On the Power of Play-Out for Scenario-Based Programs [⋆]

(preliminary version)

David Harel, Amir Kantor, and Shahar Maoz

The Weizmann Institute of Science, Rehovot, Israel
{dharel,amir.kantor,shahar.maoz}@weizmann.ac.il

Abstract. We investigate the power of play-out, the execution mechanism associated with scenario-based programming, which was defined as the operational semantics of live sequence charts (LSC). We compare some of the play-out strategies and mechanisms suggested in the literature, and discuss their strengths and limitations. Specifically, we define a simple infinite hierarchy of LSC programs, and use it to show that smart play-out, the lookahead version of play-out guided by model-checking, is strictly weaker than full synthesis from LSC.

This paper is dedicated to Prof. Willem de Roever, friend and colleague, with admiration and respect for his scientific achievements, his impact on the community, and his boundless energy.

1 Introduction

Live Sequence Charts (LSC) [3] is a visual formalism for inter-object scenario-based specification, and is especially useful for programming reactive systems. The language extends classical Message Sequence Charts (MSC) [14], mainly by being multi-modal, i.e., incorporating universal and existential modalities. Thus, LSC distinguishes between behaviors that may happen in the system (existential, cold) and those that must happen (universal, hot). It can also naturally express a variety of constraints, such as behaviors that are forbidden.

Most importantly in the context of this paper, an executable (operational) semantics for LSC, termed *play-out*, was defined in [11]. Thus, LSC can be viewed not only as a specification language but also as a high-level programming language for reactive systems. The play-out idea can be applied to any scenario-based formalism that concentrates on inter-object behavior with multi-modal constraints; thus, in principle it can be used also to execute certain variants of temporal logic.

[⋆] 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.

In this paper we describe some of the play-out strategies and mechanisms suggested since the publication of [3], and discuss their strengths and limitations. These include the original *(naïve) play-out* of [11] and two stronger strategies, *smart play-out* [7, 11] and *planned play-out* [13]. We are particularly interested here in smart play-out, due to its novel nature, whereby a verification technique (specifically, model-checking) is used to run a program rather than to prove properties thereof. Thus, we define a simple infinite hierarchy of LSC programs, and use it to show that smart play-out is strictly weaker than full synthesis from LSC.

2 Naïve Play-Out of Scenario-Based Programs

Finding ways to construct executable systems based on inter-object, scenario-based specifications, appears to be an interesting challenge [4]. Many researchers have approached the issue as a synthesis problem; see, e.g., [1, 6, 15, 20], where inter-object specifications, given in variants of Message Sequence Charts (MSC) [14], are translated into intra-object state-based executable specifications for each of the participating objects or components.

Play-out, defined in [12, 11], is a recent example of a different approach. Instead of synthesizing intra-object state-based specifications for each of the components, the play-out algorithm executes the scenarios directly, keeping track of all user (or environment) events, as well as system events, for all objects or components simultaneously, and causing other events and actions to occur as dictated by the specified scenarios. No intra-object model for any of the participating components needs to be built in the process.

To date, two implementations of this basic play-out idea are available. The original one is part of the Play-Engine tool [11], which works as an LSC interpreter and drives the simulation of an application execution, provided it implements certain custom interfaces. A more recent implementation is part of the S2A compiler [5], which is based on a compilation scheme for translating LSCs into AspectJ [16]. This second implementation uses a UML2-compliant and slightly generalized variant of the LSC language, defined in [10].

Here now is a brief and simplified description of the basic play-out technique for LSC specifications consisting of universal charts, which is sufficient for our needs in this paper. The full LSC language of [11] is much richer than the simplified fragment we use here. It includes conditions, variables, structural constructs (e.g., if-then-else), symbolic instance lifelines, symbolic methods, time, etc. Play-out handles all of these. However, in this paper we concentrate on a basic version of the language.

We assume the reader is familiar with elementary LSC notions, such as the partial order of events defined by an LSC, an LSC cut, etc. A thorough treatment may be found in [11]. Roughly, a *run* of an LSC program may be seen as sequence of execution *configurations*, each representing a global *cut* (a tuple of all LSC current cuts). A cut induces a set of enabled and violating events; enabled events are the ones immediately after the current cut in the partial order defined by the chart; violating events are all events that appear in the chart but are not currently enabled (see [11]). Whenever a minimal event of a chart occurs, a live copy of the chart is created. We say that a live copy of a chart is *preactive* (*active*) if its cut is in its prechart (main chart). A configuration is *stable* if it includes no active live copy.

The play-out execution mechanism works in phases of a *step* followed by a valid *superstep*. A step is the execution of a single external event, performed by the environment or the user (both will be referred to here as the environment). A superstep is a finite (possibly empty) sequence of events executed by the system as a reaction to a single external event. A *valid* superstep handles iteratively all enabled main chart events in a non-violating way (a violation of a preactive LSC is allowed), in order to reach a stable configuration. More live copies may be created and activated during the execution of a superstep, but all copies must be completed or preactive when a superstep ends.

Note that, in general, LSC is a nondeterministic language: after an external event occurs, many possible supersteps may exist. There are two possible causes for this. One is the partial ordering of events, which can lead to more than one enabled event at a given cut. The second is the specification of pieces of behavior in more than one chart. Events that do not appear in a chart do not violate it, but when a number of charts are active, even if they are all totally ordered, there may be more than one valid choice for the next event. Note, however, that once an event is executed in a configuration, the next configuration is uniquely determined, though some steps may yield a configuration for which no valid superstep exists.

Different play-out algorithms differ in their approach to finding a valid superstep. If a valid superstep is not realized, we say the execution *violates* the specification. The *naïve play-out* strategy [11] executes supersteps in a greedy manner. It iteratively seeks and executes some enabled main chart event that does not immediately violate an active LSC, until no such events are available. When several such events are available, naïve play-out selects one arbitrarily.

Consider a specification that contains the single LSC of Figure 1. It is a universal LSC, with three instances, one of which is the environment.

If the environment sends message $\mathtt{m}_1$ to $\mathtt{obj}_1$, a live copy of LSC_A is created, the prechart completes and as the cut enters the main chart the LSC becomes active. The following superstep starts with two enabled events, $\mathtt{m}_2$ and $\mathtt{m}_3$, waiting to be executed in some order. Naïve play-out picks any one of these, say $\mathtt{m}_3$, executes it (in this case $\mathtt{obj}_2$ sends message $\mathtt{m}_3$ to itself) and advances the cut. Self messages may represent internal actions of an object. The superstep is still not over, as $\mathtt{m}_2$ has yet to be sent. Being the only enabled event, $\mathtt{m}_2$ is promptly executed. Now $\mathtt{m}_4$ is enabled, so it is sent, and the chart is completed. The configuration reached at this point is stable, since there are no active live copies, and the superstep ends.

In this example, the specification contains only one LSC. Scenario-based programs typically include many LSCs acting together, as we shall see in the next section.

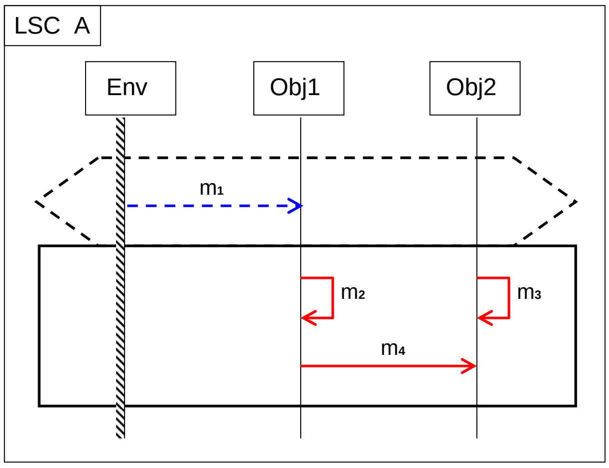

Fig. 1. LSC$_A$

3 More Powerful Play-Out Strategies

The original play-out [11] process is indeed naïve. Some of the sequences of events possible as a response to an external (user or environment) event may eventually lead to violations, and these cannot be avoided by the non-backtracking, non-looking-ahead, play-out process. Moreover, the partial order semantics among events in each chart and the ability to separate scenarios into different charts without having to say explicitly how they are to be composed are very useful in early requirement stages, but can cause under-specification and nondeterminism when one attempts to execute them.

Consider a specification consisting of LSC$_A$ from Figure 1 and LSC$_B$ from Figure 2. We first demonstrate how naïve play-out may fail to find a valid superstep when executing this specification.

Again, assume that the environment sends message $\mathtt{m}_1$ to $\mathtt{obj}_1$. This causes activation of a live copy of LSC$_A$, and at the beginning of the following superstep events $\mathtt{m}_2$ and $\mathtt{m}_3$ are enabled. Naïve play-out again may choose to execute $\mathtt{m}_3$, but in the presence of LSC$_B$ this leads to a violation of the specification, as follows: After executing $\mathtt{m}_3$ and advancing the cut of LSC$_A$, $\mathtt{m}_2$ is still enabled, so it is now executed. After the execution of $\mathtt{m}_2$, being the minimal event of LSC$_B$, a live copy of LSC$_B$ is created and activated. The enabled events are currently $\mathtt{m}_4$ of LSC$_A$ and $\mathtt{m}_3$ of LSC$_B$. Note however, that $\mathtt{m}_4$ violates LSC$_B$ and $\mathtt{m}_3$ violated LSC$_A$, so the execution of superstep is in deadlock, and a valid superstep cannot be realized.

After the environment performed $\mathtt{m}_1$, the play-out algorithm could have chosen to execute $\mathtt{m}_2$ before $\mathtt{m}_3$. This, however, would have conformed with the order induced by LSC$_B$, yielding a valid superstep, consisting of $\mathtt{m}_2$, $\mathtt{m}_3$ and $\mathtt{m}_4$ (in

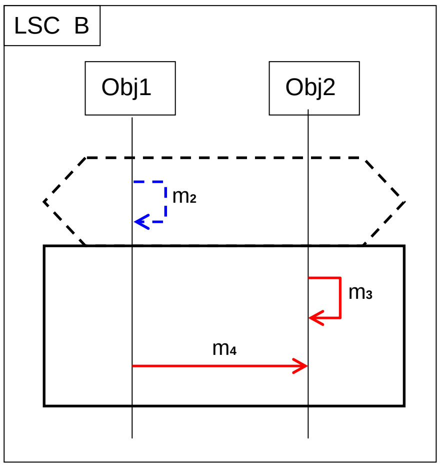

Fig. 2. LSC$_B$

that order), after which both LSCs are completed. This rather simple example demonstrates the need for a 'smarter' play-out strategy.

3.1 Smart play-out

The strategy of *smart play-out* (SPO) [7, 11] partially addresses these limitations, by using model-checking techniques to avoid violations within a superstep.

The approach taken is to formulate the play-out task as a verification problem, and to use a counterexample provided by a model-checking algorithm as the desired superstep. The system on which we perform model-checking is constructed from 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 execution just after the last external event occurred, including the current values of object properties, information about 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 complete successfully, and by the definition of the transition relation no violations occurred. Such a counter-example is exactly the desired superstep, and it is then fed into the Play-Engine for execution. If the model-checker verifies the property then no correct superstep exists and execution terminates in failure. Smart play-out is sound and complete for a single superstep.

In [8] smart play-out was extended to support LSC time constructs and forbidden elements.

3.2 Planned play-out

Often it is useful to discover more than a single valid superstep, but model checkers are usually unable to provide more than one counter-example. In [13], a variant of smart play-out was proposed, and termed *planned play-out.* Planned play-out uses AI planning algorithms to find many valid supersteps in a single run.

Technically, the problem of finding a valid superstep is translated into a planning problem, and a planner is employed in order to solve it. The resulting plans are then translated back into supersteps.

An interesting feature of this approach, and the way it was implemented in the Play-Engine, is that it supports interactive play-out: taking advantage of the interpreter/simulation nature of the Play-Engine, the user is allowed to backtrack during execution and to choose between possible steps in the quest for an acceptable superstep. This interactive, user-guided process, was called *traversable play-out* in [13].

4 Smart Play-Out Is Not Enough

We now show that smart play-out is strictly weaker than full synthesis from LSC, as defined in [6].

A superstep from a given configuration is *k-valid* (for $k \geq 1$) if the environment may not force the execution into a violation in the following $k-1$ play-out iterations. A superstep is thus 1-valid iff it is valid. A superstep is 2-valid iff it is 1-valid and from the configuration reached, any external event performed by the environment leads to a configuration from which another valid superstep exists.

A superstep from a given configuration is *ω-valid* iff from the configuration reached, the environment cannot force the execution into a violation at all. Note that if during the execution, a superstep that is not ω-valid is taken, the environment may eventually force the execution into a violation.

For any $k \geq 1$ we define SPO^k to be a generalization of smart play-out that is intended to find a k-valid superstep, if one exists. This may be realized by looking k supersteps ahead, and checking all possible events carried out by the environment between them. SPO^1 is the original smart play-out, which looks one superstep ahead.

Intuitively, looking a finite number of supersteps ahead is myopic. If the specification is 'too deep', smart play-out, however often repeated, may not be able to distinguish between a choice that will allow the specification to continue playing (an ω-valid superstep), and a choice that will allow the environment to force the execution into a violation of the specification. In LSC synthesis, however, the synthesized controller allows only ω-valid supersteps [6].

We now construct a sequence of LSC specifications. For any $k \geq 1$, $\text{Spec}_k = \{\text{LSC}_1, \text{LSC}_2, \ldots, \text{LSC}_{k+3}\}$ is a set of universal LSCs, presented in Figures 3 and 4. In each chart there are two instances: the system and the environment.

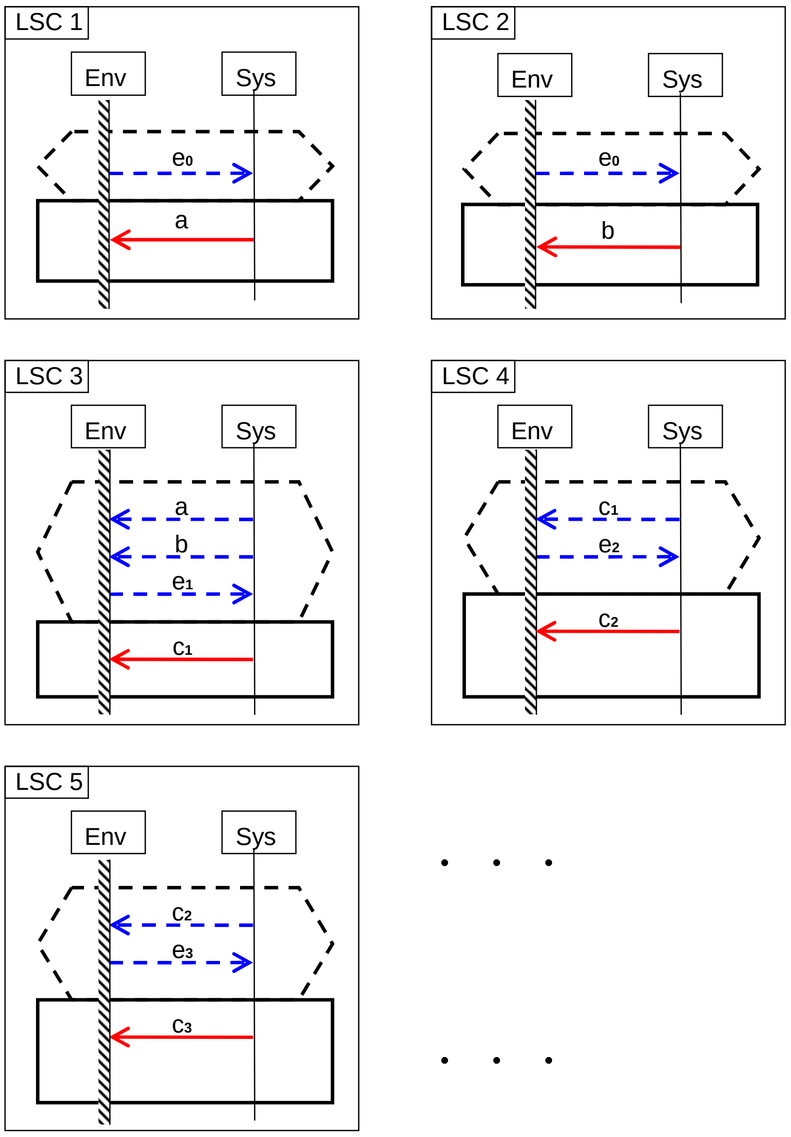

Fig. 3. $\text{Spec}_k = \{\text{LSC}_1, \text{LSC}_2, \ldots, \text{LSC}_{k+3}\}$

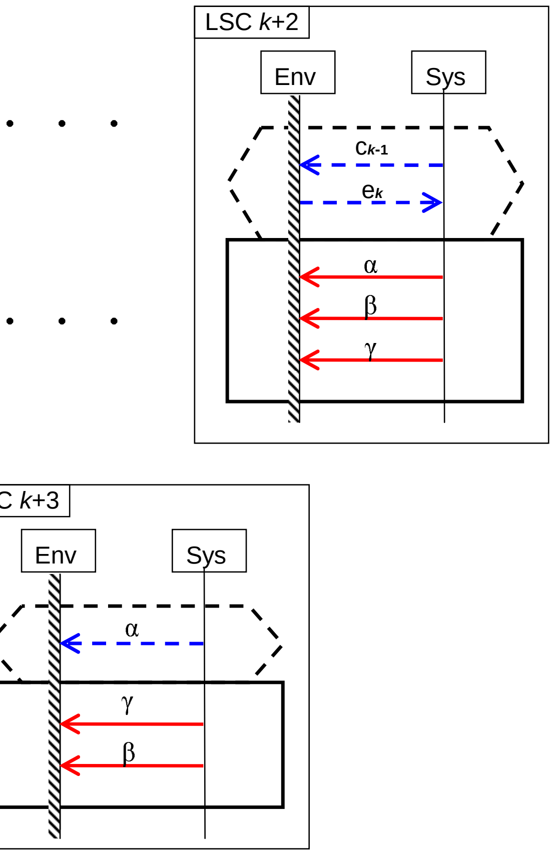

Fig. 4. Spec$_k$ — continued

All the charts are linearly ordered, therefore we may describe Spec$_k$ more concisely by:

$$\begin{array}{lr}
\text{LSC}_1\text{:} & e_0 \Rightarrow a \\
\text{LSC}_2\text{:} & e_0 \Rightarrow b \\
\text{LSC}_3\text{:} & a,\, b,\, e_1 \Rightarrow c_1 \\
\text{LSC}_4\text{:} & c_1,\, e_2 \Rightarrow c_2 \\
\text{LSC}_5\text{:} & c_2,\, e_3 \Rightarrow c_3 \\
\vdots & \\
\text{LSC}_{k+2}\text{:} & c_{k-1},\, e_k \Rightarrow \alpha,\, \beta,\, \gamma \\
\text{LSC}_{k+3}\text{:} & \alpha \Rightarrow \gamma,\, \beta
\end{array}$$

In this notation, each row represents an LSC. Time advances from left to right, and '⇒' separates the prechart from the main chart. $E_{env} = \{e_0, e_1, \ldots, e_k\}$ is the set of external events controlled by the environ-

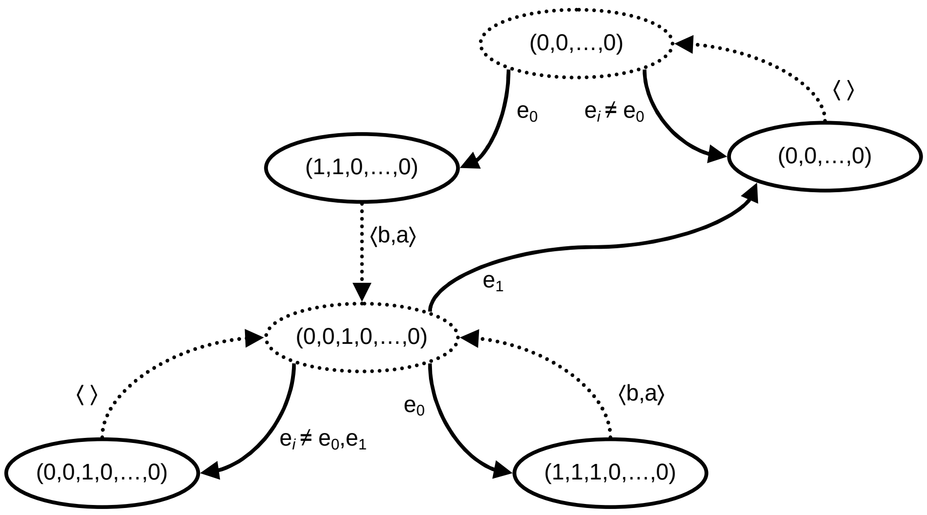

Fig. 5. Starting from the initial configuration, the environment may not force the execution into a violation.

ment. $E_{sys} = \{a, b, c_1, \ldots, c_{k-1}, \alpha, \beta, \gamma\}$ is the set of all other events controlled by the system.

Theorem 1. *SPO^k is not enough to determine a non-violating execution of Spec$_k$ ($k \geq 1$). That is, SPO^k cannot decide for two possible supersteps which one is ω-valid and which is not.*

Proof sketch Consider Spec$_k$. Note that in each chart the events are linearly ordered. Moreover, the minimal event does not reoccur in the chart, so at any point during execution there exists at most one live copy of each chart. We denote the global state of execution as a configuration $(l_1, l_2, \ldots, l_{k+3})$, where each l_i is a natural number representing the current cut of the i^{th} chart, starting from zero. The initial configuration is $(0, \ldots, 0)$, in which all charts are not active.

Claim 1 *Starting from the initial configuration, the environment may not force the execution into violation; i.e., for any future action of the environment there exists a valid superstep as a reaction, ad infinitum.*

Figure 5 presents an execution graph for Spec$_k$, starting from the initial configuration. The nodes represent configurations, edges represent environment controlled events or system controlled supersteps. For each action of the environment (an environment controlled event from E_{env}), the graph shows a possible superstep as a reaction. When formulated as a game between the system and the environment, this may be seen as describing a winning strategy for the system (which in fact does not depend on k).

Assume now that the environment starts by performing e_0 ('step zero'). LSC_1 and LSC_2 are activated and their precharts are completed, yielding the configuration $(1, 1, 0, \ldots, 0)$, so a and b are enabled and must be carried out (in any order) in the following superstep ('superstep zero').

First, consider the possibility of executing b and a, in this order (denoted $\langle b, a\rangle$) in superstep zero. As shown in Figure 5, $\langle b, a\rangle$ is a valid superstep (yielding the configuration $(0, 0, 1, 0, \ldots, 0)$), where for any future action of the environment there exists a valid superstep as a reaction. Thus, it is ω-valid for superstep zero and, in particular, it is k-valid.

Second, consider the possibility of executing $\langle a, b\rangle$ in superstep zero.

Claim 2 *$\langle a, b\rangle$ is k-valid for superstep zero, but is not ω-valid.*

After e_0, $\langle a, b\rangle$ leads to configuration $(0, 0, 2, 0, \ldots, 0)$. To show that $\langle a, b\rangle$ is k-*valid* for superstep zero, we show the environment cannot force a violation in $(k - 1)$ steps starting from $(0, 0, 2, 0, \ldots, 0)$.

Observe that in Spec_k, in order for a violation to occur in any run starting from the initial configuration, LSC_{k+2} must proceed to its main chart. In the following we show that if a superstep begins with a configuration in which LSC_{k+2} is not active, a valid superstep exists.

During the execution of the superstep, LSC_{k+2} will not be activated, because its prechart ends with an external event. Thus, α will not be executed during the superstep, and a live copy of LSC_{k+3} will not be created. Any live copy created during the superstep has an external event in its prechart, so it may not become active. Therefore, the superstep must terminate. It is now sufficient to notice that as long as there is an enabled main chart event during the execution of the superstep, there exists one that does not violate any active LSC. Only LSC_1, $\ldots$, LSC_{k+1} may be active, so the event enabled in the active LSC of the largest index among these does not violate any active live copy.

LSC_{k+2} may proceed to its main chart only after the events e_k, c_{k-1}, e_{k-1}, c_{k-2}, $\ldots$, e_1 occur. Thus, at least k external events occurred after $\langle a, b\rangle$ was executed in superstep zero. Therefore, the environment may not violate the specification in $(k - 1)$ actions, and $\langle a, b\rangle$ is k-valid.

However, $\langle a, b\rangle$ is not ω-valid for superstep zero. If it is chosen, the environment may force the system into a violation of the specification (i.e., there is a winning strategy for the environment). This, of course, cannot take place in $(k-1)$ steps (as shown above), but k steps are enough. Figure 6 shows an execution graph starting from $(0, 0, 2, 0, \ldots, 0)$. For any valid superstep, the graph shows a possible action of the environment that eventually leads to a violation; that is, it shows how the environment can force the system into a violation in k steps.

In conclusion, starting from the initial configuration and following the external event e_0, SPO^k cannot choose between $\langle b, a\rangle$ and $\langle a, b\rangle$ for superstep zero, as both are k-valid supersteps. However, the former is ω-valid while the latter is not. $\square$

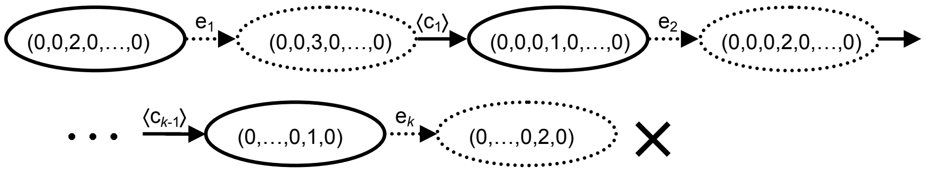

Fig. 6. If $\langle a, b\rangle$ is chosen for superstep zero, the environment may force the execution into a violation. In the last configuration presented, $(0, \ldots, 0, 2, 0)$, a valid superstep does not exist due to a contradiction between LSC_{k+2} and LSC_{k+3}.

We remark that the set of events of Spec_k depends upon the index k. As k grows, this alphabet of events grows. A stronger proof of Theorem 1 can be formulated, which employs a different sequence of specifications, $\widehat{\mathrm{Spec}}_k$, in which all specifications share the same set of events.

We use our concise notation to describe $\widehat{\mathrm{Spec}}_k = \{\mathrm{LSC}_1, \mathrm{LSC}_2, \ldots, \mathrm{LSC}_{k+3}\}$:

$$\begin{array}{rr}
\mathrm{LSC}_1: & e_0 \Rightarrow a \\
\mathrm{LSC}_2: & e_0 \Rightarrow b \\
\mathrm{LSC}_3: & a,\, b,\, e_1 \Rightarrow c \\
\mathrm{LSC}_4: & c,\, e_1 \Rightarrow c,\, c \\
\mathrm{LSC}_5: & c,\, c,\, e_1 \Rightarrow c,\, c,\, c \\
\vdots & \\
\mathrm{LSC}_{k+2}: & \overbrace{c,\, c,\, \ldots,\, c}^{k-1},\, e_1 \Rightarrow \alpha,\, \beta,\, \gamma \\
\mathrm{LSC}_{k+3}: & \alpha \Rightarrow \gamma,\, \beta
\end{array}$$

These can be easily transformed into LSCs. $\widehat{E}_{env} = \{e_0, e_1\}$ is the set of external events controlled by the environment, while $\widehat{E}_{sys} = \{a, b, c, \alpha, \beta, \gamma\}$ is the set of all other events controlled by the system.

The proof of Theorem 1 for the sequence $\widehat{\mathrm{Spec}}_k$ is similar to the proof for Spec_k. More specifically, proving that $\langle b, a\rangle$ is ω-valid for superstep zero but that $\langle a, b\rangle$ is not ω-valid, is roughly the same as in the proof of Theorem 1 above (although the proof of the second of these claims is slightly more complicated, because there are multiple live copies of an LSC). Showing that $\langle a, b\rangle$ is k-valid for superstep zero turns out to be somewhat more difficult, as it requires formulating a certain property of execution configurations, and proving that this property may be enforced by the system in the next $(k-1)$ iterations.

Now, although SPO^k does not determine a non-violating execution in all LSC specifications, we show that for a given specification $\mathcal{S}$ there is $\tilde{k}$ such that $\mathrm{SPO}^{\tilde{k}}$ is 'good enough' for $\mathcal{S}$.

Theorem 2. *Given an LSC specification $\mathcal{S}$, one may compute a number k such that SPO^k is enough to determine a non-violating execution of $\mathcal{S}$. That is, SPO^k, when executing $\mathcal{S}$, always returns an ω-valid superstep if one exists.*

Proof sketch Consider an LSC specification $\mathcal{S} = \{L_1, \ldots, L_n\}$. We define k to be larger than the number of execution configurations of $\mathcal{S}$, as follows.

Each LSC L_i has a bounded number of live copies during execution; e.g., there are no more live copies than the number of cuts in L_i (a much tighter bound will often exist). Let b_i and c_i be the number of cuts in L_i and an upper bound on the number of live copies of L_i, respectively. There are at most ${c_1}^{b_1} \cdot \ldots \cdot {c_n}^{b_n}$ execution configurations for $\mathcal{S}$. Let $k \triangleq {c_1}^{b_1} \cdot \ldots \cdot {c_n}^{b_n} + 1$.

Assume that after some external event, an ω-valid superstep exists. In particular, a k-valid superstep exists, so SPO^k returns a k-valid superstep. We show that any k-valid superstep is, in fact, ω-valid, as required.

Assume to the contrary, that the environment may force the execution into a violation. Let t be the smallest number of steps in which the environment may thus force a violation. Since there are at most $(k-1)$ configurations, if $t \geq k$, then there are two steps after which the execution is in the same configuration, thus contradicting the minimality of t. Consequently, $t < k$, and the superstep returned by SPO^k is not k-valid. □

5 Related and Future Work

There have been several efforts to generate executable programs from LSC specifications. Synthesis from LSC specifications was investigated in [6, 9]. Bontemps et al. [2] investigate different variants of synthesis from LSC. In [19], Wang *et al.* describe the synthesis of SystemVerilog programs from an LSC specification. In [17], Sun and Dong present a translation from an LSC specification to a CSP specification, for the purpose of synthesizing executable distributed processes. In [18], the same authors investigate also the generation of a distributed design from a combination of interaction-based and state-based modeling, given in LSC and Z.

In other work in our group (in progress), we suggest optimizations for smart play-out. These are based on an approximation of the set of LSCs that may participate in the current superstep, and on other methods, inspired by standard compiler optimizations. All this is aimed at reducing the size of the model without affecting the soundness and completeness of finding a correct superstep.

Finally, finding effective and efficient ways to compute a reasonably small k such that SPO^k is good enough for a given specification as a whole, or for certain possible execution configurations, might very well be of theoretical and practical interest.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In *Proc. 22nd Int. Conf. on Soft. Eng. (ICSE'00)*, pages 304–313, New York, NY, 2000. ACM Press.
2. Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of Open Reactive Systems from Scenario-Based Specifications. *Fundam. Inform.*, 62(2):139–169, 2004.

3. 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.
4. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *IEEE Computer*, 34(1):53–60, 2001. Also, Proc. Fundamental Approaches to Software Engineering (FASE), LNCS, Vol. 1783 (Tom Maibaum, ed.), Springer-Verlag.
5. D. Harel, A. Kleinbort, and S. Maoz. S2A: A Compiler for Multi-Modal UML Sequence Diagrams. In M. B. Dwyer and A. Lopes, editors, *Proc. Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *LNCS*, pages 121–124. Springer, 2007.
6. 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.).
7. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD '02)*, pages 378–398, London, UK, 2002. Springer-Verlag.
8. D. Harel, H. Kugler, and A. Pnueli. Smart Play-Out Extended: Time and Forbidden Elements. In *Proc. 4th Int. Conf. on Quality Software (QSIC'04)*, pages 2–10. IEEE Computer Society, 2004.
9. D. Harel, H. Kugler, and A. Pnueli. Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*, volume 3393 of *LNCS*, pages 309–324. Springer, 2005.
10. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, 2008. (Preliminary version appeared in Proc. 5th Int. Workshop on Scenarios and State-Machines (SCESM'06) at the 28th Int. Conf. on Soft. Eng. (ICSE'06), May, 2006. ACM Press, 13-19).
11. D. Harel and R. Marelly. *Come , Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
12. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach. *Software and Systems Modeling (SoSyM)*, 2(2):82–107, 2003.
13. 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 *LNCS*, pages 485–499. Springer, 2007.
14. ITU. International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report, 1996.
15. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In *Proc. Int. Workshop on Distributed and Parallel Embedded Systems (DIPES'98), IFIP WG10.3/WG10.5*, volume 155 of *IFIP Proc.*, pages 61–72. Kluwer, 1998.
16. S. Maoz and D. Harel. From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ. In M. Young and P. T. Devanbu, editors, *Proc. 14th ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE'06)*, pages 219–229, Portland, Oregon, November 2006. ACM.

17. J. Sun and J. S. Dong. Synthesis of Distributed Processes from Scenario-Based Specifications. In *Proc. Int. Symp. of Formal Methods (FM'05)*, volume 3582 of *LNCS*, pages 415–431. Springer, 2005.
18. J. Sun and J. S. Dong. Design Synthesis from Interaction and State-Based Specifications. *IEEE Transactions on Software Engineering*, 32(6):349–364, 2006.
19. H. H. Wang, S. Qin, J. Sun, and J. S. Dong. Realizing Live Sequence Charts in SystemVerilog. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE'07)*, pages 379–388, Washington, DC, USA, 2007. IEEE Computer Society.
20. J. Whittle, R. Kwan, and J. Saboo. From Scenarios to Code: an Air Traffic Control Case Study. *Software and Systems Modeling (SoSyM)*, 4(1):71–93, 2005.