Capturing and Executing Behavioral Requirements: The Play-In/Play-Out Approach

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

Technical Report MCS01-15, The Weizmann Institute of Science
Submitted for publication; September 2001

Abstract

A powerful methodology for specifying scenario-based requirements 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 requirements are played in, the play-engine automatically generates a formal version in 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 requirements. The ideas appear to be relevant to many stages of system development, including requirements engineering, specification, testing, analysis and implementation.

1 Introduction

Two kinds of behavior in object-oriented analysis and design are identified and discussed in [5, 9]: *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 [9] there is a discussion of the different roles of these as a requirements and a modeling language, respectively. For modeling intra-object behavior, most object-oriented modeling approaches adopt *statecharts* [8, 10]. For the requirements aspect, one of the most widely used languages is that of *message sequence charts* (MSCs), adopted long ago by the ITU [34], or its UML variant, *sequence diagrams* [31, 25].

According to many OO-based methodologies for system development, the user first specifies the system's *use cases* [16], and the different instantiations of each use case is 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 [9]. 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 [27], and by the Rhapsody tool [15] (based on the executable object modeling work of [10]). In fact the main pair of languages of [10, 15] – namely, object model diagrams and statecharts – constitute the core executable part of the UML.

As discussed in [5], 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. 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) [5]. LSCs distinguish between scenarios that *may* 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 [9], 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 recently [12]. 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 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.

[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.

Towards the end of [9], 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 now worked out the details of this proposal, and have finished building the initial version of its implementation — the *play-engine*, which we describe in this paper. One of the most interesting things to come out of this is the ability to *play out* the requirements, i.e., to execute the LSCs directly, without the need to build or synthesize a 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 a high level, intuitive and user-friendly mechanism. These could include domain experts, application engineers, requirements engineers, and 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 would 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 would click 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 built 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 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 variants of temporal logic [22] or timing diagrams [26].

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 [7]. 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 the requirements 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

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

executing a system model, or the final system, but limiting him/herself to "end-user" and external environment actions only. 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 specification 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.

In this paper we describe in detail the play-in (capture) methodology and implementation, and due to space limitations describe the play-out (execution) mechanism only briefly. A more comprehensive description of the play-out and its underlying algorithms will be published separately. Related work is discussed in Section 8.

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 [5]. There are a couple of extensions we have made to enable the specification of richer and more realistic behaviors. The user is referred to [5] for a more complete description of the original language. We illustrate the LSC elements we use by the sample chart of Fig. 1.

LSCs have two types of charts: *universal* (annotated by a solid borderline) and *existential* (annotated by a dashed borderline). Universal charts are used to specify restrictions over all possible system runs. A universal chart is typically associated with a *prechart* that specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. Existential charts are used in LSCs to specify sample interactions between the system and its environment. Existential charts must be satisfied by at least one system run. They thus do not force the application to behave in a certain way in all cases, but rather state that there is at least one set of circumstances under which a certain behavior occurs. Existential charts can 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.

In the LSC of Fig. 1, the prechart (top dashed hexagon) contains a single message denoting the event of the user clicking the switch to be *on*. Following this, in the chart body, there is an *if-then-else* construct: if the switch's state is *on*, then the light goes *on*, otherwise, it goes *off*. After the *if-then-else* comes a *loop* construct. There are three types of such constructs; this one is an unbounded loop, denoted by a '*', which means that it is performed an *a priori* unknown number of times. It can be exited when a *cold condition* inside it is violated, as described

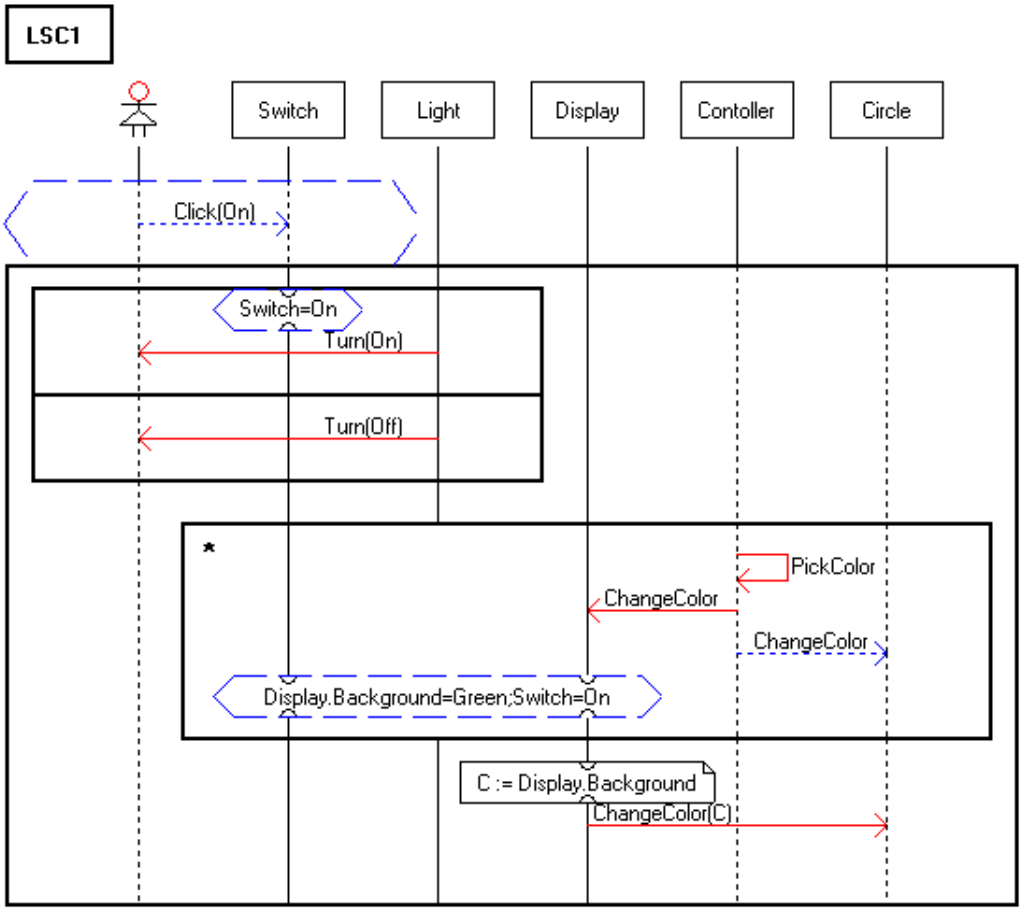

Figure 1: Example of an LSC

shortly. There are also *fixed* loops, annotated by a number or a variable name and performed a fixed known number of times, and *dynamic* loops, annotated with a '?' and for which the user determines the number of iterations at run time. Inside the loop of Fig. 1, the controller picks a color (say, at random) and sends it to the *display* object and to the *circle* object.[3] The message to the display is *hot*, denoted by a solid line, while the message to the circle is *cold*, denoted by a dashed line. The semantics is that if a hot message is sent it must be received, whereas a cold message may be lost and never received.

The loop ends with a *cold condition* that requires the background color of the display to be *green* and the state of the switch to be *on*. If a cold condition is *true*, the chart progresses to the location that immediately follows the condition, whereas if it is *false*, the surrounding (sub)chart is exited. A *hot* condition, on the other hand, must always be met, otherwise the requirements are violated and the system aborts. Note that placing a cold condition C at the

[3]We have added two rather unusual objects to the calculator, a *circle* and a *slider*, which don't play important parts in calculating with the calculator but serve us in examples. They appear at the bottom left of the GUI.

beginning or end of an unbounded loop creates *while* C *do* and *repeat until* $\neg C$ constructs. In our current implementation, we support *conjunctive-query* conditions, namely ones that are conjunctions of primitive equalities or inequalities. In Fig. 1, the controller will continue picking colors until a color other than *green* is chosen, or the switch is turned *off*.

After the loop comes an *assignment* element. Assignments are internal to a chart and 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, as in conditions, synchronize at the location of the assignment. Synchronizing at an assignment (condition) means that none of the synchronized instances may progress beyond the assignment (or condition) until all of them reach it and it is actually performed (or evaluated). In the assignment shown in Fig. 1, the background color of the display is stored in the variable C.

Following this assignment, the display sends the stored color to the circle object, this time using a hot message. Here also, we extend the language of LSCs, by enabling messages to be *exact*, as in the original definition of [5], or *symbolic*. A symbolic message uses variables whose values may vary in different runs.

The switch, light and display objects all consist of hot locations (denoted by solid instance lines), thus forcing their progress, while the circle and controller 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 [34] and UML sequence diagrams [31]), 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 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.[4]

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 system reactions. User actions are specified simply by operating the GUI application in the way it would be done in the final product. This includes clicking buttons, rotating knobs, flipping switches, etc. 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. The user defines conjunctive conditions also 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 the 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).

[4]The current implementation of our play-engine uses a COM [4] interface, but we could have used any appropriate agreed-upon format.

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 Section 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).

4 Playing out Behavior

Playing out is the process of testing the behavior of the system by providing 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 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.

We should emphasize that in the presence of symbolic messages and dynamic/unbounded loops, 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. (Dynamic loops enable the user to decide on the fly how many times they should be executed.)

Thus, by playing out scenarios the user actually tests the behavior of the specified system directly from the 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. This process is simple enough for many kinds of end-users and domain experts, and can greatly increase the chance of finding errors early on.

If the specification is large and the user wishes to focus only on certain aspects of the system behavior, he/she may specify which universal charts will participate in the play-out process. Note that a single universal chart may become activated (i.e., its prechart is successfully completed) several times during a system run. Some of these activations might overlap, resulting in a situation where there are several "copies" of the same chart active simultaneously.

A number of things happen during play-out, including the following:

1. Charts are opened whenever they are activated and are closed when they are violated or when they terminate. Each displayed chart shows a "cut" (a kind of rectilinear "slice"), denoting the current location of each instance.

2. The currently executed event is highlighted in the relevant LSCs.

3. The play-engine interacts with the GUI application, causing it to reflect the change in the GUI, as prescribed by the executed event.

4. The user may examine values of assignments and conditions by moving the mouse over them in the chart. Whenever relevant, the effects show up in the GUI.

Play-out sessions can also be recorded and re-played later on.

So much for the universal charts, which drive the behavior and are activated when needed. In contrast, 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*. By monitoring a chart, the play-engine simply tracks the events in the chart as they occur. When (and if) the chart reaches its end, it is highlighted and the user is informed that it was successfully traced to completion. These runs can be recorded as well, to provide testimonies (that can be re-played) for fulfilling the promises made by existential LSCs. 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.

We should note that no intermediate representation is constructed during play-out. Thus, the amount of memory used is linear in the number of simultaneously active charts. This is significant, since even in very large systems the requirements specification itself is expected to be highly modular, and only a small fraction of the behavioral specification is expected to be active at any given time

Due to space limitations, the detailed description of the play-out process and its implementation is deferred to a separate paper.

5 The Play-Engine Environment

We now describe the main elements involved in play-engine's development environment; see Fig. 2. They are numbered 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:

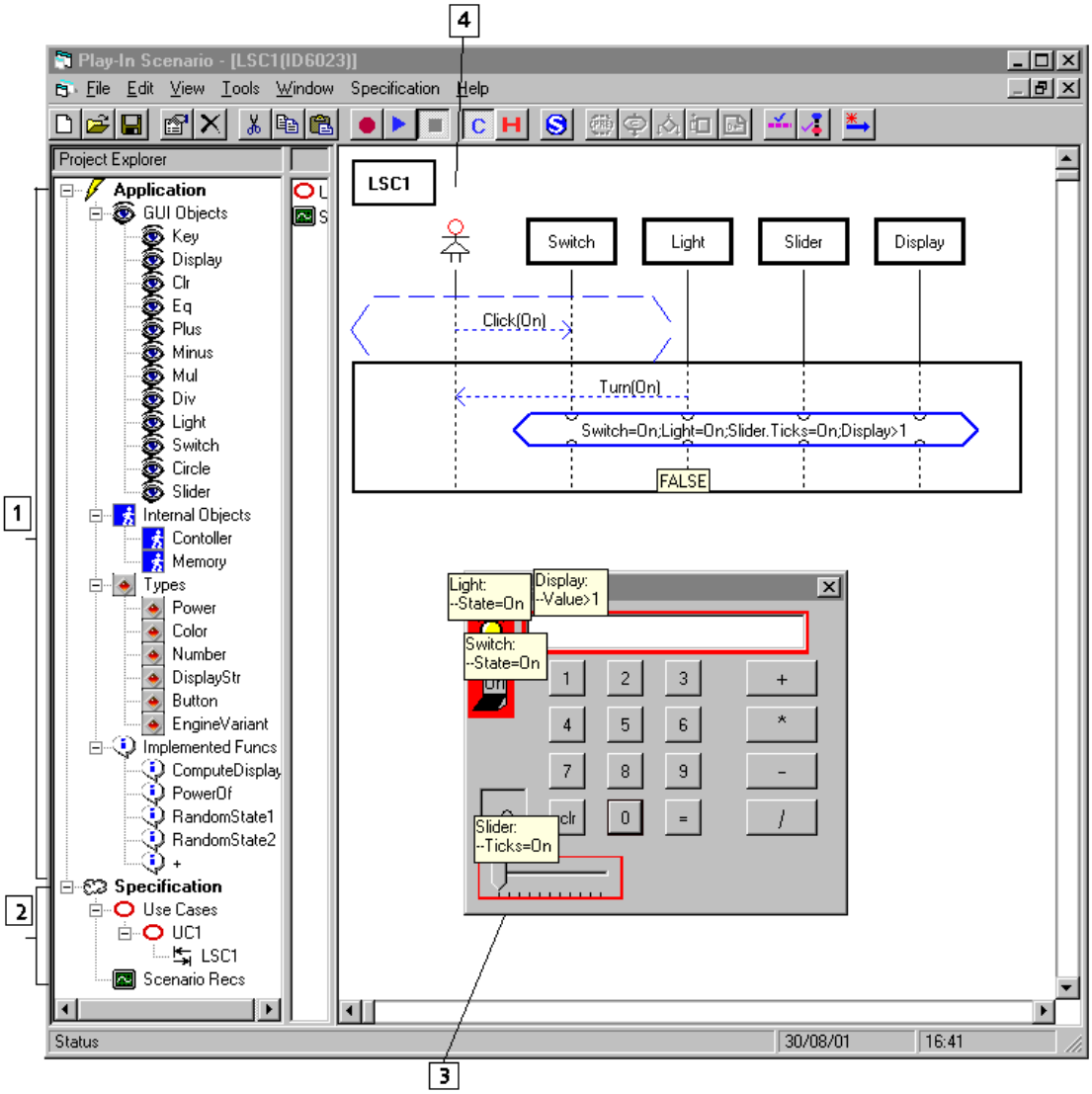

Figure 2: The play-engine environment

- The 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.
- The internal objects defined in the application (in our example there are two — Controller and Memory).
- The types upon which the properties of objects in the application can be based (e.g., Color, Number, etc.)
- The 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.

   - The use cases and LSCs. This is the main part of the requirements specification, and it consists of the use cases defined by the user and the LSCs implementing/instantiating them. As in many methodologies, the user starts by identifying a use case and giving it a name and a short description. The LSCs are those constructed by the engine as a result of the play-in.
   - Scenario recordings of system executions. They are intended to help in future uses of the engine, and will not be discussed here.

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 was written in Visual Basic. In Section 9 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.[5]

[5]Since the rectangular tool tips may overlap, we have used a variant of the layout algorithm of [11] 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.

6 A Sample Play Session

We now illustrate the methodology with a short play-in and play-out session. We use the pocket calculator even though it is somewhat trivial, since it combines characteristics of real systems, like high reactivity and computations with the need for symbolic representations of certain scenarios.

6.1 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[6] 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*. Fig. 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 a message in the LSC from the instance representing the selected object to the *user* instance, 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

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

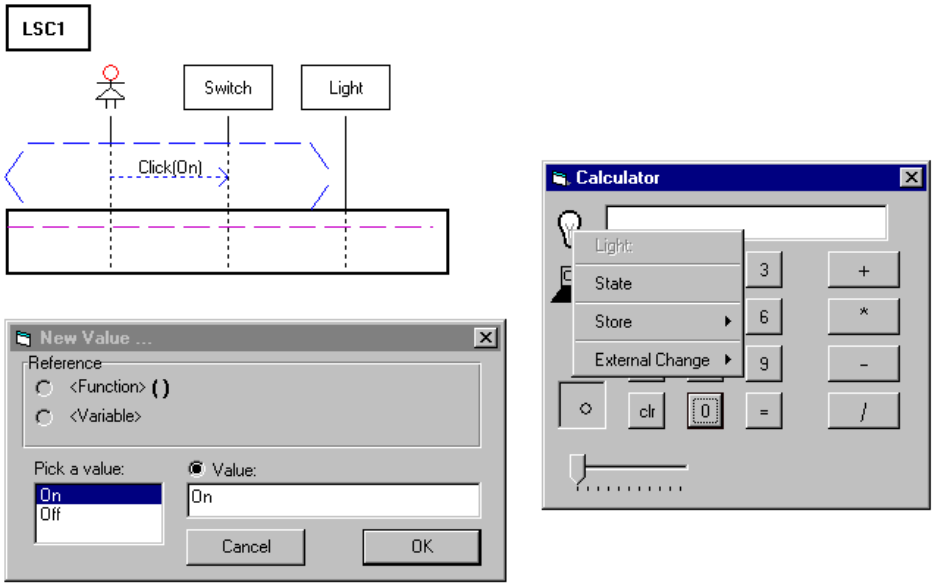

Figure 3: Turning the calculator on

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 here several kinds of relation operators can be used (e.g., $<, \leq, >$, etc.). Fig. 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 being 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

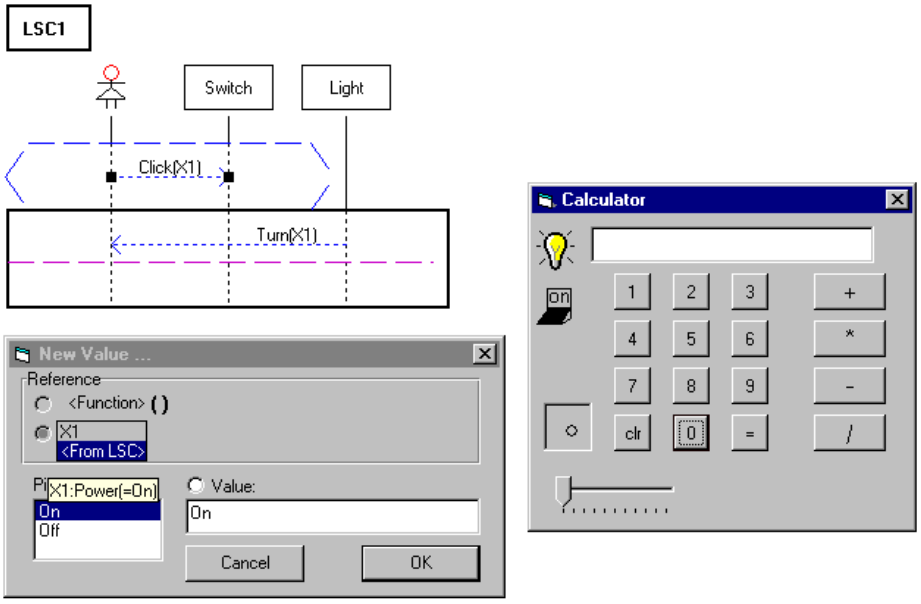

Figure 4: Using symbolic messages

`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 connectors at the intersection points of the 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. 9 (a)). 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

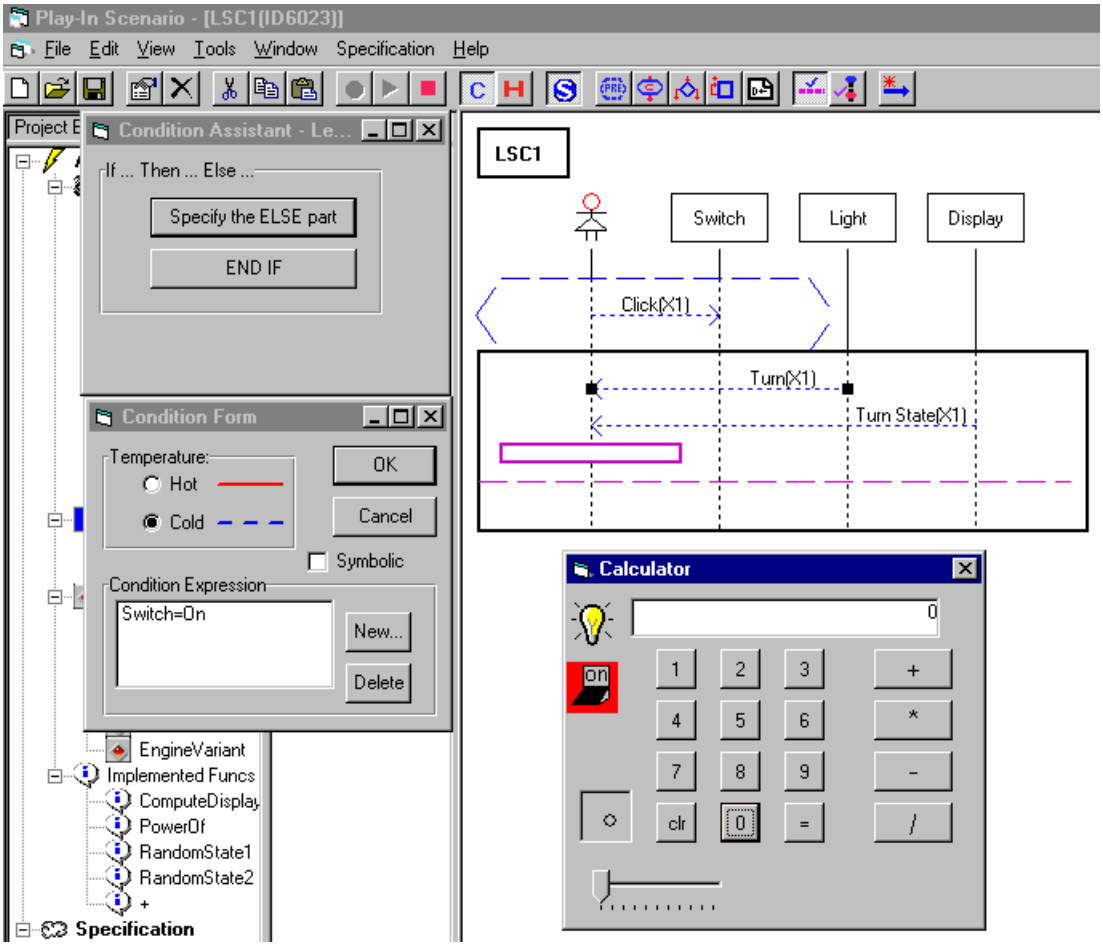

Figure 5: Specifying a condition

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 $Num1 + Num2$.[7] 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

[7] 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.

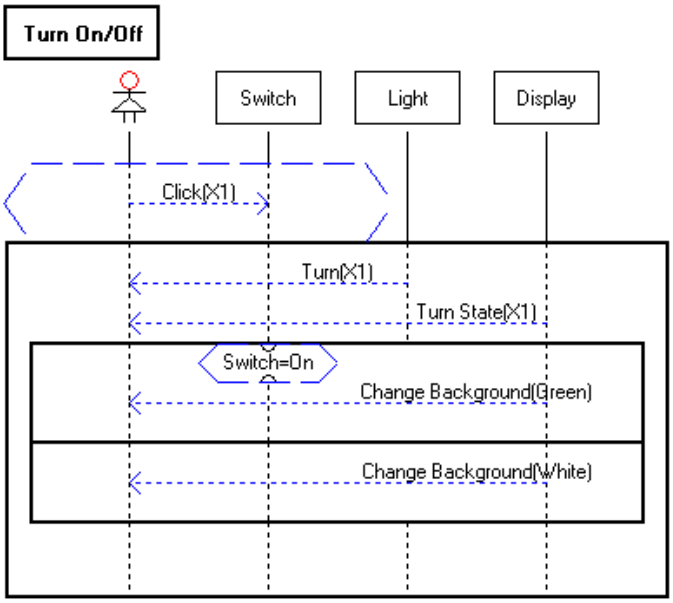

Figure 6: Turning the calculator on or off

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. Fig. 7 shows that the '+' function has two parameters, and that the one currently pointed at is of type "Number". Fig. 8 shows the final LSC for summation.

6.2 Play-Out

After (any part of) the system behavior has been specified, the user can test and debug it using play-out. As mentioned, only a very short illustration of a play-out session will be shown here. There are many more aspects of play-out that warrant reporting on, and these will appear in a separate paper.

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. 9. The LSCs "Show Number" and "Plus - New" in Fig. 9 take care of showing the clicked digits on the display. When the '+' button is clicked, the slider changes its value to 10. When a digit is clicked, if the slider's value is 10, the digit is displayed as is and the slider's is zeroed. If the slider's value is 0, the clicked digit is concatenated to the end of the currently displayed number.[8] The concatenation is provided by the implemented function *ComputeDisplay*. Chart "Sample Sum" is an existential LSC, denoted by a dashed borderline. It shows a sample scenario

[8] As mentioned in an earlier footnote, the slider does not play any interesting role in calculating, and is added just for illustration.

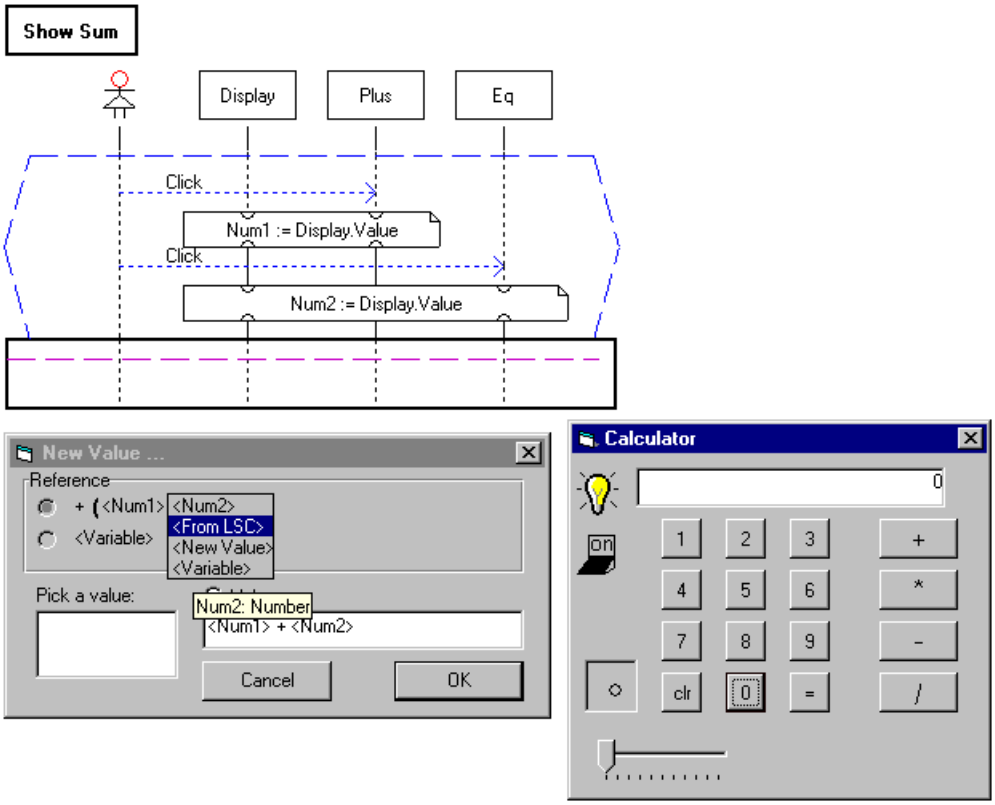

Figure 7: Prechart of the sum operation

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 $Num1 + Num2$.

When playing out a scenario, the user can choose which universal charts are to participate and which (existential) charts are to be monitored. We have decided that all the aforementioned universal charts will participate and that the existential chart "Sample 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. Fig. 10 shows the situation after the user has turned the calculator on and has clicked in the sequence $3, 4, 5, +, 1, 2$.

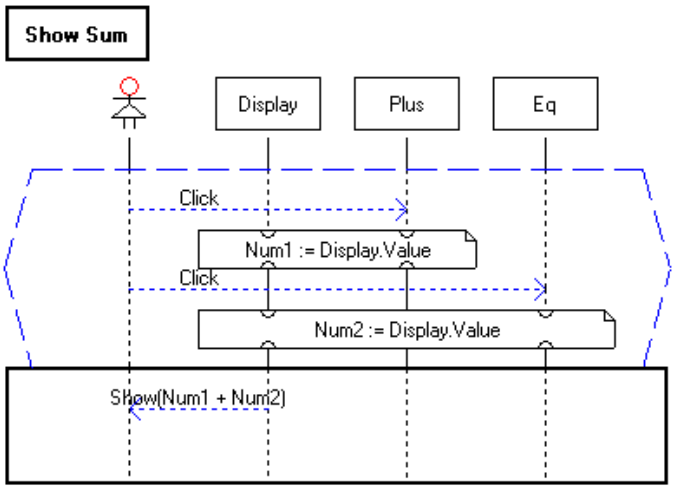

Figure 8: LSC for the sum operation

The top chart on the left, "Show Number", was activated by the click on 2. Since the slider's value was 0, not 10, 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[9], 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 $Num1$. On the right is the existential chart, "Sample 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.

7 Integration with Other Tools

We have set up the play-engine to store played-in specifications in XML format [32].[10] 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

[9]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.

[10]For a more detailed discussion of the advantages of XML as an interchange format, we refer the reader to [29].

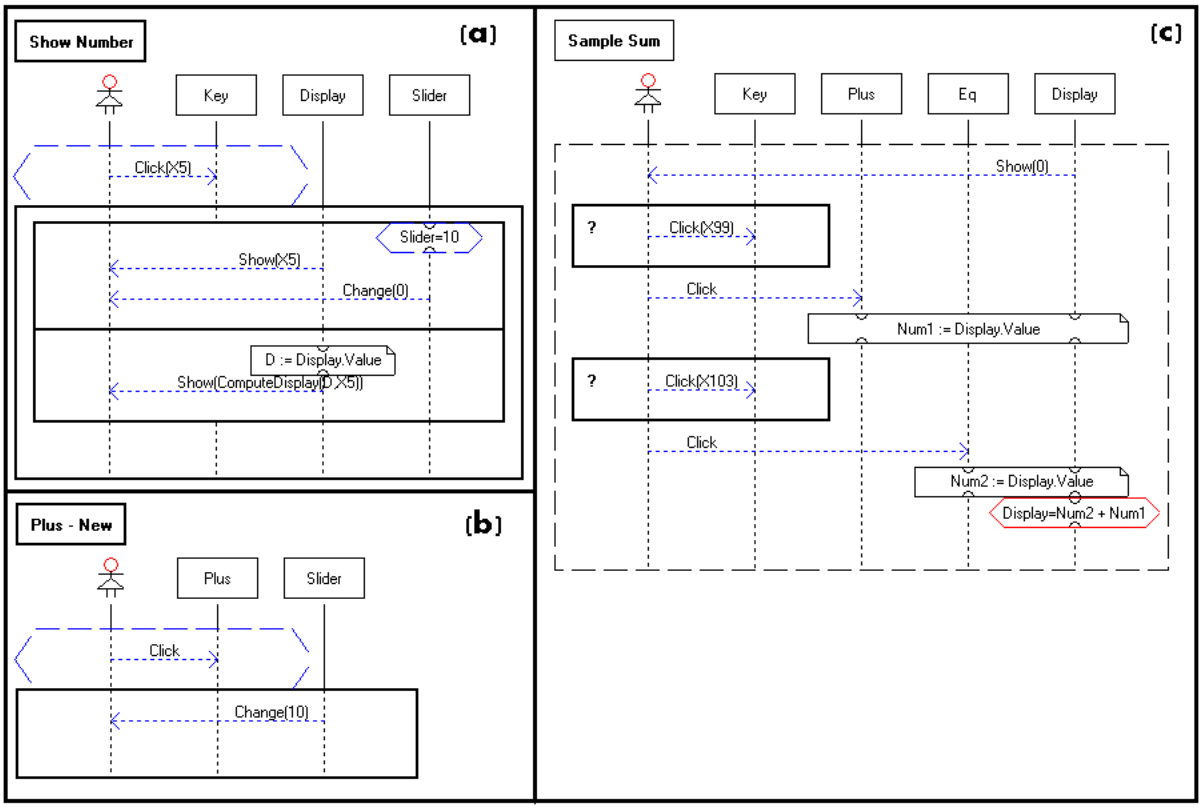

Figure 9: Additional charts used in the play-out

(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 another source can be very useful. For example, in [12] 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 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). The implementation can be set up to record the runs it produces, and these can then be re-played by the engine, so the user can see if they comply with the original requirements.

Besides these, 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 [17]. The TL version of the specification can then be read in by model checkers and other verification tools,

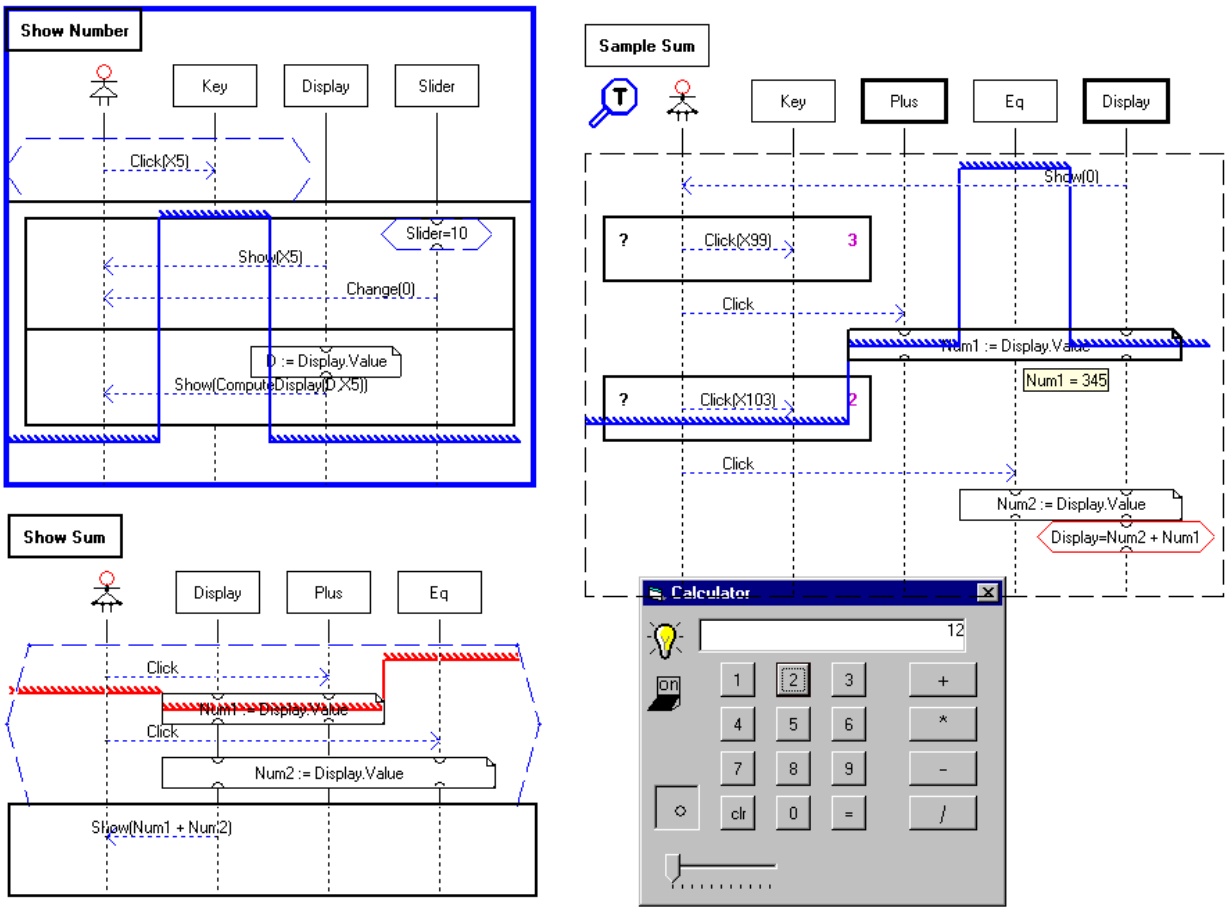

Figure 10: Playing out a scenario

as discussed in the next section.

8 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 [2] 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 played-in behavior, 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 [2] does not refer to some of the additional issues crucial to sequence-based languages, that were raised in [5], such as the ability to specify

anti-scenarios, and to distinguish between"must" and "may" behaviors, etc.

There are a number of commercial tools that handle the execution of graphical models quite successfully (e.g., Statemate [13] and Rhapsody by I-Logix [15], ObjectTime [27], and Rose-RT by Rational [24]). 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, by and large, such tools execute an intra-object design model (statecharts). 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, they 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.

Lettrai and Klose [18] present a methodology supported by a tool integrated into Rhapsody [15], for monitoring and testing a model using sequence charts. The language used in [18] is a modest subset of LSCs. Sequence charts can be monitored in a way that appears to be similar to ours. The main and most significant difference is that in order to be executable their sequence charts are transformed into Büchi automata (which is more like synthesizing statecharts or automata from LSCs, as is done in [12]), which may become very large. This should be contrasted with the fact, mentioned earlier, that our play-out process uses no accumulated intermediate representation, but works directly on the active charts.

There is also work concerned with the execution of formal specifications in non-graphical languages. For example, [28] and [21] present an execution and animation framework for specifications in Z, whereas [14] 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. [20, 30] 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 [23], 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 sequence charts, and it will usually be larger and more detailed than sequence charts. The behavior is written in FSP [19] and is compiled into LTSs, a process which appears to be less intuitive then 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 [6] presents a methodology called *genetic software engineering* (GSE), in which a requirement written in natural language is formalized by a "behavior tree". These 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. However, GSE does not include any play-in mechanism or model execution capabilities on the requirements level.

Boger et. al. [3] present a development methodology, called *extreme modeling* (XM), which tries to combine the advantages of the programming methodology of *extreme programming* [33] with the UML [31]. 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.

9 Conclusions and Future Work

In summary, we have substantiated the idea of specifying system behavior by playing in scenarios directly from friendly GUI applications [9], and in so doing have also worked out a method to play the behavior out directly. Play-in makes capturing requirements quite intuitive (someone commented to us that it was "cool" ...), thus enabling non-professional end-users to participate in the process. 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 makes the link between informal use cases and detailed requirements (e.g., in LSCs or temporal logic) 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 too. We are also coming to believe that for certain 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:

Handling Internal Objects:

We are in the process of incorporating internal objects into our work. This entails dealing with hierarchical behavior, by playing in and out the interaction between internal (non-GUI) objects. We expand the methodology to handle such interactions, which consist mainly of sending messages to other objects and calling their functions and methods. The interaction between internal objects is specified using an object map, which is a layout of the relevant objects (either defined abstractly, or actually drawn by the user). The user can specify actions by dragging elements from the sender to the receiver. These interactions may (and typically will) be interlaced with GUI interactions within a single LSC. Since at the start the user does not always know exactly which objects will constitute the system, the play-engine makes it possible to declare new objects and new operations on the fly, and carry out top-down or bottom up specification.

Using Consistency Algorithms:

The play-out algorithms we have developed will not always provide a complete picture of what the behavioral possibilities are. Although we have not yet encountered such cases, there could be ones where the play-engine chooses the system reactions in an order that is erroneous and leads to chart violation. What we mean by erroneous is that there is some "correct" order (or several) for these system events that would have managed to run to completion successfully, but the engine does not find it and reports chart violation. (This, depending on the hot or cold nature of active elements, could lead to abortion of the entire run.) With Hillel Kugler, we are now working on developing a consistency algorithm, which will hopefully eliminate this problem, by augmenting the play-out mechanism with the ability to find a "good" order of events whenever there is one. It will also be able to find a sequence of events that will lead to the successful completion of a monitored existential chart without violating the universal ones. Of course, an appropriate consistency algorithm could also be applied during play-in, to detect inconsistencies in the requirements specification. As explained earlier, the results of such checks can be given as counter example runs, and used to track the reasons for the inconsistencies.

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 would be to move smoothly into the next phase — preparing an intra-object model behavior — that would lead to implementation (see [9]). 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 [12], 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 are in the initial phase of developing an environment for building GUI applications that are "aware" of the play-engine's environment. This will most likely consist of an editor and a set of components that can be used to create GUI applications. The components will 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]). We intend to see to it that such tools can be used in concert 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 several kinds of systems for which the LSC specification, together with the play-engine, may be considered to be not just requirements but the final implementation. For example, a simple desktop utility such as a phone book could be created (given a pre-determined 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. 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.

Acknowledgements: We would like to thank Hillel Kugler for many inspiring discussions on the material reported upon here, and the referees of a previous version of the paper for their helpful comments.

References

[1] Altia Design & Altia FacePlate, http://www.altia.com.

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

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

[4] http://www.microsoft.com/com.

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

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

[7] M. Fränzel and K. Lüth. Visual Temporal Logic as a Rapid Prototyping Tool. *Vis. Lang. and Compu.*, to appear 2001.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Prog.*, pages 231–274, 1987. (Preliminary version: Tech. Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.).

[9] D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, pages 53–60, January 2001.

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

[11] D. Harel and Y. Koren. Drawing Graphs with Non-Uniform Vertices. Tech. Report MCS00-09, The Weizmann Institute of Science, 2000.

[12] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. of Foundations of Computer Science (IJFCS), in press.*, 2001. (Also,*Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, July 2000, Lecture Notes in Computer Science, Springer-Verlag, 2000, to appear.).

[13] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.

[14] P. Heymans and E. Dubois. Scenario-Based Techniques for Supporting the Elaboration and the Validation of Formal Requirements. *Requirements Engineering Journal*, 3:202–218, 1998. Springer-Verlag.

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

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

[17] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Live Sequence Charts. Manuscript, Weizmann Institute, 2000.

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

[19] J. Magee and J. Kramer. *Concurrency - State Models & Java Programs.* Chichester: John Wiley & Sons, 1999.

[20] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. *22nd Int. Conf. on Soft. Eng. (ICSE'00), Limeric, Ireland*, 2000.

[21] M. Özcan, P. Parry, I. Morrey, and J. Siddiqi. Visualization of Executable Formal Specifications for User Validation. *Ann. Soft. Eng.*, 3:131–155, 1997.

[22] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:1–20, 1981.

[23] N. Pryce and J. Magee. SceneBeans: A Component-Based Animation Framework for Java. http://www-dse.doc.ic.ac.uk/Software/SceneBeans/.

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

[25] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[26] R. Schlor and W. Damm. Specification and verification of system-level hardware designs using timing diagram. In *European Conference on Design Automation*, pages 518–524, Paris, France, 1993. IEEE Computer Society Press.

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

[28] J. I. Siddiqi, I. C. Morrey, C. R. Roast, and M. B. Ozcan. Towards Quality Requirements via Animated Formal Specifications. *Ann. Soft. Eng.*, 3:131–155, 1997.

[29] J. Suzuki and Y. Yamamoto. Extending UML for Modelling Reflective Software Components. In R. France and B. Rumpe, eds., *2nd Int. Conf. on the Unified Modeling Language* (UML'99), Lecture Notes in Computer Science, vol. 1723, Springer Verlag, pp. 220–235, 1999.

[30] S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in MSCs using LTSA. Technical Report 2001/4, Department of Computing, Imperial College, London, 2001.

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

[32] http://www.xml.com.

[33] http://www.extremeprogramming.org.

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