tool called TestConductor, which is integrated into I-Logix's tool Rhapsody [26]. The TestConductor is used for monitoring and testing a model using a subset of LSCs. The charts can be monitored in a way that is similar to the way the Play-Engine traces existential charts. In order to be monitored, their charts are transformed into Büchi automata. Their work also briefly mentions the ability to test an implementation using these sequence charts, by generating messages on behalf of the environment (or other un-implemented classes).

5 Communicating Transaction Processes

We now present another to approach to specifying system behavior using sequence charts. It is related to LSC-based executable specifications in the following way: We use simple universal LSCs with pre-conditions and post-conditions. However, the evaluation of the pre-conditions is carried out in an asynchronous manner. Further, the *control flow* between the different LSCs is specified explicitly. This leads to an execution model that is asynchronous and distributed. We shall present the resulting model, called *communicating transaction processes* (CTPs) in an informal fashion. The full details can be found in [28].

Consider a network of communicating sequential processes that synchronize by performing common actions together. The main idea underlying the CTP model is to refine each common action into a transaction scheme. A transaction scheme consists of a set of (universal) charts, each with a pre-condition and a post-condition. Figure 8 shows an example of a CTP, using the familiar Petri net notation to describe the control flow between three processes, I1, B, and I2.

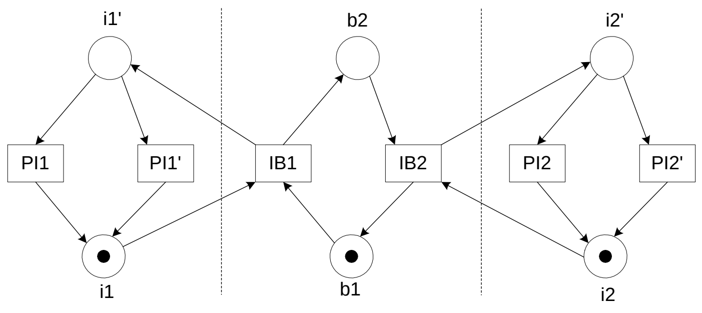

Figure 8: The control flow of a CTP

The Figure captures a simplified and high level protocol that enables the two processors P1 and P2, via their interfaces I1 and I2, to transfer data to a shared memory via the bus B. The transitions of the Petri net are labeled with the names of the transaction schemes, and the schemes are shown separately in Figure 9.

The scheme IB1 captures the interaction between I1 and B. If I1 does not have data to transfer ($\sim$ *data.present*) then the transaction IB11 takes place:

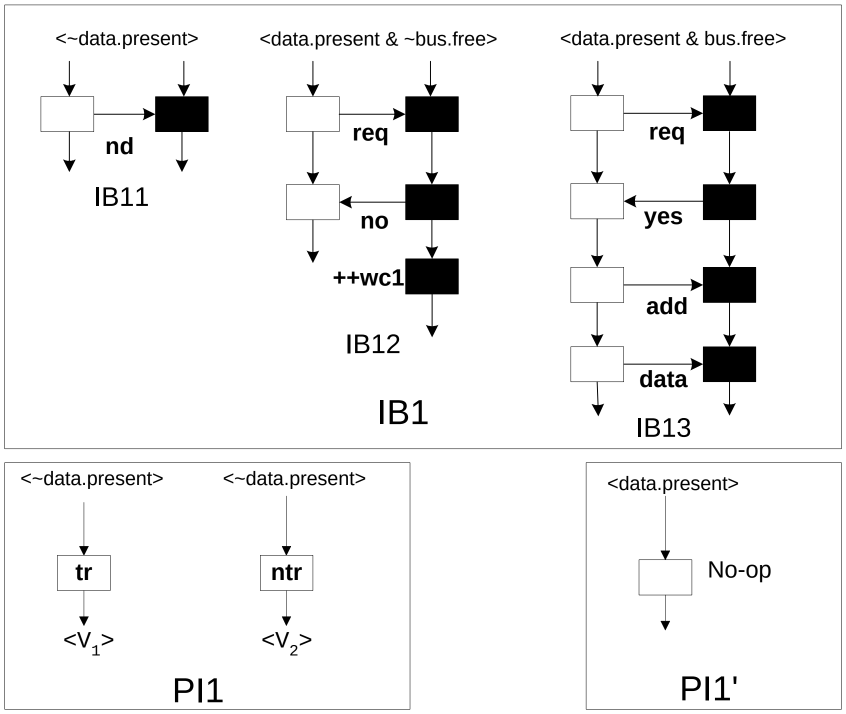

Figure 9: The transaction schemes of the CTP example

I1 informs B that it has no data to transfer (*nd*) and they both proceed to their respective next interactions. If I1 does have data to transfer but B is not free, the transaction IB12 takes place ($data.present \wedge \sim bus.free$): I1 sends a request, B sends back a "no" and increments wc1, its wait-count corresponding to I1. Finally, if B is free and I1 has data to transfer ($data.present \wedge bus.free$), the transaction IB13 takes place.

The transaction scheme IB2 is identical to IB1, except the I1 is replaced by I2 and can be read off in a similar fashion. We shall return to this example later to explain the pre-conditions and post-conditions associated with the individual transactions.

The schemes PI1 and PI1' are really degenerate schemes, where each transaction consists of a single internal action. They represent simple local choices. For example, in PI1, which takes place if I1 is "free" ($\sim data.present$), there are two transactions with identical guards. Each of these transactions consists of a single internal action. The choice between the two internal actions is an external one, that depends on whether or not the processor P1 has data to transfer via the interface. The action *tr* (i.e., the associated transaction) has as a post-condition the I1-valuation V_1, of which *data.present* will be a member. Thus, *tr* represents I1 dequeuing a data-address pair. On the other hand, *ntr* ("no transfer") has the post-condition V_2 which is a I1-valuation of which *data.present* will not be a member. If I1 is not free, it engages in the scheme PI1'. This scheme's transaction consists of a single internal "no-op" action.

In this simple example, when we view PI1 and PI1' as transitions of a Petri net, their sets of output places are identical; it is the set $\{i1\}$. In general, these two sets of output places will be neither singletons nor equal. We could have merged PI1 and PI1' into a single, more complex transaction scheme. Instead, we have decided to remain with two transaction schemes in order to illustrate the possibility of locally branching control flow.

Continuing with the example shown in Figure 8 and elaborated in Figure 9, for most of the transactions we have not shown the post-condition. By convention, it is assumed that in such cases, the post-condition is the same as the condition that held when transaction began to execute. For more precise details, the reader is referred to [28].

The key feature of the CTP model is that it too yields an executable specification. The inter-object interactions are given in terms of the transactions schemes and the control flow is captured via standard notations, such as a Petri net, as illustrated in Figure 8. One can now extract, in a systematic manner, the intra-object behaviors. The main step is to combine the different guarded transactions within a transaction scheme into a single structure consisting of a parallel composition of computation trees: one tree for each process participating in the transaction scheme. The tree will glue together the individual behaviors from the different transactions. Finite labeled *event structures*, again a standard operational model [23], can be conveniently used for representing such a parallel composition of computation trees. For example, in our operational semantics the transaction scheme IP1 will be converted into the labeled event structure shown in Figure 10. In this diagram, in order to avoid clutter, we have not shown the labels of all the events. The labels incorporate information about the guards. As usual, the directed arrows represent the immediate causality relation and the squiggly undirected edges represent the immediate conflict relation [24].

We shall not present here the formal extraction of a labeled event structure from a transaction scheme; the details can be found in [28]. The transition system associated with the event structure captures the execution of the body charts and the setting of the post-conditions, all done in an asynchronous manner. The particular transaction that is chosen for execution will be determined by the valuation holding when control reaches the input places of the transaction scheme. The full operational semantics can now be obtained with the help of the top level Petri net, its initial marking and the

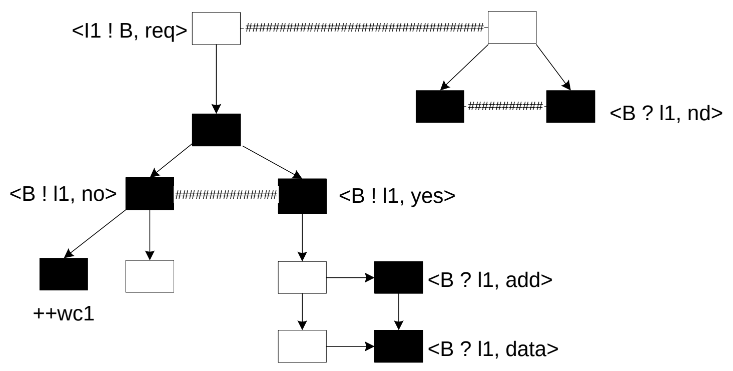

Figure 10: Event structure representation of a scheme

initial valuation of the atomic propositions. Again, the details can be found in [28].

To conclude this section, let us briefly compare the CTP formalism with HMSCs. The main difference is that HMSCs are a mechanism for specifying a *collection* of charts, whereas CTPs constitute a system model in which component interactions are non-atomic and are specified as a guarded choice of charts. Admittedly, one must extract from such transaction schemes an executable thread for each of the participating processes (we achieve this via the event structure semantics of transaction schemes), but this appears to be easier than extracting an executable specification from an HMSC.

6 Some Extensions

As the preceding sections suggest, MSCs can form the basis of powerful requirements and executable specification languages. However, as presented, these languages are still at a low level of abstraction for describing reactive real time embedded systems. A minor limitation, especially following from Section 2, is the rigid nature of the partial order associated with MSCs and the implicit assumption that the interaction between the life-lines is solely in terms of sending and receiving messages via reliable FIFOs. These limitations can be easily removed by adding additional syntactic features, to model synchronization, for example, as is already done in LSCs. One can also relax the definition of the partial order to allow messages to "overtake" each other along the life-line of a receiving agent, etc.

In addition, given the intended domain of applications, we also need major extensions along (at least) two dimensions: object features and timing constraints.

6.1 Object Features

In MSCs, only specific objects have been associated with the life-lines and only fixed and limited information is exchanged between them. As a result, in systems of realistic size, an unacceptably large number of scenarios would have to be specified in order to get a complete description. This problem is yet to be addressed in any serious way for HMSCs and the CTP formalism. However, the LSC language, together with its Play-Engine tool, has been extended in a number of ways to deal with symbolic instances (see [21] and Chapters 7 and 15 of [15]).

One of these extensions is to use *symbolic messages*. In other words, the messages are viewed as variables rather than concrete values. These variables are local to an LSC and an occurrence of the variable may take any value from its type. By using the same variable in the chart, one can specify that the same value will occur in different places in the chart for each specific run (but of course not necessarily the same value for all runs). Thus, for example, we can use a variable "op" with type $\{open, closed\}$ to specify: whenever the user opens or closes the cover of a cell phone (cover(op)) , the antenna also opens or closes (antenna(op)) correspondingly.

The more complex extension described in [21] is to allow for a life-line to be a *symbolic instance* of a class rather than a concrete object in the specification. Figure 11 illustrates the main idea with a very simple example.

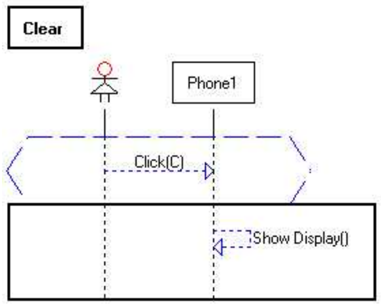

Figure 11: A Generalized Scenario

The life-line denoted by Phone:: represents the *class* Phone rather than a concrete phone. The chart says that its scenario should hold for every concrete phone in the Phone class regardless of the number of phones and whether they were statically declared or created dynamically during the system run. The scenario itself just says that whenever the "C" button is clicked, the display should be cleared.

This extension throws up a host of problems related to the use of variables and expressions to specify symbolic instances, how to bind a symbolic instance to a concrete object (and possibly even to a *set* of objects) during run time. A detailed discussion here will take us too far afield, and the interested reader is referred to [21] and Chapter 15 of [15].

6.2 Timing Constraints

Many reactive systems must explicitly refer to and react to the passage of time. Hence a second type of extension to MSCs is to add timing features. Recommendation Z.120 [31] provides timers for expressing timing constraints within an MSC along a single life-line. The timer can be set to a value, can be reset to 0 and observed for a timeout. Thus, timers can be used to express a minimal delay between two consecutive events or a maximal delay between a sequence of events. Since timers cannot be shared by different life-lines, timing constraints between events lying on different life-lines can not be captured using these timers. A survey of HMSC-related timing issues can be found in [5] and a result relating timed HMSCs to timed automata can be found in [19].

In the UML, timing constraints in sequence diagrams can be represented by drawing rules and timing markers. Horizontally drawn arrows indicate synchronous messages while downward slanted arrows indicate a required delay between the send and receive events of the message. Timing markers are used to specify quantitative timing constraints. They are boolean expressions placed in braces that can constrain particular events or the entire chart. However, neither the syntax nor the semantics of timing markers are precisely defined in the UML.

The LSC formalism has been augmented with an expressive and natural mechanism for specifying timing constraints and this extension is implemented in the Play-Engine [14]; see also Chapter 16 of [15]. The basic idea is to introduce a single clock object (called *Time*) and to use constructs already existing in LSCs, namely assignments and conditions. These, together with their hot and cold variants, allow one to define a rich set of timing constraints. The clock variable Time is assumed to evolve inexorably and its value is readily available to the other life-instances. The *Time* object has one method *Tick*, and *Time* is linked to the host machine's internal clock. For convenience, the standard synchrony hypothesis [8] is assumed in the execution of the Play-Engine.

Suppose we wish to specify the minimum and maximum delay between two consecutive events $e1$ and $e2$ along a life-line. The current time (provided by the clock variable Time) is stored immediately after e, using the variable $T1$. The minimum delay is specified by placing a hot condition of the form $Time > T1 + min\_delay$ just before the second event $e2$. Under the Play-Engine semantics, hot conditions are evaluated "continuously" until they become true. Hence this condition will be advanced only after the required amount of time has passed. If the condition is reached after min_delay units of time, then run will be advanced past this condition immediately. The maximal delay, on the other hand, is specified by placing a hot condition of the form $Time < T1 + max\_delay$ just after $e2$. If the condition is reached before the maximal delay has elapsed, it will evaluate to true and will be advanced immediately. If the condition is reached after the maximal delay has elapsed, then it evaluates to false and since Time can not decrease, this is treated as a constant *false* condition and will be reported as a violation of the requirements. This is illustrated in Figure 12(a) where the minimum and maximum delay between the receive event and the later send event of $O2$ is specified.

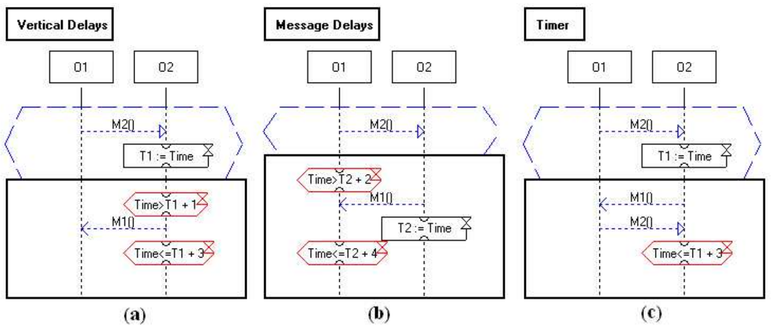

Figure 12: Expressing Timing Constraints

Figure 12(b) shows the specification of the minimal and maximal delays allowed from the moment a message is sent until the moment it is received. It works like the specification of the vertical delay between two consecutive events on a life-line as in Figure 12(a) except that here time is stored on one instance life-line ($O2$) and is checked on another ($O1$). We also note that here the first condition ($Time > T2 + 2$) and the assignment ($T2 = Time$) are not related in the LSC partial order and so the condition may be reached before the assignment. However, the LSC semantics regarding the binding of variables and their subsequent usage will ensure that the condition will not be evaluated before the assignment is performed. Finally, Figure 12(c) illustrates the specification of a timer along a life-line.

Many other means for expressing timing constraints and the techniques for handling them can be found in chapter 16 of [15].

References

[1] Z. Su A. Muscholl, D. Peled. Deciding properties for message sequence charts. In *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures FOSSACS'98, Lecture Notes in Computer Science* ***1378***. Springer-Verlag, 1998.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2001.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory CONCUR'99, Lecture Notes in Computer Science* ***1664***. Springer-Verlag, 1999.

[4] P. Ward B. Selic, G. Gullekson. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[5] H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. In *Proceedings of the Tenth International Conference on Formal Description Techniques, FORTE/PSTV'97*. Chapman & Hall, 1997.

[6] B. Caillaud, P. Darondeau, L. Helouet, and G. Lesventes. HMSCs as partial specifications ... with PNs as completions. In *Modeling and Verification of Parallel Processes 4th Summer School, MOVEP 2000, LNCS 2067*, 2001.

[7] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publications, 1993.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[10] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 2001.

[11] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7), 1997.

[12] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *International Journal on Foundations of Computer Science*, 13(1), 2002.

[13] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2002.

[14] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched lscs. In *Proceedings of the 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS 2002*. ACM Press, 2002.

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

[16] J.G. Hendriksen, M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Message sequence graphs and finitely generated regular MSC languages. In *International Colloquium on Automata, Languages and Programming (ICALP), LNCS 1853*, 2000.

[17] J.G. Hendriksen, M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Regular collections of message sequence charts. In *Mathematical Foundations of Computer Science (MFCS), LNCS 1893*, 2000.

[18] M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time uml models. In *4th Int. Conf. on the Unified Modeling Language, Toronto*, 2001.

[19] P. Lucas. Timed semantics of message sequence charts based on timed automata. In Oded Maler Eugene Asarin and Sergio Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[20] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *22nd Int. Conf. on Soft. Eng. ICSE'00, Limeric, Ireland*, 2000.

[21] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2002.

[22] A. Muscholl and D. Peled. Message sequence graphs and decision problems on mazurkiewicz traces. In *Proceedings of the 24th International Symposium on Mathematical Foundations of Computer Science MFCS'99, Lecture Notes in Computer Science **1672***. Springer-Verlag, 1999.

[23] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science (TCS)*, 13, 1981.

[24] M. Nielsen and P.S. Thiagarajan. Regular event structures and finite petri nets: The conflict-free case. In *Proceedings of the 23rd International Conference on the Applications and Theory of Petri Nets ICATPN 2002, Springer Lecture Notes in Computer Science 2360*. Springer-Verlag, 2002.

[25] S. Leue P. B. Ladkin. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5), 1995.

[26] I-Logix Inc. Products Web Page. `http://www.ilogix.com/fs_prod.htm`.

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

[28] A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *Proceedings of Third International Conference on Applications of Concurrency to System Design ACSD'03 (to appear)*. IEEE Press, 2003.

[29] W. Thomas. Languages, automata, and logic. In A. Salomaa G. Rozenberg, editor, *Handbook of Formal Language Theory, Volume III*. Springer-Verlag, 1997.

[30] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *9th European Software Engineering Conferece and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering ESEC/FSE'01. Vienna, Austria*, 2001.

[31] Z.120. ITU-TS recommendation Z.120: Message Sequence Chart (MSC), 1996.