Softw Syst Model (2003) 2: 82–107 / Digital Object Identifier (DOI) 10.1007/s10270-002-0015-5

Regular papers

Specifying and executing behavioral requirements: the play-in/play-out approach

David Harel, Rami Marelly

The Weizmann Institute of Science, Rehovot, 76100 Israel

Received: 10 September 2002/Accepted: 28 November 2002
Published online: 2 April 2003 – © Springer-Verlag 2003

Abstract. A powerful methodology for scenario-based specification of reactive systems is described, in which the behavior is "played in" directly from the system's GUI or some abstract version thereof, and can then be "played out". The approach is supported and illustrated by a tool, which we call the *play-engine*. As the behavior is played in, the play-engine automatically generates a formal version in an extended version of the language of live sequence charts (LSCs). As they are played out, it causes the application to react according to the universal ("must") parts of the specification; the existential ("may") parts can be monitored to check their successful completion. Play-in is a user-friendly high-level way of specifying behavior and play-out is a rather surprising way of working with a fully operational system directly from its inter-object requirements. The ideas appear to be relevant to many stages of system development, including requirements engineering, specification, testing, analysis and implementation.

Keywords: Live sequence charts (LSCs) – Requirements engineering – System modeling and execution – Scenarios – Testing – UML

1 Introduction

Two kinds of behavior in object-oriented analysis and design are identified and discussed in [8, 13]: *inter-object behavior*, which describes the interaction between objects per scenario, and *intra-object behavior*, which describes the way a single object behaves under all possible circumstances. In [13] there is a discussion of the different roles of these within requirements and modeling languages, respectively. For modeling intra-object behavior, most object-oriented modeling approaches adopt *statecharts* [12, 14]. For the requirements aspect, one of the most widely used languages is that of *message sequence charts* (MSCs), adopted long ago by the ITU [46], or its UML variant, *sequence diagrams* [37, 43].

According to many OO-based methodologies for system development, the user first specifies the system's *use cases* [24], and the different instantiations of each use case are then described using sequence charts. In a later modeling step, the behavior of a class is described by an associated statechart, which prescribes the behavior of each of its instances. Finally, the objects are implemented as code in a specific programming language.[1]

Parts of this process can be automated, as discussed in [13]. In particular, the generation of code from object model diagrams and their statecharts can be carried out, e.g., by tools based on the ROOM method of [39], and by the Rhapsody tool [23] (based on the executable object modeling work of [14]). In fact the main pair of languages of [14, 23] – namely, object model diagrams and statecharts – constitute the core executable part of the UML.

As discussed in [8], using sequence charts to specify requirements and substantiate use-cases leaves a lot to be desired: sequence charts (whether MSCs or the UML variant) possess an extremely weak partial-order semantics that does not make it possible to capture interesting behavioral requirements of a system. They are far weaker than, e.g., temporal logic or other formal languages for requirements and constraints, and are used in practice mainly to specify possible scenarios against which to test the system later on. To address this, while remaining within the general spirit of scenario-based visual formalisms, a rather broad extension of the language of MSCs was proposed in 1999, called *live sequence charts* (LSCs) [8]. LSCs distinguish between scenarios that *may*

[1] The paper uses object-oriented terminology quite extensively. However, there is very little here that is particularly object-orientation-oriented. The ideas can be used equally well within a non-OO system development approach.

happen in the system (existential charts) from those that *must* happen (universal charts). Also, they can 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).

Since its expressive power is far greater than that of MSCs, the language of LSCs makes it possible to start looking more seriously at the relationships and possible transitions between the behavioral artifacts of these modeling steps: on the one hand use cases and LSCs, which represent the system's requirements in an inter-object style, and on the other hand statecharts, which represent its implementable model in the intra-object style. Given the discussion in [13], we should strive to be able to verify that the former is true of the latter, but also to synthesize the latter from the former. Indeed, a first-cut algorithm for synthesis has been proposed in [16]. This algorithm is only a first step, since the resulting statecharts can be extremely large. However, we do believe that useful and efficient synthesis algorithms will become available in due time, and are working towards that end. So much for the relationships between the requirements and the system.

How should the more expressive requirements themselves be specified? One cannot imagine automatically synthesizing the LSCs or temporal logic from the use cases, since use cases are informal and highly abstract. This leaves us with having to construct the LSCs manually. In a world in which we would like as much automation as possible this is somewhat problematic, because LSCs constitute a formal (albeit, visual) language, and constructing them requires the skill of working in an abstract language, and detailed knowledge of its syntax and semantics. It would be better to find a way to avoid having to actually construct the charts.

Towards the end of [13], this problem was addressed, and a higher-level approach to the problem of specifying scenario-based behavior – termed *play-in scenarios* – was proposed and briefly sketched. We have worked out the details of this proposal, and have implemented it in our *play-engine*, which we describe in this paper. However, play-in has turned out to be a secondary benefit. The main contribution we shall discuss is *play-out*, which is the ability to execute the requirements directly, without the need to build or synthesize an intra-object state-based system model.[2]

The main idea of the play-in process is to raise the level of abstraction in requirements engineering, and to work with a look-alike version of the system under development. This enables people who are unfamiliar with LSCs, or who do not want to work with such formal languages directly, to specify the behavioral requirements of systems using an intuitive and user-friendly mechanism. These could include domain experts, application engineers, requirements engineers, and even potential users.

What "play-in" means is that the system's developer (we will often call him/her a *user* – not to be confused with the eventual end users of the system under development, which we will refer to as an *end user* or an *actor*) first builds the GUI of the system, with no behavior built into it. In systems for which there is a meaning to the layout of hidden objects (e.g., a board of an electrical system), the user may build the graphical representation of these objects as well. In fact, for GUI-less systems, or for sets of internal objects, we simply use the object model diagram as a GUI. In any case, the user "plays" the GUI by clicking buttons, rotating knobs and sending messages (calling functions) to hidden objects in an intuitive drag & drop manner. (With an object model diagram as the interface, the user clicks the objects and/or the methods and the parameters). By similarly playing the GUI, the user describes the desired reactions of the system and the conditions that may or must hold. As this is being done, the play-engine continuously constructs LSCs automatically. It queries the application GUI (that was provided by the user) for its structure, and interacts with it, thus manipulating the information entered by the user and building and exhibiting the appropriate LSCs. We have attempted to enable the user to carry out as much of the play-in as possible by manipulating the GUI directly.

We should remark that there is no inherent difficulty in modifying the play-engine to produce the formal version of the behavior in scenario-oriented languages other than LSCs, such as appropriate variants of temporal logic [33] or timing diagrams [38].

After playing in (a part of) the specification, the natural thing to do is to verify that it reflects what the user intended to say. One way of testing an LSC specification is by constructing a prototype implementation and using model execution to test it. Instead, we would like to extend the power of our interface-oriented play methodology, to not only specify the behavior, but to test and validate it as well. And here is where the play-out mechanism enters.

In play-out, the user simply plays the GUI application as he/she would have done when executing a system model, or the final system, but limiting him/herself to "end-user" and external environment actions. While doing this, the play-engine keeps track of the actions and causes other actions and events to occur as dictated by the universal charts in the specification. Here too, the engine interacts with the GUI application and uses it to reflect the system state at any given moment. This process of the user operating the GUI application and the play-engine causing it to react according to the specification has the effect of working with an executable model, but with no intra-object model having to be built or synthesized. This makes it very easy to let all kinds of people participate in the process of debugging the specification, since they do not need to know anything about the speci-

[2] The play-in and play-out methodology and the algorithms underlying the play-engine are patent pending.

fication or the language used. It yields a specification that is well tested and which has a lower probability of errors in later phases, which are a lot more expensive to detect and eliminate.

We should emphasize that the behavior played out need not be the behavior that was played in. As we shall see later, the language of LSCs enables the user to specify scenario fragments that interleave with each other and combine in different ways under different circumstances, as well as forbidden scenarios and other modalities of behavior. Therefore, the user is not merely tracing scenarios, but is executing the requirements freely, as he/she sees fit. The algorithmic mechanism underlying play-out is nontrivial, and will be described in more detail later on.

Section 2 gives a short description of the LSC language as utilized in our methodology and implementation. Section 3 overviews the principles of the play-in/play-out approach. The play-engine tool is described in Sect. 4 and a sample session is given in Sect. 5. Section 6 shows how the play-in/play-out methodology can be used when making the transition from requirements to design. An overview of the execution mechanisms and the main algorithms is given in Sect. 7, while Sect. 8 shows how the play-engine can interact with external tools. Section 9 overviews advanced features of LSCs that we have developed and implemented, and which are not described in detail here. Related work is discussed in Sect. 10. We conclude with a discussion of some future plans.

A full-scale manuscript we have written [18], contains a far more detailed exposition of the entire play-in/play-out approach, including a fully worked-out operational semantics of the LSCs language and its extensions.

2 The language of LSCs

In this section we go briefly through the elements and constructs of LSCs that we use in our work. Some of these are taken as is from the original definition in [8]. There are a couple of extensions we have made to enable the specification of richer and more realistic behavior. Section 9 contains several additional extensions. The user is referred to [8] for a more complete description of the original language.

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 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. Universal charts can also be viewed as behavioral constraints of the form *if <Prechart> then <Chart-Body>* that must be satisfied by all systems runs. 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 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.

We will use a cellular phone system to illustrate the basic concepts and constructs of the language. In the LSC of Fig. 1, the prechart (top dashed hexagon) contains three messages denoting the events of the user clicking the '*' key, then clicking some digit (denoted by *Digit*), and then clicking the `SEND` button. Following this, in the chart body, the chip sends a message to the memory asking it to retrieve the number stored in cell #*Digit*.

After this message comes an assignment in which the variable `Num` is assigned the value of the `Number` property of the memory. Assignments are internal to a chart and

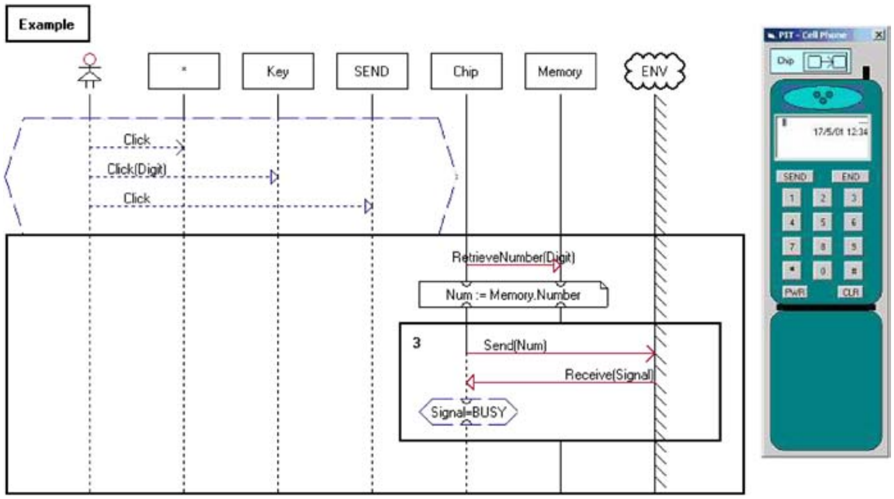


Fig. 1. LSC Sample – Quick Dialing

are something we propose adding to pure LSCs. 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 synchronize at the location of the assignment. Synchronizing at an assignment means that none of the synchronized instances may progress beyond the assignment until all of them reach it and it is actually performed. In the assignment shown in Fig. 1, the current number present in the memory is stored in a variable *Num*.

After the assignment comes a *loop* construct. There are three types of such constructs; this one is a bounded loop, denoted by a constant number (3 in this case), which means that it is performed at most that number of times. It can also be exited when a *cold condition* inside it is violated, as described shortly. There are also *unbounded* loops, denoted by a '*' and performed an *a priori* unknown number of times, and *dynamic* loops, annotated by a '?', for which the user determines the number of iterations at run time. Inside the loop of Fig. 1, the chip tries (at most three times) to call the number `Num`. After sending the message to the environment, the chip waits for a signal to come back from it.

The loop ends with a *cold condition* that requires *Signal* to be *Busy*. 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. A *hot* condition, on the other hand, must always be true, otherwise the requirements are violated and the system aborts.

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, respectively. In our current implementation, we support *conjunctive-query* conditions, namely ones that are conjunctions of primitive equalities or inequalities.

In Fig. 1, the chip will continue sending messages to the environment as long as the received signal is *Busy*, but no more than three times. Note how the use of variables and assignments in the chart makes this scenario a generic one, standing for many different specific scenarios.

The chip and memory objects consist of hot locations (denoted by solid instance lines), thus forcing their progress, while the '*', 'Key' and 'SEND' objects have cold locations (denoted by dashed instance lines), meaning that they need not progress and may stay at a location forever without violating the chart.

We end this section with a short discussion regarding the choice of LSCs as our specification language. Like other scenario-based languages (e.g., MSCs [46] and UML sequence diagrams [43]), 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. 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 response in a specific way. Events and conditions may be symbolic and 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 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.

3 The play-in/play-out approach

3.1 Playing in behavior

Consider a typical situation, where a user and a system designer meet in order to specify a new reactive system. One of the first things they might do is to discuss the functionality of the system on an abstract level and to prepare a first-cut drawing of the system's graphical user interface (GUI). At this point, our methodology recommends that the designer prepare an application representing the GUI. The GUI application has no logic built into it, but should provide a trivial predefined interface required by the engine, containing such functions as setting and receiving object values, highlighting objects, and being able to retrieve information about an object's properties.[3]

With the GUI application at hand, the user may specify use cases. In most current methodologies, a domain expert writes a use case description in an informal language and then has the system engineers describe its implementation, or instantiations, formally using more rigorous means, such as sequence charts. In contrast, we provide means for the domain expert to "play in" the instantiations of the use cases directly (including constraints and forbidden scenarios), and the play-engine then creates the charts automatically. The system engineers can then continue from these same scenarios by adding objects and refining the system design incrementally.

Playing in behavior consists of demonstrating user actions and specifying possible or mandatory system reactions. User actions are specified simply by operating the GUI application in the way it would be done in the final

[3] The current implementation of our play-engine uses a COM [7] interface, but we could have used any appropriate agreed-upon format.

product. This includes clicking buttons, rotating knobs, flipping switches, etc. System reactions are specified in a similar way, only now the user sets values for displays, indicates the status of LEDs and lights, and specifies the state of other output devices. This is done typically by right-clicking the relevant element in the GUI to access the possibilities. Each object may have several properties, that can be changed independently. Thus, the user may specify that after switching on a calculator, the display should become green (using a background color property) and should show 0 (a value property). (If an abstract GUI is used, such as an object model diagram, methods and properties can be specified by a similar click/select mechanism.)

The play-engine provides convenient, user-friendly means to state the modality (hot or cold) of messages, locations and conditions. 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.

Often it is natural to play-in a small number of sample cases that represent more general scenarios. For example, in a standard pocket calculator, the user might describe a scenario where pressing 9, + and 7 in that order (prechart) causes 16 to be displayed as a result (chart body). The play-engine can be instructed to consider this kind of play-in process as the generalized version, using loops and symbolic messages, as explained in Sect. 2. For example, if the $9+7=16$ scenario is played in via symbolic mode, this might be shown in the chart as a sequence in which “$X1$”, “+” and “$X2$” are pressed in order, and the result is shown to be “$X1+X2$”.

The play-engine also allows specifying an event value as a function applied to variables. These functions may be predefined, like the identity function and constant functions, or can be user-implemented. The latter makes it possible to include external activities that cannot be easily specified otherwise (e.g., computing a complicated mathematical function, executing an algorithm or retrieving data from a database).

3.2 Playing out behavior

Playing out is the process of testing the behavior of the system by providing any user actions, in any order, and checking the system’s ongoing responses. The play-out process calls for the play-engine to monitor the applicable precharts of all universal charts, and if successfully completed to then execute their bodies. As discussed earlier, the universal charts contain the system’s required reactions to other actions. By executing the events in these charts and causing the GUI application and object map to reflect the effect of these events on the system objects, the user is provided with a simulation of an 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.

Moreover, the possible runs of the system are not simply different orders of the same sequences of inputs, but can include totally different runs that contain unexpected events and messages. This is doubly true in the presence of symbolic messages and dynamic/unbounded loops.

The underlying play-out mechanism can be likened to an over-obedient citizen who walks around with the Grand Book of Rules on him/her at all times. He/she doesn’t lift a finger unless some rule in the book explicitly prescribes it, and never does anything if it violates some other rule. He/she constantly scans and monitors all rules at all times and upon executing any action (e.g., lifting a finger), carries out any required consequences thereof, in an iterative manner. Clearly, in so acting, there might be choices to be made, and inconsistencies in the book could be discovered. More about this later.

Thus, by playing out scenarios the user actually tests the behavior of the specified system directly from the inter-object behavioral requirements – scenarios and forbidden scenarios as well as other constraints – without the need to prepare statecharts, to write or generate code, or to provide any other detailed intra-object behavioral specification. The process is simple enough for many kinds of end-users and domain experts, and can greatly increase the chance of finding errors early on. If the specification is large and the user wishes to focus only on certain aspects of the system behavior, he/she may specify to the engine which universal charts should participate in the play-out process.

The play-engine can react to user and environment actions in two modes: *step* and *super-step*. When in step mode, the user is prompted before each event, condition evaluation, etc, and the next event to be carried out is marked on all relevant charts. In the super-step mode, the play-engine carries out as many events as possible, until reaching a “stable” state in which the system can do nothing further, and then simply waits for some input from the user.

During play-out, charts are opened whenever they are activated (possibly including multiple occurrences of the same chart) 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. When in step mode the currently executed event is highlighted in the relevant LSCs. The play-engine continuously interacts with the GUI application and the object map, causing them to reflect the changes prescribed by the executed events. As this is happening, the user may examine values of assignments, conditions and message variables by moving the mouse over them in the chart. Whenever relevant, we have programmed the play-engine to cause the effects to show up in the GUI.

3.3 Using play-out for testing

Universal charts "drive" the model by their *action/reaction* nature, whereas existential charts can be used as system tests or as examples of required interactions. Rather than serving to drive the play-out, existential charts are *monitored*, that is, the play-engine simply tracks the events in the chart as they occur. When (and if) a traced chart reaches its end, it is highlighted, and the user is informed that it was successfully traced to completion. Here also, the user may select the charts to be monitored, thus saving the play-engine the need to track charts which might currently not be of interest.

When playing out the GUI application, a run trace is produced, which includes all the user actions and the system reactions. Such runs can be recorded and saved (in XML format [44]) and then reloaded, to provide testimonies (that can be re-played) of satisfied existential LSCs. In re-playing a run, the user may select both existential and universal charts to be traced. Existential LSCs can thus be shown to be satisfied, and with universal charts the run shows when the charts were activated and how they participated in creating the system's reactions in the run. And, of course, the engine provides a notification if a universal chart is violated.

Recorded runs can be manipulated by the user, by changing the order of events and checking whether the resulting run is also a legal one. One of the interesting built-in manipulations is that of deleting all system reactions and leaving only user and environment actions. After applying this manipulation, the resulting run can be re-played. The user can then specify that universal charts should also be activated (and not only monitored). In this case, events from the recorded run are injected as before, but now the selected universal charts activate the system and trigger events as specified in their bodies.

This feature may be used for *regression testing* in the following way. While fulfilling (part of) the existential LSCs, the runs are recorded. Later on, if some of the universal charts change, the runs can be manipulated to contain only user actions and then re-played to verify that existential LSCs are still successfully fulfilled and no universal charts are violated.

Besides being recorded from the play-engine itself, runs can be imported from different sources and re-played in the same manner. These external sources can be different implementations of the specification, either given as executable programs or as more detailed design models, (e.g., statecharts [12], labeled transition systems [29], etc.). Importing a run from an implementation and replaying it, while tracing all charts, can be used to show that the implementation is consistent with the requirements, in the sense that existential charts are successfully traced and universal charts are not violated.

4 The play-engine environment

We now describe the main elements involved in the play-engine's development environment; see Fig. 2. They are numbered in what follows, to match the numbers overlaying the screen shot in the figure:

1. **The Application Section**
   This section contains the elements defined in the GUI application. The play-engine queries the application for this information and displays it in a tree format. The information contains:
   - Object classes (In our example there is one – Oper). Objects can be declared as instances of high-level classes, thus enabling the specification of more general behavior. More about that in Sect. 9.
   - GUI objects defined in the application (e.g., Key, Display, Switch, etc.). Each object has a unique id that is used in the interaction between the engine and the GUI application. Properties may be defined for each object, such as value, color, state, etc.
   - Internal objects defined in the application (in our example there are two – Controller and Memory).
   - Types upon which the properties of objects in the application can be based (e.g., Color, Number, etc.)
   - Functions implemented by the application. These functions are external to the engine and can be used within the played-in behavior (e.g., ComputeDisplay updates the value of the display after a digit is clicked, by multiplying the old value by 10 and adding the value of the key).
2. **The Specification Section**
   This section contains the elements specified by the user.
   - Use cases and LSCs. This is the main part of the requirements specification, and it consists of the LSCs (constructed by the engine as a result of the play-in), clustered into use cases. The idea is that the LSCs associated with a use case capture the behaviors that implement/instantiate it. As in many methodologies, the user starts by identifying a use case and giving it a name and a short description.
   - Jump Starts. Users often describe different scenarios, assuming different initial system configurations. The play-engine allows one to define *Jump Starts*. A Jump Start is a set of the properties of objects that are associated with initial values, and it can be used to move the system to one of its initial configurations by a single mouse click.
3. **The GUI Application**
   The GUI application (in our example, the calculator) is pre-created by the user. It may be constructed using any means, providing it supports the interface required by the play-engine. Our calculator GUI was

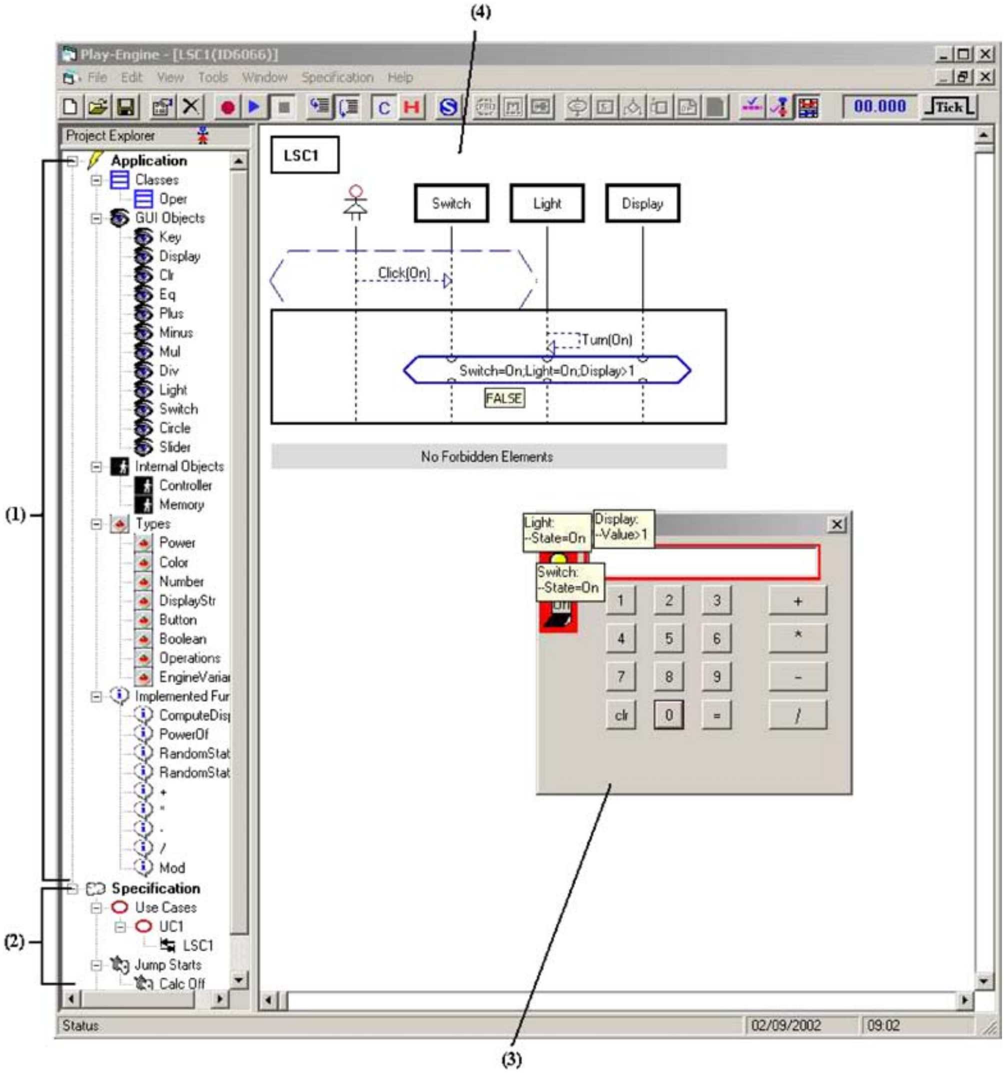


Fig. 2. The play-engine environment

written in Visual Basic. In Sect. 11 we discuss another possible way to write GUI applications.

4. **The LSCs**
   This area shows the LSC that is currently being constructed by the play-engine during play-in, or, alternatively, the LSCs that are currently being executed during play-out.

As explained earlier, we consider one of the main advantages of the play-engine to be the intuitive manner of the play-in process. We have tried to have the user interact with the GUI application directly as much as possible. One example of such a use is in the way conditions are shown. When a user points to a condition in an LSC (see Fig. 2 for an example of a condition being pointed at), several useful things happen. Within the LSC itself the condition is highlighted, as are all the participating instances, and the current true/false value of the condition is shown. At the same time, the GUI application (the calculator panel in our case) highlights the objects that participate in the condition, and each of them displays a tool tip (the kind of yellow label used in standard PC tools) that contains a description of the object's part in the condition.[4] It seems obvious to us that a more powerful HCI effort could further improve the slickness and convenience of our play-in techniques.

5 A sample play session

We now illustrate the methodology with a short play-in and play-out session. As an example, we use the pocket calculator shown in Fig. 2, even though it is somewhat trivial, since it does possess characteristics of real systems, like high reactivity and computations, and exhibits the need for symbolic representations of certain scenarios.

[4] Since the rectangular tool tips may overlap, we have used a variant of the layout algorithm of [15] to arrange them nicely. This is done by defining an attractive force between each object and its tip and a repulsive force between every two tips, and then letting the physics of equilibrium do the rest.

5.1 Play-in

The first thing our user would like to specify is what happens when the calculator is turned on. Since this is done using a switch, the action of clicking the switch is put in the prechart, and the appropriate system reactions are put in the chart body. In our case, we want the system, as a response, to turn on the light, to turn on the display, to display a 0 and to change the display's color to green.

The process of specifying this behavior is very simple. First, the user clicks the switch on the GUI, thus changing its state[5] from *off* to *on*. When the play-engine is notified of this event, it adds the appropriate message in the (initially empty) prechart of the LSC from the *user* instance to the *switch* instance. See Fig. 3. The user then moves the cursor (a dashed purple line) into the chart body and right-clicks the light on the GUI. The engine knows the properties of the light (in this case, there is just one) and pops a menu, from which the user chooses the `State` property and sets it to *on*. Figure 3 shows the situation after the switch is clicked and just before the state of the light is set to *on*. Similar processes are then carried out for the *state* and *background* properties of the display. After each of these actions, the engine adds an appropriate message in the LSC, showing the change in the property. The play-engine also sends a message to the GUI application, telling it to change the object's property in the GUI itself so that it reflects the correct value after the actions were taken. Thus, when this stage is finished, the GUI shows the switch on, the light on, and the display colored green and displaying 0.

Suppose now that the user wishes to specify what happens when the switch is turned off. In this case the light and display should turn off and the display should change its color to white and erase any displayed characters. The user may of course play in another scenario for this, but these two scenarios will be very similar, and they are better represented in a single LSC. This can be done using symbolic messages. We play a scenario as before, with the switch being clicked as part of the prechart, and the system's reactions being played-in as the chart's body. However, this time we do it with the *symbolic* flag on. When in symbolic mode, the values shown in message's labels are the names of variables (or functions) rather than actual values. So the user will now not say that the light should turn on or off as a result of the prechart, but that it should take on the same state as the switch did in the prechart. The play-engine provides a number of ways of doing this. A variable can be selected from a table of defined variables, or, as shown in Fig. 4, we can indicate that the value should be the same as in some message in the LSC. If the second option is taken, the user simply clicks the desired message inside the LSC, and its variable will be attached to the new message as well. Note that after clicking the message, the selected variable with its type and value are shown to the user as a tool tip. In case the selected message is associated with a function that has more than one variable, a dialog pops up, showing the function with its actual parameters, and the user can then click any one of these parameters, to be attached to the newly created message.

This takes care of turning the light on or off. We now want to deal with the display's color. In one case it should become green and in the other white. We can use an *if-then-else* construct for this. The user clicks the `If-Then` button on the toolbar and in response a wizard and a condition form are opened. Conditions can be specified conveniently via the GUI, as when operating objects or specifying system reactions, except that

[5] We use the word "state" to describe a property of the switch. This should not to be confused with the term "state" from statecharts.

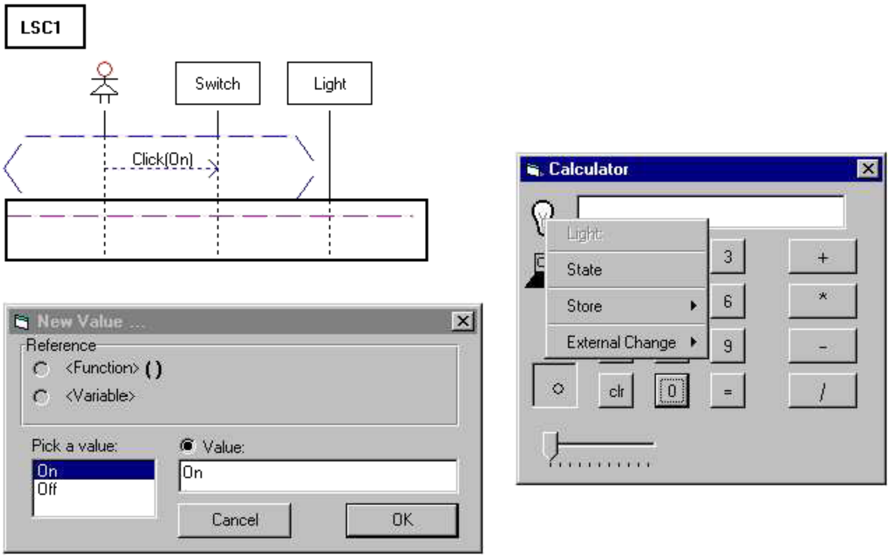
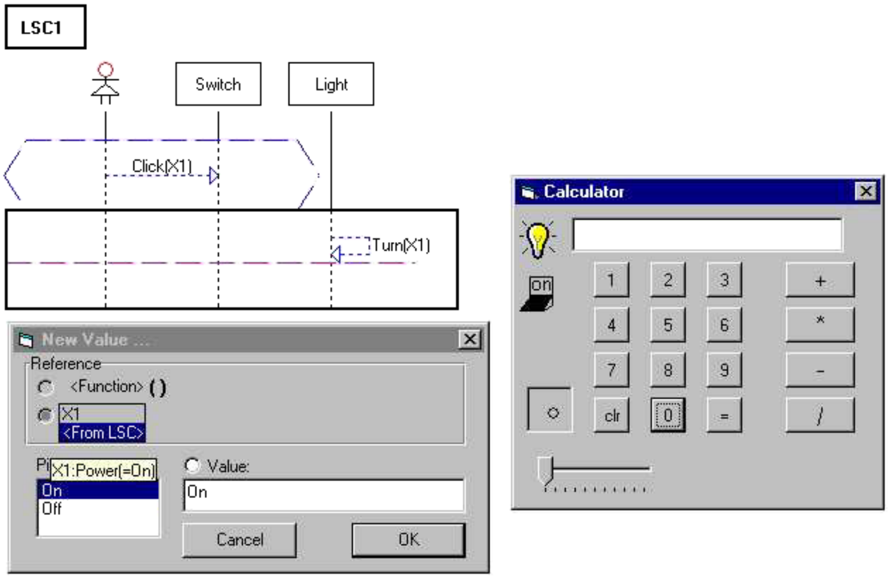


Fig. 3. Turning the calculator on



Reference: ● X1 <From LSC>

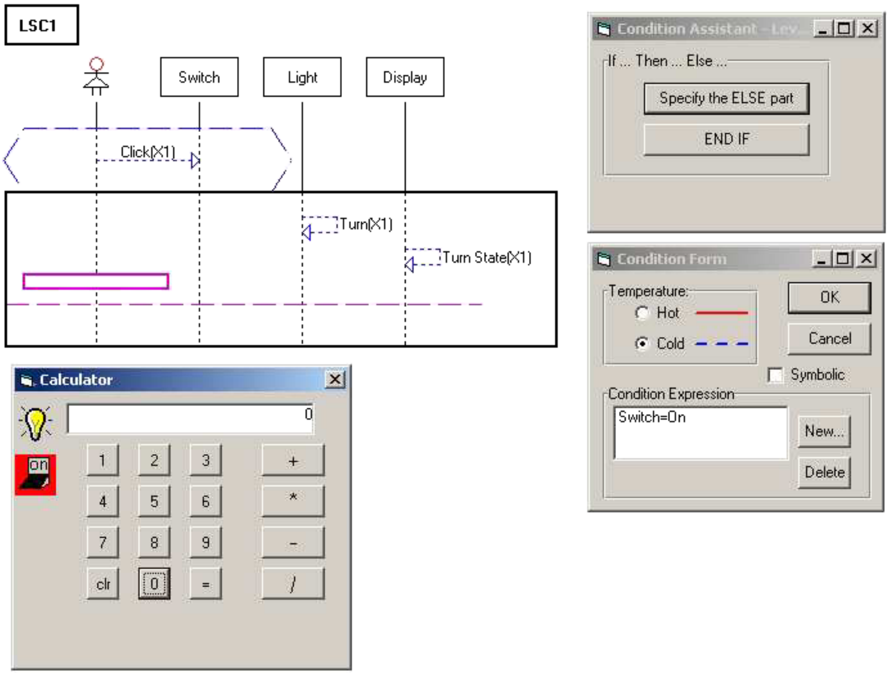
○ Value: On

Fig. 4. Using symbolic messages

here several kinds of relation operators can be used (e.g., $<, \leq, >$, etc.). Figure 5 shows the system after the wizard opens and the user clicks the switch on the GUI. Note that in the condition form, the value of the switch is specified, and the switch itself is highlighted in the GUI. Conditions may refer to properties of GUI objects, to values of variables, or even contain free expressions that the user will be requested to instantiate during play-out.

An object can participate in a condition without having any of its property values actually constrained. This is usually done when we want the object's progress to be synchronized with the condition's evaluation, but to have no effect on its value. Synchronizing an object with a condition (i.e., making the object a non-influential part of the condition) is done by right-clicking the object and choosing `Synchronize` from the popup menu.

A condition hexagon can be stretched along several instances in the LSC in order to reach those that it refers to. To distinguish those from the instances that do not participate in the condition's definition or are not to be synchronized with it, we draw small semi-circular connec-



Temperature: ○ Hot ● Cold

□ Symbolic

Condition Expression: Switch=On

Fig. 5. Specifying a condition

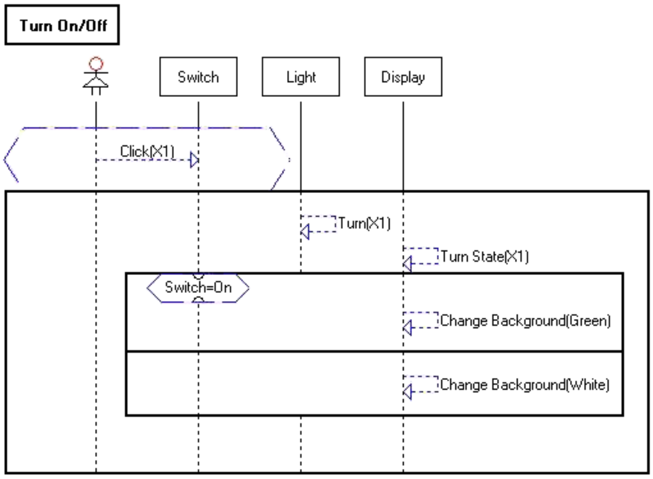


Fig. 6. Turning the calculator on or off

tors at the intersection points of a participating instance line with the condition.

After the If-Then condition is specified, the user continues playing in the behavior of the If part in the usual way. When this is completed, he/she clicks the `Specify the ELSE part` on the wizard and plays in the behavior for the Else part. The resulting LSC is shown in Fig. 6. Similar assistance is provided by the play-engine for specifying the various kinds of loops.

Sometimes we want to use data manipulation algorithms and functions, that are applied to specified variables. These functions cannot (and should not) be described using LSC-style interactions between objects but rather as external pieces of computation or logic to be worked into the requirements. Accordingly, we now play in the procedure for summing two numbers, which will illustrate how the play-engine supports such *implemented functions*. Since in our calculator example the process of entering numbers and displaying them on the screen is common to many scenarios, we have handled it in a separate chart (explained later; Fig. 10a). Therefore, what we show now deals only with the *sum* operation itself, assuming that entering the numbers has been specified separately.

To specify the prechart (see Fig. 7) the user first clicks the '+' button on the GUI. We now want the value of the display to be stored. This is done by right-clicking the GUI's display, choosing `Store` and then `Value` from the popup menu, which will result in an appropriate assignment statement in the LSC. Since after storing a value we might like to refer to it later, and for this a meaningful name is helpful, the play-engine lets the user name the assigned variable; here we use *Num1*. Note that even though the assignment refers only to the display, the *Plus* object can be seen in the figure to also be synchronized with it. This forces the assignment to be carried out only after the '+' was clicked (otherwise, there is no partial order restriction to prevent the assignment from being performed immediately upon activation of the chart). The same actions repeat with the '=' button clicked and the display's value stored in *Num2*. We thus arrive at the situation shown in Fig. 7.

After the prechart is specified, the user wants to say that the display should show the value of $Num1+$

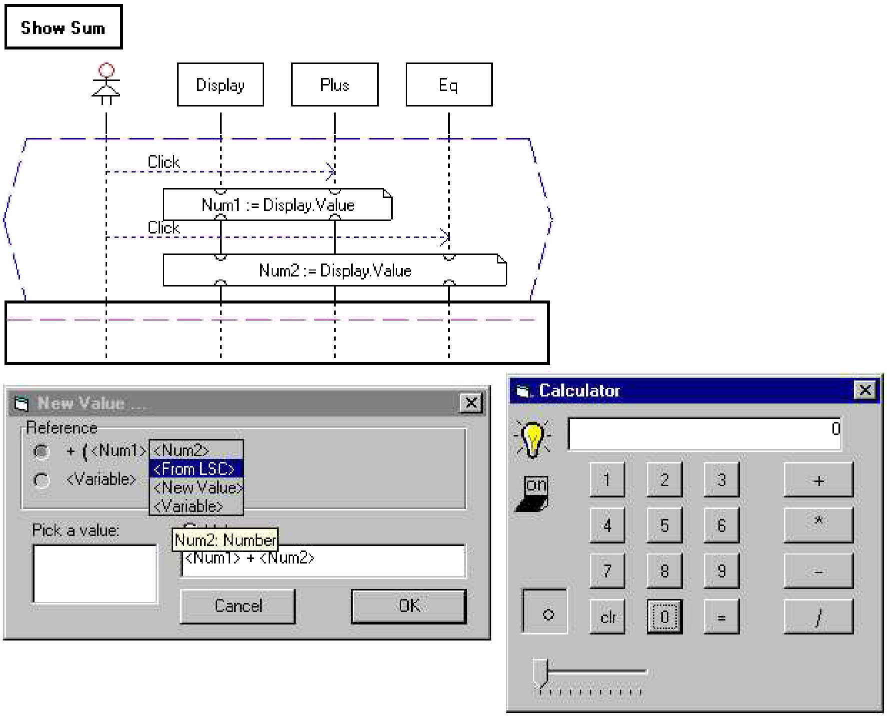


Fig. 7. Prechart of the sum operation

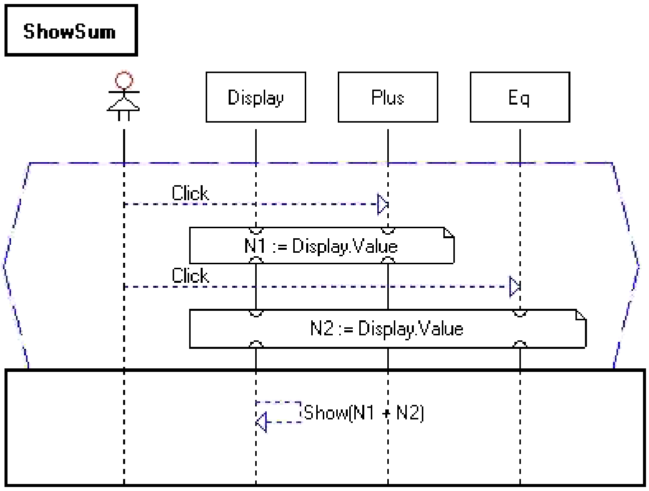


Fig. 8. LSC for the sum operation

$Num2$.[6] The user right-clicks the GUI's display and chooses the `Value` property from the popup menu. Now, instead of entering a fixed value or choosing an existing variable, the user clicks the `Function` button (which in Fig. 7 happens to be hidden by the selected function), and a list of implemented functions pops up. He/she selects one of them, and proceeds to substitute each of its formal parameters with a fixed value or a variable from the LSC. Figure 7 shows that the '+' function has two parameters, and that the one currently pointed at is of type "Number". Figure 8 shows the final LSC for summation.

Often, a reactive system works in the presence of other elements, besides the user interacting with it (other machines, computers, sensors etc.). The collection of all these elements is referred to as the system's *environment*. When playing in the required behavior of a reactive system, it is necessary to be able to express its reactions with this environment. The play-engine allows the designer to specify how the environment reacts to the system in a way that is very similar to the way the interaction with the user is played in. In order to specify a change in one of the properties of some object, caused by the environment, the object is right-clicked, and the designer chooses `External Change`. Then, from the sub-menu, the designer chooses the property to be changed. The property's value is then inserted in the same way as it would have been for any other change. The play-engine, then inserts a special instance representing the environment, and an appropriate message going from it to the target object. Figure 9 shows the menu opened when specifying an external (environment) change in the state of the light. The chart as a whole specifies that whenever the light is turned off by some external environment (not the user), the display shows "Power Off".

5.2 Play-out

Here is a short illustration of a play-out session.

We illustrate play-out using the LSCs for turning the calculator on or off (Fig. 6) and for showing the sum operation (Fig. 8), as well as the three additional charts shown in Fig. 10. The LSCs "Show Number" and "Plus – New" in Fig. 10 take care of showing the clicked digits on the display. When the '+' button is clicked, the display changes its *NewNum* property to true. When a digit is clicked, if the display's NewNum property is true, the digit is displayed as is and NewNum is set to false. If NewNum is false, the clicked digit is concatenated to the end of the currently displayed number. The concatenation is provided by the implemented function *Compute-Display*. Chart "Sum" is an existential LSC, denoted by a dashed borderline. It shows a sample scenario for summing two numbers. Dynamic loops are used in it to specify an unknown number of key clicks, and the hot condition at the end is intended to enforce the fact that when the scenario terminates the display's value should indeed be the number $Num1 + Num2$.

As mentioned earlier, when playing out a scenario, the user can choose which universal charts are to participate, and thus drive the execution, and which (existential) charts are to be monitored. In this case, we have

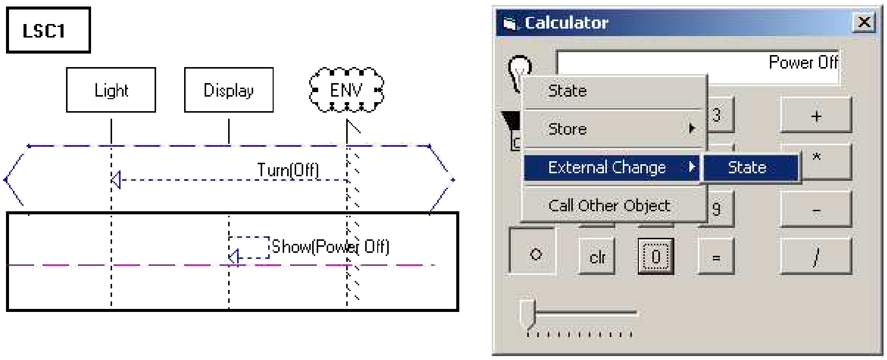


Fig. 9. Referring to the environment

[6] Even though the summation operation is simple and could have been provided by the play-engine itself, we consider it, for the sake of the example, as a function taken from the application domain, which could not be provided by a general purpose tool.

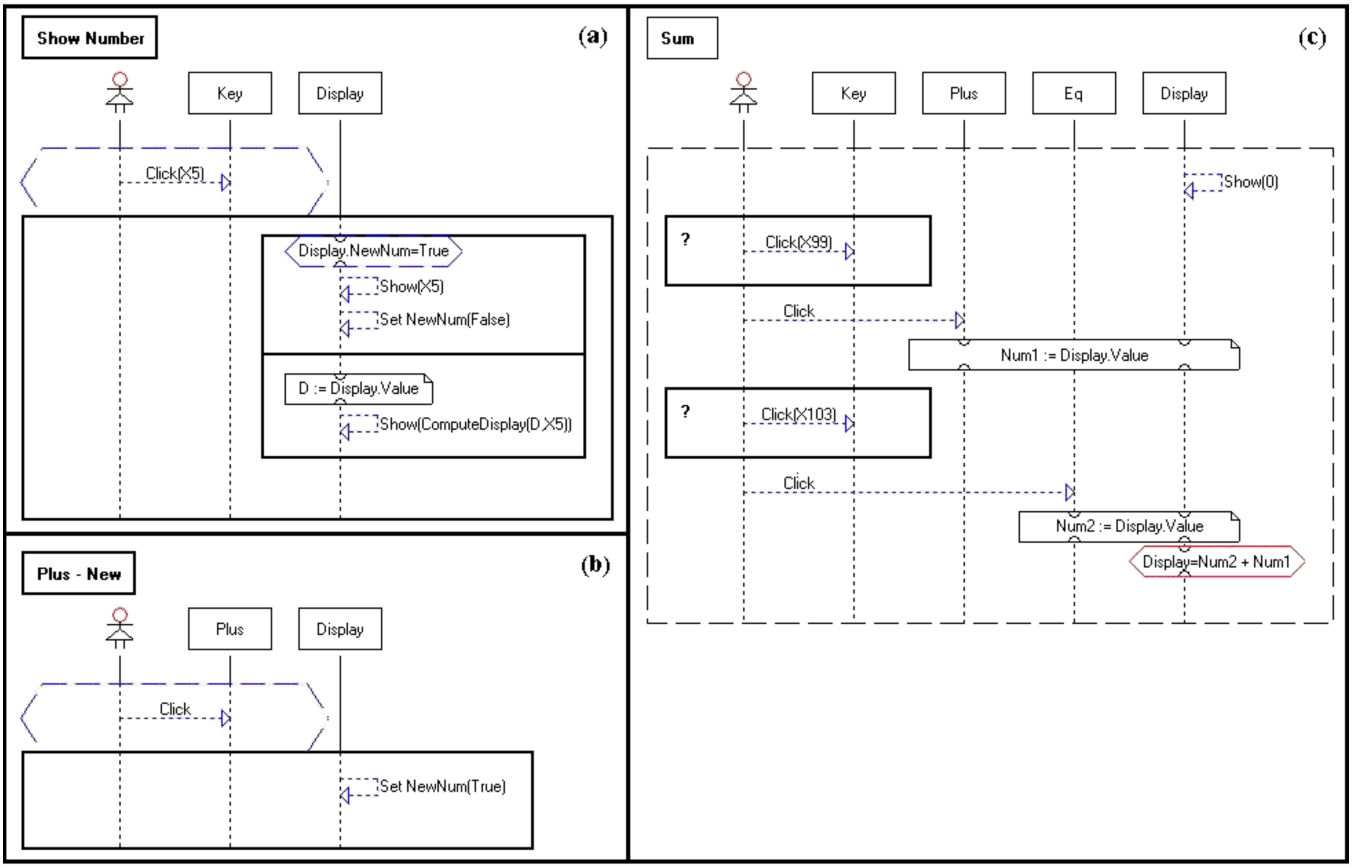


Fig. 10. Additional charts used in the play-out

decided that all the aforementioned universal charts will participate and that the existential chart "Sum" will be monitored. The charts actually shown during the play-out are those that are currently active. New charts appear as they become active. A comb-like thick line in each chart indicates the current cut of the chart – blue for cold cuts and red for hot ones. The cut line moves along in an animated fashion as the play-out proceeds. (Of course, one doesn't *have* to see the charts being animated during play-out; playing out the requirements can be done with the GUI only, with everything else being invisible to the user.)

We would now like to play out the scenario of calculating $345 + 121$. Accordingly, we select the `Play-out` mode from the play-engine's menu and then simply work with the calculator. Figure 11 shows the situation after the user has turned the calculator on and has clicked in the sequence $3, 4, 5, +, 1, 2$.

The top chart on the left, "Show Number", was activated by the click on 2. Since the display's NewNum property was false, it went to the Else part of the chart body and arranged for the new display value to be displayed (as can be seen in the calculator GUI). This chart has essentially terminated, as can be seen from the cut[7], and is thus enclosed in a thick blue frame. At this point a message to the user pops up – not shown here – indicating that the chart has ended. Once the user OKs the message, the chart is closed.

The bottom chart on the left, "Show Sum", is at the point immediately after the '+' was clicked and the value of the display was stored in $N1$. On the right is the existential chart, "Sum", which is monitored, or traced, by the play-out. This is depicted by a magnifying glass containing a "T". The chart is currently inside the second loop, at the end of its second traversal, as the numbers on the top right of the loop box shows. The first loop was traversed three times. The existential chart is shown when the mouse is located over the first assignment. Locating the mouse over the assignment in this chart has two effects: the current value of the assigned-to variable is shown in a tooltip (345 in our case), and a line is stretched to all the conditions that refer to this variable (in our case only the bottom condition).

Also, as mentioned, runs can be recorded and replayed. Figure 12 shows a situation in which a replayed run causes a violation of an existential chart. The violation is caused since the run contains two clicks on the '+' button, whereas the chart specifies only one. When the

[7] Notice that the cut line has moved beyond the If-Then-Else box, but only along the instance lines that are relevant to the box; this excludes the Key object, which is not relevant to the If-Then-Else.

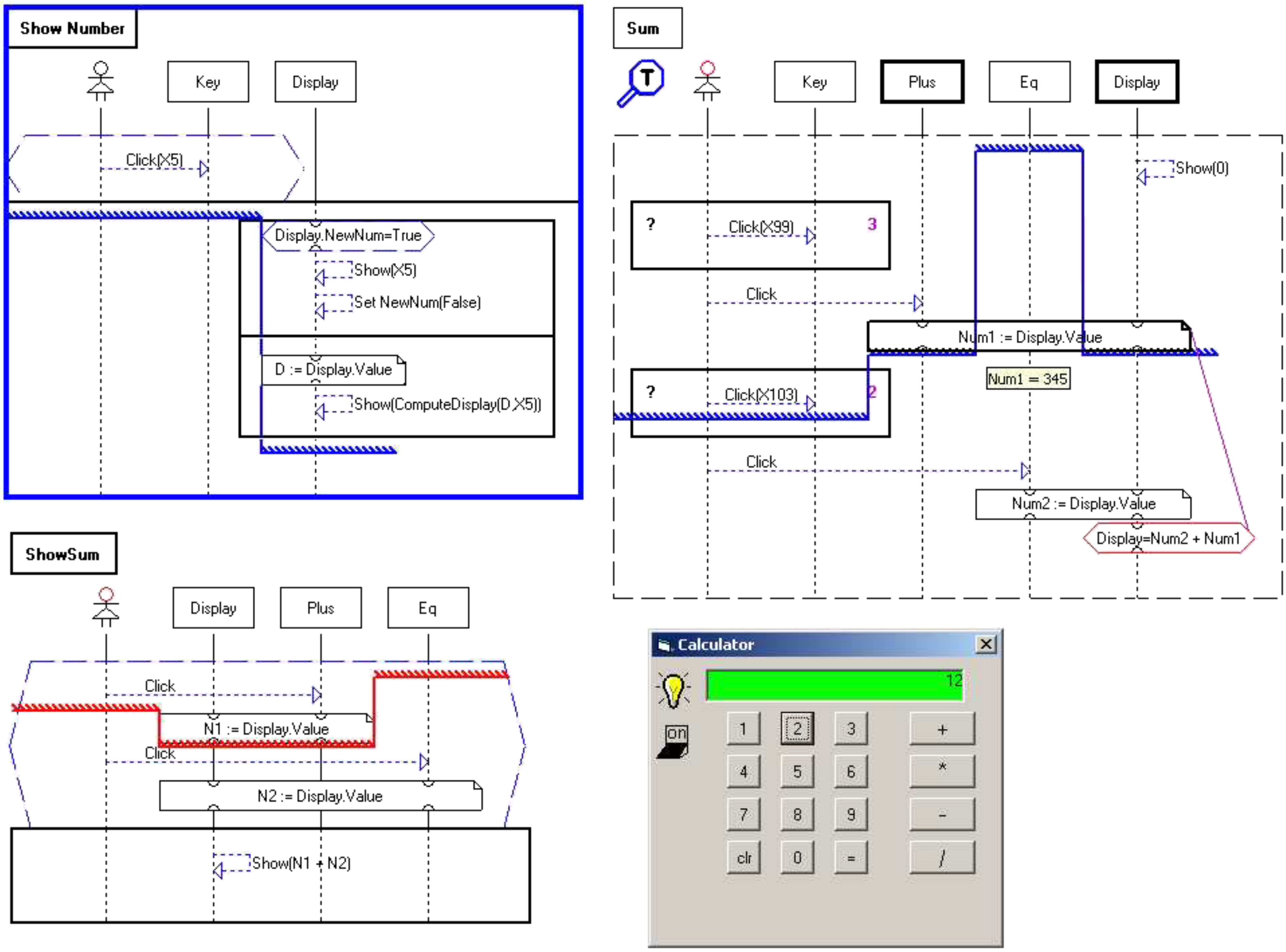
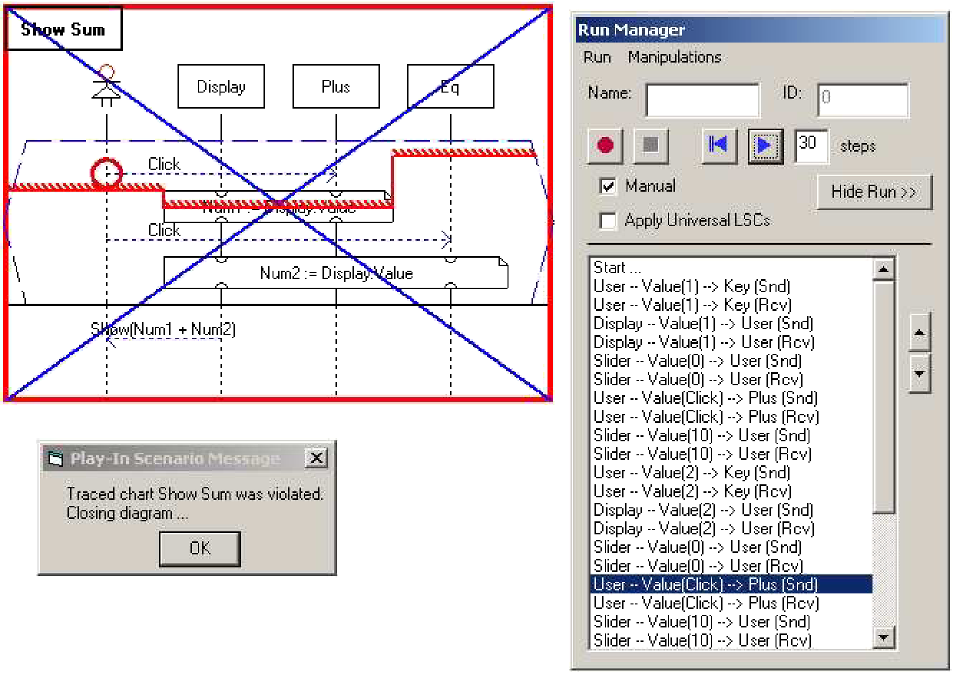


Fig. 11. Playing out a scenario



Traced chart Show Sum was violated.
Closing diagram ...

```
Start ...
User -- Value(1) --> Key (Snd)
User -- Value(1) --> Key (Rcv)
Display -- Value(1) --> User (Snd)
Display -- Value(1) --> User (Rcv)
Slider -- Value(0) --> User (Snd)
Slider -- Value(0) --> User (Rcv)
User -- Value(Click) --> Plus (Snd)
User -- Value(Click) --> Plus (Rcv)
Slider -- Value(10) --> User (Snd)
Slider -- Value(10) --> User (Rcv)
User -- Value(2) --> Key (Snd)
User -- Value(2) --> Key (Rcv)
Display -- Value(2) --> User (Snd)
Display -- Value(2) --> User (Rcv)
Slider -- Value(0) --> User (Snd)
Slider -- Value(0) --> User (Rcv)
User -- Value(Click) --> Plus (Snd)
User -- Value(Click) --> Plus (Rcv)
Slider -- Value(10) --> User (Snd)
Slider -- Value(10) --> User (Rcv)
```

Fig. 12. Replaying a recorded violation

second click is encountered (highlighted in the events list on the right), a violation is found. The play-engine indicates the violation by "crossing out" the violated LSC and circling the violating event.

6 Transition to design with internal objects

After playing in the requirements using the GUI application, and validating them using play-out, the play-engine can be used to make a smooth transition into the design phase. In many development methodologies, the designer begins the design phase with a requirements specification, usually given as a text document, and then constructs scenarios (say, in some variant of sequence diagrams) to show the interaction between the objects comprising the system, and to become convinced that these scenarios satisfy the original requirements. Using the play-in methodology, the designer can begin the design phase with a set of given (and debugged) LSCs describing the requirements in terms of allowed, forced, and forbidden interactions between the system and its users and environment. The design phase would then consist of going through the universal charts and refining them by adding internal objects and the interactions of these objects with the GUI objects and with other internal objects. Adding this information to the charts fills the gap between *what* the system should do and *how* it does it. Note that by starting with LSCs created by, or on behalf of, the user, we achieve direct tracability between the behavioral requirements and the design. Moreover, by leaving the existential charts unmodified, the designer may prove the correctness of the modified LSCs by performing regression testing to satisfy the original existential charts.

The play-engine provides means for adding internal objects on the fly. It also enables the adding of properties and methods to the new objects and also to the objects exported by the GUI application. The user may specify method calls between these objects by right-clicking the calling object and selecting `Call Other Object`. Then, the target object is right-clicked and the appropriate method is selected. Finally, the user instantiates the method's formal parameters with actual ones.

Figure 13 shows a snapshot of the play-engine while a user plays in the interaction between an internal object (the controller in this case) and a GUI object (the light). The internal object is operated from within an "Object Map" which is an object model diagram of sorts, making the interaction with GUI-less objects quite intuitive. The LSC in this figure shows that earlier a similar interaction was specified, this time from the switch to the controller.

When LSCs containing method calls are played out, the play-engine animates them by drawing thick red arrows between the involved objects and highlighting them in red. Thus, the play-out mechanism can be used not only to enable end-users to validate requirements but also as a useful tool for demonstrating, reviewing and debugging a design. Figure 14 shows how the interaction between objects is animated in the play-engine. Note that the arrows are not limited to the GUI application or to the object map, but can also run from one to the other.

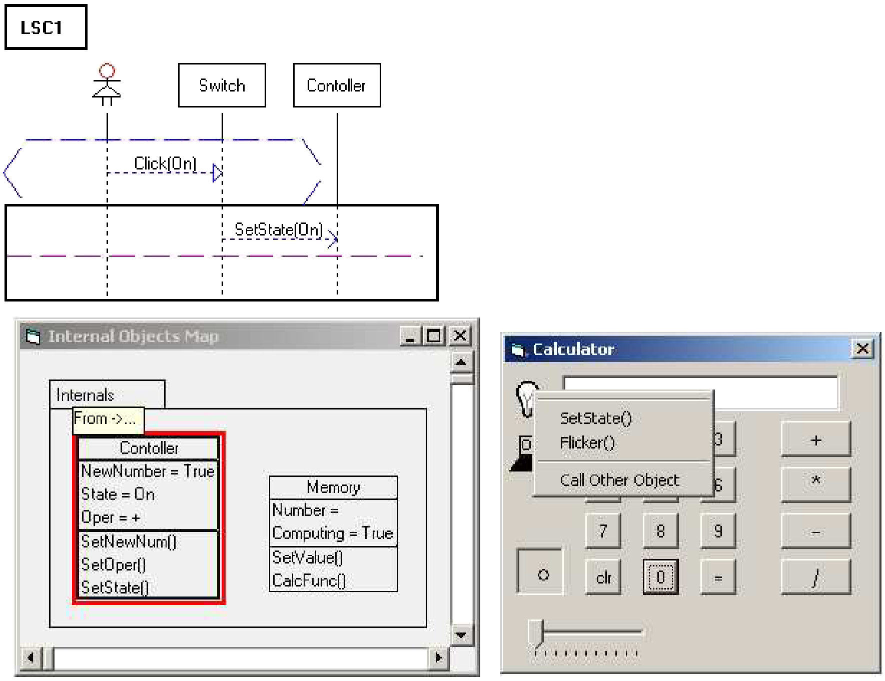
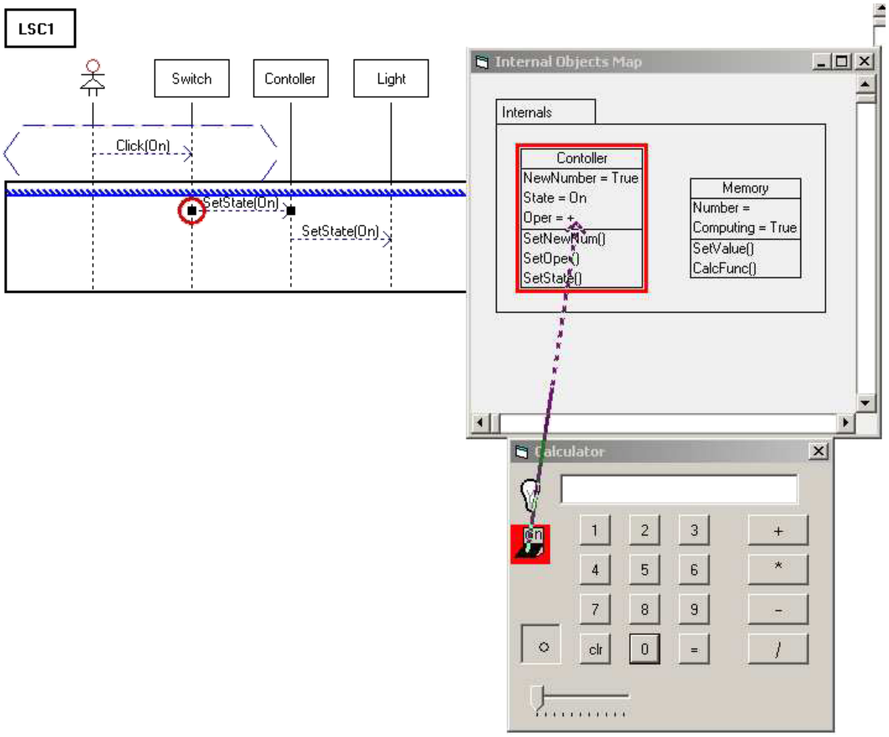


Fig. 13. Playing in internal objects: This snapshot shows the situation right after the controller was set as the source of the message and before the appropriate method of the light is selected.



Contoller
NewNumber = True
State = On
Oper = +
SetNewNum()
SetOper()
SetState()

Memory
Number =
Computing = True
SetValue()
CalcFunc()

Fig. 14. Playing out internal objects

7 The execution mechanism

In this section we overview the underlying execution mechanisms for play-out, their high-level architecture and the main components involved in them.

7.1 An LSC copy and its life cycle

Our behavioral requirements end up being a set of LSCs, termed an *LSC specification* in [8]. 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 "live 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 notion of a live copy is strongly utilized in our play-out algorithms.

Definition 1 (LSC live copy). *Given an LSC L, the live copy of L, denoted by C_L, is defined by*

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

where L is a copy of the original chart (containing distinct copies of the chart variables), $M \in \{PreActive, Active, Monitored\}$ 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.

Definition 2 (minimal event). *An event e is* minimal in a chart *L if there is no event e' in L such that $e' <_L e$, where $<_L$ is the partial order induced by the LSC.*[8]

Definition 3 (enabled event). *An event e is* enabled with respect to a cut *C if the location in C of every instance*[9] *participating in the event e is the one exactly prior to e, and there is no $e' <_L e$ that was not already processed.*

Definition 4 (violating event). *An event e* violates a chart *L* in a cut *C if e appears in L but is not enabled with respect to C.*

The general life cycle of an LSC live copy is illustrated in Fig. 15. We begin by looking at the life cycle of a universal LSC. Initially, the copy does not exist. Whenever a minimal event that appears in L's prechart occurs, the copy is created in *preactive* mode. As long as events occur and conditions are reached and evaluate to true, the cut of the copy is propagated. When all locations in the prechart have been traversed, the copy moves to *active* mode. Again, as long as events occur and conditions evaluate to true, the cut is propagated. If all locations in the chart are reached, the copy terminates and stops existing.

[8] *We have extended the standard partial order defined on sequence diagrams, so that the first (played-in) message using a variable precedes other messages that use the same variable.*

[9] *Usually, there will be only one such instance, but some constructs (e.g., precharts, conditions, loops, etc.) may have several participating instances.*

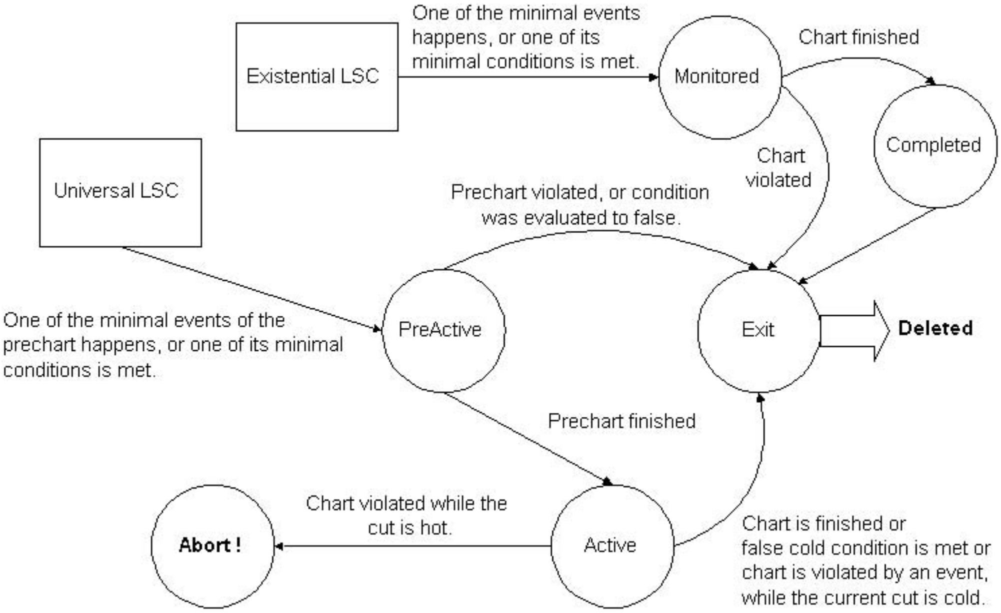

Existential LSC
One of the minimal events happens, or one of its minimal conditions is met.
Monitored
Chart finished
Completed
Chart violated
Universal LSC
Prechart violated, or condition was evaluated to false.
One of the minimal events of the prechart happens, or one of its minimal conditions is met.
PreActive
Exit
Deleted
Prechart finished
Chart violated while the cut is hot.
Abort !
Active
Chart is finished or false cold condition is met or chart is violated by an event, while the current cut is cold.

Fig. 15. The life cycle of an LSC live copy

If the prechart is violated or if it meets a false cold condition, the copy terminates and stops existing. A chart in active mode may exit or abort in several cases:

- If a *cold* condition in the main chart evaluates to false, the copy terminates and stops existing. Note that an evaluation of a cold condition located in a sub-chart just changes the flow of events but does not cause the copy to terminate.
- If the chart is violated by an event (i.e., sending or receiving a message) and the temperature of the current cut is *cold*[10] the copy terminates and stops existing.
- If the chart is violated by an event (i.e., sending or receiving a message) and the temperature of the current cut is *hot*, the chart aborts, since an illegal run was just produced.
- If a *hot* condition is evaluated to false, the chart aborts, since an illegal run was just produced.

The last case does not occur during the execution of our algorithms (with one exception), since these algorithms avoid evaluating false hot conditions. The exceptional false hot conditions that are evaluated by the execution mechanisms are those that contain the reserved word 'FALSE' explicitly. Such conditions can never become true and are usually used to indicate anti-scenarios. Therefore, the user would like to be notified when such a condition is reached. The third case cannot be totally prevented by the choices of the algorithms, since violating events may be caused by the user or the environment. However, the algorithms will avoid the initiation of violating events. This policy may yield a situation where the play-engine does not trigger events dictated by one chart because of other charts forbidding them.

Existential LSCs go through a similar, though simpler, life cycle. When one of the minimal events in the chart occurs, the copy is created and enters the *monitored* state. If the chart is violated, it is simply exited and deleted. If the chart completes successfully, it moves through a temporary *completed* state where different registration and management actions are taken and then is exited and deleted.

7.2 High level architecture and main functions

Figure 16 shows the main components involved in the way the play-engine executes, monitors, records and re-plays, and the ways these activities interact with each other.

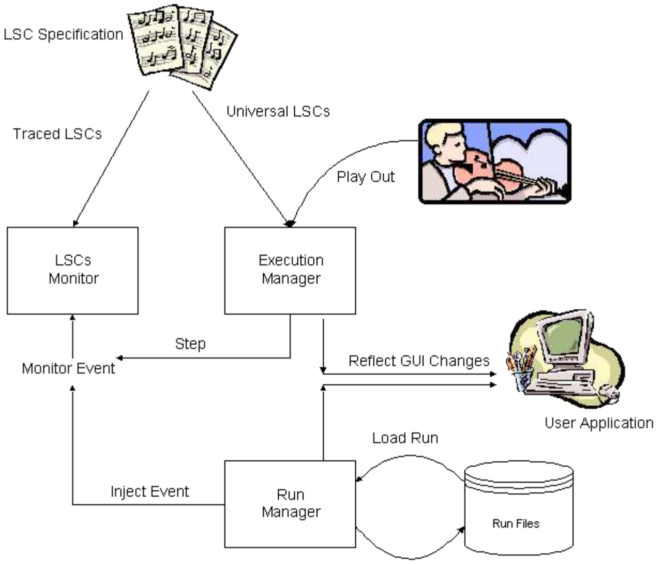


Fig. 16. High-level Architecture of LSCs execution

[10] 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.

The *Execution Manager* is the main component, responsible for processing user actions, activating universal charts accordingly and generating system reactions as dictated by the active universal charts. The *Run Manager* is responsible for recording runs, saving them to files, loading runs from files and re-playing loaded runs. Both components send information about the events played, generated and re-played to the *LSCs Monitor*, which is responsible for managing and monitoring the LSCs selected to be traced. The execution manager and run manager also send information about generated and re-played events to the GUI application, so it can reflect the changes that result from system reactions.

Figure 17 shows the call graph of the main functions used in the various execution processes. We describe the purpose of the algorithms and the way they are used from bottom to top.

The procedure *Unify Messages* handles message unification [36]. Since messages may be symbolic and contain variables as well as functions, we must determine when two messages can be unified. We distinguish between *positive* unification, which is used to find enabled events that can be advanced simultaneously, and *negative* unification, which is used to find events that may cause chart violation if some event is to be carried out. In positive unification we allow the binding of variables, while in negative unification we do not. Thus, if there is a disabled event whose variables are not bounded, it will not be unified with an event that is about to be carried out, and thus will not cause a violation. We have adopted this approach, since it is possible that by the time the event is enabled it will be bound to other values. A more formal discussion of variables and unification within LSCs is given in [31].

The function *Find Unifiable Event* gets an event and an LSC and determines whether there is an event in the chart

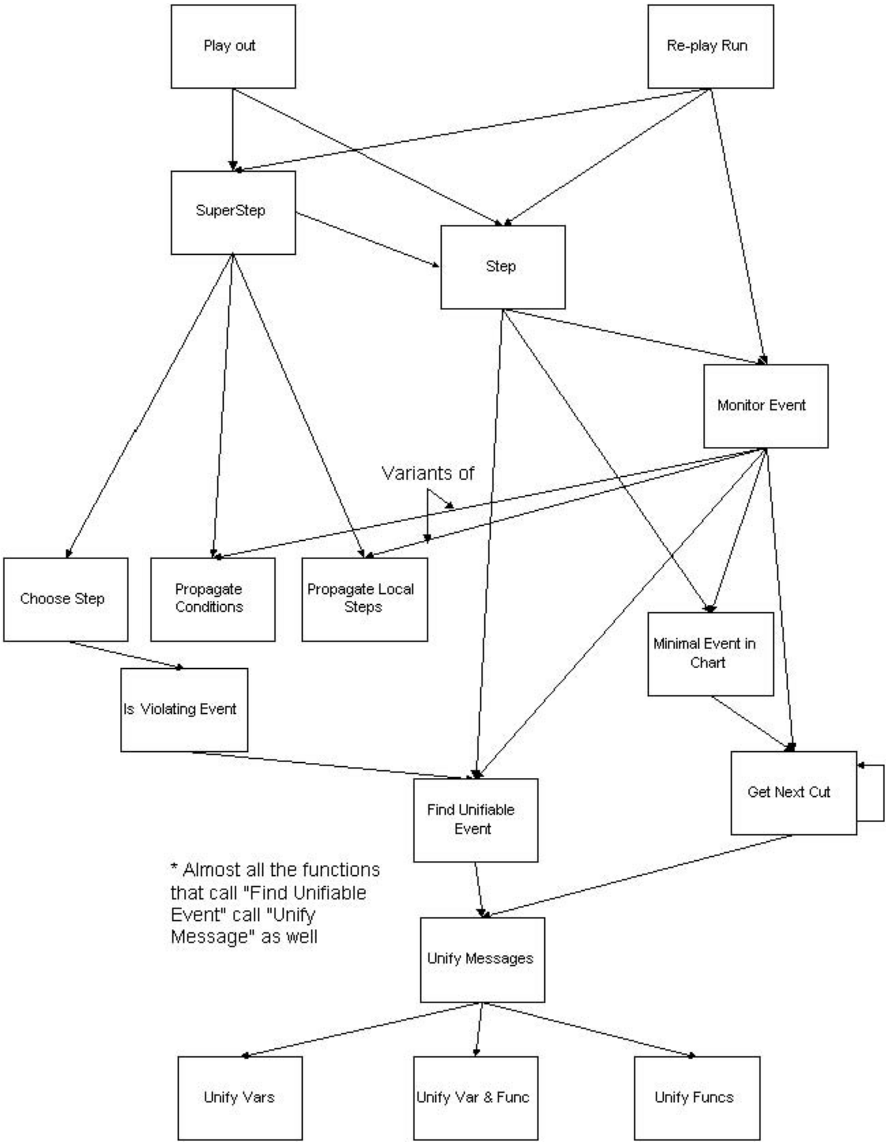


Fig. 17. Main algorithms in the LSC execution process

that is unifiable with the given one. The function may be restricted to look only for events that are enabled throughout the whole chart. The recursive function *Get Next Cut* receives an LSC with a given cut and an event. This function determines whether there is a unifiable event in the chart that is reachable from the given cut by performing local steps. The local steps include performing assignments, evaluating conditions, entering flow constructs, and even skipping *dynamic loops* (which the user might indicate should be carried out 0 times). The function returns the new cut, as well as the unifiable event, since it could be that other instances that are not directly involved with the event have changed their location.

The function *Minimal Event in Chart* receives an LSC and an event and determines whether there is a minimal event that is unifiable with the given one. This function uses *Get Next Cut*, since it is possible that the event may be reached only after propagating some local elements. *Is Violating Event* is a function that determines whether a given event has a matching unifiable event in the currently active LSC copies, which may cause chart violation if performed. The function *Choose Step* is responsible for finding the next event to be carried out. It searches through the live LSC copies and tries to find an event that does not violate any other chart. The procedures *Propagate Local*

```
procedure Step (Evnt)
begin
  To-Be-Executed ← $\phi$
  Context ← new Context
  foreach $C_L \in \mathcal{RL}$ (according to LSC-Search List) do
    if there is an event $e$ enabled by $C_L$ and (positively) unifiable with Evnt then
      Context.Unify(Evnt,e)
      To-Be-Executed ← To-Be-Executed $\cup\{\langle C_L, e\rangle\}$
    else if the mode of $C_L$ is preactive or active in a cold cut and
      $e$ is an event that violates $L$ (negatively) unifiable with Evnt then
      delete $C_L$ from $\mathcal{RL}$
    else if $C_L$ is active in a hot cut and
      $e$ is an event that violates $L$ (negatively) unifiable with Evnt then
      delete $C_L$ from $\mathcal{RL}$
      Report a Requirements Violation Error
    end if
  end foreach
  foreach $L \in \mathcal{S_U}$ do
    if $e$ is a minimal event in the prechart of $L$ unifiable with Evnt then
      Context.Unify(Evnt,e)
      create a copy of $L$, $C_L$.
      set the mode of $C_L$ to preactive
      insert $C_L$ into $\mathcal{RL}$
      To-Be-Executed ← To-Be-Executed $\cup\{\langle C_L, e\rangle\}$
    end if
  end foreach
  if Evnt is free then
    assign values to the free variables (according to Context) of Evnt
  end if
  bind variables in Symbol Table according to Context.
  foreach $\langle C_L, e\rangle \in$ To-Be-Executed do
    Advance the cut of $C_L$ to be the $e - successor$ of the current cut.
    if all events in the prechart of $C_L$ were traced then
      set the mode of $C_L$ to active
    end if
    if all events in the main chart of $C_L$ were traced then
      delete $C_L$ from $\mathcal{RL}$
    end if
  end foreach
  // Send the event to LSCs-Monitor so it can be monitored
  LSCs-Monitor.Monitor-Event(Evnt)
  if the run is recorded then
    Run-Manager.Record-Event(Evnt)
  end if
end .
```

Fig. 18. Procedure step

Steps and *Propagate Conditions* are responsible for propagating local steps (e.g., performing assignments, entering and exiting flow constructs etc.) and conditions, respectively.

The procedure *Step* is responsible for executing a single event. Given an event, it finds all the enabled events that are unifiable with the given one, advances the appropriate copies, activates universal charts that have this event as a minimal one in their prechart and terminates charts that are completed or violated. The pseudo-code of *Step* is given in Fig. 18.

The procedure *Super-Step* tries to perform as many *steps* as possible (excluding user and environment initiated actions). It works in iterations, where in each iteration it chooses a step to be taken and then performs it. Between any two steps, the procedure tries to evaluate conditions according to the *condition evaluation policy* and to propagate local steps. During play-out, this procedure is invoked after every user action, thus causing the system to complete its derived reaction to that action. The procedure *Monitor Event* is very similar to *Step*, in the sense that it also finds all the events that are unifiable with the one to be carried out and tracks them. It activates traced charts when one of their reachable minimal events occurs and closes them when completed or violated. The main difference is that this procedure is more "passive" than Step, in the sense that it waits for an event to happen before it evaluates conditions or enters loops, so that these actions will be taken only when needed and will not block other potential courses of progress.

Given all the procedures described so far, the ones for playing out a scenario and replaying recorded or imported runs are quite straightforward. These are shown in Fig. 19. Playing out a scenario is carried out by a loop. In each iteration a user action is translated into an event, the event is executed and a super step is performed to complete the system reaction.

Re-playing a run is done by going through the list of events and injecting them into the *LSCs-Monitor* so that they can be monitored. A run can also drive universal LSCs, which is useful when the run contains only user and environment events, and is thus used as the "recording" of a regression test. If the option of activating universal charts is chosen, every event is sent to the procedure *Step* instead of *Monitor-Event*, and then a *Super-Step* is performed to complete the derived system response.

8 Integration with other tools

We have set up the play-engine to store played-in specifications in XML format [44].[11] This enables the engine to inter-operate with other kinds of applications, regardless of their internal representation, as we now show.

The play-engine is capable of receiving a system run in a given format (also written as XML) and playing it, as if the run was recorded using the play-engine itself. If the run is complete (i.e., it contains all the events it calls for), the play-engine simply traces all the charts, showing those that are activated at any given point in time . If the run contains only user/environment actions, the play-engine will operate as if the run were input by the user doing playing out, by activating the universal charts and causing the application to react according to them. This capability of playing runs that come from other sources can be very useful. For example, in [16] an algorithm is given for checking the consistency of an LSC specification. This algorithm can be implemented to provide a counter example run when the specification is inconsistent, which can then be played out by the engine, so that the user can track the reason for the inconsistency.

Another kind of inter-operability can be achieved with system implementations. Suppose that an implementation is constructed after the specification has been written (by applying an appropriate synthesis algorithm, by constructing a statechart model or by writing code explicitly). This implementation can be set up to record the runs it produces, and these can then be re-played by the engine, so that the user can see if they comply with the original requirements.

Besides these possibilities, the engine is currently able to create an LSC representation in a format readable by a tool we have developed for transforming LSCs into temporal logic [26]. The TL version of the specification can

(a)

```
procedure Play-Out-Scenario ()
begin
  e ← Get-User-Action()
  while e ≠ stop do
    Step(e)
    Super-Step()
    e ← Get-User-Action()
  end while
end .
```

(b)

```
procedure Inject-Event (Evnt, Apply-Univ-LSCs)
begin
  if Apply-Univ-LSCs then
    Step(Evnt)
    Reflect the effect of Evnt in the GUI
    Super-Step()
  else // Monitor events only
    monitor-Event(Evnt)
    Reflect the effect of Evnt in the GUI
  end if
end .
```

Fig. 19. Top level procedures

[11] For a more detailed discussion of the advantages of XML as an interchange format, see [41].

then be read in by model checkers and other verification tools, as discussed in Sect. 9.6.

9 Overview of advanced topics

In this section we give a short overview of some advanced extensions to the language of LSCs and to the play-in/play-out approach. All these extensions have been implemented within the play-engine, and are, or will be, described in separate publications.

9.1 Extending LSCs with multiple instances

Sequence diagrams, including MSCs and LSCs, are of limited value if the requirements themselves can refer only to fixed concrete objects and to constant and limited information being passed between them.

To overcome these deficiencies, we have extended the language of LSCs. A symbolic instance, associated with a class rather than with an object (possibly parameterized by a variable or other expression), may stand for any object that is an instance of the class (and which satisfies the expression). We extend the language by defining new constructs and their visual representations. We also give rigorous semantics (which has to deal with many subtle issues such as identification, unification and binding mechanisms) and extend the play-out mechanism to fully execute symbolic LSCs as well.

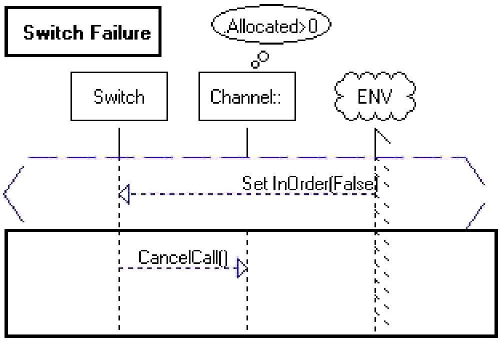


Fig. 20. A universally quantified symbolic instance

Figure 20 shows an LSC with a universally quantified symbolic instance (denoted by the solid line of the instance head and balloon). This LSC states that when the switch gets out of order, it sends a cancellation message to all the currently allocated channels.

The various issues, challenges and algorithms involved with the issue of symbolic instances are described in detail in [18, 31].

9.2 Time and real-time systems

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 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.

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 basic approach of 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.

Figure 21 shows how commonly used timing constraints can be expressed using our extensions to LSCs. Figure 22 shows how a single time event can be referred to in the extension.

As to the play-engine implementation, the play-in part provides convenient ways to define timing con-

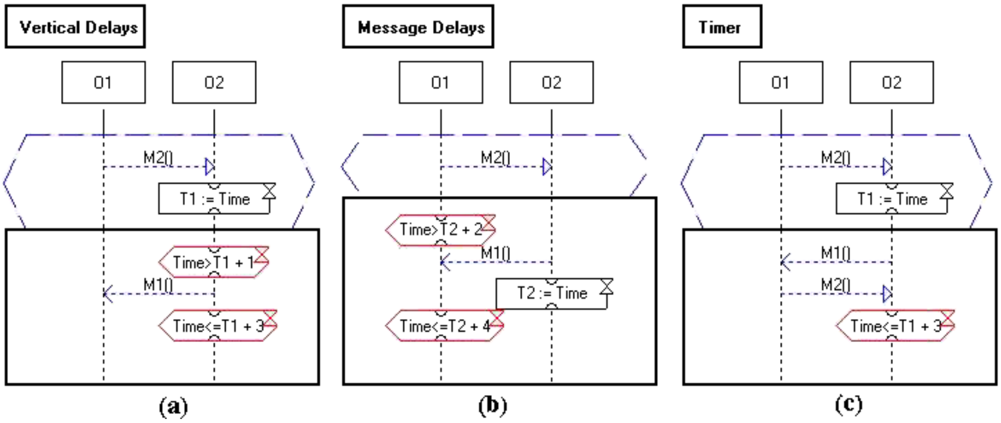


Fig. 21. Expressing basic timing constraints in time-enriched LSCs

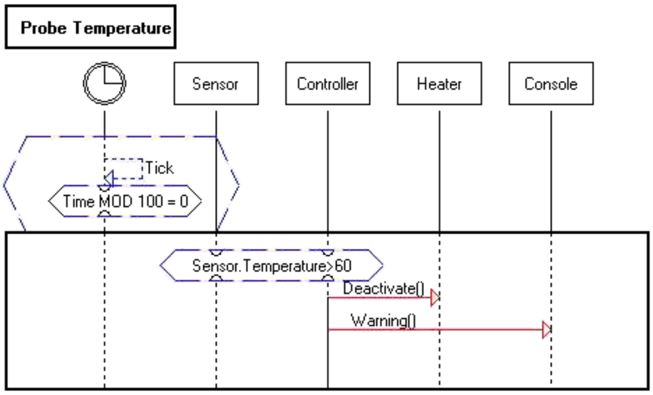


Fig. 22. Time events in time-enriched LSCs

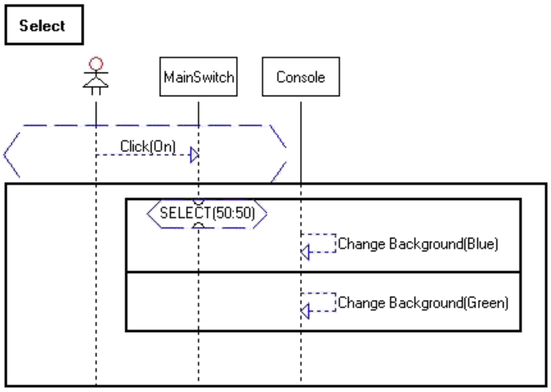


Fig. 23. Non deterministic choice in LSCs

straints between any two events (not restricted to being in a single instance), and for defining time events. 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 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.[12]

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.

The time extensions of the LSC language and the modifications to the execution mechanism are described in detail in [18, 19].

9.3 Non-deterministic choice

When executing an LSC specification, there are many non-deterministic choices that are carried out by the play-engine. This non-determinism originates from the fact that the events in a single chart are only partially ordered and due to the interleaving of events from multiple, simultaneously active, LSCs.

We have extended the language with a *SELECT(P)* command to enable the specification of an intended non-deterministic behavior of the system. A SELECT command is placed in a condition construct and is associated with a probability for a successful evaluation. Using SELECT in a cold condition enables a non-deterministic exit from a (sub)chart, while using it in an if-then-else construct allows for a non-deterministic choice between alternatives. Figure 23 shows an example of non-deterministic choice with equal probabilities for both alternatives. Non-determinism and the SELECT command are described in [18, 30].

9.4 Forbidden elements

A message that appears in a chart must occur only when it is expected, otherwise, the chart is violated. We sometimes want to say that a message should not appear while a chart is active, even if it does not appear explicitly in the chart. A condition too is evaluated only when reached. Similarly, we sometimes wish to specify an invariant, i.e., a condition that must (or is not allowed to) hold while a chart is active.

We have extended the LSC language with forbidden messages and conditions. A forbidden element can constrain the entire LSC, its prechart, its main chart or any subchart therein. Forbidden elements can be hot or cold. If a hot forbidden message occurs or a hot forbidden condition is evaluated to true, while the LSC is in the constrained scope, this is considered a violation of the requirements. If a cold forbidden message occurs or a cold forbidden condition becomes true, the constrained scope is gracefully exited. Figure 24 shows an LSC with forbidden messages and conditions. As the mouse is placed over a forbidden element, the play-engine graphically connects the element with its forbidden scope and as it is placed over a subchart, the subchart is connected with all the elements it is constrained by (as shown in the figure).

We have modified the monitoring mechanism to support detection of forbidden messages and conditions, so that when such a message occurs or a condition becomes true, the proper actions are taken. We have also modified the execution mechanism to consider forbidden messages and conditions in the algorithm for choosing the next event to be carried out during play-out. The algorithm tries to avoid choosing events that are forbidden in some active charts. Only if no other event is enabled, events that are coldly forbidden may be selected. Forbidden conditions are also considered in this algorithm. Before an event is selected, the algorithm simulates the immediate impact of the event on the objects in the system, and if

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

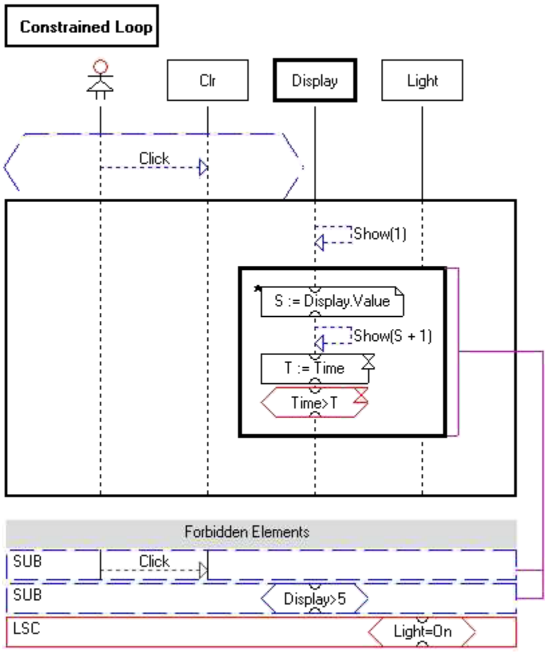


Fig. 24. Forbidden messages and conditions

this event causes some hot forbidden message to become true, it is not selected to be executed.

Forbidden elements are described in detail in [18, 30].

9.5 External objects

Often, a reactive system works in the presence of other elements, besides the user who interacts with it. Such elements might include other machines, computers, sensors, and even Mother Nature. The collection of all these elements is referred to as the system's *environment*. When playing in the required behavior of a reactive system, it is necessary to be able to express its interaction with this environment. We have shown how the play-engine allows the user to specify the environment's interaction with the system in a way similar to the interaction with the end user. The interaction with the environment is limited to object property changes, and it thus suits interactions with nature more than communication with interfacing systems.

We have extended the LSC language to refer to external objects. External objects can be added on the fly in the play-engine, in a similar way to internal objects. The execution mechanism was modified not to initiate events that originate from external objects, much as it does not initiate events from the user or the external environment. While playing out, the user can simulate property changes of external objects and can initiate calls from external objects to objects that are internal to the system. Thus, the user can control the behavior of different external components and systems while the target system is being executed and analyzed. Figure 25 shows an LSC with an external object (the switch). External objects are discussed in [18, 30].

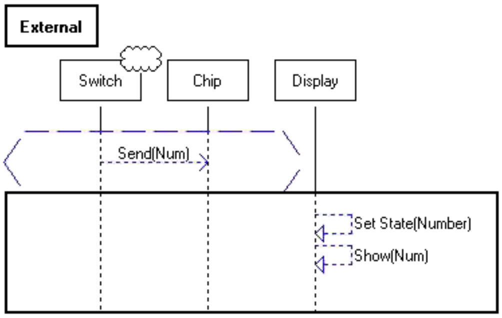


Fig. 25. External objects in LSCs

9.6 Smart play-out

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. This process is rather naive: For example, there can be many sequences of events possible as a response to a user event, and some of these may not constitute a "correct" super-step. We consider a super-step to be correct if when it is executed no active universal chart is violated. The multiplicity of possible sequences of reactions to a user event is due to the fact that a declarative inter-object behavior language such as LSCs allows high-level requirements given in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. The partial order semantics among events in each chart and the ability to separate scenarios in different charts without saying explicitly how they should be composed are very useful in early requirement stages, but can cause under-specification and non-determinism when one attempts to execute them.

With Hillel Kugler, we have enhanced the play-engine with an ability we call *smart play-out*. Smart play-out focuses on executing the behavioral requirements with the aid of formal analysis methods, mainly model-checking. Our smart play-out uses model-checking to find a correct super-step if one exists, or proves that there isn't one. We do this by formulating the play-out task as a verification problem, in such a way that a counter-example resulting from the model-checking will constitute the desired super-step. The transition relation is defined so that it allows progress of active universal charts but prevents violations. The property to be checked is one that states that always at least one of the universal charts is active. In order to falsify it, the model-checker searches for a run in which eventually none of the universal charts is active; i.e., all active universal charts completed successfully, and by the definition of the transition relation no violations occurred. Such a counter-example is exactly the desired super-step. If the model-checker manages to verify the property then no correct super-step exists.

The other kind of thing smart play-out can do is to find a way to satisfy an existential chart. Here we cannot limit ourselves to a single super-step, since the chart under scrutiny can contain external events, each of which triggers a super-step of the system. Nevertheless, the formulation as a model-checking problem can be used with slight modifications for this task too. Also, when trying to satisfy an existential LSC, we take the approach that assumes the cooperation of the environment. We should add that the method for satisfying existential LSCs can also be used to verify safety properties that take the form of an assertion on the system state. This is done by putting the property's negation in an existential chart and verifying that it cannot be satisfied.

Currently, smart play-out can be applied only to a subset of our extended LSC language, including messages, conditions and assignments, but excluding, time, symbolic instances and forbidden elements. Smart play-out is integrated as a module in the play-engine, and is discussed in detail in [17].

10 Related work

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

Amyot and Eberlein [3] provide an extensive survey of scenario notations. Their paper also defines several comparison criteria and then uses them to compare the different notations. It seems that LSCs, the scenario language we use as the formal rendition of our behavioral requirements, scores high on most of the criteria presented therein: it is component centered, it can encapsulate several runs in a single scenario, it can be abstract, and can relate to internal objects and not only to the system as a whole. Moreover, it is highly visual, a criterion which is very important when dealing with complex systems. The survey in [3] does not refer to some of the additional issues crucial to sequence-based languages, that were raised in [8], such as the ability to specify anti-scenarios, and to distinguish between "must" and "may" behaviors, etc., for which LSCs were in fact developed.

There are a number of commercial tools that successfully handle the execution of graphical models (e.g., Statemate [20] and Rhapsody by I-Logix [23], ObjectTime [39], and Rose-RT by Rational [35]). Some of these tools can be connected to a GUI mockup (or a real target system) and will activate it as the execution progresses. However, these tools all execute an intra-object design model (usually, statecharts) rather than an inter-object scenario-based 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, and they use GUIs for model execution but not for capturing the requirements.

In a recent, independently written paper, Lettrai and Klose [27] present a methodology supported by a tool called TestConductor, which is integrated into Rhapsody [23]. The tool is used for monitoring and testing a model using a subset of LSCs. The charts can be monitored in a way that is similar to the way we trace existential charts. In order to be monitored, however, their charts are transformed into Büchi automata (which could significantly increase the size required to store each chart). Their work also briefly mentions the ability to test an implementation using these sequence charts, by generating messages on behalf of the environment (or other un-implemented classes).

Work has been done concerning the execution of formal specifications in non-graphical languages. For example, [40] and [32] present an execution and animation framework for specifications in Z, whereas [22] does so for the language Albert II. In addition to being non-sequence-based design models and not having a play-in-like capability, the animation in these tools does not use the target application GUI.

Magee et al. [29, 42] present a methodology supported by a tool called LTSA for specifying and analyzing labeled transition systems (LTSs). This tool works with an animation framework called SceneBeans [34], yielding a nicely animated executable model. The model has to be an LTS, which, again, is more akin to the intra-object statecharts than to inter-object sequence based behavior, and it will usually be larger and more detailed than sequence charts. The behavior is written in FSP [28] and is compiled into LTSs, a process that appears to be somewhat less intuitive than play-in. An interesting idea would be to use SceneBeans as an animation engine to describe the behavior of internal (non-GUI) objects in our play-engine.

Dromey [9] 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. GSE is similar to our work in two aspects: it 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). It also uses a richer specification language 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 any play-in mechanism or model execution capabilities on the requirements level.

The Software Cost Reduction (SCR) method described in [21] also allows the specification and simulation of requirements for reactive systems. The SCR method provides a tabular notation for specifying the required relation between system and environment variables. According to this method, the specification is first modified, with the aid of an automated tool, to be deterministic and can then be simulated, using a graphical user interface for capturing user inputs and reflecting the system state. In [4] model-checking methods are used to verify that a complete SCR model satisfies certain properties, by using SMV and Spin model-checkers. SCR is similar to our work (and to other work as described above) in the fact that it uses a GUI in the final phase of the simulation. It is different from our work in the languages used (i.e., tables of variables vs. visual formalisms) and in the fact that the requirements are not played in. The SCR requirement that the final requirements should reflect a deterministic model is also different in concept from our work.

Interaction interfaces [5] are used to formally specify the interaction between two or more components that co-operate as subsystems of a distributed system. Their format uses predicates to characterize sets of interaction histories, and they show how to derive component specifications from a general interaction interface specification (though for liveness properties some external intervention is required to assign responsibilities to specific components). The issue of realizability of components is also studied in that paper. The same authors also co-designed a synthesis algorithm from MSCs to state machines, appearing in [25].

Boger et al. [6] present a development methodology, called *extreme modeling* (XM), which tries to combine the advantages of the programming methodology of *extreme programming* [45] with the UML [43]. 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 a requirements scenario-based language, yet the effect of the execution can also be shown on collaboration diagrams. Here also, no GUI is used in the requirements capturing process nor in the model execution.

11 Conclusions and future work

In summary, we have substantiated the idea of specifying system behavior by playing in scenarios directly from friendly GUI applications [13], and in so doing have also worked out a method to play the behavior out directly. Play-in makes capturing requirements quite intuitive, thus enabling non-professional end-users to participate in the process. It is worth noting that as more complex and sophisticated features of the language are used, the user is expected to be more familiar with the language of LSCs. Hence, playing in is somewhat like programming in an intuitive, visual and high-level programming environment. Play-out allows even more end-users to operate the GUI and validate the requirements by actually operating the application. All this seems to have far-reaching potential applications in many stages of system development, including requirements engineering, specification, testing, analysis and implementation. To support the methodology we have built a play-engine development environment.

Among other things, the play-in/play-out methodology substantiates the link between informal use cases and detailed requirements (e.g., in LSCs or temporal logic), and makes it more rigorous and useful. In particular, one could view our work as providing an approach and a tool for *executable use cases*. Using the concepts and techniques described herein, we may use the GUI application, or some abstract version thereof, both in specifying desired behavior and in testing and debugging it. When more powerful synthesis algorithms become available this could lead to the automatic generation of implementable models. We are also coming to believe that for many kinds of systems the play-engine methodology could serve as the final implementation too, with the play-out being all that is needed for running the system itself.

Several issues have not yet found their way into the play-engine. Some of these are in research stages, and others we have already worked out and are being implemented. Here are brief descriptions of some of them:

Running Coordinated Play-Engines:

We intend to modify the play-engine to be able to work in coordination with other simulation tools (or more play-engines) in a coordinated mode, thus supporting component based development from the initial phase of requirements analysis and validation. In a project called SEC (Simulation Engines Coordinator) we are in the midst of defining both a standard interface and an interaction protocol so that several simulation engines (specifically, play-engines) can be used together, each executing a different part of the system. By using SEC, one could execute a system, some parts of which are implemented, some only designed (e.g., with statecharts), and some only in the requirements phase, described by LSCs in the play-engine.

Integration of Synthesis Tools:

After the user has finished playing in the system's behavior and has debugged it by playing out, the next desired step (if the resulting executable model is still inadequate) would be to move smoothly into the next phase – preparing an intra-object model – that would lead to implementation (see [13]). Accordingly, we would like to integrate into our environment a tool to synthesize a system model from the requirements; say, statecharts from the LSCs. A first-cut algorithm for this appears in [16], and efforts are underway to improve it and come up with a practical and implementable approach to synthesis that will link fruitfully with the play-engine.

GUI Development Environments:

We have developed a prototype environment for constructing GUI applications that are "aware" of the play-engine. This environment consists of an add-on to Microsoft Visual Basic and a set of components that can be used to create GUI applications. The constructed applications implement all the interfaces required by the play-engine, thus enabling the end user to create GUI applications simply by dragging the components onto an initially blank GUI application. There are also several commercial products that enable engineers to avoid building a real hardware prototype, by having them build one in software (e.g., Altia FacePlate & Design [1], E-Sim Rapid [10], Macromedia Flash [11], etc.). We intend to see to it that such tools can be used together with the play-engine, in order to facilitate the construction of more complicated and sophisticated GUI applications.

LSCs as the Final System?

There appear to be many kinds of systems for which the LSC specification, together with the play-engine, acting as a *universal reactive machine*, may be considered to be not just requirements but the final implementation. For example, a 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 a single line of code, and play-out could serve very well as the executable system. In general, the play-in/play-out methodology could be used to create web-like applications where most of the user interaction is done by clicking objects. 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. Furthermore, electronic home devices such as VCR, toaster-oven, microwave, etc., can be instrumented with a play-engine and their behavior could be determined by end-users, using play-in to create an LSC specification and then loading the specification into the device. In short, we strongly believe there is a potential to use the suggested methodology and tool not only for isolated parts of a development cycle, but throughout the entire cycle. These ideas do not hold as is for systems which are time-critical, or ones that have to be distributed over several machines or processes, but we have some ideas about these too.

Acknowledgements. We would like to thank Hillel Kugler for many inspiring discussions on the material reported upon here. We would also like to thank Evgeniy Bart, Maksim Frenkel and Dan Barak for helping to implement some of the features discussed in Section 11. Finally, we thank the referees of a previous version of the paper for their helpful comments.

References

1. Altia Design & Altia FacePlate, web page: `http://www.altia.com`
2. Alur, R., Henzinger, T.: Real-time System = Discrete System + Clock Variables. Software Tools for Technology Transfer 1: 86–109, 1997
3. Amyot, D., Eberlein, A.: An Evaluation of Scenario Notations for Telecommunication Systems Development. In: Int. Conf. on Telecommunication Systems, 2001
4. Bharadwaj, R., Heitmeyer, C.: Model Checking Complete Requirements Specifications Using Abstraction. Automated Software Engineering, 6(1): 37–68, January 1999
5. Broy, M., Krüger, I.: Interaction Interfaces – Towards a Scientific Foundation of a Methodological Usage of Message Sequence Charts. In: Staples, J., Hinchey, M.G., Liu, S. (eds.) Formal Engineering Methods, IEEE Computer Society, 1998, pp. 2–15
6. Boger, M., Baier, T., Wienberg, F., Lamersdorf, W.: Extreme Modeling. In: Extreme Programming and Flexible Processes in Software Engineering – XP2000. Addison Wesley, 6 2000
7. Microsoft COM, web page: `http://www.microsoft.com/com`
8. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design, 19(1) 2001. Preliminary version in: Ciancarini, P., Fantechi, A., Gorrieri, R. (eds.) Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), Kluwer Academic Publishers, 1999, pp. 293–312
9. Dromey, R.: Genetic Software Engineering. Manuscript, 2001
10. e-SIM Rapid, web page `http://www.e-sim.com/home/`
11. Macromedia Flash, web page: `http://www.macromedia.com/software/flash/`
12. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Prog., 8: 231–274, 1987. (Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)
13. Harel, D.: From Play-In Scenarios to Code: An Achievable Dream. IEEE Computer, 34(1): 53–60, January 2001
14. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. IEEE Computer, 30(7): 31–42, 1997
15. Harel, D., Koren, Y.: Drawing Graphs with Non-Uniform Vertices. In: Proc. of Working Conference on Advanced Visual Interfaces (AVI'02). ACM Press, 2002, pp. 157–166
16. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. Int. J. of Foundations of Computer Science (IJFCS)., 13(1): 5–51, Febuary 2002. (Also, Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000), July 2000, Lecture Notes in Computer Science, Springer-Verlag, 2000.)
17. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements. In: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon, 2002, pp. 378–398. Also available as Tech. Report MCS02-08, Weizmann Institute of Science, 2002
18. Harel, D., Marelly, R.: Come, Let's Play: An Executable Scenario-Based Approach to Reactive Systems. (tentative title), manuscript, 2002
19. Harel, D., Marelly, R.: Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In: Proc. 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02), Fort Worth, Texas, 2002, pp. 193–202
20. Harel, D., Politi, M.: 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

21. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A Toolset for Specifying and Analyzing Software Requirements. In: Hu, A., Vardi, e.M.Y. (eds.) Intl. Conference on Computer Aided Verification (CAV'98), Lecture Notes in Computer Science, vol. 1427. Springer-Verlag, New York, 1998, pp. 5–51
22. Heymans, P., Dubois, E.: Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements. Requirements Engineering Journal 3: 202–218, Springer-Verlag, 1998
23. I-Logix,Inc., products web page: `http://www.ilogix.com/fs_prod.htm`
24. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA, 1992
25. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. Proc. DIPES'98, Kluwer, 1999
26. Kugler, H., Harel, D., Pnueli, A., Lu, Y., Bontemps, Y.: Temporal Logic for Live Sequence Charts. Technical report, Weizmann Institute, 2000
27. Lettrari, M., Klose, J.: Scenario-Based Monitoring and Testing of Real-Time UML Models. In: 4th Int. Conf. on the Unified Modeling Language, Toronto, Lecture Notes in Computer Science, vol. 2185, October 2001, pp. 317–328
28. Magee, J., Kramer, J.: Concurrency – State Models & Java Programs. John Wiley & Sons, Chichester, 1999
29. Magee, J., Pryce, N., Giannakopoulou, D., Kramer, J.: Graphical Animation of Behavior Models. 22nd Int. Conf. on Soft. Eng. (ICSE'00), Limeric, Ireland, 2000
30. Marelly, R.: Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. PhD thesis, The Weizmann Institute of Science, 2002
31. Marelly, R., Harel, D., Kugler, H.: Multiple Instances and Symbolic Variables in Executable Sequence Charts. In: Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), Seattle, WA, 2002, pp. 83–100. Also available as Tech. Report MCS02-05, Weizmann Institute of Science, 2002
32. Özcan, M., Parry, P., Morrey, I., Siddiqi, J.: Visualization of Executable Formal Specifications for User Validation. Ann. Soft. Eng., 3: 131–155, 1997
33. Pnueli, A.: The Temporal Semantics of Concurrent Programs. Theoretical Computer Science, 13: 1–20, 1981
34. Pryce, N., Magee, J.: SceneBeans: A Component-Based Animation Framework for Java. `http://www-dse.doc.ic.ac.uk/Software/SceneBeans/`
35. Rational,Inc., web page: `http://www.rational.com`
36. Robinson, J.: Logic: Form and Function, chap. 11. North-Holland, 1979, pp. 182–198
37. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA, 1999
38. Schlor, R., Damm, W.: Specification and Verification of System-Level Hardware Designs using Timing Diagram. In: Proc. European Conference on Design Automation. IEEE Computer Society Press, Paris, France, 1993, pp. 518–524
39. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons, New York, 1994
40. Siddiqi, J.I., Morrey, I.C., Roast, C.R., Ozcan, M.B.: Towards Quality Requirements via Animated Formal Specifications. Ann. Soft. Eng., 3: 131–155, 1997
41. Suzuki, J., Yamamoto, Y.: Extending UML for Modelling Reflective Software Components. In: France, R., Rumpe, B. (eds.) UML'99 – The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28–30 1999, Proceedings, Lecture Notes in Computer Science, vol. 1723. Springer, New York, 1999, pp. 220–235
42. Uchitel, S., Kramer, J., Magee, J.: Detecting Implied Scenarios in Message Sequence Chart Specifications. In: 9th European Software Engineering Conferece and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'01). Vienna, Austria, September 2001
43. Documentation of the Unified Modeling Language (UML), available from the Object Management Group(OMG): `http://www.omg.org`
44. Web page: `http://www.xml.com`
45. Web page: `http://www.extremeprogramming.org`
46. Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996

David Harel is the William Sussman Professor of Mathematics at The Weizmann Institute of Science in Israel, and has been Dean of the Faculty of Mathematics and Computer Science there since 1998. He is also co-founder of I-Logix, Inc., Andover, MA. He received his PhD from MIT in 1978. He has worked in several areas of theoretical computer science, and in recent years has become involved in other areas, including software and systems engineering, visual languages, graph layout, modeling and analysis of biological systems, and smell communication. He is the inventor of statecharts, and co-inventor of live sequence charts (LSCs), and was part of the team that designed the Statemate and Rhapsody tools. He has received a number of awards, including ACM's Karlstrom Outstanding Educator Award in 1992. His latest books are "Dynamic Logic" (with Kozen and Tiuryn), MIT Press, 2000, and "Computers Ltd.: What They Really Can't Do", Oxford, 2000.

Rami Marelly received his M.Sc in Computer Science in 1991 from the Technion – Israel Institute of Technology. His M.Sc. thesis was in the area of formal verification. In the years that followed, he worked as a programmer, a team leader and a project manager in various kinds of systems, including real-time and GIS systems. He has just finished his Ph.D. in the area of requirements engineering and visual languages at the Weizmann Institute of Science.