Smart Play-Out Extended: Time and Forbidden Elements *

David Harel, Hillel Kugler and Amir Pnueli
The Weizmann Institute of Science, Rehovot, Israel
{dharel,kugler,amir}@wisdom.weizmann.ac.il

Abstract

Smart play-out is a powerful technique for executing live sequence charts (LSCs). It uses verification techniques to help run a program, rather than to prove properties thereof. In this paper we extend smart play-out to cover a larger set of the LSC language features and to deal more efficiently with larger models. The extensions cover two key features of the rich version of LSCs, namely, time and forbidden elements. The former is crucial for systems with time constraints and/or time-driven behavior, and the latter allows specifying invariants and contracts on behavior. Forbidden elements can also help reduce the state space considered, thus enabling smart play-out to handle larger models.

1. Background

Understanding system and software behavior by considering the various "stories" or scenarios it entails is a promising approach, which has resulted in intensive research efforts in the last few years. One of the most widely used languages for capturing scenario-based behavior is that of *message sequence charts* (MSCs), proposed long ago by the ITU [24], or its UML variant, *sequence diagrams* [23]. More recently, a broad extension of MSCs has been proposed, called *live sequence charts* (LSCs) [8], which are multi-modal in nature. They distinguish between behaviors that may happen in the system (existential) from those that must happen (universal), both on the chart level and when referring to the elements within a chart. A universal chart contains a *prechart*, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body.

In [14] a methodology for specifying scenario-based behavior, termed the *play-in/play-out approach* is described. Scenarios are captured by the user playing them in directly from a graphical interface of the system to be developed (or a virtual interface, in the form of an object model diagram). During play-in, the supporting tool, called the *Play-Engine*, constructs a formal version of the behavior in the form of LSCs.

Play-out is a complementary idea to play-in, which makes it possible to execute the behavior directly. During play-out, the user also interacts directly with the application GUI, and the Play-Engine reflects the system state at any given moment on the GUI. Play-out is actually an iterative process, where after each step taken by the user, the Play-Engine computes a super-step, which is a sequence of events carried out by the system as its response to the event input by the user. The play-out mechanism of [14] is rather naive when faced with nondeterminism, and makes essentially an arbitrary choice among the possible responses. This choice may later cause a violation of the requirements, whereas by making a different choice the requirements could have been satisfied.

One of the problems with play-out in its original form, is the inherent nondeterminism allowed by the LSC language. LSCs is a declarative, inter-object language, and as such it enables formulating high level behavior in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. Technically, the two sources of this nondeterminism are the partial order semantics among events in each chart, and the liberal nature of the interleaving between different charts during execution. These features are very useful in early requirement stages, but can cause undesired under-specification when one attempts to consider them as the system's executable behavior. Moreover, in the spirit of most tools that execute high-level system models, the naive play-out mechanism deals with nondeterminism in a way that is not controllable by the user, making choices that might be "good", but which also might cause violations that lead to aborting the run.

In [12], we introduced a more powerful technique for executing LSCs, called *smart play-out*. It takes a significant step towards removing the sources of nondeterminism during execution, proceeding in a way that eliminates some of the dead-end executions that lead to violations. In the cur-

* This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems, by the European Commission project OMEGA (IST-2001-33522) and by the Israel Science Foundation (grant No. 287/02-1).

rent paper, we extend this technique to cover two of the more advanced central features of LSCs, and in so doing also provide a means for dealing with larger models.

2. Smart Play-Out

The idea of smart play-out is to formulate the play-out task as a verification problem, and to use a model-checking algorithm [7] to find a "good" super-step (i.e., a chain reaction of system events that constitute the reaction to an external event), if one exists. Thus, we use verification techniques to help run a program, rather than to prove properties thereof.

The model-checking procedure is handed as input a transition system that is constructed from to the universal charts in the LSC specification. (These are the charts that drive the execution in the naive play-out process too.) The transition relation is designed to allow progress of the active universal charts, but to prevent violations. The system is initialized to reflect the status of the execution just after the last external event occurred, including the current values of object properties, 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 it is always the case that at least one of the universal charts is active. This is really the negation of what we want, since in order to falsify the property, the model-checker searches for a run in which eventually none of the universal charts is active. That is, all active universal charts complete successfully, so that by the definition of the transition relation no violations occurred in the process. Such a counter-example is the desired super-step. If the model-checker is able to verify the property then no correct super-step exists, but if it is not able to, the counter-example is exactly what we seek. For more details see [12].

Smart play-out can also be used to satisfy existential charts, which can be used to specify system tests. It automatically finds a trace (if there is one) that satisfies the existential chart without violating any universal charts in the process. This can be useful in understanding the possible behavior of a system and also in detecting problems, by, e.g., asking if there is some way for a certain scenario, which we believe cannot be realized by the system, to be satisfied. If smart play-out manages to satisfy the chart it will execute the trace, thus providing evidence for the cause of the problem.

Since the appearance of [12], in which we reported on smart play-out as applied to a basic kernel version of LSCs (more or less the one appearing in [8]), we have gained experience in applying the method to several applications and case studies. These include a computerized system — a machine for manufacturing smart-cards [17] — as well as a biological system — parts of the vulval development process of the *C. elegans* nematode worm [19].

We have also been working on extending smart play-out to cover a larger set of the LSC language features and to deal more efficiently with larger models, and this is the subject matter of the present paper. Specifically, we show how smart play-out has extends to cover two key features of the rich version of LSCs described in [14], namely, time and forbidden elements. The former is crucial for systems with time constraints and/or time-driven behavior, and the latter allows specifying invariants and contracts on behavior. Forbidden elements can also help reduce the state space that has to be considered by the model-checking, thus enabling smart play-out to handle larger models.

A short summary of the translation of the basic LSCs language that was defined in the original presentation of smart play-out in [12] appears in the Appendix. It is rather technical and may be skipped in a first reading of the paper, or consulted when getting into the details of the extensions for time and forbidden element.

3. Time-Enriched LSCs

In [13, 14] LSCs have been extended with timing requirements, thus making the language suitable for specifying the behavioral requirements of time intensive systems. The approach follows Alur and Henzinger [1] in basing the extension on a single clock object. The extensions have been implemented in full in the Play-Engine tool. This extension assumes a discrete time model and adopts the synchrony hypothesis, according to which the system events themselves consume no actual time, and time may pass only between events.

When handling time, play-out takes an "eager" approach, progressing with system events as far as possible, and only when faced with hot timing requirements that are not yet satisfied does it wait and allow time to pass. In contrast, smart play-out may decide not to eagerly perform all enabled events, but rather to allow some time progress before continuing the execution. As will be shown below this may help smart play-out to satisfy the LSC requirements while the "naive" play-out may cause a violation. Thus, smart play-out in effect refines the semantics of LSCs and makes it more liberal. Our extension of smart play-out is also effective in the mode where an existential chart is to be satisfied, allowing queries of the form "what is the minimal time in which some objective can be achieved?".

We now use a few simple examples to illustrate the role and possible usage of smart play-out as applied to timed behavioral requirements. Consider the two charts of Fig. 1.

The first, "*Time1*", states that when the phone's `Cover` is opened, after more than 2 time units the display sets its

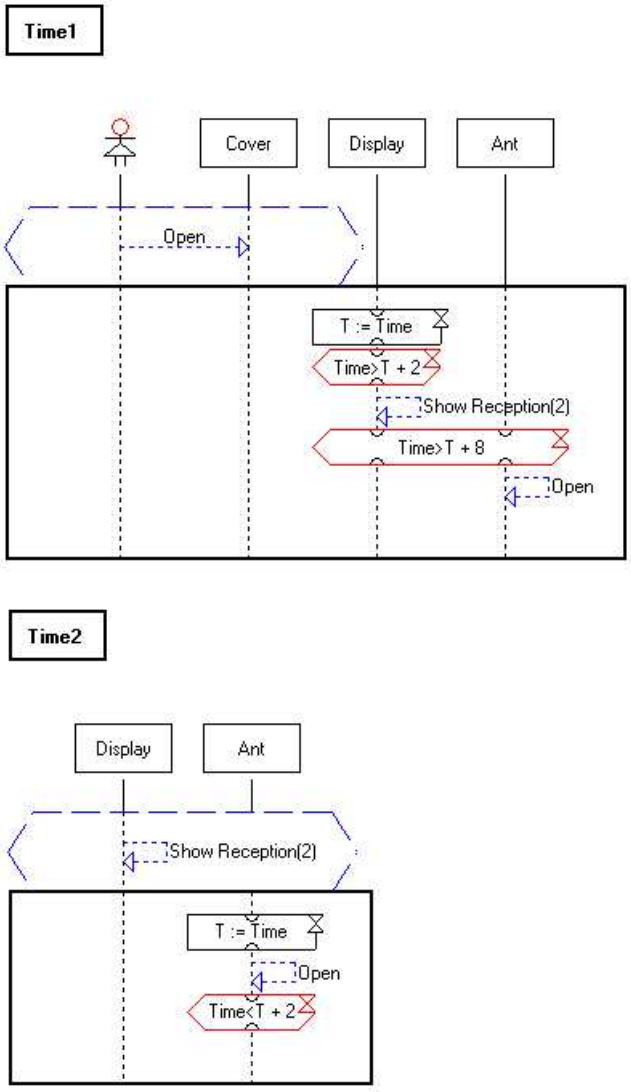

Figure 1. Smart play-out helps with time

reception level to 2 and after more than 8 time units the `Antenna` is opened. The message `Open` must occur after the message `ShowReception(2)` as implied by the partial order defined by chart "*Time1*", taking into account the synchronization enforced by the timed condition labeled `Time > T + 8`.

The second chart in the figure, "*Time2*", states that whenever the the `Display` sets its reception level to 2, the `Antenna` should open within less than 2 time units, as specified by the timed condition labeled `Time < T + 2`.

Assuming that these are the only two relevant charts of the system, and that the user opens the cover during naive play-out, the chart "*Time1*" becomes active and the play-out mechanism then immediately stores the time. After 3 time units pass, the timed condition `Time > T + 2` is satisfied, and then the message `ShowReception(2)` occurs, activating the chart "*Time2*" and the timed assignment in this chart. The timed condition `Time > T + 8` forces 6 additional time units to pass before it is satisfied and the message `Open` is taken, causing chart "*Time1*" to be completed successfully. The timed condition `Time < T + 2` is now detected as violated by play-out, causing a violation of chart "*Time2*", as shown in Figure 2 .

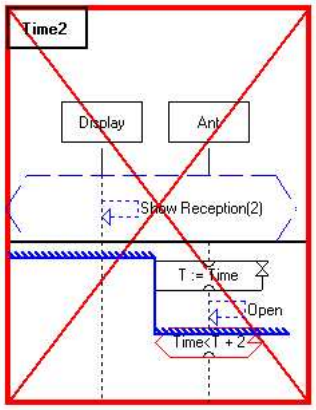

Figure 2. Violation by naive play-out

In contrast, if we apply the smart play-out process to this example, it computes and carries out a different order of events. After the user opens the cover, smart play-out allows 9 time units to pass, and only then the message `ShowReception(2)` is taken. Now, without any further time delays the message `Open` occurs, causing the successful completion of both charts.

Another issue concerns There are LSC specifications that are inconsistent due to contradicting time requirements. Consider the two charts of Fig. 3. The first, "*Time3*", states that when the phone's `Cover` is opened, the `Antenna` should open, the background of the `Display` should change to green and its reception level to 4; all according to this ordering and within 3 time units. The second chart in the figure, "*Time4*", states that whenever the the `Antenna` becomes open, the reception level of the `Display` should change to level 4, but only after at least 5 time units have passed from the opening of the `Antenna`. Smart play-out would prove that in such a case no correct super-step exists, which by the semantics of LSCs means that the requirements are inconsistent; see [11].

3.1. The Translation

We now provide some details on how our extension of smart play-out translates the time features of LSCs to the

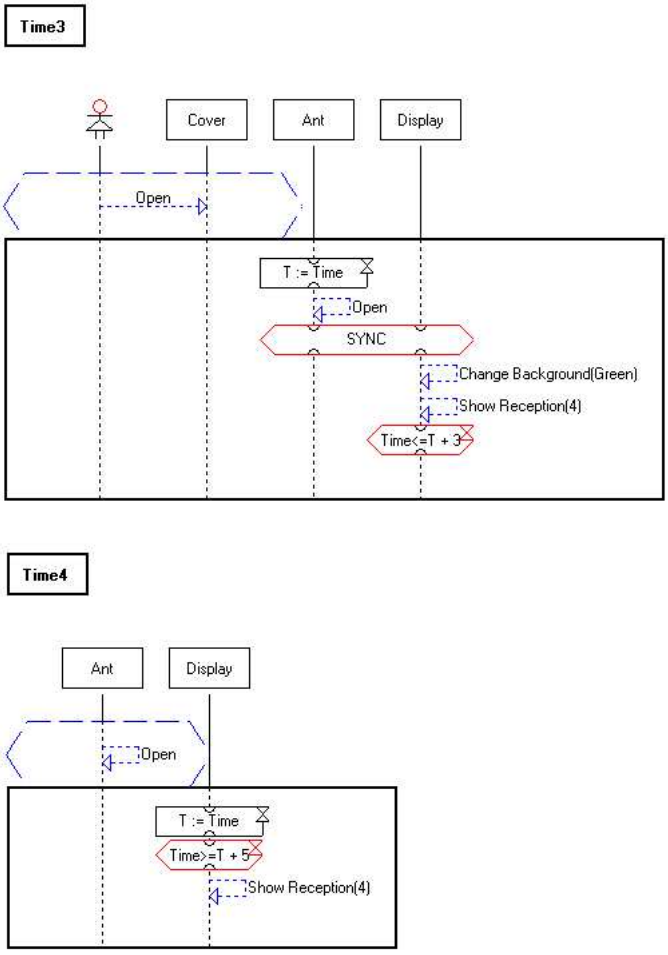

Figure 3. Inconsistent LSCs

transition relation used by the model-checking algorithms.

The time features supported are timed assignments, timed conditions and an explicit TICK message; see [13, 14]. A timed assignment is of the form $T_i := TIME$, Where T_i is a clock variable (local to the chart), and $TIME$ is the global clock. A timed condition is of the form $Time\ op\ T_i + d$, where op is any of the standard operations $=, <, \leq, >, \geq, \neq$. The delay d has an integer value, and can be a constant (the usual case), a variable or a function. An explicit $TICK$ is a self message of the clock object, and causes the global clock to progress by one time unit.

In the smart play-out translation we add a new integer variable T_i corresponding to each clock variable appearing in the LSC specification. T_i is defined to range over the domain $-1 \cdots d^{max} + 1$, where d^{max} is the maximal delay value appearing in any timed condition for this variable. If we restrict the delays d appearing in the timed conditions to be constants, then d^{max} is found simply by taking the maximum of the constant values, and for a variable or a function we take the maximal value the variable or function can return while calculating d^{max}.

$$T_i' = \begin{cases} -1 & \text{if } act_{m_i} = 1 \wedge act'_{m_i} = 0 \\ 0 & \text{if } l_{m_i,O_k} = l - 1 \wedge l_{m_i,O_k} = l \\ T_i + 1 & \text{if } TICK' = 1 \\ \{T_i, T_i + 1\} & \text{if } 0 \leq T_i \leq d^{max} \\ T_i & \text{otherwise} \end{cases}$$

A clock variable is initially set to value -1 and is set again to this value when the chart changes from active to non-active. If object O_k is at location $l - 1$ in chart m_i, and the next location of O_k corresponds to a timed assignment to the clock variable T_i, then T_i is set to 0. Once the clock variable is in the range $0 \leq T_i \leq d^{max}$, it is incremented by one if an explicit $TICK$ occurs. In case an explicit $TICK$ message does not appear in the relevant LSCs or is not enabled, a nondeterministic choice can allow to increment T_i by one or to leave it unchanged. Actually, to achieve a more efficient implementation we support "acceleration", by allowing time increments of $1, 2, 4, 8 \cdots$, which in certain cases may help find a correct run faster.

Intuitively, the specifier can add explicit time ticks to the charts, determining time progress, but may choose to specify only the timed assignments and time conditions without explicit time ticks, and then the nondeterministic choice will allow time progress. If T_i reaches its maximum value $d^{max} + 1$, it will remain with this value until the chart ends or a new timed assignment is taken. The fact that T_i remains at value $d^{max} + 1$ and that this does not change the evaluation of the timed conditions is part of the proof of the correctness of our translation and is omitted from this version of the paper.

Timed conditions are a special form of conditions and are thus handled within the framework of conditions as defined in the original version of smart play-out [12]. We define how to evaluate a timed condition using our timed clock definitions. Given a timed condition of the form $Time\ op\ T_i + d$ appearing in an LSC, we evaluate it as $T_i\ op\ d$. Since we reset T_i to 0 on a timed assignment, our evaluation of timed conditions is equivalent, and we are not forced to maintain the global clock. This is also more efficient, since maintaining global time values would force us to allocate larger ranges for the clock variables, which would have a strong effect on the performance of the model-checking.

Timed assignments can be specified also in the prechart, as shown in Fig. 4. Time is assigned to the variable T immediately after the message `ShowReception(2)` occurs, and this should be equivalent to performing the timed assignment in the main chart, as specified in Fig. 1 in chart "*Time2*". To ensure that timed assignments in the prechart are handled correctly, we modify the model-checking problem of smart play-out that was originally given as:

$$G(\bigvee_{m_i \in \mathcal{S}_\mathcal{U}} (act_{m_i} = 1))$$

to refer also to enabled timed assignments in the prechart:

$$G(\bigvee_{m_i \in \mathcal{S}_\mathcal{U}} (act_{m_i} = 1) \vee \bigvee_{m_i \in \mathcal{S}_\mathcal{U}} (ETA_{pch(m_i)} = 1)$$

Here $ETA_{pch(m_i)}$ gets the value 1, if there is an enabled timed assignment in the prechart of chart m_i, and 0 otherwise. This states that at least one of the universal charts is active or at least one of the timed assignments in a prechart is enabled. Falsifying this modified property amounts to finding a run that leads to a point in which all active universal charts have been completed successfully and there are no enabled timed assignments that have not been performed, which is exactly the desired run.

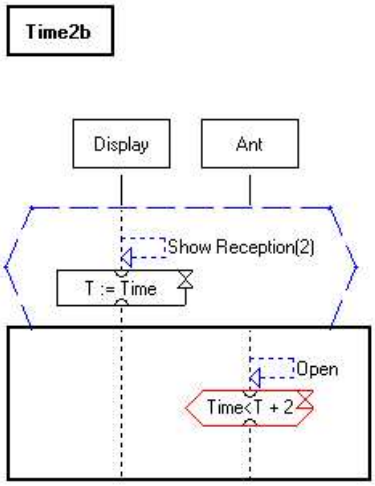

Figure 4. Timed assignment in a prechart

4. Forbidden Elements

Using forbidden elements [14] one can specify events that are not allowed to occur or conditions that are not allowed to hold during specific intervals within a chart's execution. Forbidden elements can be used to express invariants, i.e., expressions that must hold during specified execution intervals. In this section we explain how forbidden elements are fully supported by our extension of smart play-out in a direct and natural way.

The use of forbidden elements is especially important in smart play-out. Apart from the fact that forbidden elements are one of the LSC features supported in the standard play-in/play-out approach and the Play-Engine tool, they have a significant role for smart play-out since they allow the user to provide additional knowledge about the system, e.g., invariants or preconditions, which can reduce the state-space dramatically and allow smart play-out to handle much larger designs.

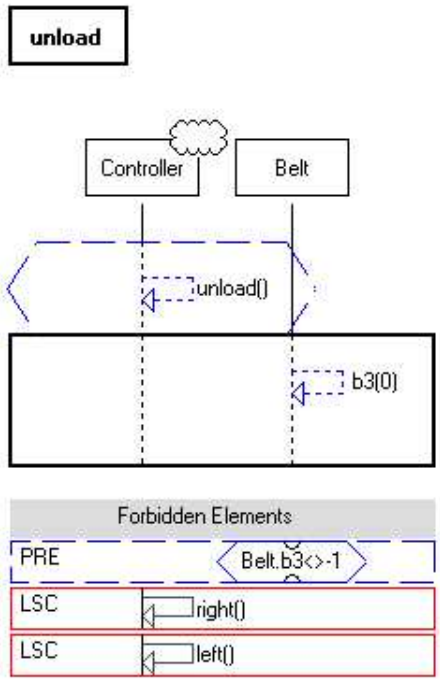

Figure 5. Forbidden elements

Forbidden elements are either messages or conditions, and are specified in a special area at the bottom of the LSC, separated by the `Forbidden Elements` header. An example of both element types appears in Fig. 5. The chart describes the *Unload* scenario for a model of a smart-card manufacturing machine [17]. When the `Controller` sends the self message `unload`, as specified in the prechart, a new empty card is placed on the belt, as specified by the message `b3(0)` appearing in the main chart. No belt movements are allowed during this scenario. This is specified by designating the `right` and `left` messages as forbidden while the unload chart is active.

The distinction between hot and cold applies also to forbidden messages, where a hot forbidden message is not allowed to occur in the designated scope, and if it does it causes a violation and the system aborts, while the occurrence of a cold forbidden message causes the exit of the rel-

evant (sub) chart but it does not mean a violation. In our example of Fig. 5 the forbidden messages `right` and `left` are hot, since they are strictly not allowed during the unload scenario.

Fig. 5 also shows the use of a forbidden condition. The *Unload* scenario should not be performed if there is already a card on the belt in the relevant slot `b3`. This is specified by the forbidden condition `Belt.b3 <> -1`, where by convention -1 denotes the fact that no card is placed on the belt slot, 0 denotes an empty card, and a positive value denotes a personalized card. The forbidden condition is cold and has the prechart as its scope. Thus, if the controller performs the `unload` message while the slot `b3` is not empty, the forbidden condition `Belt.b3 <> -1` becomes true and the prechart is exited without activating the main chart. As a result the unload scenario designated in the main chart will not be taken. In general, the scope for forbidden elements can be the entire LSC, its prechart, its main chart, or any subchart thereof.

4.1. The Translation

We first explain how we handle forbidden messages. As explained in the Appendix, without considering forbidden messages we define the transition relation for the occurrence of a message as follows:

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

In order for the event of sending msg from O_j to O_k to occur, we require the condition ϕ to hold. This condition is defined in a way that requires that at least one of the main charts in which this message appears is active, and that all active charts must "agree" on this message. When considering also forbidden messages we conjunct ϕ with a formula ψ' specifying that $msg^s_{O_j \to O_k}{}'$ is not a hot forbidden message in the current scope.

$$\phi' = \bigwedge_{m_i \in \mathcal{S}_\mathcal{U} \wedge msg^s_{O_j \to O_k} \in ForbMessages(m_i)} \chi(m_i)$$

$$\chi(m_i) \triangleq (act_{m_i} = 0 \vee \psi(m_i)) \vee ((l_{O_1} < L_1 \vee l_{O_1} > H_1) \wedge (l_{O_2} < L_2 \vee l_{O_2} > H_2) \wedge \cdots \wedge (l_{O_n} < L_n \vee l_{O_n} > H_n))$$

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

We require that for each universal chart m_i in which $msg^s_{O_j \to O_k}{}'$ is designated as a hot forbidden message, either m_i is not active (that is, $act_{m_i} = 0$), or all of the objects participating in the subchart are outside the subchart, or the message $msg^s_{O_j \to O_k}{}'$ appears in the subchart and is enabled (that is, $\psi(m_i)$). Assuming $O_1, O_2 \cdots O_n$ are the objects participating in the subchart, an object O_i is outside the subchart if $l_{O_i} < L_i \vee l_{O_i} > H_i$ holds. Here L_i is the minimal location for object O_i in the subchart while H_i is the maximum location in the subchart. In case the scope is the entire LSC or the main chart the clause relating to participating objects being outside the (sub) chart evaluates to `FALSE` and can thus be omitted.

A cold forbidden message affects the transition relation of the location of an object participating in the (sub) chart. If this cold forbidden message occurs, the participating objects exit the subchart.

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

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 m from O_j to O_k, then if in the next state the message m is sent, the location is advanced; otherwise it remains where it is, unless the cold forbidden message $msg^s_{O_j \to O_k}{}'$ occurs, in which case object O_j exits the subchart to location $H_j + 1$.

The treatment of forbidden conditions follows along similar lines. A condition is 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 it can relate to the properties of several objects. Here, the properties appearing in the condition are $P_1, P_2, \cdots P_r$. The values of properties can change only as an effect of a message occurring, so for a hot forbidden conditions we disallow a message if in the next state we are in the scope of the forbidden condition and the condition holds. Also, cold forbidden conditions affect the transition relation of the location of the participating (sub) chart objects, but we omit the details in this version of the paper. Notice that forbidden elements do not introduce additional variables into our transition system, and thus they constitute an effective means for reducing the state space and allowing smart play-out to handle larger models.

5. Related Work

A large amount of work has been carried out on formal requirements, sequence charts, and model execution. Amyot and Eberlein [4] provide an extensive survey of scenario notations. Their paper also defines several comparison criteria and then uses them to compare the different notations. The idea of using sequence charts to discover design errors at early stages of development has been investigated in [2, 22] for detecting race conditions, time conflicts and pattern matching. The language used in these papers is that

of classical message sequence charts, with the semantics being simply the partial order of events in a chart. In order to describe system behavior, such MSCs are composed into hierarchal message sequence charts (HMSCs) which are basically graphs whose nodes are MSCs. As has been observed in several papers, e.g. [3], 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 HMSCs in order to avoid these problems appear in [15, 16, 10].

Live sequence charts have been used for the testing and verification of system models. Lettrai and Klose [21] present a methodology supported by a tool called TestConductor, which is integrated into Rhapsody [18]. The tool is used for monitoring and testing a model using a restricted subset of LSCs. Damm and Klose [9, 20] describe a verification environment in which LSCs are used to describe requirements that are verified against a Statemate model implementation. LSCs have also been applied to the specification and verification of hardware systems [5, 6].

References

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

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

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *10th International Conference on Concurrency Theory (CONCUR99)*, volume 1664 of *Lect. Notes in Comp. Sci.*, pages 114–129. Springer-Verlag, 1999.

[4] D. Amyot and A. Eberlein. An evaluation of scenario notations for telecommunication systems development. In *Int. Conf. on Telecommunication Systems*, 2001.

[5] A. Bunker and G. Gopalakrishnan. Verifying a VCI Bus Interface Model Using an LSC-based Specification. In H. Ehrig, B. J. Kramer, and A. Ertas, editors, *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology*, pages 1–12, 2002.

[6] A. Bunker and K. Slind. Propert Generation for Live Sequence Charts. Technical report, University of Utah, 2003.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).

[9] 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.

[10] E. L. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. In *Proc. 7^{th} Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), volume 2031 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, pages 496–511, 2001.

[11] 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.).

[12] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart playout of behavioral requirements. In *Proc. 4^{th} Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *Lect. Notes in Comp. Sci.*, pages 378–398, 2002. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.

[13] 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.

[14] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[15] J. Henriksen, M. Mukund, K. Kumar, and P. Thiagarajan. On message sequence graphs and finitely generated regular MSC languages. In J. R. U. Montanari and E. Welzl, editors, *Proc. 27th Int. Colloq. Aut. Lang. Prog.*, volume 1853 of *Lect. Notes in Comp. Sci.*, pages 675–686. Springer-Verlag, 2000.

[16] J. Henriksen, M. Mukund, K. Kumar, and P. Thiagarajan. Regular collections of Message Sequence Charts. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science(MFCS'2000)*, volume 1893 of *Lect. Notes in Comp. Sci.*, pages 675–686. Springer-Verlag, 2000.

[17] H.Kugler and G. Weiss. Planning a Production Line with LSCs. Technical report, Weizmann Institute, 2004.

[18] Rhapsody. I-Logix, Inc., products web page. http://www.ilogix.com/products/.

[19] N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E. Hubbard, and M. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. In Corrado Priami, editor, *Proc. Int. Workshop on Computational Methods in Systems Biology (CMSB 2003)*, pages 4–20. Springer-Verlag, 2003. Extended version To appear in Modeling in Molecular Biology, G.Ciobanu (Ed.), Natural Computing Series, Springer, 2003 .

[20] 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), volume 2031 of* Lect. Notes in Comp. Sci., *Springer-Verlag*, 2001.

[21] 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.

[22] A. Muscholl, D. Peled, and Z. Su. Deciding properties for message sequence charts. In *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures (FOSSACS '98)*, number 1378 in Lect. Notes in Comp. Sci., pages 226–242. Springer-Verlag, 1998.

[23] UML. Documentation of the unified modeling language (UML). Available from the Object Management Group (OMG), http://www.omg.org.

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

Appendix A: The Basics of Smart Play-Out

We provide here a short summary of the notations and translation used in the basic smart play-out process. For more details see [12, 14].

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

We define a transition system with the following variables:

act_{m_i} , determining if universal chart m_i is active. Its value is 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. Its 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. Its value is set to 1 at the occurrence of the receive and is changed to 0 at the next state.

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

We denote by $f(l) = evnt(l)$ 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. organized by the various features of the LSC language.

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 where it is. 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 the happenings in some other chart. This is one of the places were the interaction between the different charts becomes important.

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

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

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

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

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

Let us explain. 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 the 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 that contains $msg^s_{O_j \to O_k}$, we require that object O_j progress to a location l_t corresponding to this message, as expressed by the formula $\psi(m_i)$. Formula ϕ_2 states that for all charts m_i containing $msg^s_{O_j \to O_k}$ (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 (i.e., its sending and receipt) is allowed to occur 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 containing $msg^s_{O_j \to O_k}$.

Precharts

The prechart of a universal chart describes the scenario which, if completed successfully, forces the scenario described in the main chart to occur. The main chart becomes active if all locations of the prechart have reached maximal positions, which is what successful completion of the prechart means. A central feature of play-out a sequence of events in a super-step causing the activation of some additional universal chart, which now must also be completed successfully as part of the same super-step. For this purpose precharts are monitored, and locations along instance lines are advanced when messages are sent and received.

Activation of Charts

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

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

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

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.

The Model-Checking Formula

To compute a super-step in the execution of an LSC system 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. After each external event, the Play-Engine decides which precharts have completed and sets their corresponding main charts to be active. We also set the properties of the objects 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:

$$G(\bigvee_{m_i \in \mathcal{S}_\mathcal{U}} (act_{m_i} = 1))$$

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 super-step.