Multiple Instances and Symbolic Variables in Executable Sequence Charts

Rami Marelly, David Harel & Hillel Kugler
The Weizmann Institute of Science

Technical Report MCS02-05, March 2002
submitted for publication.

Abstract

We extend live sequence charts (LSCs), a highly expressive variant of sequence diagrams, and provide the extension with an executable semantics. The extension involves support for instances that can bind to multiple objects and symbolic variables that can bind to arbitrary values. The result is a powerful executable language for expressing behavioral requirements on the level of inter-object interaction. The extension is implemented in full in our *play-engine* tool, with which one can execute the requirements directly without the need to build or synthesize an intra-object system model. It seems that in addition to many advantages in testing and requirements engineering, for some kinds of systems this could lead to the requirements actually serving as the final implementation.

1 Introduction

Sequence charts (whether MSCs [22] or their UML variant, sequence diagrams [20]) possess a rather weak partial-order semantics that does not make it possible to capture many kinds of behavioral requirements of a system. This is mostly due to the fact that they do not distinguish between possible and mandatory behavior, and are thus far weaker than, e.g., temporal logic or other formal languages for requirements and constraints.

To address this, while remaining within the general spirit of scenario-based visual formalisms, a broad extension was proposed in 1999, called *live sequence charts* (LSCs) [6]. LSCs distinguish between scenarios that *may* happen in the system (existential) from those that *must* happen (universal). They can also specify messages that *may* be received (cold) and ones that *must* (hot). A condition too can be cold, meaning that it *may* be true (otherwise control moves out of the current block or chart), or hot, meaning that it *must* be true (otherwise the system aborts). Moreover, the progress of instances may be defined to be *hot* thus enforcing the instance to progress, or *cold* thus enabling the instance to remain in its location. Among many other things, LSCs can express forbidden behavior ('anti-scenarios').

The expressiveness of the language makes it possible to view LSCs as an executable model, and not only as a transient development product used for verification and documentation. Indeed, in [11] a *play-out* execution mechanism is described. Using play-out, the user can execute requirement specifications given in LSCs directly, without the need to build or synthesize a system model consisting of statecharts or code. The play-out mechanism is one part of a wider methodology, called *play-in/out*; the other part enables scenarios to be 'played-in' directly from a GUI or an object model diagram, using user-friendly and intuitive means. Both the play-in and play-out parts of the methodology are implemented in a tool called the *play-engine*[11]. We should emphasize that the behavior played out need not be the behavior that was played in. The user is not merely tracing scenarios, but is executing the requirements freely, as he/she sees fit.

This makes it possible to utilize LSCs throughout the development cycle. As in many object oriented methodologies, the user first specifies the system's *use cases* [14], and then instantiates these use cases using sequence charts. Since use cases are intended to describe the observable reactions of the system to actions coming from the user or the external environment, universal charts are perfect for the job. Thus, while requirements would be described as universal charts, system tests that are intended to demonstrate provisional behavior under specified conditions can be captured by existential charts. When moving to the design phase, the universal charts can be refined, and be used to model how the objects that constitute the system design interact in order to satisfy the original requirements.

Since the play-engine renders LSC specifications executable at every point along the way, these (refined) universal charts can be executed, and the existential charts can be monitored to check that system tests hold continuously. In a later phase, the behavior of each class can be described as a statechart, or be ultimately implemented as code in a specific programming language. Now, since the LSC specification is executable, it is possible that for some kinds of systems the last phase may be omitted: the LSCs, together with the play-engine acting as a *universal requirements execution machine*, may serve as the final implementation.

All this might sound very nice, but it is of limited value if the LSC requirements themselves can refer only to specific objects and to constant and limited information being passed between them. In such a case, and in order for the specification to be executable as expected, the user would have to specify an unacceptably large number of scenarios.

In this paper, we address this problem and extend the LSCs of [6] with symbolic instances. A symbolic instance, associated with a class rather than with an object, and parameterized by a variable or other expression, may stand for any object that is an instance of the class. We also allow the information passed between the instances to be parameterized, using symbolic variables. A symbolic message may stand for any message of the same kind, with actual values bound to its parameterized variables. In the case where objects can be created and destroyed dynamically and the number of objects is *a priori* unknown, using classes allows specifying requirements that could not have been expressed using concrete objects only. The extension is also useful for specifying parameterized systems, where an actual instantiation of the system has a bounded number of objects, but this number is given as a parameter.

In our setting, the binding of classes to concrete objects may be restricted by conditions that are dynamically evaluated. This allows for one generic scenario to describe the interaction of objects that are associated by dynamically changing relations and are not restricted to a static object model. An LSC specification utilizing both symbolic instances and messages is compact, yet represents a large number of specific scenario realizations.

We provide an operational semantics for the extended language and describe how it is implemented and incorporated into the play-engine's execution mechanism.

We note that even though the extensions and semantics are given for LSCs, they apply in a straightforward manner to weaker scenario languages, such as MSCs and UML sequence diagrams.

2 Executing Behavioral System Requirements

In this section we define our system model and explain what we mean by executing requirements and what is expected from a tool that executes them.

A system includes a set of objects and a set of messages that can be sent and received by them. Some of the messages can be sent by the system's external environment and some may be received by it. We denote by $\mathcal{M}$ the alphabet of messages and by $\mathcal{E}$ the alphabet of system events consisting of sending and receiving these messages.

$$
\begin{aligned}
&\mathcal{S}ys = \langle \mathcal{O}, \mathcal{M} \rangle \\
&\mathcal{O} = \{O_1, O_2, \ldots, O_n\} \\
&\mathcal{M} = \{M_1, M_2, \ldots, M_k\} \\
&\mathcal{E} = \{e_1, e_2, \ldots, e_{2k}\} = \mathcal{M} \times \{Send, Recv\}
\end{aligned}
$$

Behavioral requirements for a reactive system specify constraints on how the system should react with respect to external stimuli such as user actions, events coming from external environment, timing events, etc. In way of executing requirements, we would like to enable the user to specify end-user or environment actions and for him/her to then be able to observe the system responses to these stimuli. The play-out methodology described in [11] adopts the kind of interaction framework present in model execution tools such as Statemate and Rhapsody [13]: the user demonstrates end-user and environment actions by operating the objects in a GUI application and the system responses are reflected in the GUI objects. The external GUI can be replaced or augmented with an object model diagram to show internal object behavior. This mode of action is very intuitive and gives the effect of working with a fully operational system.

One must realize, however, that it is much harder to execute requirements given in an inter-object language, such as sequence diagrams or LSCs, than it is for statecharts or code. The latter are intra-object in nature, providing clear information on each object's reactions to every possible event. In contrast, sequence diagrams or LSCs are scenario-based and do not contain explicit instructions for each object under any set of circumstances. This is the central issue that our play-out mechanism had to address.

Our approach in building the play-engine can be described as the following general set of capabilities that an execution mechanism for requirements should supply:

1. Identify an external stimulus and find all the requirements that should be considered when resolving the system's response to it (e.g., identify the event of flipping a switch and find all the requirements that specify the system's responses to the switch being flipped).

2. Apply the relevant requirements to create the sequence of system reactions. This includes identifying additional requirements that become relevant as the system is responding and applying them too (e.g., after the switch is flipped, send a signal to the controller, which, in turn, may result in sending a signal to the light and turning it on).

3. Identify scenarios that are forbidden according to the requirements, and avoid generating them (e.g., make sure that if the requirements say that a certain signal cannot be sent to the light while it is on, the signal will indeed not be sent when the light is on).

4. In case a forbidden scenario does occur, indicate a violation.

5. Indicate when an existential (provisional) scenario completes successfully.

A requirements execution mechanism can thus be likened to an over-obedient citizen who walks around with the 'Grand Book of Rules' on him at all times. He doesn't lift a finger unless some rule in the book says he has to, and never does anything that violates some other rule. He constantly scans and monitors all rules at all times. Thus, the engine does only those things it is required to do, while avoiding those it is forbidden to do. This is a minimalistic, but completely safe way for a system to behave exactly according to the requirements. To make sure the system doesn't just sit around doing nothing, it is up to the requirement engineers to make sure that liveness properties they want the system to satisfy should be incorporated into the requirements. Clearly, in so acting, nondeterministic choices could arise, and inconsistencies in the requirements could be discovered. More about this later.

3 The LSC Language

Like other scenario-based languages (e.g., MSCs [22] and UML sequence diagrams [20]), LSCs are visual, which appeals to engineers, but they are far more expressive and are thus suitable for specifying the actual behavioral properties of reactive systems [6]. For example, conventional sequence languages mostly specify scenarios that *may* happen during a system run, whereas LSCs can also specify what *must* happen. Precharts in universal charts can specify that *whenever* some behavior occurs, the system is *obligated* to respond in a specific way. Events and conditions can themselves be hot (mandatory) or cold (provisional), which provides considerable additional power. A cold condition at the beginning of a chart, for example, is equivalent to specifying a precondition. A *while* loop can be obtained by placing a cold condition at the beginning of an unbounded loop, while placing it at the end of the loop results in a *repeat-until*. A hot constant *false* condition standing alone in a chart means that the scenarios specified in the prechart are forbidden, thus enabling the user to specify *anti-scenarios* (forbidden ones) as an integral part of the language.

The expressive power of the LSCs language caused us to choose it over far weaker variants of sequence charts. However, the extensions described in this paper are valid for most variants of sequence charts, including, of course, UML's sequence diagrams. There is also an effort underway, inspired by the work on LSCs, to extend UML sequence diagrams for UML 2.0 by universal and hot elements. The ideas in the current paper can be easily applied to that language too.

An LSC specification is defined as:

$$\mathcal{S} = \mathcal{S}_{\mathcal{U}} \cup \mathcal{S}_E$$

where $\mathcal{S}_{\mathcal{U}}$ is a set of universal charts and $\mathcal{S}_E$ is a set of existential charts. Universal charts are used to specify restrictions over all possible system runs. A universal chart is associated with a *prechart* that specifies the scenario(s) which, if successfully executed, forces the system to satisfy the scenario(s) given in the actual chart body. In contrast, an existential chart is required to be satisfied by at least one system run. Existential charts thus do not force the application to behave in a certain way in all cases, but rather state that there is at least one set of circumstances under which a certain behavior occurs. Existential charts can be used to specify system tests, or simply to illustrate longer (non-restricting) scenarios that provide a broader picture of the behavioral possibilities to which the system gives rise.

An LSC L is defined to be:

$$L = \langle I_L, M_L, Cond_L, Sub_L \rangle$$

where I_L is a set of instances, M_L is a set of messages over the alphabet $\mathcal{M}$, $Cond_L$ is a set of conditions (stand-alone guards) and Sub_L is a set of subcharts. Each instance represents an object and each message represents some information sent from one object to another (or to itself). Instances are drawn as vertical lines and messages are drawn as horizontal lines. Time is assumed to go from top to bottom. We define the set of events in L as:

$$E_L = M_L \times \{Send, Recv\} \cup Cond_L \cup Sub_L \times \{Start, End\}$$

That is, an LSC event can be either an actual system event of sending or receiving a message, or it could be one of the acts of evaluating a condition, entering a subchart or exiting it. Every instance line contains *locations*. Each location is an intersection of the instance line with an event from E_L. We denote by l_x^i the x^{th} location of instance I_i, and by $\ell(I)$ the set of locations of instance I. Each location may be *hot* (which the semantics will take to mean that eventual progress of the instance beyond that location is forced), or *cold* (the instance can remain in that location without violating the chart).

We define the function $evnt : \bigcup_{I \in L} \ell(I) \rightarrow E_L$, mapping a location to the event it is associated with, and its inverse $loc : E_L \rightarrow 2^{\bigcup_{I \in L} \ell(I)} = evnt^{-1}$, mapping an event to the set of locations associated with it. Note, that when restricting the domain of loc to $M_L \times \{Send, Recv\}$, loc becomes single-valued.

In the present paper, we focus mainly on instances and messages. For details about conditions, if-then-else constructs, various kinds of loops, and the way these constructs are utilized in our setting, see [11]. For the original definition of LSCs see [6].

Fig. 1 shows an example of a universal LSC, which describes the following requirement:

Whenever the user dials '2' and then clicks the 'Call' button on Phone1, the phone sends the message 'Call(2)' to its channel. If the channel is out of order, the scenario ends. If it is in order, the channel forwards the message to the switch. The switch then sends a message 'Call(1)' to Chan2. If Chan2 is in order it forwards the message to Phone2. Otherwise, it sends a message 'DenyCaller(1)' to the switch, which in turn sends an error message to Phone1.

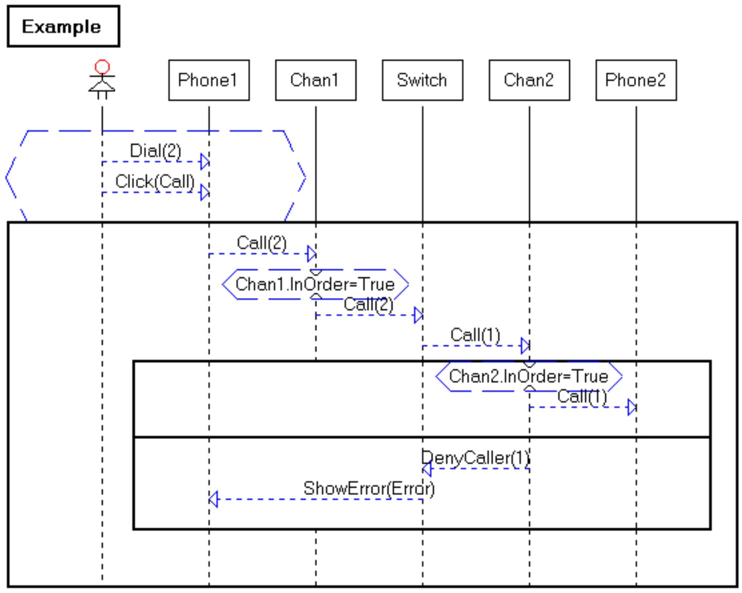

Figure 1: Example of an LSC

Note how the triggering scenario is placed in the prechart (top dashed hexagon), while the chart body contains the consequential part of the requirement. Cold elements are denoted by dashed lines and hot elements by solid ones.

3.1 The semantics of LSCs

In this section we overview the semantics of non-symbolic LSCs and the main principles of our LSC execution mechanism.

Definition 3.1 (LSC cut) *An* LSC cut *is a mapping of every instance to one of its possible locations in the LSC. Cuts are used to indicate the progress line of each chart during the execution. The* temperature *of a cut is hot if at least one of the instances is in a hot location and cold if all the instances are in cold locations.*

Cuts are drawn in the diagrams as thick comb-like broken lines.

Every LSC (the same goes for other variants of sequence diagrams) induces a partial order between locations, which is the central aspect in determining the order of execution. In the following, we shall be using terms related to partial orders and to execution and synchronization to explain the concepts. The LSC partial order $<_L$ induced by a chart L is obtained by the following relations:

Instance line - The locations along a single instance line are ordered top-down, beginning with the prechart start and ending with the chart end. Thus, things higher up are carried out earlier:

$$x < y \Rightarrow l_x^i <_L l_y^i$$

Send-Receive - For an asynchronous message $m \in M_L$, the location of the $\langle m, Send \rangle$ event precedes the location of the $\langle m, Recv \rangle$ event. Thus, an asynchronous message is sent before it is received. For synchronous messages, the two events take place simultaneously:

$$\begin{aligned}&\forall m \in M_L : \\ &(async(m) \Rightarrow loc(\langle m, Send \rangle) <_L loc(\langle m, Recv \rangle)) \wedge \\ &(sync(m) \Rightarrow loc(\langle m, Send \rangle) =_L loc(\langle m, Recv \rangle))\end{aligned}$$

Conditions - All the locations of instances participating in a condition are at the same place in the partial order. Thus, the condition is processed at a single time for all its relevant instances, and they are all synchronized to that event:

$$\forall C \in Cond_L \forall l_x^i, l_y^j \in loc(C) : l_x^i =_L l_y^j$$

Subcharts - All the instances participating in a subchart (e.g., prechart, if-then-else, loop) are synchronized at the beginning of the subchart and at its end. That is, no instance is allowed to move into the subchart before all other instances have arrived at their entry points. And the same applies to ending the subchart; participating instances wait for all others before proceeding to whatever comes after the subchart .

$$\begin{aligned}\forall Sub \in Sub_L \qquad &\forall l_x^i, l_y^j \in loc((Sub, Start)) : l_x^i =_L l_y^j \\ &\forall l_x^i, l_y^j \in loc((Sub, End)) : l_x^i =_L l_y^j\end{aligned}$$

We extend the partial order $<_L$ to events in the following way: $e' <_L e$ if $\exists l \in loc(e), \exists l' \in loc(e')$ $s.t.$ $l' <_L l$.

Definition 3.2 (minimal event in a chart) *An event e is* minimal *in a chart L if there is no event e' in L such that $e' <_L e$.*

Minimal events are important in our executability approach: whenever an event e occurs, all charts that feature e as a minimal event in their prechart are activated and start being monitored to see if their prechart completes successfully.

Definition 3.3 (enabled event) *An event e is* enabled *with respect to a cut C if the location in C of every instance participating in the event e is the one exactly prior to e.*[1]

At each step of executing the specification the engine chooses from among the enabled events the one it will execute next.

Definition 3.4 (violating event) *A system event e* violates *a chart L in a cut C if $e \in M_L \times \{Send, Recv\}$ but e is not enabled with respect to C.*

[1]Usually, there will be only one such instance, but conditions and subcharts may have several participating instances.

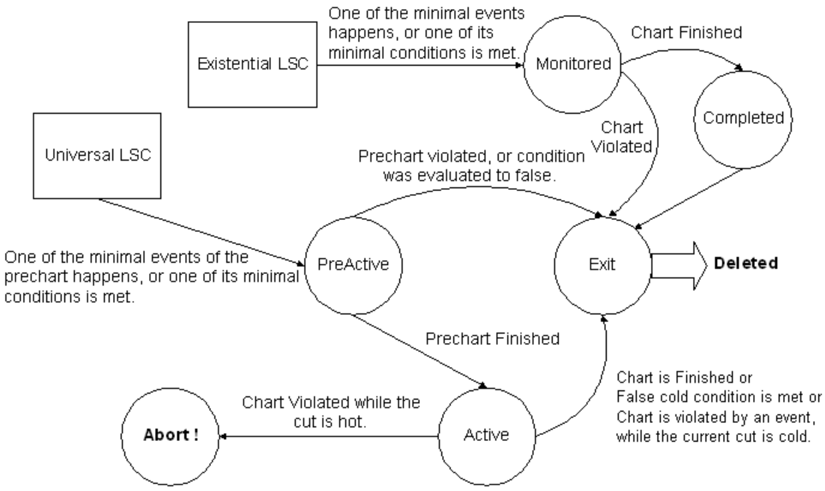

Figure 2: The life cycle of an LSC live copy

Before an event e that is enabled in chart L is chosen to be executed, the execution engine verifies that it does not violate any other chart. Note that an event that does not appear in chart L can occur during the execution of that chart without causing any violation.

A single universal chart may become activated (i.e., its prechart is successfully completed) several times during a system run. Some of these activations might overlap, resulting in a situation where there are several copies of the same chart active simultaneously. In order to correctly identify the activation of universal charts, there is also a need to have several copies of the prechart (each representing a different tracking status) monitored at the same time. The following notion will be instrumental for this:

Definition 3.5 (LSC live copy) *Given an LSC L, a* live copy *of L, denoted by C_L, is defined as:*

$$C_L = \langle L, M, Cut \rangle,$$

where L is a copy of the original chart, $M \in \{PreActive,\ Active,\ Monitored\}$ is the execution mode of this copy, and Cut is some legal cut of L representing the current location of the instances of L in this particular copy.

The general life cycle of an LSC live copy is illustrated in Fig. 2.

Having all these definitions in mind, our execution mechanism works in iterations, each of which consists of a *step*, which is an event initiated by the user, followed by a *super-step*, which is a sequence of events that are selected from the set of enabled events that are not violating. As this is going on, minimal events are identified and new LSC copies are created. The super-step ends when there are no more enabled events that can be carried out. For a more detailed description of our play-out execution mechanism, see [11].

It is now clear that executing requirements given as LSCs has a lot to do with event matching and unifying. For example, when an event e occurs we have to find all the matching events in the set of LSCs in order to create new copies of them and monitor their execution too. Matching events must also be found in copies that are already active, both to detect violations and in order for enabled matching events to be propagated simultaneously in their respective charts. From now on, we shall use the term *unifiable events* instead of matching events, since as we will show, the algorithms for matching events are based mainly on classical unification [18].

Definition 3.6 (level$_0$-unifiable events) *Two events $e = \langle m, t\rangle$ and $e' = \langle m', t'\rangle$, with $t, t' \in \{Send, Recv\}$ are* level$_0$-unifiable *if $t = t'$, $m = m'$, $sender(m) = sender(m')$ and $receiver(m) = receiver(m')$.*

4 Symbolic Messages and Assignments

4.1 Symbolic messages

Often it is natural to specify a small number of sample cases that represent more general scenarios. For example, we might describe a scenario in a calculator where pressing 9, + and 7 in that order (prechart) causes 16 to be displayed as a result (chart body). We would then like to generalize this scenario and show it in the chart as a sequence in which "$X1$", "+" and "$X2$" are pressed in order, and the result is shown to be "$X1 + X2$".

To enable such generic scenarios we extend the definition of a message to contain formal parameters. Now, instead of a set of constant messages:

$$\mathcal{M} = \{m_1, \ldots, m_k\}$$

we have a set of symbolic messages, each with (zero or more) parameter variables, where each variable x_i is defined over a type D_i from the application domain:

$$\mathcal{M} = \{m_1(x_1, \ldots, x_{n_1}), \ldots, m_k(x_1, \ldots, x_{n_k})\}$$

An occurrence of a variable in an LSC, means that it may take on any value from the variable's type. Using the same variable in different places in a chart allows the user to specify that the same value will occur in these places in a specific run (not necessarily the same value in all runs).

Fig. 3(a) shows a simple example of using variables to denote the link between the state of a switch, which may be changed by the user, and the state of a light which is required to be the same as the state of the switch; when one is *On* the other is too, and the same for *Off*. In this particular case, the variable *Xpower* is of type *Power* which is an enumeration containing the values *On* and *Off*.

A careful look at Fig. 3(a) reveals an interesting issue concerning the partial order defined by the chart. The chart seems to say that "the user changes the state of the switch to *On* or *Off*, and the light then changes its state to be the same as that of the switch". However, the partial order of LSCs does not restrict the clicking of the switch to come before the light changing its state. This is a problem, since we really want the first to happen before the second. This problem is often solved by introducing a control object that receives the state from the switch and sends it to the light (Fig. 3(b)), thus restricting the partial order, as required.

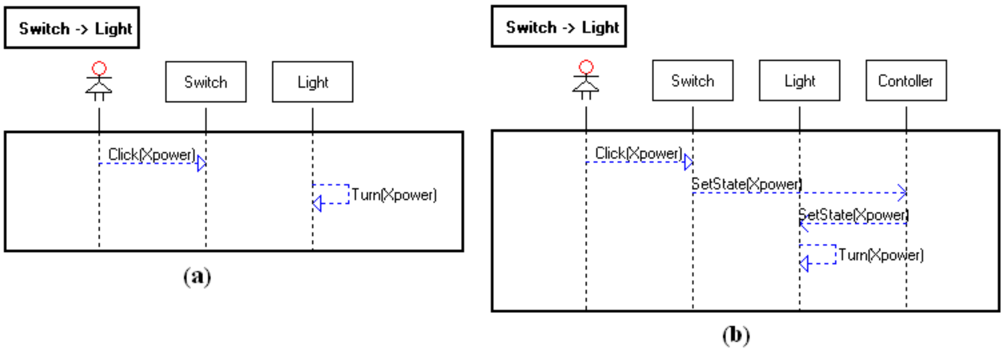

Figure 3: The effects of variables on LSCs partial order

However, we most often wish to remain in a higher, more abstract, level of the specification and do not want to get into the details of how information about the value of the variable is transferred. Yet, on the other hand we do want some initial event to determine the value of the variable, so that subsequent usages of the same variable in different places will have the same value. We make this possible by a slight extension of the partial order $<_L$ induced by an LSC L.

4.2 Enriching the partial order

We make use of the vertical placing of locations that are not on the same instance line. Imagine a vertical line T aligned with an LSC L. Consider the ordering of locations from top to bottom, as projected on T. See Fig. 4, in which locations L_1 and L_2 are not ordered by $<_L$ but are vertically ordered; we might write $L_1 <_V L_2$.

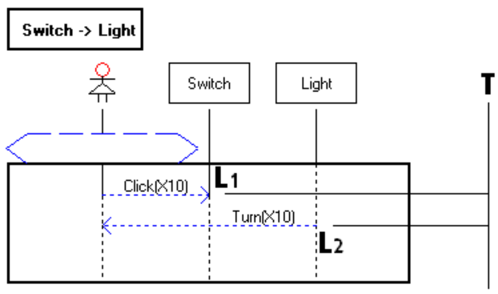

Figure 4: Vertical order: $L_1 <_V L_2$

The partial order $<_L$ is now extended to handle the order induced by variables. We say that l_x^i is the *first* location in L to use variable V, if $evnt(l_x^i) = \langle m, Send \rangle$, and with m having V as one of its parameters, and such that for any other location l_y^j of an event e' that uses V, either $l_x^i <_L l_y^j$ or $l_x^i <_V l_y^j$. Now, for every location l_x^i which is the first in L to use the variable V, and every other location l_y^j in L that uses the same variable, we add the relation $l_x^i <_L l_y^j$ to the partial order $<_L$.

Thus, the extension causes the first occurrence of a variable to come before all the others, but no new order is imposed on the subsequent occurrences. It is very natural for users building LSCs to specify dependent events in the order they occur, and to cause this order to be reflected in the geometry of their LSCs (even though the official partial order is not necessarily affected by this order). While we do not go as far as to attach significance to all vertical ordering, we do think that the first occurrence of each variable is a significant part of the way the user chose to enter the information, taken to mean "this is the first occurrence of this symbolic value, and the value given here will impact later occurrences". By the way, if the requirements are *played in* [11], events that are played in first will appear higher up in the generated LSC anyway, which is consistent with this philosophy.

Having said all this, we should note that this decision is not sacred, and was made for convenience. We could have required the user to specify separately which of the minimal occurrences of each variable is the dominating one for the purpose of receiving and spreading around a value.

4.3 Assignments

Most sequence diagram languages, including the original definition of LSCs, seem to lack the capability of referring to values of system properties after they are set. In [11] a new *assignment* construct was proposed as an extension to pure LSCs.

Assignments are local to a chart, and can be used by the user to save values of the properties of objects, or of functions applied to variables that hold such values. The assigned-to variable stores the value for later use in the LSC. The expression on the right hand side may be a constant value, a reference to a property of some object (this is the typical usage), or a function applied to some predefined variables. Each assignment may have several participating objects, which, as in conditions, synchronize at the location of the assignment. In contrast to the system's state variables, which may be used in several charts, the locality of an assignment variable means that it can be used in the containing chart only.

An assignment is also considered an event for the purpose of finding the first event that uses a variable. Hence, the user may specify that a variable V is assigned some value, and from then on V can be used by other objects as a parameter in messages being sent and received. Assignments, although a simple construct, are very useful when dealing with symbolic charts, as we will show later on.

4.4 Message unification

Now that messages can be symbolic, we can redefine event unification.

Definition 4.1 (bound and free variable) *A variable at a particular location in an LSC is said to be* bound *if it has been assigned a value. It is* free *if it is not bound. The 'value' of a free variable is denoted* $\perp$.

Definition 4.2 (bound and free messages) *A message is said to be* bound *if all its variables are bound. It is* free *if it is not bound.*

We shall use simple predicate notation for notions like free and bound, as in $free(x)$.

Definition 4.3 (level$_1$-unifiable events) *Two events,* $e = \langle m(x_1, \ldots, x_n), t\rangle$ *and* $e' = \langle m'(y_1, \ldots, y_n), t'\rangle$*, are* level$_1$-unifiable *if* $\langle m, t\rangle$ *and* $\langle m', t'\rangle$ *are* level$_0$-Unifiable *and:*

$$\forall i \in \{1..n\} : (value(x_i) = value(y_i)) \vee free(x_i) \vee free(y_i)$$

Here is how the unification of two level$_1$-unifiable events works: If x_i is bound and y_i is free, than after the events are unified y_i will be bound to the value of x_i. The same for the case where y_i is bound and x_i is free. If both x_i and y_i are free, they are connected in a connection list. Later, when one of the variables in the connection list is bound to some value, all other variables in the connection list will be bound to the same value.

As a simple example, suppose one chart contains an event $e = \langle m(x), Send\rangle$, and the sending is from O_1 to O_2, a second chart contains $e' = \langle m(y), Send\rangle$, also from O_1 to O_2. Now, suppose that the event of O_1 sending $m(3)$ to O_2 has occurred. There are a number of cases to consider:

- If both e and e' are enabled in their charts, and x and y are free, we would like to bind x and y to 3 and advance both charts.
- Suppose y is already bound to the value 4. Clearly, after binding x to 3, the messages are not the same and therefore cannot be unified. In this case, e will be advanced and e' will neither be advanced nor be considered violating.
- The last and most interesting case is when y is free and e' is currently not enabled. There are really two possibilities we could adopt here. The first is to succeed in unifying e and e', thus advancing the first chart and causing violation of the second, and the second is to fail in the unification, thus advancing the first chart and leaving the second as is. The first approach maintains that if it is possible for an event to cause a violation it should be handled as if it indeed will, and the second says that if it is possible that an event will not cause a violation, it should be handled as if it won't. Our experience with several examples shows that the second approach is significantly more practical. For example, suppose there is a system with a display that shows several warnings while running, using messages of the form "show(x_i)" where the x_i's are assigned values during the execution. These messages are spread throughout several charts and should not necessarily be synchronized with each other. If the first approach is taken, whenever the display tries to show something as dictated by one chart, it will cause a violation of other charts. This limits the specification in a way that is very difficult to overcome.

We adopt the second approach, and thus distinguish between *positive* unification, in which we allow variables to be bound, and *negative* unification, in which variable binding is forbidden. Positive unification is used to find minimal events in precharts that will cause the activation of new LSC copies and also to find events in different active charts that should be propagated simultaneously. Negative unification is used to find violating events in active LSCs, so that only if a violating event is already bound to a violating value will it cause a violation. If the event is not yet bound, it means that we currently do not know that it will cause a violation, and we therefore leave it as is. Hence, for negative unification, Def. 4.3 is modified to require that:

$$\forall i \in \{1..n\} : (value(x_i) = value(y_i) \neq \perp)$$

In the presence of symbolic messages, our execution algorithm now operates as follows. When a user initiates an event e, this event is by its very nature bound (e.g., the user has clicked the digit 7, turned the switch *on* etc.). The universal charts are scanned for minimal events that are level_1-unifiable with e and are then unified with it, thus (possibly) causing their variables to be bound. Once these variables are bound, new events that depending on these now definitely-valued variables may become enabled (e.g., displaying the digit 7, turning a light *on* etc.). A super-step is then performed, where enabled events that are positively level_1-unifiable are carried out simultaneously. Before triggering any event, however, the active LSC copies are scanned for events that are negatively level_1-unifiable with this event. If such a violating event is found, the original event is not taken.

5 Symbolic Instances

Symbolic messages are but one aspect of the task of defining generic scenarios. There is another major issue to address. Many systems feature multiple objects that are instances of the same class. This is one of the central maxims of the object-oriented paradigm. For example, a communication system contains many phones, a railroad control system may have not only many trains and terminals but also many distributed controllers, etc. We would like to be able to specify behavioral requirements in a general way, on the level of classes and their parameterized instances, not necessarily restricting them to concrete objects. In this section we extend the LSC language with symbolic instances and present the semantics of executing charts with symbolic instances.

First, we extend the system model to contain not only specific objects but also abstract classes:

$$\mathcal{S}ys = \langle \mathcal{C}, \mathcal{O}, \mathcal{M} \rangle$$
$$\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$$

where $\mathcal{C}$ is the set of classes and the sets of objects and messages remain the same. Next, we define the *class* function that maps each object to the class it is an instance of:

$$class : \mathcal{O} \rightarrow \mathcal{C}$$

5.1 Symbolic instances in precharts

Consider the prechart of Fig. 5(a). The instance name *Phone::* actually denotes a symbolic instance, representing the class *Phone* that contains all actual phones in the system. As discussed before, events in precharts are monitored, and are propagated as a result of the user initiating an event or of an event occurring in some universal chart body as part of a super-step. In this example, when the user clicks a digit on one of the phones, say *Phone1*, the event can be identified and the prechart can be activated, with the instance *Phone::* being bound to *Phone1*.

The extended unification principle we will be using can be derived from this example quite naturally: level$_2$-unifiable events are defined by relaxing the requirement for identical senders and receivers:

Definition 5.1 (level$_2$-unifiable events) *Two events* $e = \langle m(x_1, \ldots, x_n), t \rangle$ *and* $e' = \langle m'(y_1, \ldots, y_n), t' \rangle$ *are* level$_2$-unifiable *if they are* level$_1$-unifiable *but instead of requiring that* $sender(m) = sender(m')$ *and* $receiver(m) = receiver(m')$ *(see Def. 3.6), we require that:*

$$sender(m) = sender(m') \vee$$
$$sender(m) \textit{ is symbolic and } class(sender(m')) = sender(m) \vee$$
$$sender(m') \textit{ is symbolic and } class(sender(m)) = sender(m')$$

and the analogous clause for $receiver(m)$ *and* $receiver(m')$.

This definition could be easily extended to support class inheritance by defining a class hierarchy and associating each object with the class from which it is derived. A symbolic instance representing a class C can then bind to a concrete object O if O is either an instance of C or is an instance of class C' which (indirectly) inherits from C.

The execution mechanism is now modified as follows. When an event e occurs, the precharts of universal charts are scanned for minimal level$_2$-unifiable events. Note that if e occurred, its sender and receiver are already bound. If such a unifiable event is found, a copy of the relevant universal chart is created and the symbolic sender and receiver instances (if they exist) are bound to the actual sender and receiver, respectively. The same holds for events located in precharts of already created copies.

5.2 Symbolic instances in the main chart

Suppose that we would like our phone to have the property that any digit the user clicks will be transmitted over a channel appropriately associated with the phone. Simply adding a symbolic instance representing a *Channel::* is not enough, since there is a major difference between the phone and channel symbolic instances, in the way they can be identified. The *Phone::* instance is bound to an actual phone as the result of a user action. The channel however, should be bound by the execution mechanism (not by the user), in order for the message from the actual phone to be sent via an actual channel without the need for a user-triggered action during execution. The information on which channel should bind to the *Channel::* symbolic instance is missing.

One option for obtaining this information is to follow the approach suggested in [10], where the static object model can have each phone associated with a channel by a special relationship denoted by *itsChannel*. This information can then be used in the LSC. We suggest a somewhat more general approach, using *binding expressions*, which can involve any properties of the instance. If the first event involving a symbolic instance is in the prechart, the instance will be bound to an actual object as the result of a level$_2$-unification with some other event, but if it is in the chart body it should be possible for the execution mechanism to trigger it, and it should be identified using a binding expression, as follows.

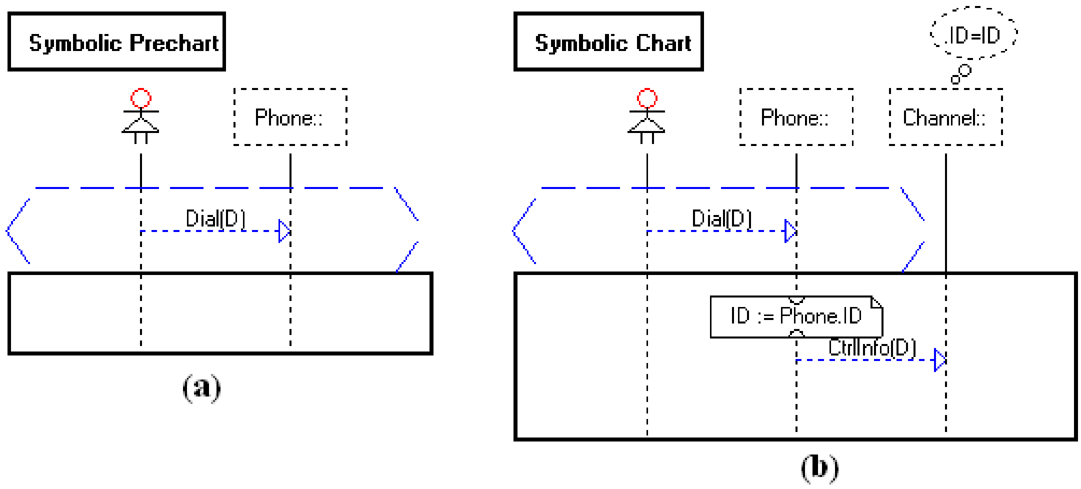

Figure 5: A symbolic instance in a prechart

Fig.5(b) shows how the *Channel::* instance is specified as binding with the actual channel associated with the phone (the example assumes that this association was done by the use of a common ID). After the *Phone::* instance is bound, the assignment can be performed and the value of ID is determined, following which the binding expression of the *Channel::* instance (shown in the small oval above the instance name[2]) can be evaluated and the appropriate channel can be bound. A binding expression can be evaluated (and actually is evaluated) as soon as all the variables it uses are bound.

The binding-expression approach can be used to associate objects using any navigation expressions derivable from the object model diagram. For each such navigation association, a property can automatically be extracted (e.g., itsChannel could be derived in the case of a one-to-one association between a phone and its channel, itsCahnnel$_i$ for the i^{th} channel in the case of a one-to-many association, etc.). We have not implemented this feature yet, since our play-engine has not yet been set up to support object associations.

Now, a binding expression can be satisfied by more than one object. This raises the options of the engine choosing one at random at runtime, or making such cases an error, or somehow binding to all satisfying objects. Since there are cases where we would like only one instance to bind and others for which we would like them all, we have decided to allow both,in the hot/cold spirit of the original definition of LSCs [6]: the association of a symbolic instance with a binding expression is quantified existentially or universally. In the existential case (denoted by a dashed instance header and expression oval), one of the satisfying objects is chosen at random and the instance is bound to it, and in the universal case (denoted by solid lines), a new copy is created for each object satisfying the binding expression.

The LSC in Fig. 6 shows the power of this approach. Whenever the central switch becomes out of order (as a result of some external environment stimulus), we want it to send a CancelCall message to all the channels that are currently allocated to a conversation. This is done using a

[2]The '.' preceding the ID property indicates a self reference.

universally quantified *Channel::* instance associated with a binding expression that is true for all allocated channels (no matter how many channels the system has).

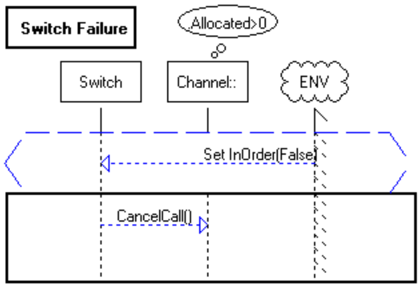

Figure 6: Using universally quantified symbolic instances

5.3 Reusing a scenario prefix

Fig. 7(a) raises an interesting issue concerning multiple bindings of a symbolic instance.

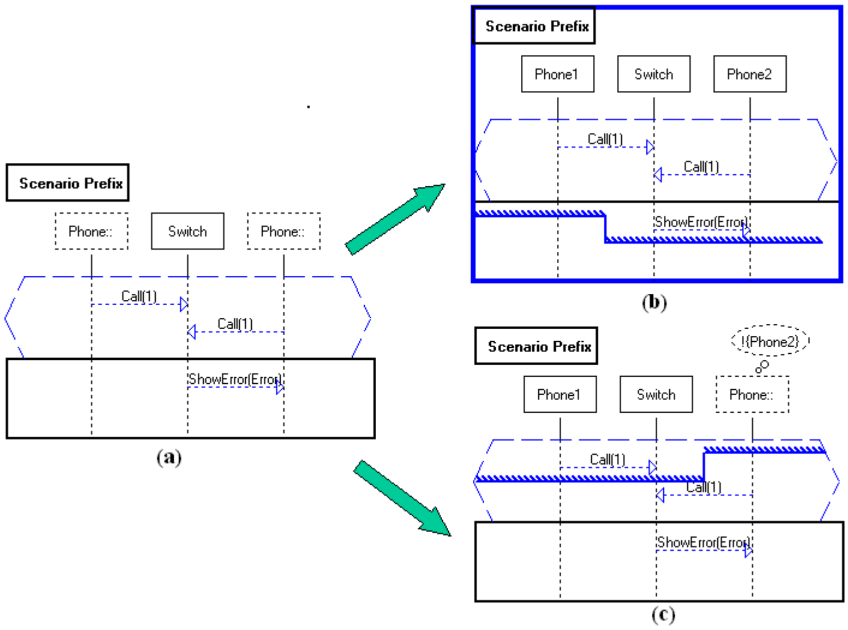

Figure 7: Keeping scenario prefixes for reuse

The LSC in the figure states that if some phone sends the message *Call(1)* to the switch and then some other phone sends the same message to the switch, the switch sends an error message to the second phone. In way of illustration, suppose *Phone1* is the first to send the message to the switch. As a consequence, the left-hand instance in the chart is bound to *Phone1*. Now, *Phone2* sends the message to the switch, the right-hand instance is now bound to *Phone2*, and the chart completes its execution by sending the error message from the switch to *Phone2*; see Fig. 7(b).[3] This is fine.

Now suppose that a third phone, *Phone3*, sends the same message to the switch. The pair of events of *Phone1* and then *Phone3* sending the same message satisfies the prechart of the LSC in Fig. 7(a), and the chart body should therefore be executed. But this is a problem since after binding the right-hand instance to *Phone2*, the LSC copy completes execution and stops existing, and the first event of *Phone1* sending the message is 'lost', making it impossible for a second event to cause another satisfaction of the chart. We solve this problem as follows. When an instance in an already activated LSC copy is bound to an object as the result of an event, the binding and consequent propagation of the execution are carried out in a new separate copy (in this case Fig. 7(b)). In addition, the original copy is left open, with the cut left as it was in the first active copy before the binding, but the symbolic instance in this remaining copy is restricted in a binding expression to not bind to the same object again. This original copy with the new binding expression is shown in Fig. 7(c). In this way, prefixes of scenarios are kept open for reuse. Proceeding with our example, when *Phone3* sends the message to the switch, a new copy is created, based on the original copy of Fig. 7(c), in which *Phone3* is bound to the right-hand instance. Then, in the original copy of Fig. 7(c), *Phone3* is added to the set of objects that the right-hand instance cannot bind to.

If the set of forbidden objects of a symbolic instance (together with the objects of the same class that are already present in the chart) becomes equal to the set of all objects in the system that are instances of the same class, the LSC copy is closed, since it has a symbolic instance that cannot bind to any more objects.

The same ideas apply also to events that cause violation of a prechart. If an event e associated with object O occurs, and is level$_2$-unifiable with an event e' associated with a symbolic instance I, and e' is not enabled, this should be a violation. However, for the same reasons discussed earlier, we want the scenario prefix to be kept open for other objects to bind to later. Therefore, a new copy is created, in which I is bound to O, and this copy is violated and closed.[4] The original copy adds O to the set of restricted objects of I, thus forbidding O to bind to I later on in the execution.

5.4 Redundant activation of symmetric precharts

Consider now Fig.8(a). The scenario in this LSC states that if one phone calls a second one (marked as event $e1$), and a third phone calls a fourth one (marked $e2$), then the fourth calls the second and the second calls the fourth. Suppose that the following two events occurred in that order: *Phone1* calls *Phone2*, and *Phone3* calls *Phone4*. Since there is no restriction on the order of $e1$ and $e2$ in the chart, and all instances are symbolic, *Phone1* and *Phone2* can bind to either the left pair of instances or to the right pair. Following this, *Phone3* and *Phone4* should bind to the remaining pair. Fig. 8(b) and (c) show these two possibilities in the form of the

[3]The solid frame surrounding the LSC in this figure denotes its completion. Notice also that the cut is located at the end of the chart.

[4]Creating a copy, violating and closing it, is done so the user can be visually notified of the effects of the last event.

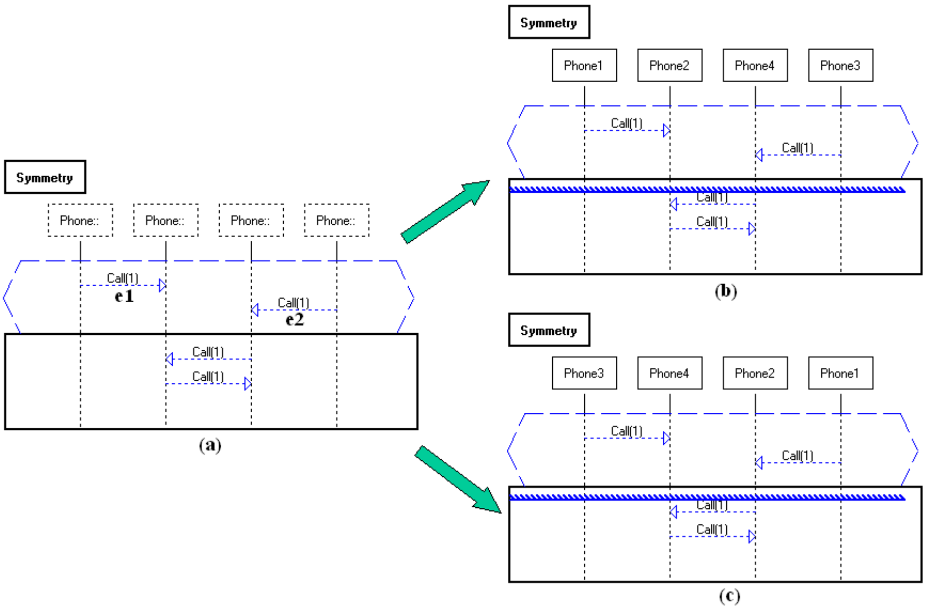

Figure 8: A symmetric prechart

two generated copies.[5]

Now, after binding all instances, neither of the charts can proceed with its body, since the order of messages is conflicting in the two copies. In the top copy, it is *Phone4* who should be sending the next message, while in the bottom copy it is *Phone2*. Although, this conflict is 'correct', and is formally compliant with the LSC semantics, it seems counter to what a designer would expect when specifying such a chart. We think that a more intuitive and practical approach would say that, since the designer left such a degree of freedom, he/she meant that it shouldn't really matter who sends the message first. Since, we prefer our execution mechanism not to initiate violating events, the correct way to solve this problem is to avoid getting into it in the first place, by identifying the symmetry such a chart embodies, and modifying the execution mechanism to avoid creating multiple copies with symmetric precharts in response to a single event.

The details of this problem and the algorithm to solve it are given in appendix A.

6 The Case of Existential Charts

Existential charts do not participate in the core of requirement execution. They are however, very useful for testing whether a behavior presented by some system implementation or some

[5]Actually, there are many other copies involved. Recalling the discussion of reusing scenario prefixes, one can see that *Phone3* and *Phone4* are created in new copies, leaving the original one with a restriction over the remaining two symbolic instances. Also, since *Phone3* calling *Phone4* is a minimal event, it also causes the creation of new copies where only these phones are bound. However, for the clarity of the discussion we focus on the two copies shown.

intra-object design model conforms with a set of expected test scenarios. In principle, the testing is done by monitoring the specified existential LSCs as events are generated by the tested implementation/model. A detailed discussion of this, and the way it is implemented in the play-engine can be found in [11]. In fact, monitoring existential LSCs is based on essentially the same ideas that are used to monitor the precharts of universal LSCs.

When it comes to monitoring scenarios, the specification language need not be as powerful as the language used for execution. For that matter, any variant of sequence diagrams can be checked against given event traces. Our extensions of symbolic messages and symbolic instances may be applied directly to any such scenario languages, thus making them more expressive and better suited to the testing of real-world applications.

7 Implementation

In this section we overview the tool we have developed, the *play-engine*, and its play-out execution mechanism. More details can be found in [11].

7.1 The system model

The play-engine is a framework for specifying and executing requirements in a user-friendly and intuitive way. Behavioral requirements are *played in* using a GUI of the application or an object model diagram, and the play-engine generates, on the fly, a formal version of the requirements in the language of LSCs. The objects in the system may be either *a priori* given as part of the GUI application or may be added during the work to the object model diagram. Classes may also be added in this way, and objects can be defined to be an instance of some class.

Each class and object have a set of properties and methods. Properties are used to reflect the state (or state variables) of objects. When using a GUI application these properties are actually shown in the GUI (e.g., the background color of a display, the external position of an antenna, a visual indication as to whether a light is on or off, etc.). When using an object model diagram, the values of properties are simply shown next to the property name in the diagram.

Methods are used to indicate sending and receiving of control information and data between objects in the system.

Thus, the messages that are generated in the LSCs as play-in is taking place represent either changes in the values of properties (caused by the user, the environment or the object itself), or flow of control and data between objects. Each message may be defined to be symbolic, in which case it involves variables rather than fixed values. The play-engine also enables importing functions to aid in domain-specific calculations. These functions can be used within messages, and their parameters can also be symbolic.

When playing in the requirements, the user operates specific objects (e.g., clicks a particular switch, or dials a number on some actual phone), and therefore the generated instances are not symbolic. To turn an instance into symbolic, the user right-clicks it and selects the 'symbolic' option from a pop-up menu. The play-engine then checks whether the first event of this instance is located in the prechart or in the chart body. If the event is in the chart body, a wizard is opened, with which the user may specify the binding expression for the symbolic instance.

Turning to the play-out mechanism, it works in iterations of steps and super-steps, as discussed earlier. The first step of each iteration is performed by the user operating the GUI application as if it were an operational system. Such steps are transformed into events that are bound to the objects and values demonstrated by the user (e.g., dial '7' on *Phone1*). Events in the universal LSCs are then unified if they are level$_2$-unifiable, where messages are identified by the property or method they refer to.

We have not yet implemented the checks for symmetric precharts, since we feel that such LSCs will occur only very rarely in real world systems. If such a case does arise, one could add a condition at the end of the prechart that will filter all options but one (e.g., require that the instances be bound to objects only in the option where they are ordered by their ID).

7.2 Experimental results: The NetPhone application

We have used the play-engine to specify and execute behavioral requirements of various systems, each emphasizing a different aspect of reactive systems (e.g., computational aspects via a pocket calculator, interaction with an external environment via a cellular phone, etc.), and we are currently in the process of modeling a nontrivial biological system, namely the egg-laying mechanism of the C. elegans worm [9].

In this section, we describe a model of a telephone network system. It was chosen because it emphasizes the strong need for symbolic instances and messages in specifying complex systems, and the power of an execution mechanism that supports such features.

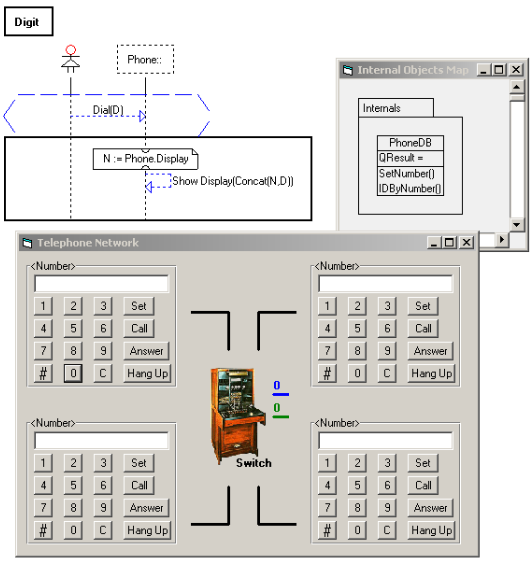

Figure 9: The NetPhone GUI Application

Fig. 9 shows the GUI of the NetPhone system, a small object model diagram and a sample LSC. The GUI consists of four telephones, four channels and one switch. The telephones are instances of class *Phone* and the channels are instances of *Channel*. The object model diagram, shown above the GUI, contains one object, *PhoneDB*, which represents the database in which the phone numbers are stored. The LSC on the top-left shows that when a user dials some digit D, this digit is to be concatenated to the number already being displayed. Note that both the *Phone::* instance and the first message are symbolic, thus allowing the LSC to be activated with a different phone and a different digit each time.

We have played in parts of a possible specification of this system. It consists of 27 universal LSCs (partitioned into 8 use cases) which comprise the following (informally stated) behavioral requirements:

1. When a user dials some digit, the digit is concatenated[6] to what is already shown in the display. The LSC for this is shown in Fig. 10(a).

2. When the C button is clicked, the display is cleared. The LSC for this is shown in Fig. 10(b).

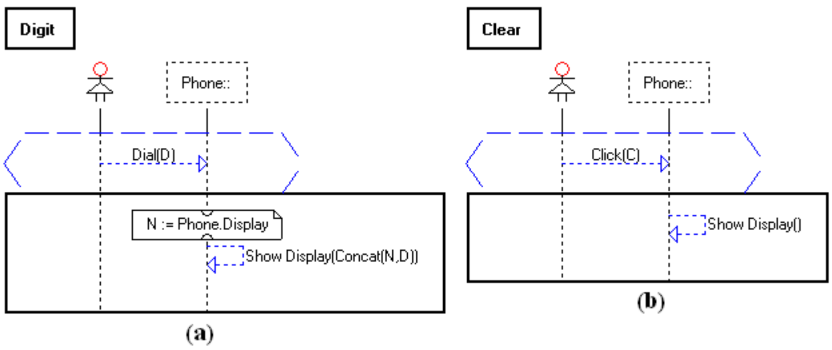

Figure 10: Simple operations in a single phone

3. When the user dials a number and then clicks the *Set* button, then:

   (a) If the number is already occupied by another phone an error message is shown.

   (b) Otherwise, it is associated with the current phone and is stored in the database as such. The number is also displayed in the phone's top-left corner.

   Actually, this requirement is given in a very detailed form in the specification, since it entails a chain of messages going from the phone to the switch, from there to the database, and then back with the appropriate messages. The LSC "Click Set" (Fig. 11(a)) shows that when the user clicks the 'Set' button, the phone ID is stored in the variable *ID* and the currently displayed number is stored in *N*. The phone then calls the switch's *SetNumber* method with *ID* and *N*. The LSC "Verify Number" (Fig. 11(b)) shows that when *some* phone calls the switch's *SetNumber* method with parameters *ID* and *N*, the PhoneDB is searched for a phone ID associated with the number *N* (denoted by the implemented

[6] The concatenation of strings is done using an implemented application-domain function. The play-engine allows applications to import such functions (for details, see [11]).

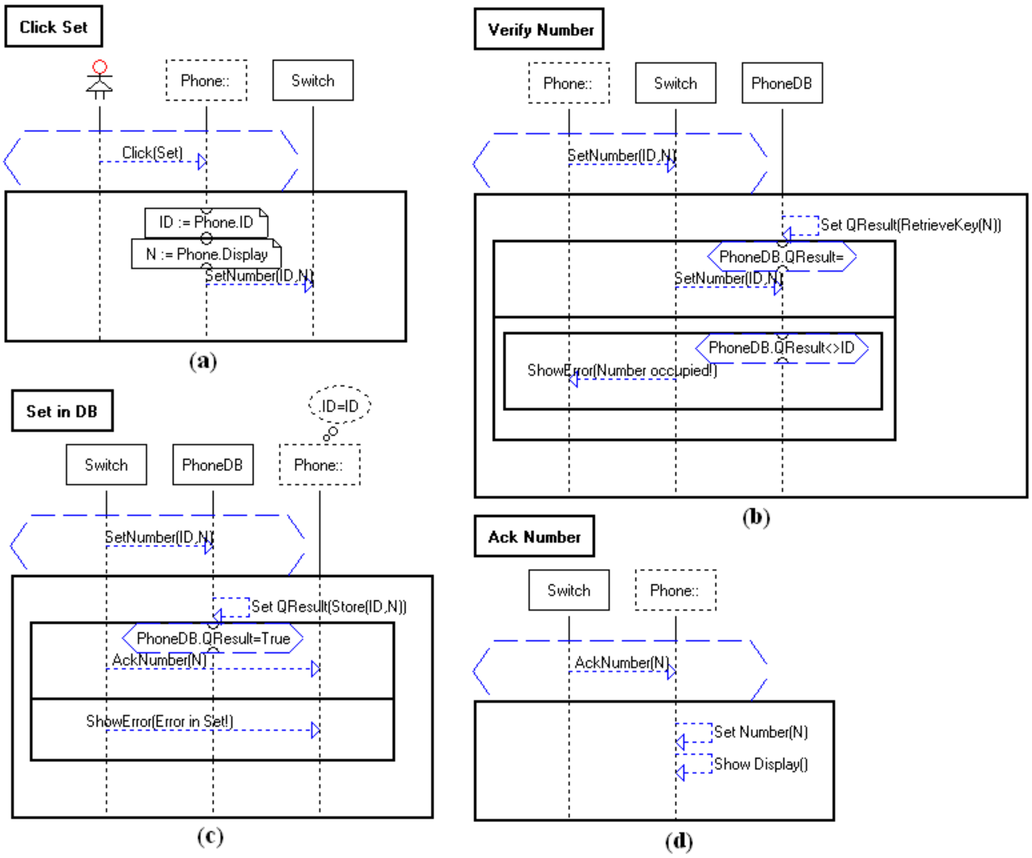

Figure 11: Setting the phone's number

function 'RetrieveKey'). If the result returned in the PhoneDB state variable *QResult* is the empty string, then no phone is associated with this number, and therefore the switch asks the PhoneDB to associate the phone ID *ID* with the number *N* and to store this information in the database. Otherwise, if the retrieved ID is not the same as the initiating phone ID, the number *N* is already associated with a different phone, and an error message should be shown on the initiating phone's display. LSC "Set in DB" (Fig. 11(c)) shows that when the switch asks the PhoneDB to set the number of *ID* to be *N*, the PhoneDB actually stores the information in the DB (using an implemented function). If the operation was successful, the switch sends *AckNumber* to the phone with ID= ID. Otherwise, an error message is sent to it. The last LSC, "AckNumber" (Fig. 11(d)), simply states that when a phone receives the *AckNumber* message with the acknowledged number *N* as a parameter, this number is shown in a proper place and the phone's display is cleared.

4. External events may cause a channel to be out of order or in order. A channel that is out of order should be colored red in the GUI. The LSCs for this requirement and for a later one describing a switch failure are given in Fig. 12.

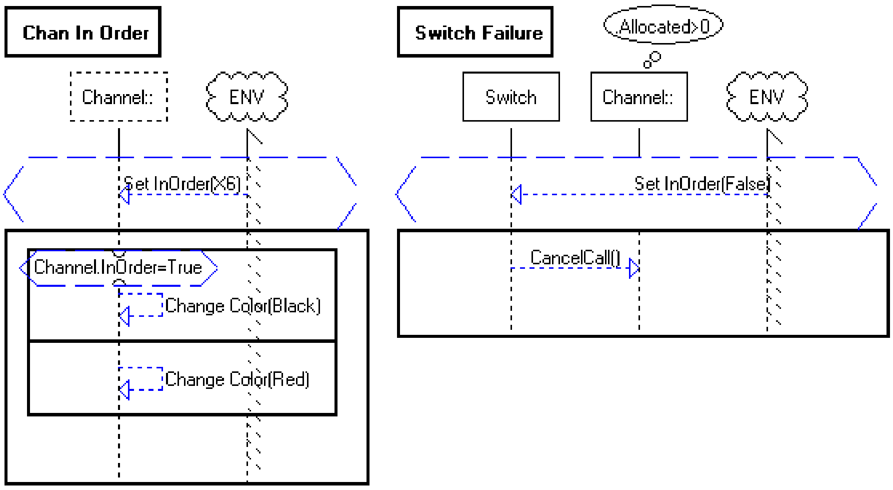

Figure 12: Failures of channels and switch

5. When a user dials a number and then clicks the *Call* button, the following possibilities hold:

   (a) If the channel is out of order, or if the number dialed is invalid (i.e., it does not exist in the database), or if the channel of the target phone is out of order, an appropriate error message is shown on the phone's display.

   (b) If the channel of the target phone is already allocated to another conversation, a "Partner Busy" message is shown.

   (c) If non of these is the case, a conversation is established between the calling and called phones. The channels participating in the same conversation are colored with a common color (different colors for different conversations).

   (d) If the calling phone is already in a conversation session, the called phone is joined to the same session (conference call), and its channel is allocated to the same session (and is colored with the same color).

   (e) When a conversation is established, the receiver phone displays the calling number on its display.

   Some of the LSCs involved in the protocol that establishes a conversation are given in Fig. 13.

   Note that our general approach of binding expressions enables us to specify symbolic LSCs even in the case where the numbers of the phones are set dynamically, during execution. This could not have been done using a static object model.

6. When the user clicks the *Answer* button, all the phones participating in the conversation display a message showing that this phone has joined.

7. When the user clicks the *Hang Up* button when the phone is participating in a conversation, its channel is deallocated and all other participants in the conversation show a

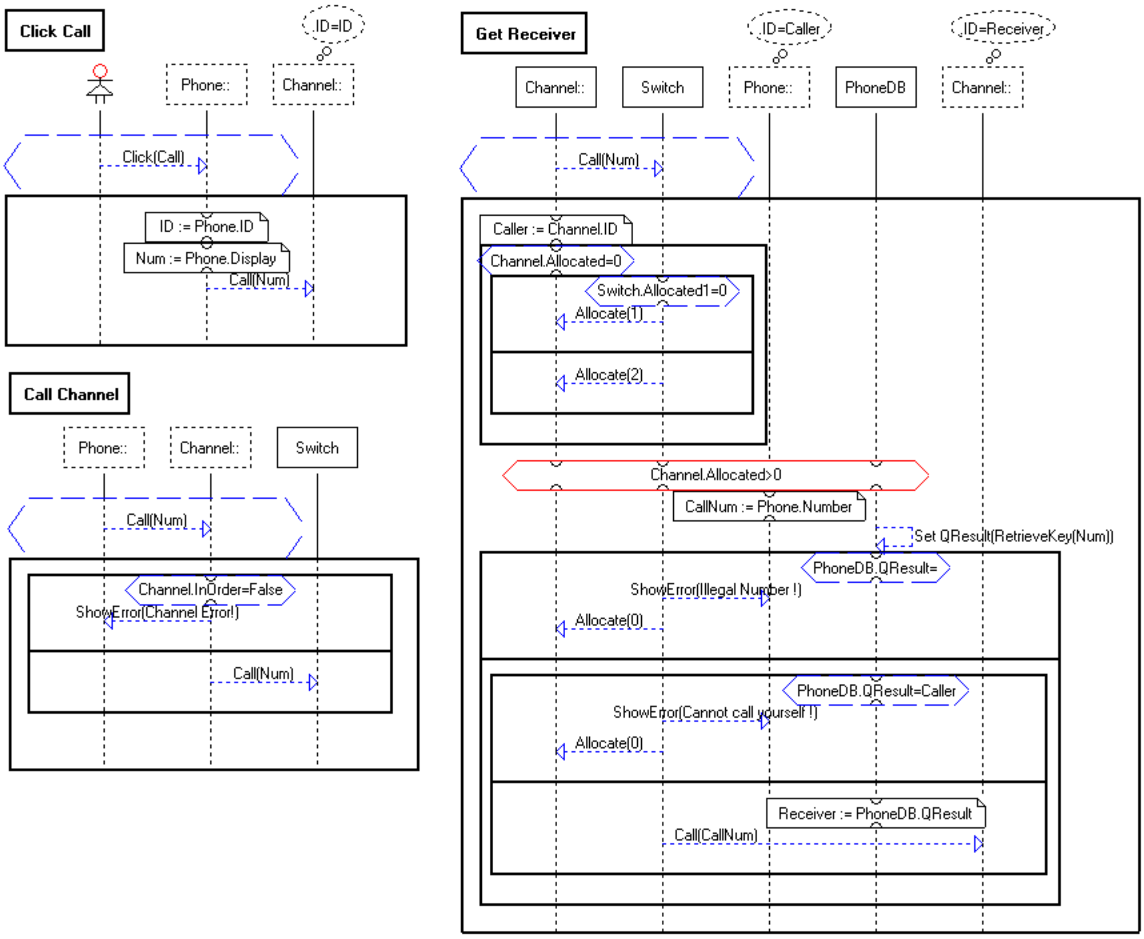

Figure 13: Part of the conversation establishment protocol

message indicating that this phone hung up. If there is only one partner to the conversation, its channel is also deallocated.

8. External events may cause the switch to be out of order or in order. When an event causes the switch to be out of order, all channels that are allocated are disconnected. Error messages are shown on the displays of the relevant phones.

9. Speech is simulated by clicking the ‘$\sharp$’ button. When a user participating in a conversation clicks the ‘$\sharp$’ button, the string composed of ‘$\sharp$’ and the phone ID (not its number) is shown on all the phones participating in the conversation.

This specification can be easily extended to support features such as leaving messages on answering machines, returning to the most recent caller, etc.

The central point here is that the specification as given consists entirely of LSCs that were played in using the engine, and that it is fully executable as is; no code or statechart or other intra-object behavioral information was needed. Table 1 shows, for each of the requirements,

Requirement♯	♯Copies	♯Steps
1	1	2
2	1	2
3(a)	2	5
3(b)	2	8
4	1	2
5(a)	2	4
5(b)	4	17
5(c)	4	20
5(d)	4	20
5(e)	4	16
5(f)	4	11
6 (using 3 phones)	2	13
7	4	16
8 (using 3 phones)	5	27
9 (using n phones)	n	n

Table 1: Experimental Results for the NetPhone Specification

the maximal number of LSC live copies that were active simultaneously while executing it and the number of steps executed within the corresponding super-step.

The execution of requirements using the play-engine is done in interactive online response time. The bottleneck is the graphical updating and animation of the active LSCs as the execution progresses. This feature can be turned off, shortening the response time radically.

8 Related Work

A large amount of work has been carried on formal requirements, sequence charts, and model execution and animation. We briefly discuss the ones most relevant to the work described here.

Amyot and Eberlein [2] provide an extensive survey of scenario notations. Their paper also defines several comparison criteria and then uses them to compare the different notations. The language of LSCs scores high according to the criteria presented there: it is component centered, it can encapsulate several runs in a single scenario, it can be abstract, and it can relate to internal objects and not only to the system as a whole. Moreover, the language is highly visual, a criterion which is very important when dealing with complex systems. The survey in [2] does not refer to some of the additional issues crucial to sequence-based languages, that were raised in [6], such as the ability to specify anti-scenarios, and to distinguish between"must" and "may" behaviors, etc., for which LSCs were in fact developed. Moreover,there is no explicit discussion of symbolic instances in [2], yet one of the evaluation criteria is *dynamicity*, which is the ability to describe behaviors that can change at run-time. The presence of symbolic instances and variables as we have described here allows the specification in a single chart of unboundedly many behaviors that can vary in run-time.

There are a number of commercial tools that successfully handle the execution of graphical models (e.g., Statemate [12] and Rhapsody by I-Logix [13], ObjectTime [19], and Rose-RT by Rational [17]). By and large, these tools can be connected to a GUI mockup (and in some cases also a real target system) and they will activate it as the execution progresses. These tools handle classes, objects and the relations between them. However, such tools execute an intra-object design model (statecharts and/or code for objects) rather than an inter-object requirement model. Rhapsody is also able to produce sequence charts showing the sequence of events generated by executing the model and to compare it with ones prepared separately by the user to help verifying the model. In general, however, these tools do not execute requirements given in LSCs or in other variants of sequence charts directly.

In a recent paper, Lettrai and Klose [16] present a methodology supported by a tool called TestConductor, which is integrated into Rhapsody [13]. The tool is used for monitoring and testing a model using a (rather modest) subset of LSCs. The charts can be monitored in a way that is similar to the way we trace existential charts. Sequence diagrams, created using the TestConductor, allow the use of variables. However, when such a chart is to be used, the user has to manually bind the variables in the chart with concrete values. Therefore, no run-time unification is performed, and if a general scenario needs to be verified for *a priori* unknown objects and values, the diagram must be manually instantiated for all possible combinations. Damm and Klose [7] describe a verification environment in which LSCs are used to describe requirements that are verified against a Statemate model implementation. The tool is commercially available with the Statemate tool. The verification is based on translating an LSC chart into a timed Buchi automaton, as described in [15], and it also handles timing issues. In this work as well as in [16], the underlying assumption is that a system model whose reactive parts are described by statecharts has already been constructed, and the aim is to test or verify that model using sequence charts (though there are plans to extend the TestConductor of [16] to enable the sequence charts to drive the behavior of selected objects too).

Dromey [8] presents a methodology called *genetic software engineering* (GSE), in which a requirement written in natural language is formalized by a "behavior tree". All such trees are then integrated into a single tree. This comprehensive system behavior tree is transformed by a variety of manipulations and projections into a components architecture diagram, and then into many component trees, each describing the internal behavior of one component. There are two similarities between GSE and our work on the play-engine: GSE tries to bridge the gap between the requirements and the design phases by using a common representation for both (i.e., behavior trees) and then attempting to move from the former to the latter by automated transformations (using domain knowledge when needed). GSE also uses a specification language that is richer than conventional sequence charts (e.g., it can specify anti-scenarios). Dromey mentions the possibility of automatic transformations from trees representing single components into their implementation code. GSE does not include model execution capabilities on the requirements level, and does not deal at all with the issues of symbolic entities and their run-time binding.

Boger et. al. [5] present a development methodology, called *extreme modeling* (XM), which tries to combine the advantages of the programming methodology of *extreme programming* [21] with the UML [20]. Since XP relies mainly on iterative coding and testing, XM must strongly rely on a modeling environment that enables execution and testing of models. For this purpose, a tool called the *UML Virtual Machine* is introduced, which can execute a sublanguage of the UML diagrams. The models that drive the execution are, again, statecharts, and not an inter-object requirements scenario-based language, yet the effect of the execution can also be shown on collaboration diagrams. Here also, no GUI of the application is used in the model execution, and no symbolic entities are supported.

The idea of using sequence charts to discover design errors at early stages of development, such as race conditions and time conflicts, has been investigated in [1, 4, 3]. The language used in these papers is classical MSCs, with the semantics being simply the partial order of events in a chart. In order to describe system behavior, such MSCs are composed into hierarchical message sequence charts (HMSCs) which are basically graphs whose nodes are MSCs. The tools presented in these papers deal with analysis of HMSC specifications, and not with their execution. All the algorithms and examples are given using fixed sets of messages and instances, although it would appear that the issue of symbolic instances and variables should not play an important role in the algorithms.

Acknowledgements: We would like to thank Marcelo Glusman and David Talby for their helpful comments.

References

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

[2] D. Amyot and A. Eberlein. An Evaluation of Scenario Notations for Telecommunication Systems Development. *Proc. 9th Int. Conf. on Telecommunication Systems*, 2001.

[3] H. Ben-Abdallah and S. Leue. Mesa: Support for scenario-based design of concurrent systems. in: B. Steffen (ed.), Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98, Lisbon, Portugal, March/April 1998, Vol. 1384 of Lecture Notes in Computer Science, p. 118 - 135, Springer Verlag, 1998.

[4] H. Ben-Abdallah and S. Leue. Timing constraints in message sequence chart specifications. in: Formal Description Techniques X, Proceedings of the Tenth International Conference on Formal Description Techniques FORTE/PSTV'97, Osaka, Japan, November 1997, Chapman & Hall, to appear., 1997.

[5] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme Modeling. In *Extreme Programming and Flexible Processes in Software Engineering - XP2000.* Addison Wesley, 6 2000.

[6] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1), 2001. Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (*FMOODS'99*), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.

[7] W. Damm and J. Klose. Verification of a Radio-based Signalling System using the STATEMATE Verification Environment. *Formal Methods in System Design*, 19(2):121–141, 2001.

[8] R. Dromey. Genetic Software Engineering. Manuscript, 2001.

[9] I. Greenwald. Development of the vulva. In: Riddle, DL., Blumenthal, T., Meyer, BJ., and Priess, JR. editors. C. elegans II. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, p 519-541., 1997.

[10] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, pages 31–42, 1997.

[11] D. Harel and R. Marelly. Specifying and Analyzing Behavioral Requirements: The Play-In/Play-Out Approach. Tech. Report MCS01-15, The Weizmann Institute of Science, 2001. Submitted.

[12] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998. Early version titled *The Languages of STATEMATE*, Technical Report, I-Logix, Inc., Andover, MA (250 pp.), 1991.

[13] I-Logix,Inc., products web page. http://www.ilogix.com/fs_prod.htm.

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

[15] J. Klose and H. Wittke. An automata based interpretation of live sequence chart. In *Proc.* 7^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

[16] M. Lettrari and J. Klose. Scenario-Based Monitoring and Testing of Real-Time UML Models. *Proc. 4th Int. Conf. on the Unified Modeling Language*, Toronto, October 2001.

[17] Rational,Inc., web page. http://www.rational.com.

[18] J. Robinson. *Logic: Form and Function*, chapter 11, pages 182–198. North-Holland, 1979.

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

[20] Documentation of the Unified Modeling Language (UML), available from the Object Management Group(OMG). http://www.omg.org.

[21] Web page http://www.extremeprogramming.org.

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

Appendix A: Redundant activation of symmetric precharts

Definition 8.1 (symmetric precharts) *We say that two LSC copies C_L and C'_L have* symmetric precharts *if there is no sequence of events that brings one prechart to its end and not the other. We denote this situation by $C_L \approx_P C'_L$.*

Clearly, $\approx_P$ is an equivalence relation. Note that two copies of a common LSC having no symbolic instances constitute a special case of symmetric precharts, so that only a single copy is created when a minimal event occurs in that LSC.

We now modify the execution mechanism as follows. When an event e occurs, the prechart of each universal LSC is scanned for unifiable events. If the instances of the sender or receiver of the event are not symbolic in the LSC, at most one event can be found, because of the partial order induced by the non-symbolic instance line. Therefore, if n events are found in an LSC L, it means that there are n pairs of symbolic instances in L, $(I_1^s, I_1^r), \ldots, (I_n^s, I_n^r)$, with events $e_i(i = 1..n)$, respectively, that are unifiable with e as their first event. Now, before creating a new copy for each pair of instances (and their corresponding event) we initialize a set of *candidate events* to contain all the events found, and then check for every two copies C_L^i and C_L^j, with $i, j = 1..n, i \neq j)$, obtained by unifying e_i and e_j with e, respectively, whether $C_L^i \approx_P C_L^j$. If this is the case, we delete e_j from the set of candidate events. The only events we are left with at the end of this process are ones whose unification with e will yield copies that do not have symmetric precharts.

To show how the $\approx_P$ relationship can be tested for, we need some more definitions:

Definition 8.2 (sequentially unifiable events) *Two events e and e' associated with instances I and I', respectively, are* sequentially unifiable *if:*

1. *e is the first event in I, e' is the first event in I', and e and e' are level$_2$-unifiable,*[7] *or*
2. *e and e' have the same location number in I and I', respectively, (say i), the events $evnt(l_{i-1}^I)$ and $evnt(l_{i-1}^{I'})$ are sequentially unifiable, and after unifying all events preceding e and e', e and e' are level$_2$-unifiable.*

This is a recursive definition, and it induces an algorithm for checking whether two instances are matching. We simply scan the events in the prechart along the lines of the instances and see whether they are unifiable in pairs. If the events are unifiable, they are unified, thus binding some of them to actual values and connecting others in connection lists. After $i - 1$ pairs of events are unified, the i^th pair is checked in the context of the already bound variables. The next definition summarizes this algorithm.

Definition 8.3 (matching instances) *Two instances I and I' are* matching *if they have the same number of locations in the prechart, and for each $i \in \ell_P(I)$, (where $\ell_P(I)$ is the set of locations of I in the prechart P of L), $evnt(l_i^I)$ and $evnt(l_i^{I'})$ are sequentially unifiable. We denote matching instances I and I' by $I \sim I'$.*

[7] Actually, here we use a more strict version of unification. When checking two variables, we require that they are both bound to the same value or they are both free. This is to prevent a case in which one instance is allowed to send/receive only a subset of messages allowed by another.

Definition 8.4 (automorphic prechart) *The prechart P of LSC L is said to be* automorphic *with respect to a pair of events (e_i, e_j), if the following hold:*

1. *The set of instances I_L can be split into three sets S_1, S_2 and S_3 (with S_3 possibly empty) such that no message is sent between sets.*
2. *There is an isomorphism $h : S_1 \to S_2$ such that*

$$\forall I \in S_1, I \sim h(I)$$

Intuitively, a prechart is automorphic with respect to events e_i and e_j if it can be split into three sets of instances such that the instances involved with e_i, and all other instances they communicate with, are in one set, and the instances involved with e_j, and all instances they communicate with, are in a different set, and these sets are equivalent in the behaviors they allow. The third set may contain irrelevant instances (i.e., ones that are orthogonal to these two sets and do not interact with them).

Given an LSC L and an event $e = \langle m, Send \rangle$, where m is a message sent from O_v to O_u, denote by e_i and e_j two events that are level$_2$-unifiable with e going from I_i^s to I_i^r and from I_j^s to I_j^r respectively. We then have:

Lemma 8.5 *If the prechart P of L is* automorphic *with respect to (e_i, e_j), and if C_L^i and C_L^j are the LSC copies obtained by unifying e with e_i and e_j, respectively, then $C_L^i \approx_P C_L^j$*

Proof: suppose that P is *automorphic* with respect to (e_i, e_j). Then both $I_i^s \sim I_j^s$ and $I_i^r \sim I_j^r$. Now, suppose there is a sequence of events $e_1, \ldots e_n$ that w.l.o.g brings the prechart of C_L^i to its end but not the prechart of C_L^j. Let e_k be the first event that separates these charts; and, again w.l.o.g, suppose that when e_k occurs, some instance $I \in S_1$ moves from location x to $x+1$ and $I' = h(I)$ remains in location x. Let $e_x = evnt(l_x^I)$ and $e'_x = evnt(l_x^{I'})$. Since $I \sim I'$, e_x and e'_x are sequentially unifiable. But since all the events up to e_k caused simultaneous progress in both copies, it means that all previous events were bound simultaneously and to the same values. Therefore, e_x and e'_x must both be unifiable with e_k. Moreover, if e_k is a *Send* event, then the receivers of e_x and e'_x are matching instances. Thus, if e_k is synchronous and the receiver of e_x is ready, then the receiver of e'_x is ready too. This is in contradiction with the fact that I was able to progress in response to e_k, and I' was not. ∎