Playing with Time: On the Specification and Execution of Time-Enriched LSCs

David Harel* Rami Marelly
The Weizmann Institute of Science, Rehovot, Israel

Abstract

We extend live sequence charts (LSCs), a highly expressive variant of sequence diagrams, with timing constructs, thus making the language suitable for specifying the behavioral requirements of time-intensive systems. We follow Alur and Henzinger in basing the extension on a single clock object. We have implemented the extension in full in our play-engine tool, which provides user-friendly ways to play in the timing constraints, together with a powerful mechanism that can execute, or play out, the time-enriched requirements directly, without the need for 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 the original MSCs [21] 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 are not able to distinguish between possible and mandatory behavior. They are thus far weaker than, e.g., temporal logic or other formal languages for specifying behavioral requirements and constraints.

To address this, while remaining within the general spirit of scenario-based visual formalisms, a broad extension was proposed in 1999, termed *live sequence charts* (LSCs) [10]. 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').

In previous work [12, 15] we have extended the language of LSCs with additional constructs and features, such as assignments, loops, variables and symbolic object instances. Assignments allow values of object properties or functions to be stored at one point in a chart, and referred to later. Loops provide means for iteration within a single chart. Variables allow for the information sent from one object to another to be symbolic and thus to represent several instantiations of the same scenario with different specific values for each one. Symbolic instances take the generalization mechanism one step further, allowing for the instances themselves to be symbolic and parameterized, and thus to represent classes rather than actual objects.

The expressiveness of the extended language makes it possible to view LSCs as a rather powerful executable model, and not only as a transient development product to be used for verification and documentation. Indeed, in [12, 15] a *play-out* execution mechanism is described. Using play-out, the user can execute requirement specifications given in the inter-object style of LSCs directly, without the need to build or synthesize a system model consisting of intra-object 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 we call the *play-engine*[12].[1] 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 operating the system freely, as he/she sees fit.

Coming to the subject matter of the present paper, we note that many kinds of reactive systems must explicitly refer and react to time. For this purpose, a variety of programming language constructs have been proposed, including delays, timeouts, watchdogs and clock variables. Extensions of temporal logic, for example, have been proposed in order to enable quantification of time. These extensions include bounded temporal operators, freeze quantifiers and

*Parts of this author's work were carried out during a visit to VERIMAG in Grenoble, in May, 2002.

[1] Short animations demonstrating some capabilities of the play-engine tool are available on the web: `http://www.wisdom.weizmann.ac.il/~rami/PlayEngine`

the use of explicit clock variables. Visual scenario-based languages have also been extended with time constructs, such as timers, delay intervals, drawing rules and timing markers.

In this paper we extend the language of LSCs so it can refer to time and react to it, and implement the extensions in full in the play-engine. We adopt the approach presented by Alur and Henzinger [2], according to which a real-time system can be viewed as a discrete system with clock variables. We show how by adding a single clock object and using constructs already existing in (extended) LSCs — namely, assignments and conditions — we can define rich timing constraints. We also show how time events can be triggered simply by referring to the clock's 'tick' event. For reasons elaborated upon later, we assume the synchrony hypothesis of zero-time actions, though we do not really have to.

As to the play-engine implementation, the play-in part provides intuitive and friendly ways for defining timing constraints between any two events (not restricted to being in a single instance), and for defining time events. Since the play-engine renders LSC specifications executable at every point along the way, the timed universal charts can be executed, fully adhering to the timing constraints and the existential charts can be monitored to check that system tests hold continuously.

In a later phase, the behavior of a class can be described using a statechart, or be ultimately implemented as code in a specific programming language. But since the LSC specification is executable, it is possible that for some kinds of systems this phase may be omitted: the LSCs, together with the play-engine acting as a kind of universal requirements execution machine, may serve as the final implementation.

In Section 2 we consider some of the time models and notations present in the literature and discuss their advantages and disadvantages. In Section 3 we overview the language of LSCs, and in Section 4 we extend it to handle time. Section 5 describes how we have extended the play-in and play-out parts of the play-engine in order to support the time extensions. We conclude in Section 6.

2 Related Work

LSCs constitute a scenario-based visual language. It is thus natural to compare it to similar languages such as MSCs [21] and UML sequence diagrams [20], and to try to handle time in a way that is as similar as possible to the way it is done there. However, since the language of LSCs is a lot more expressive than these languages and is closer to temporal logic in terms of expressive power, we have tried to handle time so that timing constraints expressible using (extended) temporal logic formulae will be expressible here too. In this section we overview the most common time notations used in sequence charts and temporal logic and discuss their pros and cons. For a detailed survey and comparison of the different time notations in sequence charts, the reader is referred to [5].

There are essentially four classes of syntactic constructs to express timing constraints in (variants of) MSCs: timers [4, 21], delay intervals[4, 16], drawing rules and other timing markers[7].

Recommendation Z.120 provides timers for expressing timing constraints within a single MSC, and along a single instance. Such a timer can be set to a value, reset to 0 and observed for timeout. Timers can be used to express a minimal delay between two consecutive events or a maximal delay between two or more consecutive events. However, timers cannot be shared among different instances in an MSC and can therefore be used to constrain events occurring only within a single object.

Delay intervals are also used to express timing constraints in a single MSC. They may express three types of constraints: An *event associated* interval [16] indicates the global minimal and maximal delays within which the event should occur with respect to any previous event in the trace. A *message delivery* interval indicates the minimal and maximal delays allowed from the moment a message is sent until it is received. A *processor speed* interval indicates the minimal and maximal delays allowed between two consecutive events along an instance line. In [16] the author generalizes the message delivery and processor speed delay intervals by using the semantic notion of *consecutive* events. In addition, the syntax of MSCs is extended in [16] with *precedence edges* that connect unrelated events and thus allow the user to provide delay intervals for them. While precedence edges allow the expression of additional timing constraints, they may result in a cluttered graph.

In the UML [7], timing constraints in sequence diagrams can be represented by drawing rules and by 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. To describe more quantitative timing constraints, timing markers can be attached to the diagram. Timing markers are Boolean expressions placed in braces; they can constrain particular events or the entire diagram. Timing markers are not shown visually in the diagram. Since neither the precise syntax of timing markers nor their formal semantics are defined in the UML, we cannot adequately assess their expressiveness.

In [13] Klose and Wittke present an automata-based semantics for LSCs. While the LSC language in [13] does not contain some of the extensions present in our language (e.g., symbolic instances), time is indeed dealt with. LSCs can be annotated by timers and by delay intervals, thus limiting the timing constraints to pairs of events that are either on the same instance line or are connected by a message. The LSC is unwinded into a timed Buchi automaton with

unique clocks serving for each constraint.

Turning now to temporal logics, these are usually interpreted over linear traces or branching computation trees. Temporal logic formulae disregard aspects of visualization or the fact that two constrained events may occur in different objects/processes. An extensive survey of notations for timing constraints in temporal logics, their comparisons and a discussion of their expressive power is found in [1].

A common way of introducing real time in temporal logic is by replacing the unrestricted temporal operators by time-bounded versions [14]. For example the *bounded operator* $\Diamond_{[1,3]}$ is interpreted as "eventually within 1 to 3 time units". This notation can relate only adjacent temporal contexts. For example, there is no direct way of expressing the following property using time-bounded operators: "every stimulus *p* is followed by a response *q* and then by another response *r*, such that *r* is within 5 time units of the stimulus *p*" [1].

This shortcoming can be remedied by extending temporal logic with explicit references to the times of temporal contexts. This can be done by *freeze quantifiers* [3]. The freeze quantifier "$x.$" binds the associated variable x to the time of the current temporal context. For example, the formula $\Diamond x.\phi$ binds the variable x to the time of the state at which ϕ "eventually" becomes true.

A third way of writing real-time requirements is based on standard first-order logic. The syntax uses a dynamic state variable T — the clock variable — and first order quantification for global variables over the time domain. The clock variable T assumes, in each state, the value of the corresponding time. Examples of using this method for expressing timing constraints can be found in e.g., [17, 18].

3 The Language of LSCs

In this section we go briefly through the elements and constructs of LSCs that we use in our work. For more details of the original language and the extensions we made, the reader is referred to [10] and [12], respectively. We illustrate the LSC elements we use by the sample chart of Fig. 1.

LSCs have two types of charts: *universal* (annotated by a solid borderline) and *existential* (annotated by a dashed borderline). Universal charts are used to specify restrictions over all possible system runs. A universal chart is typically associated with a *prechart* that specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. Existential charts are used in LSCs to specify sample interactions between the system and its environment. Existential charts must be satisfied by at least one system run. They 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

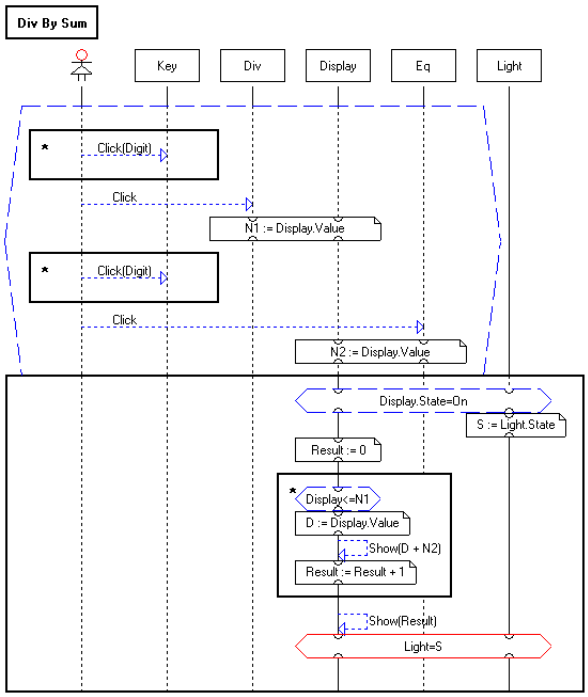

Figure 1. Example of an LSC

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.

The LSC of Fig. 1 specifies the following requirement:

> "If the user enters a number, say $N1$, then clicks the '$/$' button, then enters a second number, say $N2$ and then clicks '$=$', then, given that the display is on, it should compute the integer quotient $N1/N2$ by adding $N2$ to the displayed value until the number is greater then $N1$. The requested result is the number of times $N2$ was added. In addition, the state of the light (i.e., *on* or *off*) at the end of the operation *must* be the same as it was at the beginning."

The prechart (top dashed hexagon) contains the triggering scenario. Note how unbounded loops are used to specify an *a priori* unknown number of digit clicks. Assignments (denoted by rectangles folded in their top-right corner) are used to store the numbers $N1$ and $N2$. Using an assignment, the user may save values of the properties of objects, or of functions applied to variables holding such values. The assigned-to variable stores the value for later use in the LSC. The expression on the right hand side contains either a reference to a property of some object (this is the typical usage) or a function applied to some predefined variables. It is important to note that the assignment's variable is local to the containing chart and can be used for the specification of that

chart only, as opposed to the system's state variables, which may be used in several charts. Each assignment may have several participating objects, which, as in conditions, synchronize at the location of the assignment. Synchronizing at an assignment (condition) means that none of the synchronized instances may progress beyond the assignment (or condition) until all of them reach it and it is actually performed (or evaluated).

The first element in the chart body is a *cold* condition (denoted by a dashed blue line) stating that the state of the display is on. If a cold condition is *true*, the chart progresses to the location that immediately follows the condition, whereas if it is *false*, the surrounding (sub)chart is exited. In this case, the condition means that if the state of the display is not *on*, the chart is exited gracefully, causing no violation. After the first condition, the state of the light is stored in the variable S, again using an assignment. The last condition in the chart is a hot one (denoted by a solid red line) saying that the state of the light is equal to S, i.e., to the state it had at the beginning. A *hot* condition, must always be met, otherwise the requirements are violated and the system should abort.

The division algorithm is carried out as described earlier, using the previously stored numbers $N1$ and $N2$ and the new variable *Result*. Note that placing a cold condition C at the beginning or end of an unbounded loop creates *while* C *do* and *repeat until* $\neg C$ constructs. In our current implementation, we support *conjunctive-query* conditions, namely ones that are conjunctions of primitive equalities or inequalities; see [9].

More details about the temperatures of messages and instance locations and about the different kinds of constructs used in LSCs can be found in [12].

Conditions and assignments are non-observable events, meaning that they do not correspond directly to an event carried out by the system. Therefore, when building a tool that monitors system runs and/or executes the LSC specification, one has to decide when to perform assignments and when to evaluate conditions. We now briefly describe the semantics we chose in our play-out mechanism, used in both the execution and the monitoring parts [12]. As we show later, these decisions fit very nicely with the time extensions we propose.

The case of hot conditions is easy. Since such a condition must be true, when it is reached it is constantly evaluated and is advanced if and when it becomes true. The execution will remain stuck at that point if the specification is incorrect and the condition indeed never becomes true. Anti-scenarios can be expressed naturally within LSCs by putting the forbidden scenario in a prechart and placing a single hot condition in the main chart containing the reserved word *False*. When a hot condition containing the word *False* is encountered, there is no point in the engine waiting for it to become true; the chart is therefore considered as causing an immediate violation, and in the play-engine implementation it is crossed out and a violation message pops up. Obviously, we could have used theorem proving techniques to find other, less trivial, everywhere-false conditions and treat them as immediate violations too, but this is beyond the scope of our research. We have implemented only the most basic and practical feature needed to support the expression and detection of anti-scenarios.

One can think of several possible policies for executing cold conditions, such as waiting for them to be true if they are currently false, or waiting for them to be false if currently true, deciding at random whether to evaluate the condition, or evaluating it immediately. We have implemented the policy where cold conditions are evaluated immediately, mainly because this seems more intuitive for users to follow. Assignments are also performed as soon as they are enabled. Note that the example given in Fig. 1 justifies the intuition behind these two decisions. One would expect $N1$ and $N2$ to be stored immediately after the '$/$' and '$=$' buttons are clicked, respectively. Also, if the scenario given in the prechart occurs while the calculator is off, there is no point in waiting for the first cold condition to become true and then continue.

4 Adding Time to LSCs

Following the approach taken by Alur and Henzinger [2], we extend LSCs to handle time by adding a single *clock* object with one property, *time*, and one operation/method,*Tick*. The *clock* object and its *Tick* operation can be referred to in the LSCs by a special instance (with a clock icon) and a special constant tick message, respectively. We show how, using the clock object together with the existing constructs of LSCs, we can specify a wide variety of timing constraints and time-driven events.

An important point here is that, although we did not have to do so in order to make semantic sense of our extension, we have chosen to assume the synchrony hypothesis (see, e.g., [6]). This is a well-known abstraction of real-time systems, according to which system events consume no real time and time may pass only between events.

Our agenda is not merely to construct an off-line modeling language, but to end up with an executable language — for which the execution mechanism essentially constitutes the semantics — and to put our money where our mouth is; that is, to also build the tool that actually executes the language. Therefore, we have to somehow confront the problem of model vs. implementation, which is particularly acute in the real-time arena. The synchrony hypothesis makes this easier. When our play-engine executes the model, the clock keeps ticking and the system waits for external stimuli. When such a stimulus arrives, our execution mechanism freezes the clock and performs the sequence of

events that constitutes the system's response to that stimulus (this is a *superstep*). As the sequence is completed, the clock's operation is resumed, thus causing the effect of zero-time actions.

As we shall see later, our engine performs a lot of additional work during execution, that is not related to actually executing the specified events and actions (e.g., find the next event to be taken, monitor the activated LSCs and display their progress graphically, etc.). We could easily drop our adoption of the synchrony assumption by having the clock freeze only when this kind of additional engine-specific 'meta-work' is carried out, but letting it continue to tick when events and functions from the model itself are applied. However, obviously doing so would mean that since the play-engine executes on a single-processor PC, the requirements are captured correctly only if the target system also runs on a single-processor platform.

4.1 Hot timing constraints

A timing constraint in the extended LSC language can be defined by a combination of assignments and conditions. Fig. 2 shows how three of the most standard timing constraints can be expressed. A vertical delay interval is expressed using the following steps (see Fig. 2(a), and note the small sand-clock icons added to the assignment and condition boxes):

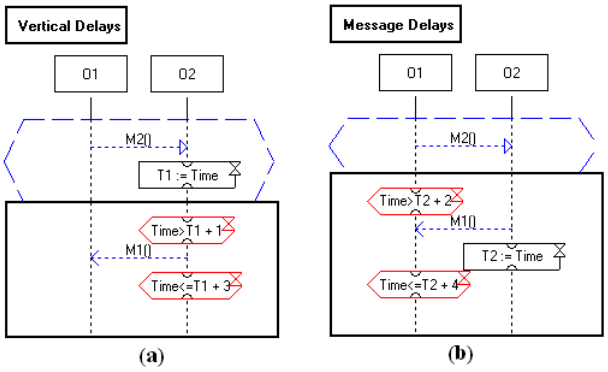

Figure 2. Basic timing constraints in LSCs

Store time: The time is stored immediately after the first event. According to our semantics, assignments are performed as soon as they are reached, so that the value of the variable will be the time instant at which the event occurs.

Specify minimal delay: The minimal delay is specified by placing a hot condition of the form *Time* $>$ *Time-Variable* $+$ *Min-Delay* just before the second event. According to the described semantics, hot conditions are evaluated continuously until becoming true. Therefore this condition will be advanced only after the required period has passed. Such a condition may be reached after the required lower bound on time has passed; in this case, it will be advanced immediately.

Specify maximal delay: The maximal delay is specified by placing a hot condition of the form *Time* $<$ *Time-Variable* $+$ *Max-Delay* just after the second event. In this case, if the condition is reached before the maximal delay has elapsed, it will evaluate to true and will be advanced immediately, causing no delay or violation. If the condition is reached after the maximal delay has elapsed, it evaluates to false, and since time cannot decrease the condition will never hold and it is therefore handled like a constant *False* condition, causing a violation of the requirements.

A message delay is specified similarly, except that now the time is stored in one instance line and is checked in another (Fig. 2(b)). Note that the assignment and the first condition in the figure are not related by the LSC partial order, so the condition may be reached before the assignment. In our semantics of conditions, a condition that refers to an unbound variable cannot be evaluated. Thus, in this case, the condition will not be evaluated until the assignment is performed.

A timer is also specified in the same manner as a vertical constraint, except that the maximal delay condition can be placed arbitrarily far from the place where the time is stored.

Here now are some more complex examples of timing constraints that can be expressed in the extended LSCs.

Fig. 3(a) shows how events occurring in different objects can be related by a timing constraint. The LSC in this figure says that whenever the switch is turned on, the light should turn on after no less than 1 time unit and no more than 2 time units. This very natural requirement cannot be expressed by the previous three standard time constructs, unless the communication between the switch and the light is given explicitly in the chart. However, this kind of constraint is often desired, especially in the requirements analysis phase, without having to get into the details of implementation.

Fig. 3(b) shows the use of conjunctive timing constraints. The LSC shows a scenario where object C sends messages to $O1$ and $O2$ and expects messages in reply from both objects. C is willing to accept the messages no later than 5 time units after the first object received its message. Conjunctive timing constraints are very useful when it is impossible to know the exact execution order between multiple events that can affect some other event.

Fig. 3(c) shows how conditions can be used to combine timing constraints with conventional constraints. The LSC in the figure is an existential one, which describes a simple test stating our expectation that the light should be on no more than D time units after the switch was turned on. Or in other words, that if the light is still off later than D units from the switch turning on, we are in trouble. The D is taken from the *MaxDelay* state variable of the controller, which may change dynamically during the system run. Note

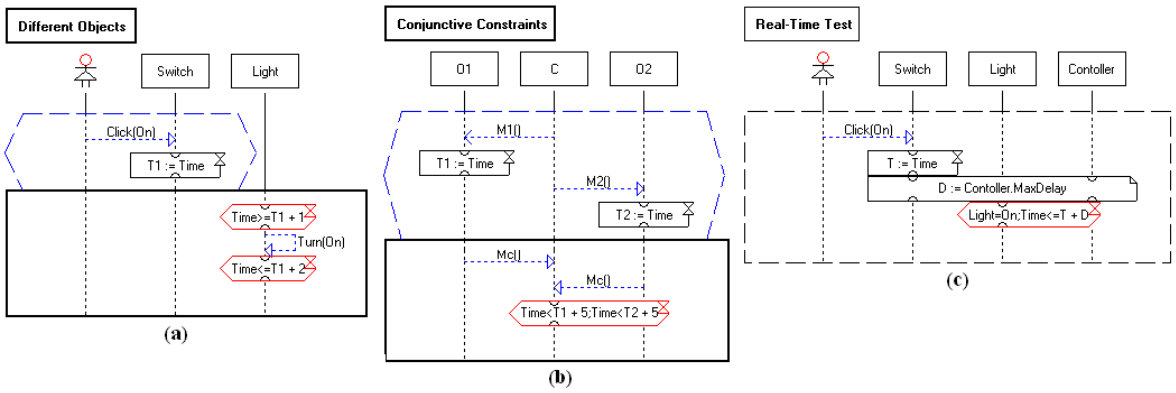

Figure 3. Expressing more complex timing constraints

that since this LSC represents a test, it does not have to specify how the result of the light being on is achieved but only that it indeed is. Placing a maximal delay in conjunction with a 'regular' guard is somewhat similar to the notion of *delayed deadline* as appears in [8].

4.2 Cold timing constraints

In the previous section we used hot conditions to force an event to occur only after a certain amount of time has elapsed (by a minimal delay) and to cause a violation if the constraint is not satisfied (by a maximum delay). The availability of cold conditions further enriches the expressive power of time-enriched LSCs. A cold timing constraint with a minimal delay states that if a certain event occurs after a specified period of time the scenario continues, otherwise the chart is exited. A cold constraint with maximal delay states that if the event occurred within a given period of time the scenario continues, otherwise the rest of it is simply ignored.

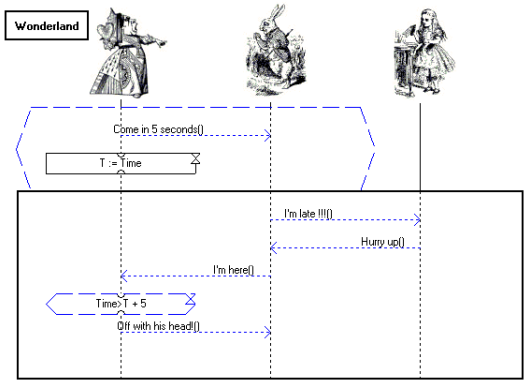

Figure 4. Having a cold time in wonderland

Fig. 4 shows an example. The queen instructs the rabbit to come to her place in 5 seconds, and proceeds to look at her watch. The rabbit meets Alice and tells her he is late, and she hurries him up. Upon arriving at the Queen's place the rabbit reports to her, and upon hearing this the Queen checks whether more than 5 seconds have passed since the last time she looked at her watch. If this is the case, she issues an order to remove rabbit's head; otherwise, the scenario ends with the rabbit's anatomy intact.

4.3 Time events

Reactive real-time systems are often required to react to the passage of time, and not only to refer to it when constraining the timing of events of interest. An example for such a requirement could be: *"The controller should probe the thermometer for a temperature value every 100 milliseconds, and if the result is more than 60 degrees, it should deactivate the heater and send a warning to the console."*

To express such requirements, generally termed *time events*, a special object instance representing the clock is available, and can be added to the LSCs. Within this instance one can refer to the 'Tick' event, which represents an actual clock tick, i.e., the passage of one time unit. This event can be placed in a prechart and thus used to trigger desired actions, or in the main chart, thus forcing delays explicitly.

Fig. 5 shows an LSC that describes the above requirement.

5 Implementation

The play-in/play-out approach is described in detail in [12]. As its name implies, the approach consists of two complementary aspects. Play-in is a method for capturing behavioral requirements in an intuitive way (e.g., following the preparation of the informal use cases), using a graphical user interface of the target system, or an abstract version

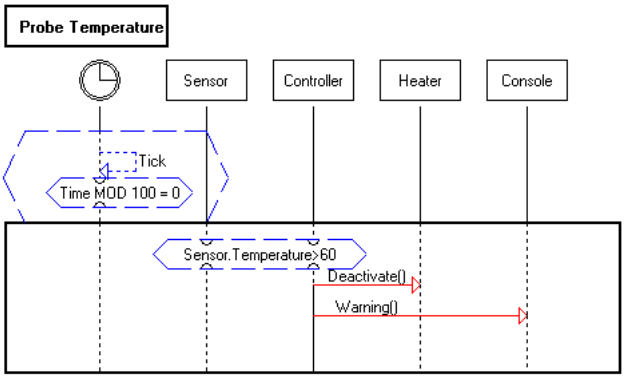

Figure 5. Time events in LSCs

thereof. The output of this process is an automatically generated formal specification in the language of LSCs. Play-out is the process of testing the requirements by executing them directly. Although, it is much more effective to play out requirements that were played in, this is not obligatory, and the LSC specification required as an input to the play-out process can be produced in any desired way.

5.1 Play-in

Playing in behavior consists of demonstrating user actions and specifying system reactions. User actions are specified simply by operating the GUI application in the way it would be done in the final product. This includes clicking buttons, rotating knobs, flipping switches, etc. Desired system reactions are specified in a similar way, by the user setting values for displays, indicating the status of LEDs and lights, and specifying the state of other output devices. This is done typically by right-clicking the relevant elements in the GUI to access and select from their possibilities. Changes in internal (GUI-less) objects are specified in the same manner, except that in this case the user works with an object model diagram (i.e., a diagram containing the objects from which the system is composed [20]) drawn by the play-engine rather than with a GUI. Here, the value of each property is shown next to the property name, thus enabling the user to view the system state at any moment. Each object may have several properties that can be changed independently. Thus, the user may specify that after switching on the phone the display should become green (using a background color property) and should show the current date (a value property). As the user plays in the desired behavior, the corresponding LSCs are generated automatically.

Conjunctive conditions are also specified by operating objects in the GUI as described above, and graphically determining the values of each object (e.g., turning a switch *on* to add to the condition that the switch ought to be on). Conditions may be used as stand-alone guards or as part of *if-then-else* constructs. In all cases, the engine provides friendly wizards to help the user define the construct of interest. Assignments are specified by right-clicking an object, choosing `Store` and then, from a popup menu, the name of the property to be stored.

It is worth noting that the behavior described in all the figures in this paper was played in and did not require any drawing or editing of elements in the generated charts.[2]

5.2 Play-out

Play-out is the process of testing the behavior of the system by providing user and environment actions in any order and observing the system's ongoing responses [12]. This calls for the play-engine to monitor the applicable precharts of all universal charts, and if successfully completed to then execute their bodies. By executing the events in these charts and causing the GUI application to reflect the effect of the events on the system objects, the user is provided with a simulation of a true executable application.

Note that in order to play out scenarios, the user does not need to know anything about LSCs or even about the use cases and requirements entered so far. All he/she has to do is to operate the GUI application as if it were a final system and check whether it reacts according to his/her expectations. Thus, by playing out scenarios the user actually tests the behavior of the specified system directly from the requirements — scenarios, scenario fragments, forbidden scenarios, etc. — without the need to prepare statecharts, to write or generate code, or to provide any other detailed intra-object behavioral specification. This process is simple enough for many kinds of end-users and domain experts, and can greatly increase the chance of finding errors early on.

A number of things happen during play-out. Charts are opened whenever they are activated and are closed when they are violated or when they terminate. Each displayed chart shows a "cut" (a kind of rectilinear "slice"), denoting the current location of each instance. The currently executed event is highlighted in the relevant LSCs. The play-engine interacts with the GUI application, causing it to reflect the change in the GUI, as prescribed by the executed event. Whenever relevant, the effects show up in the GUI.

Play-out is actually an iterative process, where after each step taken by the user, the play-engine computes a *super-step*, which is a sequence of events carried out by the system as response to the event input by the user. The super-step is executed as a continuous sequence of events that the user/environment cannot interrupt. A super-step is finished

[2] The only modification we made was to replace the instances names in Fig. 4 with pictures of the characters.

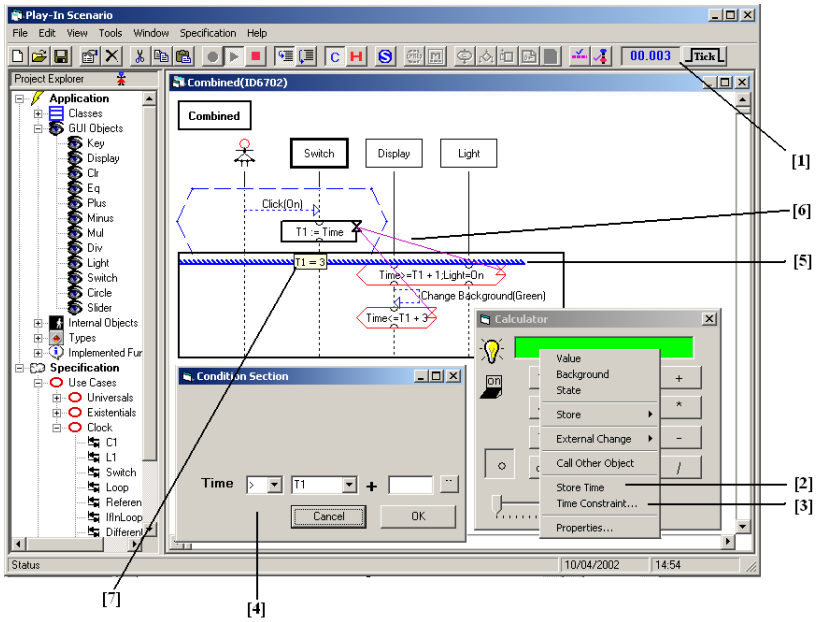

Figure 6. The play-engine environment

when there are no more events that should be carried by the system, and which do not violate other charts [12].

There are several optional modes of execution: The user may ask to execute in step mode, thus having the chance to pore over the details of each step (e.g., examining values of assignments and conditions by moving the mouse over them in the chart), or in regular mode, in which the engine pauses only when an external event has to be inserted after carrying out a full super-step. The engine can be asked to show or hide the charts themselves during execution, and so on. Play-out sessions can also be recorded and re-played later on, which is useful, e.g., for testing after making changes.

5.3 Playing with time

In order for the user to be able to refer to time while remaining within the intuitive convenience of GUI-oriented play-in, we have added a global clock object (accessible directly from the upper toolbar of the engine's layout; see [1] in Fig. 6). To specify a 'Tick' event, the user simply clicks the clock's `Tick` button on the toolbar. To store the time at any point, the user right-clicks the relevant object and chooses `Store Time` from the popup menu (see [2] in Fig. 6). To specify a timing constraint, the user right-clicks the constrained object and chooses `Time Constraint`... from the popup menu ([3] in Fig. 6), and in response a timing constraint dialog opens up, in which the user specifies the time variable, the delay and the constraint operator ([4] in Fig. 6).

The thick comb-like rendition of the current cut, marked [5] in Fig. 6, indicate the current location of each object in an active chart. More information can be obtained graphically by placing the mouse over an element in the chart. For example, in Fig. 6, the tool tip marked [7] pops up when the mouse is placed over the assignment involving the time variable $T1$ during execution, showing that it is bound to 3. For more details on run-time information in the play-engine, see [12].

As part of work on extending LSCs with time, we have added a useful feature: Placing the mouse over a timed assignment causes the play-engine to draw lines from it to all the timing constraints that may be affected by it ([6] in Fig. 6), and placing it over a timing constraint shows lines drawn to all the timed assignments that may affect it. The algorithm for doing this is not immediate, having to take into account the partial order in the chart, the bindings of variables, and a number of additional things.[3]

As we have seen, during execution the play-engine generates only system events; it does not on its own cause events that should be triggered by the user or by the external environment. In this sense, the clock is considered part of the system's external environment, and the play-engine does not trigger clock-ticks on its own account. Hence, in contrast to time-less LSCs, here a super-step can terminate in the middle of a chart's body because of as-of-yet unsatisfied minimal-delay constraints.

We have implemented two means for making the clock tick during play-out. The first is a manual mode, where

[3] We have actually implemented this algorithm for related assignments and conditions in general, and not only for those that deal with time.

user can cause a tick by simply clicking the `Tick` icon on the engine's toolbar. Note that giving the user the power to control clock ticks, over and above the normal external events, makes it possible to trigger several events within a single time unit, which can be used to set up many realistic scenarios, since it is often the case that a real-world external environment is capable of doing exactly that.

The second mode is an automatic mode. Here, the play-engine triggers a clock tick every fixed interval, using the host machine's internal clock to obtain the actual time information. The interval length, relative to the real clock, is determined by the user, and a natural thing to do would be to set the interval to be the time unit used in the requirements themselves, or the smallest such (e.g., if the time units are seconds, than the time interval will be set to 1000 msec). This allowing for true control over the execution/simulation speed, at least insofar as the host machine's clock allows, gets us as close as can be hoped to having a 'really' real-time system model.[4]

For a detailed discussion regarding the problems and challenges in the transition between models of real-time systems and their implementation, see [19].

6 The NetPhone Example

We have used the play-engine to specify and execute behavioral requirements of various kinds of example systems, each emphasizing a different aspect of typical real-world systems (e.g., computational aspects via a pocket calculator, interaction with an external environment via a cellular phone, etc.). With other colleagues we are in the process of modeling a nontrivial biological system, namely the egg-laying mechanism of the C. elegans worm [11].

In [15] we extended executable LSCs with symbolic instances representing classes, so that instead of using only concrete objects, we specify and execute generic scenarios. In that paper we describe in detail an LSC specification of a phone network system. Among other things, phones can set their own identifying numbers and store them in a database; they can call other phones to establish regular calls or conference calls; the environment can cause failures of the communication channels, of the central switching mechanism, etc. The GUI of the NetPhone application for four phones is shown in Fig. 7(a).

When preparing the present paper, we extended the NetPhone system with a number of time-related features. According to one LSC (not shown here) a user who calls some partner while the partner is busy can enter automatic mode by pressing `call` *within 5 seconds* from receiving the "busy" signal.

[4] If the interval between ticks is made sufficiently large, the user can trigger several end-user and environment events within a single time unit in this mode too.

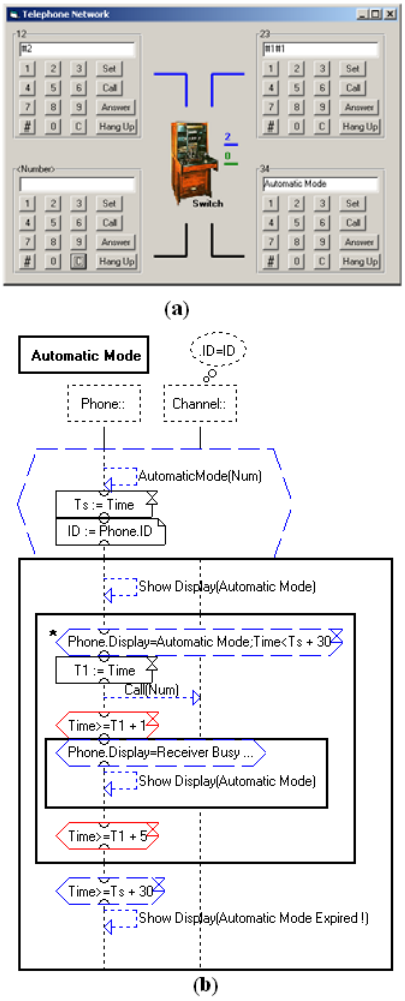

Figure 7. Some time-related behavior of the NetPhone system

The LSC in Fig. 7(b) shows the behavior of a phone in automatic mode. As soon as it enters this mode the phone stores the current time in Ts. It then displays "Automatic Mode", and enters an (unbounded) loop, which iterates for as long as the display shows "Automatic Mode", but no longer than 30 seconds after the mode was entered. This loop control is captured by the cold condition inside the loop, whose semantics prescribes exiting the current subchart — i.e., the loop — upon becoming false. The first thing the phone does inside the loop is to store the current time in $T1$ and to try calling the desired number. It then waits one second for the communication protocol to be over and checks the message on the display. If it says "Receiver Busy ...", the display is set to show "Automatic Mode" again. There follows a hot constraint that forces a 5 second wait before there is an attempt to reconnect. After

the loop, there is a cold time constraint. If the loop was exited after 30 seconds or more (i.e., it was exited not because a conversation was established), the display shows that the automatic mode has expired. Since the condition is cold, if the loop exits before 30 seconds have elapsed by establishing a conversation[5], the scenario simply ends without showing this last message.

7 Conclusions

The language of LSCs [10] is a rich visual formalism for describing inter-object, scenario-based behavioral requirements of reactive systems. Its expressive power was enhanced by the extensions described in [12, 15]. In this paper we further extended LSCs with time constructs, so that it becomes suitable for requirements of time-intensive and real-time reactive systems. The time extension we propose is relatively simple, and requires only minor changes in the syntax and semantics of the existing language, yet it can express complex time events and constraints. We have implemented the time extensions in full in our play-engine tool for requirements specification and execution.

The ideas are obviously relevant to requirements engineering and to testing. Also, system prototypes could be created by first building the application GUI and then playing in the behavior, instead of coding it. The same holds for constructing tutorials for system usage prior to actual system development.

A more extreme possibility would involve systems for which the LSC specification, together with the play-engine, can be taken to be not just the requirements but the final implementation. For example, a simple desktop utility such as a phone book could be created (given a predetermined data-retrieval function), by playing in the requirements, with no need to write any code, and play-out could serve very well as the final executable system. The same appears to hold for many kinds of small and medium-sized systems. However, only time and experience with these ideas will tell whether engineers can become comfortable with building systems to consist of a generic tool such as the play-engine executing the inter-object, scenario-based, hot and cold, may/must/may-not requirements directly. This is doubly true for time-intensive systems. Still, the possibility is tantalizing.

References

[1] R. Alur and T.A. Henzinger, "Logics and models of real time: A survey", In *Real-Time: Theory in Practice*, REX Workshop, Lecture Notes in Computer Science, Vol. 600, pp. 74–106, Springer-Verlag, 1991.

[2] R. Alur and T.A. Henzinger, "Real-time system = discrete system + clock variables", Software Tools for Technology Transfer 1:86-109, 1997. (Preliminary version in *Theories and Experiences for Real-time System Development* (T. Rus, C. Rattray, eds.), AMAST Series in Computing 2, World Scientific, pp. 1-29, 1994.

[3] R. Alur and T.A. Henzinger, "A really temporal logic", *Proc. 30th Ann. Symp. on Foundations of Computer Science*, pp. 164–169, IEEE Computer Society Press, 1989.

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

[5] H. Ben-Abdallah and S. Leue, "Timing constraints in message sequence charts specifications", *Proc. 10th Int. Conf. on Formal Description Techniques* (FORTE/PSTV'97), Chapman & Hall, 1997.

[6] G. Berry, and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", *Science of Computer Programming* **19** (1992), 87–152.

[7] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language for Object-Oriented Development* (Version 0.91 Addendum), RATIONAL Software Corporation, 1996.

[8] S. Bornot and J. Sifakis, "An Algebraic Framework for Urgency", *Information and Computation* **163** (2000), 172–202.

[9] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases", *Proc. 9th ACM Symposium on the Theory of Computing*, 1977, pp. 77–90.

[10] 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, pp. 293–312, 1999.

[11] I. Greenwald, "Development of the vulva", In *C. elegans II* (D.L. Riddle, T. Blumenthal, B.J. Meyer, and J.R. Priess, eds.), Cold Spring Harbor Laboratory Press, pp. 519–541., 1997.

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

[13] J. Klose and H. Wittke, "An automata based interpretation of live sequence charts", In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'01), 2001.

[14] R. Koymans, J. Vytopil, and W.-P. de Roever, "Real-Time Programming and Asynchronous Message Passing", In *Proc. 2nd Ann. Symp. on Principles of Distributed Computing*, pp. 187–197. ACM Press, 1983.

[15] R. Marelly, D. Harel, and H. Kugler, "Multiple Instances and Symbolic Variables in Executable Sequence Charts", *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'02), Seattle, WA, 2002. Also available as Tech. Report MCS02-05, The Weizmann Institute of Science, 2002.

[16] N. Meng-Siew. *Reasoning with timing constraints in message sequence charts*, Master's thesis, University of Stirling, Scotland, U.K., August 1993.

[17] A. Pnueli and W.-P. de Roever, "Rendez-vous with ADA: A Proof-Theoretical View", *Proc. SIGPLAN AdaTEC Conference on Ada*, pp. 129–137. ACM Press 1982.

[18] A. Pnueli and E. Harel, "Applications of temporal logic to the specification of real-time systems", In *Formal Techniques in Real-Time and Fault-tolerant Systems*, (M. Joseph, ed.), Lecture Notes in Computer Science, Vol. 331, pp. 84–98. Springer-Verlag, 1988.

[19] J. Sifakis. "Modeling Real-Time Systems — Challenges and Work Directions", *Proc. 1st Int. Workshop on Embedded Software* (EMSOFT 2001), Lecture Notes in Computer Science, Vol. 2211, Springer-Verlag, 2001.

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

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

[5] If a connection was indeed established, the receiving phone initiates a sequence of actions — not shown in this LSC — which results in the display changing to show the number of that phone, thus causing our loop to exit as it should, before the full 30 seconds have elapsed, by making the controlling condition false.