Smart Play-Out of Behavioral Requirements ⋆

David Harel, Hillel Kugler, Rami Marelly and Amir Pnueli

The Weizmann Institute of Science, Rehovot, Israel
{harel,kugler,rami,amir}@wisdom.weizmann.ac.il

Abstract. We describe a methodology for executing scenario-based requirements of reactive systems, focusing on "playing-out" the behavior using formal verification techniques for driving the execution. The methodology is implemented in full in our *play-engine* tool[1]. The approach appears to be useful in many stages in the development of reactive systems, and might also pave the way to systems that are constructed directly from their requirements, without the need for intra-object or intra-component modeling or coding.

1 Introduction

In the last few years, formal specification and verification techniques are beginning to be applied to the development of complex reactive systems. Major obstacles that still prevent even wider usage of such methods include the fact that errors are found relatively late in the development process and that high expertise is required to correctly capture the properties to be verified. Recently there has been a growing interest in the verification of software based reactive systems, especially given the success in applying verification techniques to hardware. Due to the size and complexity of such systems, it is desirable to understand all the system requirements, and to make sure they are consistent, before moving to the implementation phase. In classic verification, a model is first constructed and then verified against well defined requirements, whereas one of the main points of this paper is that verification techniques can be beneficially applied to the requirements too.

In this paper we suggest a methodology that addresses these obstacles. As our requirements language we use the *live sequence charts (LSCs)* of [7], a visual formalism based on specifying the various kinds of scenarios of the system — including those that are mandatory, those that are allowed but not mandatory, and those that are forbidden. LSCs thus extend classical message sequence charts, which do not make such distinctions. The Unified Modeling Language (UML) [33], which is the leading standard for specifying object oriented software systems, uses a variant of classical message sequence charts (MSCs) [21], called sequence diagrams, which can be viewed as a simple existential variant of LSCs.

A new approach for capturing behavioral requirements (proposed briefly in [12]) has been developed recently, and is described in detail in [14]. In it the user *plays in* the behavior using a graphical interface (GUI) of the target system or an abstract version thereof. The formal requirements in the language of LSCs are then automatically generated from the play-in by a

⋆ This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems.

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

tool called the *play-engine*, without a need to explicitly prepare the LSCs or to write complex formulas in, e.g., temporal logic.

Complementary to the play-in process is *play-out* [14]. In the play-out phase the user plays the GUI application as he/she would have done when executing a system model (or, for that matter, the final system) but limiting him/herself to "end-user" and external environment actions only. While doing so, the play-engine keeps track of the actions taken, and causes other actions and events to occur as dictated by the universal charts in the specification (these are charts describing mandatory behavior), thus giving the effect of working with a fully operational system or an executable model. It is noteworthy that no code needs to be written in order to play-out the requirements, nor does one have to prepare a conventional intra-object system model, as is required in most system development methodologies (e.g., using statecharts or some other language for describing the full behavior of each object, as in the UML, for example). We should also emphasize that the behavior played out is up to the user, and need not reflect the behavior as it was played in; the user is not merely tracing scenarios, but is executing the requirements freely, as he/she sees fit.

This idea appears to have potential in many stages of system development [14]. In particular, the ability to execute such inter-object requirements without building a system model or writing code could lead to a totally new way of building many kinds of reactive systems. The play-engine would become a sort of "universal reactive machine", which would run requirements that were played in via a GUI, or written directly as LSCs, timing diagrams or formulas in an appropriate temporal logic. You provide the global, declarative, inter-object ways you want your system to behave (or to not behave), and the engine runs the system directly from them. It works a little like a perfect citizen, who does absolutely nothing unless it is called for by the grand "book of rules", and unless it doesn't contradict anything else written in the book. Thus, the engine does only those things it is required to do, while avoiding those it is forbidden to do. This is a minimalistic, but completely safe way for a system to behave exactly according to the requirements, and to make sure that the system doesn't just sit around doing nothing, it is up to the requirement engineers to make sure that any liveness properties they want the system to satisfy should be incorporated into the requirements.

Play-out is actually an iterative process, where after each step taken by the user, the play-engine computes a *superstep*, which is a sequence of events carried out by the system as response to the event input by the user. However, the original play-out process of [14] is rather naive, for several reasons. 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" superstep. We consider a superstep to be correct if when it is executed no active universal chart is violated. By acting blindly by the "book" of requirements, reacting to a user-generated event with the first action it encounters as a possible reaction to that event, the naive play-out process could very well follow a sequence of events that eventually causes violation, although another sequence could have been chosen that would have completed successfully. 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, enables formulating high level requirements 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 having to say explicitly how they should be composed are very useful in early requirement stages, but can cause under-specification and nondeterminism when one attempts to execute them.

The work we describe here, which we term *smart play-out*, focuses on executing the behavioral requirements with the aid of formal analysis methods, mainly model-checking. Our smart play-out process uses model-checking to find a "correct" superstep if one exists, or proves that such a superstep does not exist. Model-checking is applied anew at the occurrence of each user event to examine the different potential supersteps and to find a correct sequence of system reactions if there is one. Model-checking thus drives the execution. Another way of putting it is that the "smartness" in smart play-out works as an aid, helping the objects in the system cooperate in fulfilling the requirements. Experimental results we have obtained using a prototype implementation of smart play-out are very promising.

Smart play-out illustrates the power of putting formal verification methods to use in early stages of the development process, with the potential of impacting the development of reactive systems. We believe that additional verification tools and technologies can be used to improve the ability of the play-out framework to handle large systems efficiently. And, as mentioned above, we also believe that for certain kinds of systems the play-out methodology, enhanced by formal verification techniques, could serve as the final implementation too, with the play-out being all that is needed for running the system itself.

The paper is organized as follows. Section 2 gives a brief overview of the LSC language using a cellular phone system which serves as a running example throughout the paper. Section 3 discusses the Play-in/Play-out approach focusing on play-out and explaining the need for "Smart Play-Out". Section 4 shows examples from the cellular phone system illustrating where smart play-out is helpful. Section 5 gives a high level description of the smart play-out approach and how model-checking is used to achieve it, while section 6 provides a formal description of the translation that produces the input to the model-checker. Section 7 describes experimental results obtained on the cellular phone system using our prototype tool implementation of smart play-out. We conclude with a discussion of related work in Section 8.

2 LSCs

Live sequence charts (LSCs) [7] 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 typically contains 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 specify sample interactions between the system and its environment, and 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 the cellular phone system to illustrate the main 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 *X2*), 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 #X2.

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 were proposed in [14] as an extension to LSCs. Using an assignment, the user may save values of the

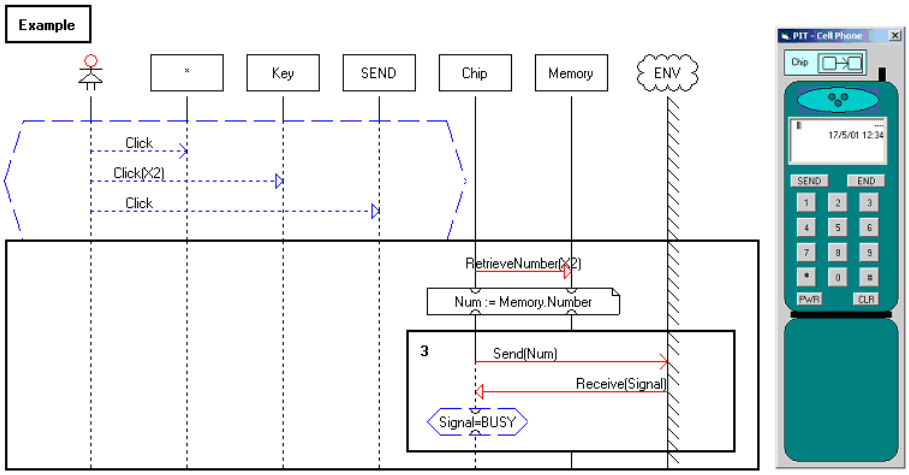

Fig. 1. LSC Sample - Quick Dialing

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. 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.

After the assignment comes a *loop* construct. This 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 be exited when a *cold condition* inside it is violated, as described shortly[2]. 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. 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.

Hot conditions can be used for many other things too. For example, a forbidden scenario can be specified by putting it into a prechart with the main chart being a hot false condition.

In general, we consider *open* reactive systems, and thus distinguish between the system and its external environment. As can be seen in Fig. 1 the system's environment is also composed of a user operating the system (denoted by the like of a person) and an abstract entity representing all other elements interacting with the system. The user interacts with the system directly by operating its user interface, while the environment interacts with the system in other ways (e.g., communicating over channels, controlling environmental settings etc.).

[2] [14] defines also *unbounded* loops and *dynamic* loops, which we will not describe here

The advantage in using LSC's is that it is an extension of sequence chart formalisms that are widely accepted and used by engineers, but is far more expressive than MSCs or UML sequence diagram. LSC's can be viewed as a visual front-end to a somewhat restricted version of temporal logic, with mechanisms enabling convenient usage of the language. The semantics of a restricted subset of LSC's in terms of temporal logic are given in [13], and a more complete treatment is in preparation. For a discussion on the advantages of LSCs as a requirements specification language see, e.g., [7, 14].

3 The Play-in/Play-out Approach

The play-in/play-out approach is described in detail in [14]. Recognizing that [14] has not yet been published, we give a brief overview here, sufficient for the purposes of the present paper. As its name states, the approach consists of two complementary aspects. Play-in is a method for capturing behavioral requirements (e.g., following the preparation of use cases) in an intuitive way, using a graphical user interface of the target system or an abstract version thereof. The output of this process is a formal specification in the language of LSCs [7]. Play-out is the process of testing the requirements by executing them directly. The input to the play-out process is a formal LSC specification. Although, it is much more effective to play out requirements that were played in, this is not obligatory, and the LSC specification can be produced in any desired way.

It is worth noting that the behavior described in Fig. 1 was played in using a GUI of a cellular phone and did not require any drawing or editing of elements in the generated chart.

Play-out is the process of testing the behavior of the system by providing user and environment 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. 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. 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.

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. 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.

A number of things happen during play-out. Charts are opened whenever they are activated and are closed when they are violated or when they terminate. Each displayed chart shows a "cut" (a kind of rectilinear "slice"), denoting the current location of each instance. The currently executed event is highlighted in the relevant LSCs. The play-engine interacts

with the GUI application, causing it to reflect the change in the GUI, as prescribed by the executed event. 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*, meaning that 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. We thus run the system in such a way as to seek satisfaction of existential promises while making sure we satisfy all universal promises.

The premise of our present work is that the play-out algorithms described in [14] are somewhat naive. For example, if there are several ways to linearize the partial order of events in an LSC, the engine might choose one that leads to a contradiction with another LSC. This, depending on the hot or cold nature of active elements, could lead to abortion of the entire run. While such an occurrence is indeed a result of what the user played in, and is a legal execution, we might want the engine to help avoid it. If in this example there is some "correct" order (or several) that manages to run to completion successfully, we would like to find it and guide the play-out accordingly.

4 Being Smart Helps : Examples

Consider the two charts LSC1 and LSC2 appearing in Fig. 2 and the following system reaction performed in response to the user clicking on the 'PWR' button:

$$\mathbf{ChangeBackground}(\mathbf{Green}), \mathbf{ChangeBackground}(\mathbf{Red}), \mathbf{Open}$$

This superstep satisfies LSC1 but LSC2 remains active with the condition **DisplayBack-**

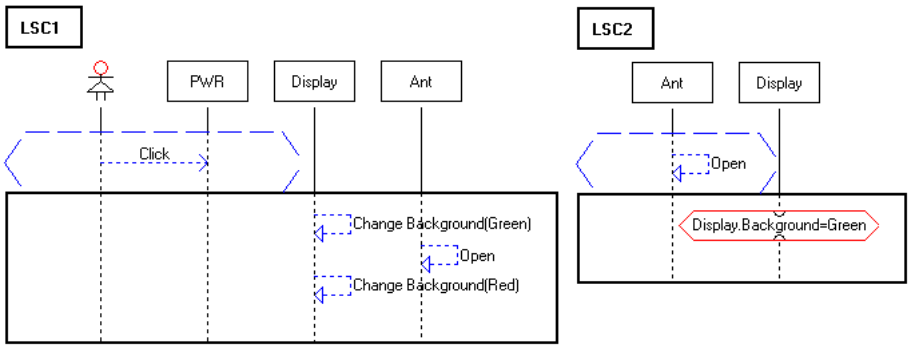

Fig. 2. Smart play-out helps

ground = Green false, because when it was activated by the **Open** event the background was already red. Notice that "locally" each event seems to be good, since it does not cause

violation and causes the execution to progress. However, “globally” these system moves do not satisfy the second LSC.

In contrast, the following system reaction satisfies both LSCs:

$$\mathbf{ChangeBackground}(\mathbf{Green}), \mathbf{Open}, \mathbf{ChangeBackground}(\mathbf{Red})$$

After changing the color to **Green** the system opens the antenna, thus causing the activation of LSC2. The **Display** color is **Green**, so the condition holds and LSC2 is satisfied. Then the color is changed to **Red** and LSC1 is satisfied. Smart play-out is designed to find a correct superstep in such cases.

Similarly, consider the two charts `State First` and `Background First` in Fig. 3. When the user opens the cover both charts are activated. However, there is no way to satisfy them both since they require the events **ChangeBackground(Green)** and **SetState(Time)** to occur in contradicting order. While this is a very simple example, such contradictions can be a lot more subtle, arising as a result of the interaction between several charts. In large specifications this can be very hard to analyze manually. The smart play-out framework would prove that in such a case no correct superstep exists, which by the semantics of LSCs means that the requirements are inconsistent; see [13].

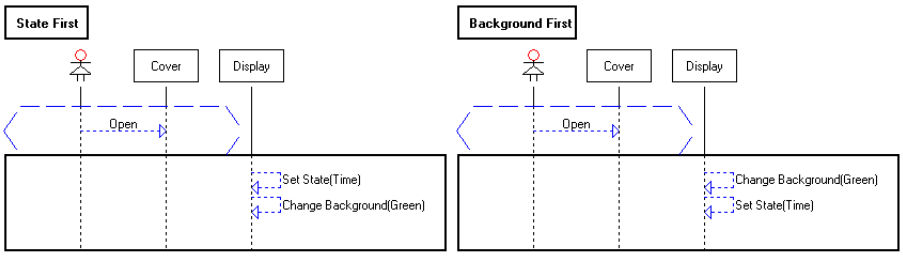

Fig. 3. Inconsistent LSCs

As discussed earlier, existential LSCs may be used to specify system tests. Smart play-out can then be used to find a trace that satisfies the chart without violating universal charts on the way. Fig. 4 shows a test in which user and external environment actions are performed and expected system responses are described using conditions. In this chart, the user opens the cover and enters the number 911. In response, the display is expected to show the dialed number. Next, the user clicks the ‘SEND’ button and the phone’s speaker is expected to ring. Finally, when a signal from the environment indicating the accepting of the call (denoted by the “ACK” reserved word) is received by the phone’s chip, the speaker turns silent.

5 Smart Play-Out: The General Approach

The approach we use is to formulate the play-out task as a verification problem, and to use a counterexample provided by model-checking as the desired superstep. The system on which we perform model-checking is constructed according to the universal charts in the specification. The transition relation is defined so that it allows progress of active universal charts but prevents any violations. The system is initialized to reflect the status of the application just after the last external event occurred, including the current values of object properties,

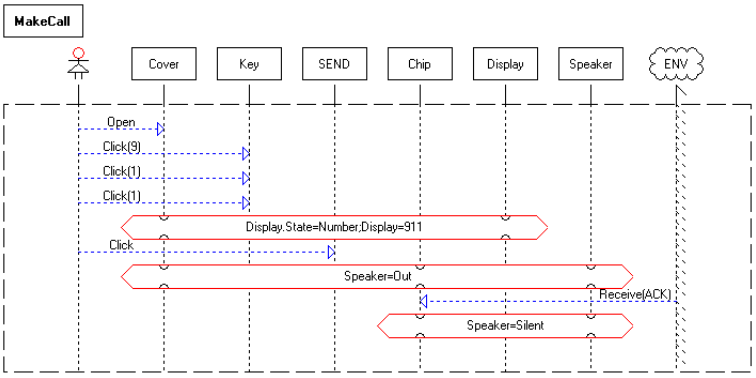

Fig. 4. Using existential charts to specify system tests

information on the universal charts that were activated as a result of the most recent external events, and the progress in all precharts.

The model-checker is then given a property claiming that always at least one of the universal charts is active. In order to falsify the property, 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 superstep. If the model-checker verifies the property then no correct superstep exists. The next section provides details of how to construct the input to the model checker.

It is important to note that smart play-out (at least as it stands today) does not backtrack over supersteps. Thus, we may get to a situation where no correct super-step exists due to moves the system made in previous super-steps, which could perhaps have been done differently. This demonstrates the difference between smart play-out, which looks one super-step ahead, and full synthesis, which performs a complete analysis.

Another important thing that we have incorporated into the smart play-out is to find a way to satisfy an entire existential chart (e.g. Fig. 4). Here we cannot limit ourselves to a single superstep, since the chart under scrutiny can contain external events, each of which triggers a superstep of the system. Nevertheless, the above 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.

6 The Translation

In the original paper defining LSCs [7] and in later work that uses LSCs for testing reactive systems [22], the semantics of LSCs is defined for a single chart. In the first one, a programmatic style is used and in the second, an automaton having legal cuts[3] as states is constructed.

[3] A cut is a configuration indicating the location of each object along its instance line

In our work, the main focus is to find a correct behavior of the system according to several charts acting together. To do that, we construct a transition system which has one process for each actual object. A state in this system indicates the currently active charts and the location of each object in these charts. The transition relation restricts the transitions of each process only to moves that are allowed by all currently active charts. Note that our translation does not explicitly construct the cuts for each chart (a construction which by itself causes an exponential growth in the size of the initial representation).

We now provide some of the details on how to translate a play-out problem into a model-checking problem.

An LSC specification LS consists of a set of charts M, where each chart $m \in M$ is existential or universal. We denote by $pch(m)$ the prechart of chart m. Assume the set of universal charts in M is $M^U = \{m_1, m_2, ..., m_t\}$, and the objects participating in the specification are $\mathcal{O} = \{O_1, ..., O_n\}$.

We define a system with the following variables:

act_{m_i} determines if universal chart m_i is active. It gets value 1 when m_i is active and 0 otherwise.

$msg^s_{O_j \to O_k}$ denoting the sending of message msg from object O_j to object O_k. The value is set to 1 at the occurrence of the send and is changed to 0 at the next state.

$msg^r_{O_j \to O_k}$ denoting the receipt by object O_k of message msg sent by object O_j. Similarly, the value is 1 at the occurrence of the receive and 0 otherwise.

l_{m_i,O_j} denoting the location of object O_j in chart m_i, ranging over $0 \cdots l^{max}$ where l^{max} is the last location of O_j in m_i.

$l_{pch(m_i),O_j}$ denoting the location of object O_j in the prechart of m_i, ranging over $0 \cdots l^{max}$ where l^{max} is the last location of O_j in $pch(m_i)$.

Throughout this paper, we use the asynchronous mode, in which a send and a receive are separate events, but we support the synchronous mode too. We denote by $f(l)$ the event associated with location l, and use the convention that primed variables denote the value of a variable in the next state while unprimed variables relate to the current state.

We will now show the definition of the transition relation as it is affected by the different features of the LSC language.

6.1 Messages

We first define the transition relation for the location variable when the location corresponds to the sending of a message:

$$l'_{m_i,O_j} = \begin{cases} l & \text{if } l_{m_i,O_j} = l-1 \wedge {msg^s_{O_j \to O_k}}' = 1 \\ l-1 & \text{if } l_{m_i,O_j} = l-1 \wedge {msg^s_{O_j \to O_k}}' = 0 \end{cases}$$

Intuitively, if object O_j is at location $l-1$ in chart m_i, and the next location of O_j corresponds to the sending of message msg from O_j to O_k, then if in the next state the message is sent, the location is advanced; otherwise it remains still. It is important to notice that the event $msg^s_{O_j \to O_k}$ may not be allowed to occur at the next state due to some other chart. This is one of the places were the interaction between the different charts becomes important.

As for the receipt of events, given that n is the location at which message msg is sent from object O_j to object O_k, we define the transition relation as:

$$l'_{m_i,O_k} = \begin{cases} l & \text{if } l_{m_i,O_k} = l-1 \wedge l_{m_i,O_j} \geq n \wedge {msg^r_{O_j \to O_k}}' = 1 \\ l-1 & \text{if } l_{m_i,O_k} = l-1 \wedge (l_{m_i,O_j} < n \vee {msg^r_{O_j \to O_k}}' = 0) \end{cases}$$

If object O_k is at location $l-1$ in chart m_i, and the next location of O_k corresponds to the receipt of the message msg sent by object O_j, and this message has already been sent , then if in the next state the message is received, the location is advanced; otherwise it remains as is.

We now define the transition relation for the variable determining the occurrence of a send event (the receive case is similar):

$${msg^s_{O_j \to O_k}}' = \begin{cases} 1 & \text{if } \phi_1 \wedge \phi_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_1 = \bigvee_{m_i \in M^U \wedge msg^s_{O_j \to O_k} \in Messages(m_i)} act_{m_i} = 1$$

$$\phi_2 = \bigwedge_{m_i \in M^U \wedge msg^s_{O_j \to O_k} \in Messages(m_i)} (act_{m_i} = 0 \vee \psi(m_i))$$

$$\psi(m_i) = \bigvee_{l_t \text{ s.t. } f(l_t) = msg^s_{O_j \to O_k}} (l_{m_i,O_j} = l_t - 1 \wedge l'_{m_i,O_j} = l_t)$$

In order for the event of sending msg from O_j to O_k to occur, we require two conditions to hold, which are expressed by formulas ϕ_1 and ϕ_2 respectively. The first, ϕ_1, states that at least one of the main charts in which this message appears is active. The assumption is that message communication is caused by universal charts that are active and does not occur spontaneously. The second requirement, ϕ_2, states that all active charts must "agree" on the message. For an active chart m_i in which $msg^s_{O_j \to O_k}$ appears we require that object O_j progress to a location l_t corresponding to this message, as expressed in formula $\psi(m_i)$. Formula ϕ_2 states that for all charts m_i in which $msg^s_{O_j \to O_k}$ appears (that is, $msg^s_{O_j \to O_k} \in Messages(m_i)$) either the chart is not active or the message can occur (that is, $\psi(m_i)$ holds). According to the semantics of LSCs, if a message does not appear in a chart explicitly it is allowed to occur in-between the messages that do appear, without violating the chart. This is reflected in ϕ_2 by the fact that the conjunction is only over the charts in which $msg^s_{O_j \to O_k}$ appears.

6.2 Precharts

A prechart of a universal chart describes a scenario which, if completed successfully, forces the scenario described in the main chart to occur. (Fig. 1 has a prechart — the portion enclosed in the dashed hexagon.) The main chart becomes active if all locations of the prechart have reached maximal positions. In play-out it is often the case that a sequence of events in a superstep causes the activation of some additional universal chart, and this chart must now also be completed successfully as part of the super-step. For this purpose precharts are monitored, and locations along instance lines are advanced while messages are being sent and received.

The transition relation for a location variable in a prechart is similar to the one defined for locations in the main chart, with one major difference; precharts may be violated. If a message is sent or received while it is not enabled in the prechart, the prechart is "reset" by moving all its instances back to their initial location. This reset action allows for the prechart to start "looking" for another option to be satisfied. In fact, in many cases when the model-checker searches for a "correct" super-step it tries to violate precharts in order not to get into the "obligations" of having to satisfy the corresponding main charts. When all locations in the prechart reach their maximal positions, they too are reset.[4]

Formally, if location $l_{pch(m_i),O_j} = l - 1$, and the next location corresponds to a message sending, then its transition relation is given by:

$$l'_{pch(m_i),O_j} = \begin{cases} l & \text{if } {msg^s_{O_j \to O_k}}' = 1 \\ 0 & {msg^s_{O_j \to O_k}}' = 0 \wedge \Phi(m_i) \\ l-1 & \text{otherwise} \end{cases}$$

$$\Phi(m_i) = \bigvee_{msg^s_{O_x \to O_y} \in Messages(m_i)} \Psi^s(msg^s_{O_x \to O_y}) \vee \bigvee_{msg^r_{O_x \to O_y} \in Messages(m_i)} \Psi^r(msg^r_{O_x \to O_y}) \vee \bigwedge_{O_j \in Obj(m_i)} (l_{pch(m_i),O_j} = l^{max}_{pch(m_i),O_j})$$

$$\Psi^s(msg^s_{O_x \to O_y}) = \begin{cases} 1 & \text{if } l_{m_i,O_x} = l_x - 1 \wedge f(l_x) \neq msg^s_{O_x \to O_y} \wedge {msg^s_{O_x \to O_y}}' = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\Psi^r(msg^r_{O_x \to O_y}) = \begin{cases} 1 & \text{if } l_{m_i,O_y} = l_y - 1 \wedge f(l_y) \neq msg^r_{O_x \to O_y} \wedge {msg^r_{O_x \to O_y}}' = 1 \\ 0 & \text{otherwise} \end{cases}$$

Ψ^s/Ψ^r checks whether a send/receive event occurred while not enabled by its sender/receiver instance in the chart. $\phi(m_i)$ checks whether all locations reached their maximal position.

6.3 Activation of charts

For a universal chart m_i, we define the transition relation for act_{m_i} as follows:

$$act'_{m_i} = \begin{cases} 1 & \text{if } \phi(pch(m_i)) \\ 0 & \text{if } \phi(m_i) \\ act_{m_i} & \text{otherwise} \end{cases}$$

$$\phi(m_i) = \bigwedge_{O_j \in Obj(m_i)} (l'_{m_i,O_j} = l^{max}_{m_i,O_j})$$

The main chart m_i becomes active when all locations of the prechart reach maximal positions, and it stops being active when all locations of the main chart reach maximal positions.[5]

[4] Our current treatment of precharts is still rather preliminary, and there are several issues we plan to consider more fully in the future. They include figuring out whether or not (or when) to use model checking to "help" precharts be successfully completed, and how to deal with loops and conditions in precharts in light of the main goals of smart play-out.

[5] When the chart body contains interactions with the user/environment, we cannot guarantee that all maximal positions are reached, because the play-out cannot initiate moves by the environment. We

In order to identify the activation of a universal chart it is sometimes necessary to maintain several copies of the same prechart, each one being in a different stage of the prechart scenario. A universal chart may also be reactivated before it has completed, causing several copies of the main chart to be active simultaneously. It can be shown that in the absence of unbounded loops, the maximal number of simultaneously active charts and precharts is bounded and can be computed. Actually, we predict that in most practical cases these bounds will be small.[6]

6.4 Object properties and conditions

Although the basic strength of scenario-based languages like LSCs is in showing message communication, the LSC language has the ability to reason about the properties of objects too. Object's properties can be referenced in condition constructs, which can be hot or cold. According to the semantics of LSCs, 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. To support this kind of reasoning, we have to update the value of each property as the system runs.

More formally, let $P^t_{O_k}$ denote the t^{th} property of object O_k, defined over a finite domain D. For many of the object properties there are simple rules — defined when the application is being constructed — that relate the value of the property to message communication. Accordingly, suppose that message msg received by O_k from O_j has the effect of changing property P^t of O_k to the value $d \in D$. We then add to the transition relation of process O_j the clause:

$$P^t_{O_k}{}' = d \text{ if } msg^r_{O_j \to O_k}{}' = 1$$

In this way, the values of the properties are updated as the objects send and receive messages.

Object properties can be referred to in conditions. In fact, we take a condition expression to be a Boolean function over the domains of the object properties, $C : D^1 \times D^2 \cdots \times D^r \to \{0, 1\}$, so that a condition can relate to the properties of several objects. Here, the properties appearing in the condition are $P_1, P_2, \cdots P_r$.

A condition affects the transition relation of the location of a participating object. If object O_j is at location $l_j - 1$ and object O_k is at location $l_k - 1$ in chart m_i, and if their next locations correspond to a *hot* condition C, we define:

$$l'_{m_i,O_j} = \begin{cases} l_j & \text{if } C(d_j, d_k)' = 1 \wedge l_{m_i,O_j} = l_j - 1 \wedge l_{m_i,O_k} = l_k - 1 \\ l_j - 1 & \text{if } l_{m_i,O_j} = l_j - 1 \wedge ((C(d_j, d_k)' = 0 \vee l_{m_i,O_k} \neq l_k - 1) \end{cases}$$

Object O_j moves to location l_j if both objects participating in the condition are ready to evaluate the condition expression, being at locations $l_j - 1$ and $l_k - 1$, respectively, and the condition C holds. Here d_j and d_k are the values of properties $P^s_{O_j}$ and $P^t_{O_k}$, respectively. The

therefore modify the transition relation to set a chart to be inactive when only user/environment events are enabled.

[6] This is because in order for the bound to be large there must be a very strong correlation between the messages in the prechart and the main chart, and this is usually not the case.

transition relation thus ensures synchronization of the objects when evaluating the condition and allows progress only if the condition expression holds, thus preventing violation of the chart. In this definition, we assumed that we have two objects, O_j and O_k, constrained by the condition, whereas in the general case there could be a single object or several objects.

For a cold condition we define:

$$l'_{m_i,O_j} = \begin{cases} l_j & \text{if } C(d_j,d_k)' = 1 \wedge l_{m_i,O_j} = l_j - 1 \wedge l_{m_i,O_k} = l_k - 1 \\ l_s & \text{if } C(d_j,d_k)' = 0 \wedge l_{m_i,O_j} = l_j - 1 \wedge l_{m_i,O_k} = l_k - 1 \\ l_j - 1 & \text{if } l_{m_i,O_j} = l_j - 1 \wedge ((C(d_j,d_k)' = 0 \vee l_{m_i,O_k} \neq l_k - 1) \end{cases}$$

The difference between this and the definition for a hot condition is that if the objects are ready for evaluating the condition but the condition does not hold, the smallest surrounding (sub)chart is exited, as per the semantics of LSCs. Here, l_s is the location of object O_j at the end of the surrounding (sub)chart. In such a case, all the other objects will also synchronize their exit of this (sub)chart. Note that this is a "peaceful exit", and does not constitute a violation of the universal chart m_i.

6.5 Assignments

Assignments enable referring to system properties after they are set. An assignment of the form $x := d$ stores the value d in the variable x. In practice, d may be a constant value, a property value of some object or the value obtained by applying some function. To handle assignments we add a boolean variable $assign(x,d)$ that is set to 1 exactly when the assignment is performed. Actually, these variables are used only for notational clarity, since in the implementation they can be computed from the values of the location variables. The translation is straightforward:

$$x' = \begin{cases} d & \text{if } l_{m_i,O_k} = l - 1 \wedge l_{m_i,O_k} = l \wedge assign(x,d) \\ x & \text{otherwise} \end{cases}$$

Intuitively, if object O_k is at location $l-1$ in chart m_i, and the next location of O_k corresponds to the assignment $x := d$ the value of x is set to d.

We also add to the system a boolean variable x_{bound}, which determines whether variable x is already bound to a concrete value. After an assignment is evaluated x_{bound} is set to 1. More information about this appears in the next subsection.

Assignments are local to a chart. Typically the variable x in the left hand side of the assignment is used later in a condition or symbolic message.

6.6 Symbolic messages

Symbolic messages are of the form $msg(x)$, where x is a parameter ranging over the finite domain D . A symbolic message represents concrete messages of the form $msg(d)$, where $d \in D$. Using symbolic messages it is possible to describe generic scenarios, which are typically instantiated and bound to concrete values during play-out.

To handle symbolic messages we add a variable representing the parameter x, which can be bound to a concrete value as the result of the occurrence of a concrete message or an assignment. The binding of this variable also affects other messages in the same chart that are parameterized by x, binding them to the same value. Once the variables of a symbolic

message are bound to concrete values, the usual rules concerning message communication apply to it, so it affects the transition relation similarly to a regular message.

Formally, for a symbolic message of the form $msg(x)$ we add a variable $x \in D$ and a boolean variable x_{bound}, which determines whether variable x is already bound to a concrete value.

Initially we set x_{bound} to 0 and define the transition relation as follows:

$$x'_{bound} = \begin{cases} 1 & \text{if } \phi_1 \vee \phi_2 \vee x_{bound} = 1 \\ 0 & \text{otherwise} \end{cases}$$

$$\phi_1 = l_{m_i,O_j} = l - 1 \wedge l'_{m_i,O_j} = l \wedge \bigvee_{d \in D} msg(d)' = 1$$

$$\phi_2 = \bigvee_{l_t \text{ s.t. } f(l_t) = assign(x)} (l_{m_i,O_k} = l_t - 1 \wedge l'_{m_i,O_k} = l_t)$$

According to the definition x_{bound} is changed to 1 in the case of the occurrence of concrete message $msg(d)$ where $d \in D$ (As defined by ϕ_1) or when x appears in the left hand side of an assignment that is being evaluated (As defined by ϕ_2).

The transition relation for the variable x is defined:

$$x' = \begin{cases} d & \text{if } l_{m_i,O_j} = l - 1 \wedge l'_{m_i,O_j} = l \wedge (msg(d)' = 1 \vee assign(x,d)' = 1) \\ x & \text{otherwise} \end{cases}$$

The first case corresponds to binding of x to value d as the result of the occurrence of concrete message $msg(d)$ or as the result of x being assigned the value d. Otherwise x remains unchanged.

We now define the transition relation for the location variable when the location corresponds to a symbolic message:

$$l'_{m_i,O_j} = \begin{cases} l & \text{if } l_{m_i,O_j} = l - 1 \wedge \bigvee_{d \in D} (msg(d)' = 1 \wedge x'_{bound} = 1 \wedge x' = d) \\ l - 1 & \text{if } l_{m_i,O_j} = l - 1 \wedge \bigwedge_{d \in D} (msg(d)' = 0 \vee x'_{bound} = 0 \vee x' \neq d) \end{cases}$$

Intuitively, if object O_j is at location $l - 1$ in chart m_i, and the next location of O_j corresponds to a symbolic message, then the location is advanced if the message $msg(d)$ occurs and x is bound to the value $d \in D$.

6.7 If-Then-Else

The transition relation of this construct is a variation on the way conditions are handled in subsection 6.4. All participating objects are synchronized when the condition is evaluated and when entering and exiting the Then and Else parts. We omit the details.

6.8 Loops

A loop is a sub-chart whose behavior is iterated, and all objects are synchronized at the beginning and end of each iteration. Loops can be of two basic types, bounded or unbounded [7, 14]. The transition relation synchronizes the objects at the beginning and end of each iteration, and for the bounded case a counter variable is added to ensure that the given bound is not exceeded. We omit the details.

6.9 Functions

As explained in the subsection dealing with object properties, message communication can have an effect on the values of object properties. In cases where there is a simple rule relating the value of a property to message communication, this can be fully handled in the transition relation. In cases where more complex functions are used, the situation is more complicated. We used a practical approach, creating a symbolic trace of events that is bound to actual values at a later stage, iteratively. Here too, we omit the details.

6.10 The Model-Checking

To compute a super-step using a model checker, the system is initialized according to the current locations of instances in precharts, while all locations in the main charts are set to 0. The main chart's activation state is also initialized to reflect the current state.[7] We also set the objects' properties to reflect their current value.

The model checker is then given the following property to prove, stating that it is always the case that at least one of the universal charts is active:

$$G(\bigvee_{m_i \in M^U} (act_{m_i} = 1))$$

As explained earlier, falsifying this property amounts to finding a run that leads to a point in which all active universal charts have completed successfully, with no violations, which is exactly the desired superstep.

7 Implementation and Experimental Results

We have implemented smart play-out as part of a prototype tool that links to the play-engine, thus supporting the general play-in/play-out approach. During play-out, the tool translates a play-out task into the corresponding model, runs the model checker and then injects the obtained counter-example into the play-engine. Thus, smart play-out drives the execution. We use the Weizmann Institute model-checker TLV [30] and the CMU SMV model-checker [6], but we can easily modify the tool to use other model-checkers too.

Before constructing the model we perform a static calculation to identify those charts that can potentially become active in the current super-step, and use only them when defining the system transition relation. This static calculation appears to reduce the size of the model-checking problem dramatically, since we have found that only a relatively small number of charts are usually active together in a single super-step even when the LSC specification itself is large.

The model-checkers we use are BDD based,[8] where the ordering of variables has a critical influence on running time. We use structural information from the LSC specification in order to derive good variable ordering. We also noticed that the message variables described in the translation section can be expressed in terms of the location variables, and can then be eliminated from the model. When obtaining the counter-example their values can be calculated and used for constructing the "correct" super-step.

[7] After each external event, the play-engine decides which precharts have completed and sets their corresponding main charts to be active.

[8] We have recently begun using bounded model checking based on SAT methods. In some cases, they prove to be very effective for smart play-out, yet this work is only in its initial phase.

A cellular phone system we use for illustration has about 35 different charts, and handles scenarios like dialing numbers, sending and receiving calls, opening the antenna, etc. It consists of 15 objects and uses 40 different types of messages. Calculating a super-step using our current implementation of smart play-out takes less than 1 second on a standard PC. This is fast enough to give the user a seamless feeling of working with an conventional executable model. The tool also manages to satisfy existential charts for which the counter-example has more than 100 events, in less than 2 minutes. A satisfying scenario for the existential chart shown in Fig. 4 was found by the play-engine in less then 7 seconds (including the translation, model checking and construction of the run). The scenario consists of 19 events and involves 5 different universal charts, one of which is activated 3 times.

Besides these rather dry algorithmic/performance issues, using the smart play-out tool seems to provide the user with an enhanced understanding of the behavioral requirements, and a smooth and realistic execution framework for LSCs.

Given these results and the major progress verification and model-checking has made in recent years, we are strongly believe that using such a methodology can be practical for handling real-world applications. And, as we have repeatedly mentioned, it brings us one step closer to the possibility of requirements-based code-less development of reactive systems.

8 Related Work

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

There are commercial tools that successfully handle the execution of graphical models (e.g., Statemate [16] and Rhapsody by I-Logix [20], ObjectTime [32], and Rose-RT by Rational [31]). However, they all execute an intra-object design model (statecharts) rather than an inter-object requirement model.

LSC's have been used for testing and verification of system models. Lettrai and Klose [26] present a methodology supported by a tool called TestConductor, which is integrated into Rhapsody [20]. The tool is used for monitoring and testing a model using a (rather restricted) subset of LSCs. During execution of a Rhapsody model the TestConductor monitors the charts and provides information on whether they have been completed successfully or if any violations have occurred. [26] also mentions the ability to test an implementation using these sequence charts, by generating messages on behalf of the environment (or other unimplemented classes termed stubs). Their algorithm selects the next event to be carried out at the appropriate time by the environment (or by unimplemented classes) based on a local choice, without considering the effects of the next step on the rest of the sequence, or the interaction between several charts.

Damm and Klose [8, 22] describe a verification environment in which LSCs are used to describe requirements that are verified against a Statemate model implementation. The verification is based on translating an LSC chart into a timed Buchi automaton , as described in [22], and it also handles timing issues. In both this work and [26], the assumption is that a system model whose reactive parts are described by statecharts has already been constructed, and the aim is to test or verify that model. We might thus say that while our work here focuses on putting together the information in the different charts, these papers treat each chart independently.

In a recent paper [27], the language of LSCs was extended with variables and symbolic instances. A symbolic instance, associated with a class rather than with an object, may stand for

any object that is an instance of the class. The information passed between the instances can also be parameterized, using symbolic variables. A symbolic message may stand for any message of the same kind, with actual values bound to its parameterized variables. The extension is useful for specifying systems with unbounded number of objects and for parameterized systems, where an actual instantiation of the system has a bounded number of objects, but this number is given as a parameter. In [15], the language of LSCs is further extended with powerful timing constructs, and the execution mechanism is modified so that real-time systems too can be specified and simulated directly from the requirements. We intend to extend the smart play-out algorithms to deal with both symbolic instances and the timing extensions.

Application of formal methods to the analysis of software requirements captured with SCR (Software Cost Reduction) is described in [17]. The SCR method provides a tabular notation for specifying the required relation between system and environment variables. In [5] model-checking methods are used to verify that a complete SCR model satisfies certain properties, by using SMV and Spin model-checkers. This work is very different from our work. In [5] model-checking is used for verifying properties of a state-based model (which is the traditional use of model-checking), while we use model-checking for driving the execution of a scenario-based specification.

The idea of using sequence charts to discover design errors at early stages of development has been investigated in [3, 28] for detecting race conditions, time conflicts and pattern matching. The language used in these papers is that of classical Message Sequence Charts, with semantics being simply the partial order of events in a chart. In order to describe system behavior, such MSC's are composed into hierarchal message sequence charts (HMSC's) which are basically graphs whose nodes are MSC's. As has been observed in several papers, e.g. [4], allowing processes to progress along the HMSC with each chart being in a different node may introduce non-regular behavior and is the cause of undecidability of certain properties. Undecidability results and approaches to restrict HMSC's in order to avoid these problems appear in [19, 18, 11]. In our work, the fact that LSC semantics requires that objects are synchronized while iterating during (unbounded) loops prevents such problems.

Another direction of research strongly related to our work is synthesis, where the goal is to automatically synthesize a correct system implementation from the requirements. Work on synthesis from MSC-like languages appears in [23, 24, 2, 34, 10], and an algorithm for synthesizing statecharts from LSC's appears in [13]. Moreover, a lot of work has been done on synthesis from temporal logic e.g., [9, 1, 29, 25]. The main difference is that in our work the play-out algorithms search one super-step ahead (or several super-steps when satisfying existential charts), whereas synthesis algorithms do not have such restrictions; they can thus be proven to behave correctly under all circumstances. Apart from the fact that smart play-out deals with an easier problem and therefore solutions may be more practical, we believe that play-out is complementary to synthesis. Making synthesis methodologies feasible requires designers to have good ways to understand and execute the requirements, in order to make sure that the input to the synthesis algorithm is exactly what is desired. Our approach is also useful in an iterative development cycle, where many modifications of requirements and implementations are performed; trying to run a synthesis algorithm after each modification, even assuming that synthesis becomes feasible, does not seem like a particularly good approach.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 1–17. Springer-Verlag, 1989.
2. R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.
3. R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
4. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99)*, Eindhoven, Netherlands, August 1999.
5. R. Bharadwaj and C. Heitmeyer. Model Checking Complete Requirements Specifications Using Abstraction. *Automated Software Engineering*, 6(1):37–68, January 1999.
6. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
7. 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.).
8. W. Damm and J. Klose. Verification of a Radio-based Signalling System using the STATEMATE Verification Environment. *Formal Methods in System Design*, 19(2):121–141, 2001.
9. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.
10. M. Fränzle and K. Lüth. Visual Temporal Logic as a Rapid Prototyping Tool. *Computer Languages*, 27:93–113, 2001.
11. Elsa L. Gunter, Anca Muscholl, and Doron Peled. Compositional message sequence charts. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 496–511, 2001.
12. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, January 2001. (Also in Fundamental Approaches to Software Engineering (*FASE*), Lecture Notes in Computer Science, Vol. 1783 (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22–34.).
13. D. Harel and H. Kugler. 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.).
14. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/ Play-Out Approach. Tech. Report MCS01-15, The Weizmann Institute of Science, 2001.
15. D. Harel and R. Marelly. 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. To appear.
16. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.
17. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A Toolset for Specifying and Analyzing Software Requirements. In *A.J. Hu and M.Y. Vardi, editors*, Proc. 10^{th} Intl. Conference on Computer Aided Verification (CAV'98), *volume 1427 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 5–51, 1998.
18. J.G. Henriksen, M. Mukund, K. Narayan Kumar, and P.S. Thiagarajan. On Message Sequence Graphs and finitely generated regular MSC languages. In *Proceedings of the 27th International Colloquium on Automata Languages and Programming (ICALP'2000)*, number 1853 in Lecture Notes in Computer Science, Geneva, Switzerland, 2000. Springer.
19. J.G. Henriksen, M. Mukund, K. Narayan Kumar, and P.S. Thiagarajan. Regular collections of Message Sequence Charts. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science (MFCS'2000)*, number 1893 in Lecture Notes in Computer Science, Bratislava, Slovakia, 2000. Springer-Verlag.
20. I-logix,inc., products web page. http://www.ilogix.com/fs_prod.htm.
21. ITU. ITU-T recommendation Z.120: Message sequence chart (MSC).
22. J. Klose and H. Wittke. An automata based interpretation of live sequence chart. In *Proc. 7^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.
23. K. Koskimies, T. Systa, J. Tuomi, and T. Mannisto. Automated support for modeling OO software. *IEEE Software*, 15(1):87–94, 1988.
24. I. Kruger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *Proc. DIPES'98*. Kluwer, 1999.
25. O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *2nd International Conference on Temporal Logic*, pages 91–106, Manchester, July 1997.
26. M. Lettrari and J. Klose. Scenario-based monitoring and testing of real-time uml models. In *Proc. 4th Int. Conf. on the Unified Modeling Language*, 2001.
27. R. Marelly, D. Harel, and H. Kugler. 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. To appear. Also available as Tech. Report MCS02-05, Weizmann Institute of Science, 2002.
28. Anca Muscholl, Doron Peled, and Zhendong Su. Deciding properties for message sequence charts. In *Foundations of Software Science and Computation Structure*, pages 226–242, 1998.
29. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.
30. A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *In R. Alur and T. Henzinger, editors*, Proc. 8^{th} Intl. Conference on Computer Aided Verification (CAV'96), *volume 1102 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 184–195, 1996.
31. Rational,inc., web page. http://www.rational.com.
32. B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, New York, 1994.
33. UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.
34. J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000.