How Hard is Smart Play-Out?
On the Complexity of Verification-Driven Execution [*]

David Harel[1] , **Hillel Kugler**[2] , **Shahar Maoz**[1] , **Itai Segall**[1]

[1] *The Weizmann Institute of Science, Israel*
{dharel,shahar.maoz,itai.segall}@weizmann.ac.il

[2] *Microsoft Research, Cambridge, UK*
hkugler@microsoft.com

Abstract

Smart play-out is a method for executing declarative scenario-based requirements, which utilizes powerful model-checking or planning algorithms to run the scenarios and avoid some of the violations that can be caused by naïve execution. In this paper, we investigate the complexity of smart play-out. Specifically, we use a reduction from QBF in order to show that smart play-out for a most basic subset of the scenario-based language of LSC is PSPACE-hard. The main advantage of our proof compared to a previous one by Bontemps and Schobbens is that ours is explicit, and takes advantage of the visual features of the LSC language. We also show that for a subset of the language, in which no multiple running copies are allowed, the problem is NP-hard.

1 Introduction

Live sequence charts (LSCs) [2] constitute a visual formalism for inter-object scenario-based specification and programming of reactive systems. The language extends classical *message sequence charts* (MSC) [10], mainly by adding universal and existential modalities. Thus, LSCs distinguish between behaviors that may happen in the system (existential, cold) and those that must happen (universal, hot). A universal chart contains a *prechart*, which specifies the scenario which, if satisfied, forces the system to satisfy also the scenario given in the actual chart body.

An operational semantics was defined for the language in [8]. The result of this semantics is an execution technique for LSCs, called *play-out*, in which an LSC specification consisting of a set of LSCs is executed directly. However, the execution engine presented in [8] is not enough, due to its naïve nature. At each point in time, it chooses one step that is legal at that time, without considering the consequences of one choice or the other. Thus, it may choose steps

[*]The research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science and by a Grant from the G.I.F., the German-Israeli Foundation for Scientific Research and Development.

that eventually lead to a violation even though other steps could have avoided it. The notion of *smart play-out* is therefore introduced in [7]. Two implementations have been suggested to date for smart play-out [7, 9].

In this paper, we analyze the theoretical bounds on the complexity of smart play-out, and prove the problem to be PSPACE-hard. We also show an interesting subset of the problem, in which no multiple running copies are allowed, to be NP-hard.

Despite the high theoretical worst case complexity, in practice, there are interesting LSC specifications for which smart play-out can be applied successfully. We believe that since LSC specifications are man-made, they will inherently have some "logical" structure, which can be exploited for optimizing smart play-out techniques, causing them to be practical. This issue is further discussed in Section 5.4. Another reason that smart play-out may be effective despite its worst case complexity is rooted in the significant advances made over the past few years in verification methods and tools for analyzing large complex systems.

The paper is organized as follows. Section 2 presents preliminary material on LSCs. Section 3 defines the problem of smart play-out. In Section 4 we prove the complexity results for the problem. In Section 5 we discuss some issues related to the smart play-out problem and its complexity. Finally, Section 6 covers some related and future work. The formal proofs omitted from the body of the paper are given in Appendix A.

2 Preliminaries

LSCs inherit the syntactic structure and visual representation from MSCs. An LSC contains vertical lines, termed *lifelines*, which denote objects, and *events*, involving one or more lifelines, thus inducing a partial order. The most basic construct of the language are messages: a message is denoted by an arrow between two lifelines (or from a lifeline to itself), representing the event of the source object sending a message to the target object. Objects are partitioned into environment-controlled and system-controlled ones, and messages are said to be controlled by the environment or system according to the sending object. A universal LSC, the kind used to drive the execution, is divided into two parts, the prechart and the main chart. The intended semantics is that whenever the prechart is satisfied, the main chart must also be satisfied. LSCs are multi-modal; almost any construct in the language can be either cold (usually denoted by the color blue) or hot (denoted by red), with a semantics of "may happen" or "must happen", respectively. If a cold element is violated (say a condition that is not true when reached), this is considered a legal behavior and some appropriate action is taken. Violation of a hot element, however, is considered a violation of the specification and must be avoided.

An LSC may also contain conditions, *if-then-else* constructs, etc., as well as more advanced constructs such as symbolic instances (representing classes instead of individual objects), and forbidden elements. Using conditions, one can also express *anti-scenarios*, i.e., scenarios that are forbidden, by introducing an LSC with the entire forbidden scenario as a prechart, and a hot FALSE condition as the main chart.

An example of an LSC can be seen in Figure 1(a). The prechart is denoted by the dashed blue hexagon, and the main chart by the solid black rectangle. The prechart contains a single message, in which the object $Y\{i-1\}T$ sends `Choose_Next` to itself. The main chart states that XiT should send `m` to XiF, then YiT should send `m` to YiF, etc. Note the `SYNC` construct, dictating an order between these two messages, which would have otherwise been unordered.

An operational semantics was defined for the language in [8]. The result of this semantics is an execution technique for LSCs, called *play-out*, in which an LSC specification consisting of a set of LSCs is executed directly. In this semantics, the notion of *cut* is used. The cut includes one location on each lifeline, separating the set of messages that have already occurred from those that have not. Whenever a message that is minimal in the prechart is sent, a *running copy* of this chart is created, and the cut starts to progress. A runtime *configuration* of an LSC specification is a set of running copies and their corresponding cuts. Messages appearing immediately after the cut are termed *enabled.* If a message that appears in a chart is sent while not being enabled in it, the chart is violated (either causing a violation of the specification or a graceful exit of the chart, according to the modalities/temperatures).

Note that if a message appears twice in a chart, once as minimal and again as non-minimal, then this chart may have multiple running copies at runtime: if the second appearance is enabled, and the message occurs, then this copy must advance its cut to after the message and another copy opens, advancing its own cut to be directly after the minimal event. The complexity for the case where multiple running copies are not allowed is discussed in Section 4.2.

3 The Smart Play-Out Problem

Given an initial configuration, we define a *superstep* to be a finite set of steps executed by the system and ending in a state in which the system has no more obligations. This means that after performing the superstep all the main charts [†] have completed successfully. We also assume that all events in the superstep were taken as a result of appearing explicitly in the main chart of an LSC and being enabled; that is, we do not consider the case of taking events 'spontaneously', which may be in general helpful by allowing violation of precharts, amounting to making less commitments to satisfy main charts.

The problem of *smart play-out* is defined as the problem of finding a legal superstep from a given initial configuration if such a superstep exists, or deciding and reporting that no such superstep exists otherwise. This problem is proven to be PSPACE-hard in [1]. In the present paper, we give an alternative proof, by a reduction from QBF, the canonical PSPACE-complete problem [3].

We prove this complexity for the subset of LSCs containing only messages. As a first step, we allow also synchronization constructs and anti-scenarios (charts for which the main chart is merely a FALSE condition, causing the completion of the prechart to be an immediate violation of the specification), and then explain how these can be removed.

Note that we assume nothing about the structure of the charts themselves. For example, a message may appear several times in a chart. In particular, a message that is minimal in the prechart can also appear again in the same chart. This feature allows multiple copies of the same chart to be active simultaneously at runtime, each at a different cut. Although our main focus is on the most general case, the case in which multiple running copies are disallowed is also discussed in Section 4.2.

4 How Hard is Smart Play-Out?

4.1 The general problem

Theorem 1 *Any smart play-out mechanism supporting messages, synchronization constructs, and anti-scenarios is PSPACE-hard.*

PROOF. We prove this by a reduction from QBF. Given a QBF formula $\varphi = \exists x_1 \forall y_1 \exists x_2 \forall y_2 \cdots \exists x_n \forall y_n (\psi)$, where ψ is a CNF formula over variables $\{x_1, y_1, \ldots, x_n, y_n\}$, we build a system and an LSC specification. The specification is built such that a superstep exists (from a specific initial configuration) if and only if the formula is true.

Intuitively, the superstep will backtrack over the variables x_i, y_i, where it will non-deterministically choose a value for the x_i's and check both options for the y_i's. For each such assignment to all variables, it checks that the CNF formula ψ holds.

[†]To simplify the discussion we assume that the main charts contain only system events.

We now describe the system and specification generated, first intuitively and then more formally. The objects in the system are v_iT, v_iF for each variable $v_i, v \in \{x, y\}$. The assignment of a TRUE value to a variable v_i is denoted by a message `m` passing from v_iT to v_iF and then back. A FALSE value is denoted by the same two messages, in the reverse order (i.e., first from v_iF to v_iT).

Some of these objects may also send various messages to themselves, as follows.

- A message `Choose_Next` sent from an object y_iT to itself denotes the fact that x_i and y_i have both received values and the execution can proceed to the next pair.
- A message `Done` sent from an object y_iT to itself indicates a backtrack from y_i — if y_i was previously TRUE, then the FALSE value should now be checked, and if it was FALSE the execution will backtrack to $i-1$.
- A message `Check` sent from either x_iT or y_iT to itself denotes the fact that all variables are assigned values. The execution will now "resend" these values and check that the assignment is legal (i.e., that ψ holds).
- A message `Done_Check` sent from either x_iT or y_iT to itself denotes the fact that this variable was checked. Once all variables x_i, y_i are checked without the specification being violated, we know that the current assignment is legal (i.e., ψ holds).

The specification consists of three groups of LSCs, with the following roles:

- Type 1 LSCs take care of the backtracking stage. They backtrack over the variables x_i and y_i, while non-deterministically choosing values for the x_i's and checking both options for the y_i's.
- Type 2 LSCs act as the memory of the system; they keep the most recent assignment to each variable, and once all variables are assigned values they "resend" these values, so that ψ can be checked.
  If a backtracking iteration changes only some of the values, these LSCs are the ones in charge of remembering the values of the rest of the variables.
- Type 3 LSCs check that ψ holds in the current assignment. Each LSC checks one clause, and if a violated clause is detected, it causes the specification to be violated.

Formally, the system consists of $4n + 1$ objects, labeled:

$$x_1T, x_1F, y_1T, y_1F, \ldots, x_nT, x_nF, y_nT, y_nF, \text{ and } y_0T$$

and of $12n + m + 1$ LSCs (where m is the number of clauses in ψ and n is half the number of variables appearing in ψ), as follows.

- Type 1 LSCs:
  - *Type1A* charts: Figure 1(a) shows n different charts (for $i = 1, \ldots, n$). Each such chart, upon seeing a `Choose_Next` message from y_{i-1} to itself, sends `m` from x_iT to x_iF, then from y_iT to y_iF and from y_iF to y_iT, and, finally, the message `Choose_Next` from y_iT to itself.
  - *Type1B* charts: Figure 1(b) shows n charts, similar to the Type 1A charts, only the first main chart message is from x_iF to x_iT (as opposed to the other way around in Type 1A).
  - *Type 1C* charts: Figure 1(c) shows n charts, stating that whenever `m` is sent from y_iT to y_iF and back, and then y_iT sends `Done` to itself, m should be sent from y_iF to y_iT and back, and, finally, y_iT should send `Choose_Next` to itself.
  - *Type 1D* charts: Figure 1(d) shows n charts, stating that whenever `m` is sent from y_iF to y_iT and back, and then y_iT sends `Done` to itself, $y_{i-1}T$ should also send `Done` to itself.
  - *Type 1E* charts: Figure 1(e) shows a chart, specifying that once y_nT sends `Choose_Next` to itself, all objects $v_iT, v \in \{x, y\}, i \in \{1, \ldots, n\}$ should send themselves `Check` and `Done_Check`. After they all do so, y_nT should send `Done` to itself.
- Type 2 LSCs:
  - *Type 2A* charts: Figure 2(a) shows $2n$ different charts (for $v \in \{x_i, y_i\}$, $i = 1, \ldots, n$). Each such chart states, for a specific object pair vT, vF, that if vT sends `m` to vF, vF sends `m` to vT, and then vT sends `Check` to itself, vT should once again send `m` to vF, vF should send `m` to vT, and, finally, vT should send `Done_Check` to itself.
  - *Type 2B* charts: Figure 2(b) shows $2n$ charts, similar to Type 2A charts, except that the order between the vT sending `m` and the vF sending `m` is switched, both in the prechart and in the main chart.
  - *Type 2C*, *Type 2D* charts: Figures 2(c) and 2(d) show $2n$ charts each, similar to Type 2A and Type 2B charts, respectively, with the addition of object vT sending `Done_Check` to itself before the message `Check` in the prechart.

- Type 3 LSCs:
  Let $\psi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. We introduce a single chart for each clause $C_j, j = 1, \ldots, m$. An example of this chart, corresponding to the clause $x_1 \vee y_1 \vee \neg y_2$, is given in Figure 3. The example states that the following scenario is forbidden: the scenario contains three independent sequences, with no explicit order between them. The first contains object x_1T sending `Check` to itself, x_1F sending `m` to x_1T, x_1T sending `m` to x_1F and x_1T sending `Done_Check` to itself. The second sequence is similar, for objects y_1T and y_1F. Finally, the third sequence is for objects y_2T and y_2F, except that the order of the two `m` messages is switched.
  It is easy to see how this example generalizes to any clause C. All literals v_k in C are represented similarly to x_1 and y_1 in the example, and all literals $\neg v_k$ are represented similarly to y_2.

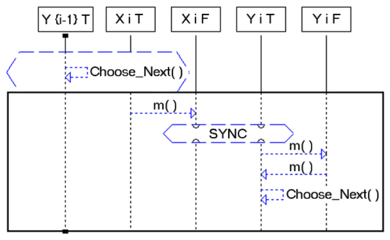
Note that the size of the specification is polynomial in the size of the input: it consists of $12n + m + 1$ LSCs, with sizes as follows. The Type 1E LSC is of size polynomial in n (it contains $2n$ objects, each sending a constant number of messages). Type 3 LSCs are of size polynomial in the size of the clauses of ψ (and all together, polynomial in the size of ψ itself). All other LSCs are of constant size.

We also introduce another set, called Type 1C′ LSCs; see Figure 4. These are similar to Type 1C charts, with an extra `Choose_Next` message sent from y_iT to itself in the prechart (before the `Done` message). These LSCs are not part of the final specification, but are used as an intermediate step in the proof below.

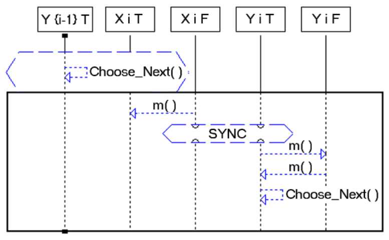
Throughout the rest of the proof, we denote the event of object o_1 sending message `msg` to object o_2 by $o_1 \xrightarrow{\texttt{msg}} o_2$.

We now explain intuitively why the specification constructed here has a legal superstep, triggered by an external $y_0T \xrightarrow{\texttt{Choose\_Next}} y_0T$ event, if and only if φ is true. The formal proof for this claim is given in Appendix A.

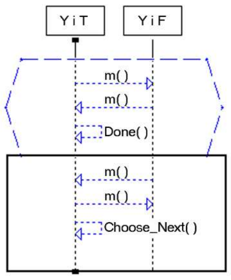
We explain this by incrementally adding the chart types to the specification. First, consider a specification containing only charts of types 1A, 1B, 1C′, 1D and 1E. These represent the backtracking "engine". This engine backtracks over the x_is and y_is, where in backtracking level i, x_iT and x_iF send each other the message `m` in a non-deterministic order, then $y_iT \xrightarrow{\texttt{m}} y_iF$ and $y_iF \xrightarrow{\texttt{m}} y_iT$ are sent in this order, and the following backtracking level is called (chart types 1A, 1B). Upon return of this call, $y_iF \xrightarrow{\texttt{m}} y_iT$ and $y_iT \xrightarrow{\texttt{m}} y_iF$ are sent in this order (which is to be contrasted to the eariler case), and the following backtracking level is called again (chart Type 1C'). Upon



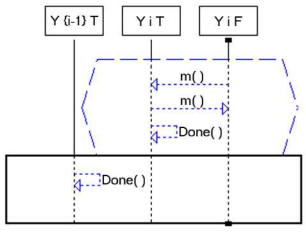
(a) Type 1A charts: n charts, which, together with Type 1B charts, nondeterministically choose an x_i, and then choose $y_i = T$.



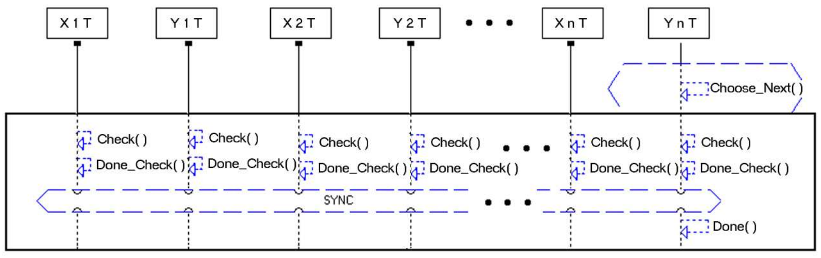
(b) Type 1B charts: n charts, which, together with Type 1A charts, nondeterministically choose an x_i, and then choose $y_i = T$.



(c) Type 1C charts: n charts, for choosing $y_i = F$.

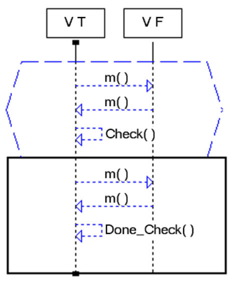


(d) Type 1D charts: n charts for signaling that both values for y_i have been chosen.

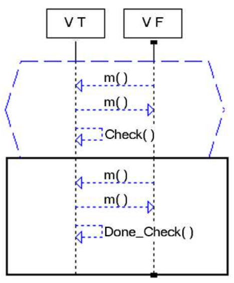


(e) Type 1E chart: A single chart for sending the `Check` and `Done_Check` messages.

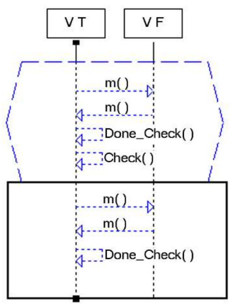
Figure 1: LSCs of Type 1, forcing the smart play-out mechanism to check all values for y variables and choose a value for each x variable.



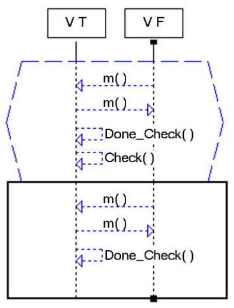
(a) Type 2A charts: for "resending" the last value, if true, of each variable v after a $v_t \xrightarrow{\texttt{Check}} v_t$ message.



(b) Type 2B charts: for "resending" the last value, if false, of each variable v after a $v_t \xrightarrow{\texttt{Check}} v_t$ message.



(c) Type 2C charts: for "resending" the last value, if true, of each variable v after a $v_t \xrightarrow{\texttt{Check}} v_t$ message, if it was not changed since the previous `Check` message.



(d) Type 2D charts: for "resending" the last value, if false, of each variable v after a $v_t \xrightarrow{\texttt{Check}} v_t$ message, if it was not changed since the previous `Check` message.

Figure 2: LSCs of Type 2, causing the last value of each variable to be "resent" after each `Check` message.

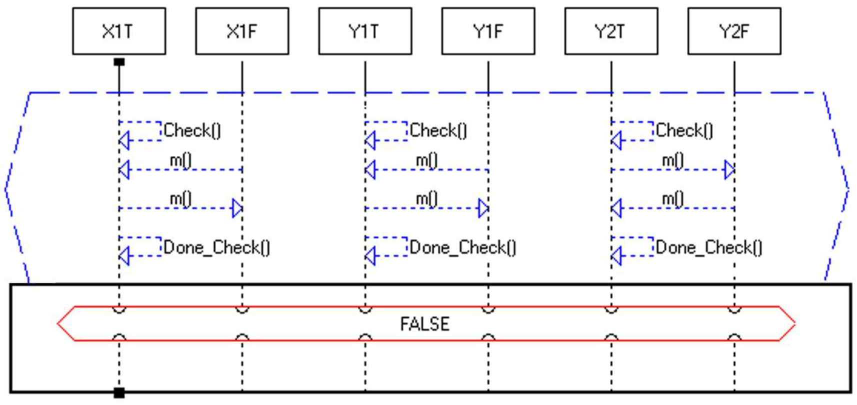


Figure 3: An example for a Type 3 LSC, corresponding to the clause $x_1 \vee y_1 \vee \neg y_2$. The chart causes the case where $x_1 = F, y_1 = F, y_2 = T$ to be forbidden.

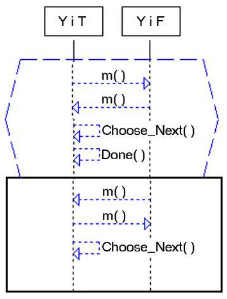


Figure 4: Type 1C′ charts: similar to Type 1C but with an extra $v_iT \xrightarrow{\texttt{Choose\_Next}} v_iT$ message in the prechart.

return of this second call, this level returns as well (chart Type 1D). At the halting condition, i.e., at level n, all x_iT and y_iT objects send themselves `Check` and `Done_Check` in this order (chart 1E). The idea is that for variable v_i, the order between $v_iT \xrightarrow{\texttt{m}} v_iF$ and $v_iF \xrightarrow{\texttt{m}} v_iT$ represents the value given to it. Thus, for x variables, the order is chosen non-deterministically and for y variables both options are checked. The halting condition is the point where the CNF formula ψ will be checked against the assigned values.

Type 2A and Type 2B charts act as memory – whenever a variable v_i is assigned some value (i.e., $v_iT \xrightarrow{\texttt{m}} v_iF$ and $v_iF \xrightarrow{\texttt{m}} v_iT$ are sent in a certain order), the appropriate 2A or 2B chart will advance past the first two prechart messages. Then, following a $vT \xrightarrow{\texttt{Check}} vT$ message, the two m messages will be resent in the same order. 2C and 2D are similar, and will resend the messages if a $vT \xrightarrow{\texttt{Done\_Check}} vT$ is sent before the $vT \xrightarrow{\texttt{Check}} vT$ (which happens whenever this variables has not been reassigned a value since the last time the halting condition was reached). Since $vT \xrightarrow{\texttt{Check}} vT$ is sent for all x_is and y_is at the halting condition of the backtracking algorithm (of the Type 1 charts), augmenting the Type 1 charts with Type 2 charts causes all variables to resend their last $v_iT \xrightarrow{\texttt{m}} v_iF$ and $v_iF \xrightarrow{\texttt{m}} v_iT$ messages (in the same order they were last sent) whenever the halting condition is reached.

Type 3 charts wait for this resend at the halting condition. Each chart of Type 3 checks a single clause in ψ. Its prechart will be satisfied if and only if the clause does not hold, i.e., all the variables relevant to it were assigned values that are negations of those in the clause. Since Type 3 charts are anti-scenarios (i.e., with a hot FALSE condition as the main chart), if the prechart of a Type 3 chart is ever satisfied, the specification is violated. Thus, augmenting the specification with Type 3 charts assures that whenever the halting condition in the backtracking algorithm is reached, all clauses are satisfied, and ψ holds.

Therefore, the full specification has a legal superstep, following an external $y_0T \xrightarrow{\texttt{Choose\_Next}} y_0T$ event, if and only if the backtracking algorithm manages to assign values to the x_is and y_is such that ψ always holds, which can happen if and only if φ is true.

∎

We are now ready to move on to the main theorem of the paper — the PSPACE-hardness of smart play-out mechanisms:

Theorem 2 *Any smart play-out mechanism supporting messages is PSPACE-hard.*

Proof. It is enough to show how the synchronization constructs and anti-scenarios of the above construction can be reduced to messages. There are three types of constructs that need to be removed, as follows.

1. The SYNC constructs on two objects, appearing in Type 1A and Type 1B charts, can easily be replaced by a newly introduced message between the two lifelines (a different message for each chart). Each of these messages appears only once in the specification, therefore their only effect is in synchronizing the two lifelines, much like the original synchronization construct.
2. The SYNC construct in the Type 1E chart can be replaced by a series of new messages from each lifeline to the following one (i.e., x_1T to y_1T, and then y_1T to x_2T, etc.). This will ensure that the $y_nT \xrightarrow{\texttt{Done}} y_nT$ message will be executed only after all `Done_Check` messages were sent, which is the purpose of the SYNC.
3. Anti-scenarios in Type 3 charts can be removed as follows. First, we duplicate the chart. In one copy, we use two newly introduced messages `msg1` and `msg2`, and specify that they must appear in a certain order. In the second, we require them to appear in the opposite order. Clearly, no superstep can satisfy both, so that the new pair of LSCs acts, in combination, as an anti-scenario that prevents the play-out mechanism from satisfying the prechart.

Clearly, after making these changes, the specification is still polynomial in the size of the input. ∎

4.2 Multiple running copies

In general, the semantics of LSC and of play-out allows multiple copies of the same chart to be simultaneously active. This may happen if messages that are minimal in the chart are allowed to reappear in it. A similar issue was mentioned in [11] as one that requires special attention — forbidding events from appearing more than once in the same chart allowed a more succinct translation of the LSC language into temporal logic. It remains an open question as to whether this is a necessary condition for such a succinct translation to exist.

Both known PSPACE-hardness proofs — the one in [1] and ours — use this feature in order to "implement memory". For example, in our proof this is heavily relied on in the Type 2 LSCs: the prechart "remembers" the last value assigned to each variable, and the main chart "resends" it. It is crucial that when the main chart resends this value, a new copy opens with its prechart "remembering" this value.

We do not know whether the complexity of the problem drops when one forbids simultaneous multiple running copies of the same chart. What we can prove, however, is that the smart play-out problem, even without multiple running copies, is NP-hard. While this does not answer the question completely, it does show that the problem is still hard enough even in this constrained case. Note that even without multiple running copies, the superstep may still be of length exponential in the size of the specification. This is discussed further in Section 5.3.

Theorem 3 *Smart play-out without multiple running copies is NP-hard.*

PROOF. We prove this by a reduction from SAT. Similarly to the proof of the main theorem above, given a CNF formula $\psi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ over variables $\{x_1, \ldots, x_n\}$, we build a system and an LSC specification. The specification is built such that a superstep exists (from a specific initial configuration) if and only if the formula is satisfiable.

The system consists of $2n+1$ objects, named x_1T, x_1F, x_2T, $x_2F, \ldots$, x_nT, x_nF and O. The truth assignment to $x_1, \ldots x_n$ is represented similarly to that in the proof Theorem 1, i.e., x_i is assigned TRUE if x_iT sends `m` to x_iF and then x_iF sends `m` to x_iT, and is assigned FALSE if the same two messages are sent in reverse order. The object O can send the message `Start` to itself, which is the event that will trigger the superstep.

The specification contains $2n+m$ LSCs, of two types, as follows.

- Type 1 charts: these will be in charge of assigning values to the variables:
  - Type 1A charts: n charts, where chart i states that whenever O sends `Start` to itself, then x_iT must send `m` to x_iF.
  - Type 1B charts: similar to Type 1A charts, except that in the main chart x_iF sends `m` to x_iT.
- Type 2 charts: These are similar to the Type 3 charts in the earlier construction, and make sure that each clause is satisfied by the assignment. Given a clause C, let X_C^T be the literals appearing positive in C, and X_c^F be those appearing negative. The LSC corresponding to C is an anti-scenario, forbidding the scenario in which $\forall x \in X_C^T$ the events $x_iF \xrightarrow{\mathtt{m}} x_iT; x_iT \xrightarrow{\mathtt{m}} x_iF$ occur and $\forall x \in X_C^F$ the events $x_iT \xrightarrow{\mathtt{m}} x_iF; x_iF \xrightarrow{\mathtt{m}} x_iT$ occur.

The specification is clearly of size polynomial in that of the input. Moreover, since there is no LSC for which a minimal event appears again in the chart, there will never be any multiple running copies of the same chart.

It is left to show that following an event $O \xrightarrow{\texttt{Start}} O$, a superstep exists if and only if the formula is satisfiable.

- Assume a superstep exists. Clearly, from the Type 1 charts, the superstep contains each object x_iT sending one `m` message to x_iF and vice versa (in some order between the two). Since no Type 2 charts are violated, it follows that for each clause C, either $\exists x \in X_C^T$ s.t. $x_iT \xrightarrow{\texttt{m}} x_iF$ was sent first, or $\exists x \in X_C^F$ s.t. $x_iF \xrightarrow{\texttt{m}} x_iT$ was sent first. If we assign TRUE to x_i if and only if $x_iT \xrightarrow{\texttt{m}} x_iF$ was sent first, it follows that for this assignment, each clause is satisfied, and therefore ψ is satisfied.
- Assume an assignment to $x_1, \ldots, x_n$ exists s.t. ψ is satisfied. Consider a superstep in which $\forall x_i$ assigned TRUE, the events $x_iT \xrightarrow{\texttt{m}} x_iF; x_iF \xrightarrow{\texttt{m}} x_iT$ are sent, and $\forall x_i$ assigned FALSE, the events $x_iF \xrightarrow{\texttt{m}} x_iT; x_iT \xrightarrow{\texttt{m}} x_iF$ are sent. Similarly to the above argument, the superstep satisfies all Type 1 charts, and does not satisfy the prechart of any anti-scenario in the Type 2 charts, and is therefore a legal superstep for the specification.

∎

5 Discussion

5.1 Advanced constructs

LSCs is a rich language, which in addition to messages and synchronization constructs that we have used in our proofs, contains also conditions, object properties, assignments, messages with parameters, and control flow constructs like if-then-else and loops. The original smart play-out indeed handles these more advanced features of the language (see [7]). Thus, it is of interest to understand whether allowing the use of these constructs increases the complexity of smart play-out.

The addition of bounded loops does not affect smart play-out complexity, as such loops can be unravelled to represent all the iterations explicitly. However, the effect of using condition evaluation and assignments over object properties and variables depends on their domain and on the operators allowed. Thus, these may add no complexity but might render smart play-out undecidable in the case of infinite domains.

We note that unbounded loop constructs (i.e., star) may introduce additional complexity, as they allow the user to add another level of non-determinism to the LSC program: one may create a specification where the next cut is not uniquely determined by the current cut and the next event. It is of interest to check how these special cases may affect the complexity of smart play-out.

5.2 Upper bound

To date, there are two implementations of smart play-out: one based on model-checking [7] and one based on AI-style planning (termed planned play-out) [9]. However, neither support multiple running copies (see 4.2). Thus, although both use a reduction into known PSPACE problems, they cannot be used as a proof for an upper bound for the complexity of the general problem. A tight upper bound for the problem is yet unknown.

5.3 Superstep length

We note that the length of the superstep may be exponential in the size of the specification. For example, in the specification used in the proof of section 4.1, a legal superstep will traverse all possible assignments to the y_i's, and will therefore be of exponential length.

Interestingly, this observation still holds even when multiple running copies are not allowed. For example, consider a one-object system, with a specification consisting of k LSCs, $S_k = \{L_1, L_2, \ldots, L_k\}$, over an alphabet $\Sigma_k = \{m_1, m_2, \ldots, m_{k+1}\}$, such that for all $1 \leq i \leq k$, L_i states that whenever m_i is sent, eventually m_{i+1} needs to be sent twice. Following m_1, the only legal superstep for S_k is of length exponential in k (for example, for S_2, the superstep is $m_1, m_2, m_3, m_3, m_2, m_3, m_3$).

Thus, one may consider the problem of *bounded smart play-out*: given a specification, an initial configuration, and an integer k, return a superstep of length up to k if and only if one exists. This problem may be of interest in practice. We leave its analysis to future work.

5.4 Accelerations

Although the worst case complexity of smart play-out is PSPACE-hard, in practice we expect LSC specifications to be less complex. The main purpose of smart play-out is to resolve dependencies between LSCs. Though there may be many such dependencies in the worst case, our experience shows that in many interesting cases they are limited. This fact gives rise to the hope that appropriate heuristics could render the problem feasible for many practical specifications.

In other work in our group [6], and inspired by standard compiler and model checking optimization techniques, we suggest an algorithm that uses various acceleration techniques for smart play-out. The accelerations are based on approximating the set of LSCs that may participate in the current superstep, and on separating the elements that may cause dependencies between the LSCs in the specification (in the context of a given configuration) from the elements that may not do so. While the former require smart play-out, the latter can be handled efficiently in a more naïve fashion. All this is aimed at reducing the size of the model without affecting the soundness and completeness of finding a correct superstep. Clearly, such accelerations are heuristic in nature, and do not reduce the complexity of the problem in the worst case.

5.5 Is smart play-out good enough?

Smart play-out addresses the limitations of naïve play-out and finds a legal superstep if one exists. However, looking only one superstep (or a finite number of supersteps) ahead is still quite limited. Intuitively, if the LSC specification is "too deep", smart play-out may not be able to distinguish between a superstep that allows the system to continue playing (forever) from one that allows the environment to eventually force the execution into a violation.

Indeed, in [4], it is shown that smart play-out, however often repeated, is strictly weaker than full synthesis from LSCs, as was defined in [5]. On the other hand, [4] also shows that for a given LSC specification, there exists a k such that smart play-out that looks k supersteps ahead is as good as full synthesis.

6 Related and Future Work

Bontemps and Schobbens [1] examine the complexity of various problems related to LSCs, including reachability, language inclusion, and variants of synthesis. Specifically, they use a reduction from the halting problem of a PSPACE-bounded Turing machine to show that for LSC, reachability, inclusion and satisfiability are PSPACE-hard. Finally, the authors of [1] claim that their results can be adapted to show that determining whether a finite superstep exists is PSPACE-complete, but an explicit proof is not given. In contrast, our proof uses a reduction from QBF and is explicit.

A number of questions related to the complexity of smart play-out remain open. As mentioned earlier, these include explicit investigation of additional langauge constructs, the question of whether allowing multiple copies really affects the difficulty of the problem, and establishing a tight upper bound for the general problem. In addition, following [9], we consider the complexity of finding all supersteps (from a given configuration) to be an interesting question.

References

[1] Y. Bontemps and P. Y. Schobbens. The complexity of live sequence charts. In V. Sassone, editor, *Proc. 8th Int. Conf. Foundations of Software Science and Computational Structures (FoSSaCS'05)*, volume 3441 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.

[2] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 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.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[4] D. Harel, A. Kantor, and S. Maoz. On the Power of Play-Out for Scenario-Based Programs. In D. Dams, U. Hannemann, and M. Steffen, editors, *Concurrency, Compositionality, and Correctness, Festschrift in Honor of Willem-Paul de Roever.* Springer, 2009. To appear.

[5] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. of Foundations of Computer Science*, 13(1):5–51, February 2002. (Also in *Proc. 5th Int. Conf. on Implementation and Application of Automata* (CIAA 2000), Springer-Verlag, pp. 1–33. Preliminary version appeared as technical report MCS99-20, Weizmann Institute of Science, 1999.).

[6] D. Harel, H. Kugler, S. Maoz, and I. Segall. Accelerating Smart Play-Out of Scenario-Based Specifications. *Submitted*, 2009.

[7] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In M. Aagaard and J. W. O'Leary, editors, *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD '02)*, pages 378–398. Springer-Verlag, 2002.

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

[9] D. Harel and I. Segall. Planned and traversable play-out: A flexible method for executing scenario-based programs. In O. Grumberg and M. Huth, editors, *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *Lecture Notes in Computer Science*, pages 485–499. Springer, 2007.

[10] ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.

[11] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In N. Halbwachs and L. D. Zuck, editors, *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 445–460. Springer, 2005.

A Completion of Proofs

In proving Theorem 1 we omitted the formal proof that a superstep exists in the specification, following an external $y_0T \xrightarrow{\texttt{Choose\_Next}} y_0T$ event, if and only if φ is true. This is proved in the following series of claims.

Claim 1 *Consider a specification consisting only of the Type 1 charts, where Type 1C charts are replaced by Type 1C′ charts. Any legal execution of it, triggered by an external event $y_0T \xrightarrow{\mathit{Choose\_Next}} y_0T$, can be described by the following backtracking pseudo-code:*

```
1: procedure Choose(i)
2:     if $i = n + 1$ then
3:         for all $v_j \in \{x_j, y_j : j = 1, \ldots, n\}$ do
4:             $v_jT \xrightarrow{\texttt{Check}} v_jT$
5:             $v_jT \xrightarrow{\texttt{Done\_Check}} v_jT$
6:         end for
7:         $y_nT \xrightarrow{\texttt{Done}} y_nT$
8:         Return
9:     end if
10:    nondeterministic switch
11:        case
12:            $x_iT \xrightarrow{\texttt{m}} x_iF$
13:            $x_iF \xrightarrow{\texttt{m}} x_iT$
14:        case
15:            $x_iF \xrightarrow{\texttt{m}} x_iT$
16:            $x_iT \xrightarrow{\texttt{m}} x_iF$
17:
18:    end switch
19:    $y_iT \xrightarrow{\texttt{m}} y_iF$
20:    $y_iF \xrightarrow{\texttt{m}} y_iT$
21:    Async call to Choose(i+1)
22:    Wait until $y_iT \xrightarrow{\texttt{Done}} y_iT$ is sent
23:    $y_iF \xrightarrow{\texttt{m}} y_iT$
24:    $y_iT \xrightarrow{\texttt{m}} y_iF$
25:    Async call to Choose(i+1)
26:    Wait until $y_iT \xrightarrow{\texttt{Done}} y_iT$ is sent
27:    $y_{i-1}T \xrightarrow{\texttt{Done}} y_{i-1}T$
28: end procedure
```

Proof. For each $i = 1, \ldots, n$, the four LSC instantiations of Type 1A, Type 1B, Type 1C′, and Type 1D charts implement the `Choose` procedure with parameter i, as follows.

The Type 1A and Type 1B charts begin the execution. Lines 10–21 are the exact description of their main charts (where the nondeterministic choice comes from choosing which one progresses first), according to LSC's operational semantics.

Whenever a Type 1A and Type 1B chart pair completes (note that this always happens simultaneously), the cut of the corresponding Type 1C′ chart is right before the last prechart message ($y_iT \xrightarrow{\texttt{Done}} y_iT$). Note that this state will not be violated, since neither of the corresponding Type 1A, Type 1B or Type 1C′ charts can be activate at that time. Once this message ($y_iT \xrightarrow{\texttt{Done}} y_iT$) is sent, the main chart will be activated. This corresponds to line 22 in the pseudocode. The main chart clearly implements lines 23 - 25.

Similarly, the Type 1D chart will be activated by the next $y_iT \xrightarrow{\texttt{Done}} y_iT$ message (thus implementing line 26), and its main chart implements line 27.

The Type 1E chart clearly implements the halting condition in lines 2 - 9. (Note that the `for all` command in Line 3 is interpreted as all j's executed in any order, not necessarily sequential, and with any possible interleaving between the messages corresponding to different values of j.)

∎

Claim 2 *Consider a specification consisting of all Type 1 and Type 2 charts (with the original Type 1C charts, not Type 1C′ as used in Claim 1). Any legal execution of it, triggered by an external* $y_0T \xrightarrow{\textit{Choose\_Next}} y_0T$ *event, can be described by the following pseudocode (changes from claim 1 underlined see line 5):*

```
1: procedure CHOOSE(i)
2:     if $i = n + 1$ then
3:         for all $v_j \in \{x_j, y_j : j = 1, \ldots, n\}$ do
4:             $v_jT \xrightarrow{\texttt{Check}} v_jT$
5:             $\underline{\text{Repeat the last message } \texttt{m} \text{ sent back and forth between}}$
               $\underline{v_jT \text{ and } v_jF \text{ (or vice versa)}}$
6:             $v_jT \xrightarrow{\texttt{Done\_Check}} v_jT$
7:         end for
8:         $y_nT \xrightarrow{\texttt{Done}} y_nT$
9:         Return
10:    end if
11:    nondeterministic switch
12:        case
```

```
13:             $x_iT \xrightarrow{\mathtt{m}} x_iF$
14:             $x_iF \xrightarrow{\mathtt{m}} x_iT$
15:         **case**
16:             $x_iF \xrightarrow{\mathtt{m}} x_iT$
17:             $x_iT \xrightarrow{\mathtt{m}} x_iF$
18:
19:     **end switch**
20:     $y_iT \xrightarrow{\mathtt{m}} y_iF$
21:     $y_iF \xrightarrow{\mathtt{m}} y_iT$
22:     Async call to Choose(i+1)
23:     Wait until $y_iT \xrightarrow{\mathtt{Done}} y_iT$ is sent
24:     $y_iF \xrightarrow{\mathtt{m}} y_iT$
25:     $y_iT \xrightarrow{\mathtt{m}} y_iF$
26:     Async call to Choose(i+1)
27:     Wait until $y_iT \xrightarrow{\mathtt{Done}} y_iT$ is sent
28:     $y_{i-1}T \xrightarrow{\mathtt{Done}} y_{i-1}T$
29: **end procedure**
```

Proof. This proof has two parts. Fact 1: Augmenting the model by the Type 2 charts, while replacing Type 1C′ charts with Type 1C charts, does not affect the backtracking part of the pseudo-code. Fact 2: The Type 2 LSCs implement line 5. We start by proving the second of these.

Fact 2: Assuming Fact 1 holds, whenever the pseudo-code reaches line 4 (in which the message $v_jT \xrightarrow{\mathtt{Check}} v_jT$ is sent), for each variable v_j, exactly one of the corresponding Type 2 charts has its cut right before the $v_jT \xrightarrow{\mathtt{Check}} v_jT$ message in the prechart, and no main chart of Type 2 charts is active. This is true, since for each v_j, both $v_jT \xrightarrow{\mathtt{m}} v_jF$ and $v_jF \xrightarrow{\mathtt{m}} v_jT$ were necessarily sent. If $v_jT \xrightarrow{\mathtt{Done\_Check}} v_jT$ was sent afterwards, then either the Type 2C or Type 2D chart has a cut in that location, otherwise this holds for the Type 2A or Type 2B chart.

The $v_jT \xrightarrow{\mathtt{Check}} v_jT$ message in line 4, therefore, necessarily activates exactly one of the Type 2 charts for each variable v_j, which corresponds to the correct order in which the last $v_jT \xrightarrow{\mathtt{m}} v_jF$ and $v_jF \xrightarrow{\mathtt{m}} v_jT$ messages were sent. This will cause them to be sent again. Also note that the message $v_jT \xrightarrow{\mathtt{Done\_Check}} v_jT$ appears now in two active main charts — the Type 1E chart, and the active Type 2 chart. According to LSC semantics, this message may not be sent until enabled in both. This causes the Type 1E chart to block until all the $v_jT \xrightarrow{\texttt{m}} v_jF$ and $v_jF \xrightarrow{\texttt{m}} v_jT$ messages are resent, therefore postponing the execution of line 6 until after line 5 completes.

Fact 1: First introduce the Type 1C charts and Type 2 charts into the specification from Claim 1 (without removing the Type 1C$'$ ones yet). Clearly, Type 2 LSCs are activated by the $v_jT \xrightarrow{\texttt{Check}} v_jT$ message, which can be sent only by the Type 1E chart. Whenever this chart is active, no other Type 1 charts are active. Moreover, for each $i = 1, \ldots, n$, either the corresponding Type 1C$'$ chart, or the Type 1D chart has a cut right before the last prechart message (the Type 1C precharts were violated by the $y_iT \xrightarrow{\texttt{Choose\_Next}} y_iT$ message). Now, whenever a Type 2 chart resends the last $v_jT \xrightarrow{\texttt{m}} v_jF$ and $v_jF \xrightarrow{\texttt{m}} v_jT$ messages, this prechart will be violated, but another copy of it will start. For Type 1D charts, it will reach the exact same location (right before the last prechart message). Both Type 1C and Type 1C$'$ precharts will open, both with a cut following the two `m` messages. Therefore, when the $v_jT \xrightarrow{\texttt{Done\_Check}} v_jT$ message is sent (and the Type 2 charts all finish), the backtracking algorithm continues from the exact same state as in Claim 1: Type 1C and Type 1D charts are activated (where the main chart of Type 1C is identical to that of Type 1C$'$), and Type 1C$'$ precharts are violated and closed.

Since Type 1C$'$ charts never become active in this run, we can now remove them from the specification without changing the execution. ∎

Claim 3 *A legal superstep in the specification defined above, triggered by an external $y_0T \xrightarrow{\textit{Choose\_Next}} y_0T$ event, exists if and only if φ is true.*

Proof. First note that adding the Type 3 charts to the specification does not change the result of the above claim (as long as no Type 3 chart causes a violation), since Type 3 charts are only anti-scenarios, and not LSCs with "real" main charts that can drive the execution. Also note that for Type 1 and Type 2 charts alone, any choice from among the non-deterministic possibilities reflects a legal superstep. Therefore, a superstep exists for the entire specification if and only if there exist choices for each of the non-deterministic possibilities such that Type 3 charts are never violated.

The minimal messages in the precharts of Type 3 charts are `Check` messages from the relevant vT objects to themselves. These are sent in the halting condition of the pseudo-code above (line 4). Therefore, these precharts will become active whenever a halting condition is reached. For each object vT, following the `Check` message, the corresponding vT and vF objects resend their last exchange of the message `m`.

Assume a superstep exists. Consider, for each variable v, the exchange of messages `m` replayed by the Type 2 charts (i.e., in line 5). Let $v =$ TRUE if the order is $vT \xrightarrow{\mathtt{m}} vF; vF \xrightarrow{\mathtt{m}} vT$, and $v =$ FALSE otherwise. Now consider the set of all values that all variables take in all the executions of line 5. It is clear that a Type 3 anti-scenario holds (i.e., it violates the specification) if and only if none of the corresponding literals is satisfied. Since a superstep exists for each such set of values, no Type 3 anti-scenario holds; i.e., for each clause c_i, at least one of its literals is satisfied, and therefore the whole CNF formula ψ holds. From the flow of the backtracking algorithm, it is clear that it finds a value for each $\exists$ term and checks all $\forall$ terms, such that ψ holds, and therefore φ is true.

Now assume a superstep does not exist, and by contradiction that φ is true. Thus, there exist decisions for the backtracking algorithm such that whenever the halting condition is reached, ψ holds. The algorithm, along with these decisions, induce a legal superstep, in contradiction to the assumption.

∎