Accelerating Smart Play-Out [⋆]

David Harel[1], Hillel Kugler[2], Shahar Maoz[1], and 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 specifications, which utilizes powerful computation methods to compute safe supersteps, thus helping to avoid violations that may be caused by naïve execution. Major challenges for smart play-out are performance and scalability. In this work we show how to accelerate smart play-out by adapting and applying ideas inspired by formal verification and compiler optimization. Specifically, we present an algorithm that can reduce the size of the specification considered for smart play-out, while maintaining soundness and completeness. Experimental results show significant performance improvements and thus open the way to the application of smart play-out to large scenario-based programs.

1 Introduction

Scenario-based modeling using various variants of sequence diagrams has attracted intensive research efforts in recent years (see, e.g., [1–5]). In this paper, we focus on the language of *live sequence charts* (LSC), which has been suggested in [3] as a highly expressive extension of message sequence charts (MSCs) [6]. LSC has been endowed with an operational semantics termed *play-out*, where a specification consisting of a set of charts is executed directly [4]. In the original play-out algorithm, non-determinism is solved ad-hoc without considering the future effects of its choices. To help alleviate this, *smart play-out* was proposed in [7], where formal reasoning (originally, model-checking) is used to compute safe execution paths. In [8] we prove smart play-out to be PSPACE-hard for the general case, and NP-hard if multiple copies of the same chart are not allowed.

In this paper we introduce an algorithm that accelerates smart play-out of scenario-based specifications. The algorithm exploits special syntactic and semantic properties of the language in order to reduce the size of the specification before smart play-out is computed. It consists of several steps, each aimed at identifying different types of constructs that may be temporarily removed from the specification without affecting correctness. First, entire charts are removed

[⋆] This research was partially supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

in a cone-of-influence-like iterative fixpoint algorithm [9], computing a safe approximation (an over approximation) of the set of charts that may influence the current computation. Second, constructs within the remaining LSCs are precomputed or eliminated using an approach inspired by compiler optimization methods, such as constant propagation and early evaluation. All these result in an overall smaller specification that is then used as the input for the smart play-out computation.

The algorithm takes advantage of the following typical features of LSC specifications. First, the breakdown of the specification into user-friendly intuitive scenarios often creates redundancies that can be removed without affecting the execution. Second, intentional under-specification can sometimes be abstracted away from the smart play-out mechanism and be left for the naïve implementation. Third, the execution paths we are looking for are typically rather short and local, involving only a subset of the specification, especially in large systems. Finally, the specification may be exponentially more succinct than the state space of the model it induces; we benefit from attacking the problem already at the level of the specification, before the input model for smart play-out is constructed.

Our work can be viewed as an adaptation and application of well-known program analysis and abstraction techniques from the domains of compiler optimization and formal verification to our specific need, which is the acceleration of smart execution of scenario-based specifications.

A technical report with additional details and proofs is available [10].

2 Preliminaries

LSC [3] is an extension of message sequence charts (MSC) [6]. Both contain vertical lines, termed *lifelines*, which denote objects, and *events*, which involve one or more of these objects. 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. More advanced constructs, like conditions, `if-then-else`, loops, etc., can also be expressed. A typical LSC consists of a prechart (denoted by a blue dashed hexagon), and a main chart (denoted by a solid frame). Roughly, the intended semantics is that whenever the prechart is satisfied in a run of the system, eventually the main chart must also be satisfied (see Fig. 1).

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 is not allowed to happen in an execution.

2.1 An example

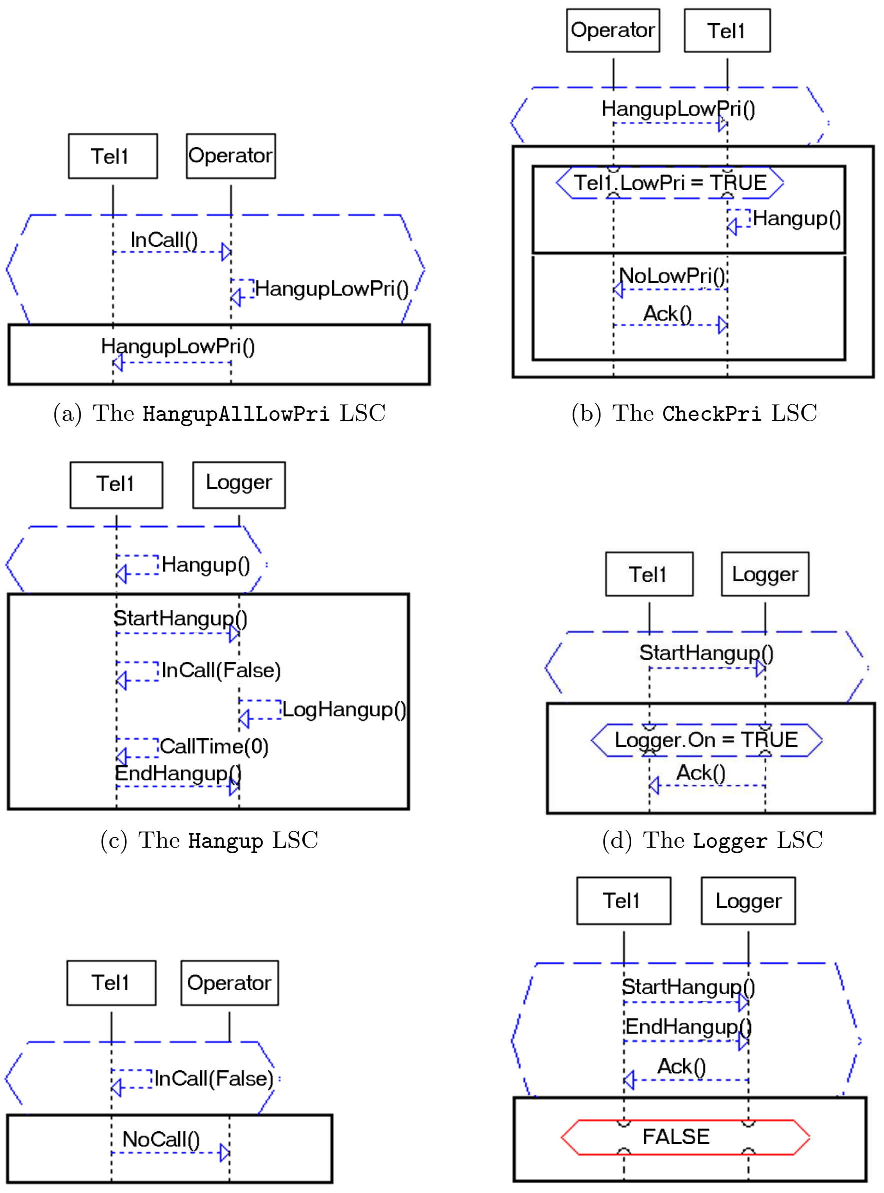
An example specification, consisting of six LSCs, is given in Fig. 1. It describes a simple telephone system with three objects – a phone, an operator, and a logger. The operator may wish to force low-priority calls to disconnect (e.g., due to a system overload). The LSC `HangupAllLowPri`, in Fig. 1(a), refers to the operator notifying the phone that a low-priority hangup is called for. Specifically, the LSC states that if the phone sends the message `InCall` to the operator, and the operator sends `HangupLowPri` to itself, then the operator sends `HangupLowPri` to the phone.

Fig. 1(b) shows LSC `CheckPri`, where the phone checks whether it is in a low-priority mode. The LSC uses an if-then-else construct, represented by two sequential boxes, with a condition at the beginning of the first box. It states that if the message `HangupLowPri` is sent from the operator to the phone (note that this is the same message sent in the main chart of the previous LSC; the process of deciding at runtime that the messages are the same is termed *unification*), then if the property *LowPri* of the telephone is true, it should send `hangup` to itself. Otherwise, it sends `NoLowPri` to the operator, and the operator replies with an `Ack`.

Fig. 1(c) shows LSC `Hangup`, which describes the hangup process. It states that whenever the phone sends `Hangup` to itself, it sends `StartHangup` to the logger, then sends `InCall(False)` to itself, and `CallTime(0)` to itself. The logger also sends `LogHangup` to itself. Finally, the phone sends `EndHangup` to the logger. Two features worth noting in this LSC are the following: (1) The partial order of an LSC is defined by the lifelines (from top to bottom) and the messages (and other multi-lifeline constructs) that synchronize between lifelines. Therefore, in this example, no explicit order is defined between the `LogHangup` message and the two messages `InCall(False)` and `CallTime(0)`, while these two must be executed in this order. The message `EndHangup`, appearing on both lifelines, is executed last. (2) Some messages change object properties. For example, `InCall(False)` changes the property *InCall* of the phone to be false. Similarly, `CallTime(0)` sets $Tel1.CallTime$ to be 0.

The LSC `Logger`, in Fig. 1(d), specifies how the logger replies to the telephone: If the phone sends `StartHangup` to the logger, the *Logger.On* condition is checked. If it is true, the execution continues, and the logger sends `Ack` to the phone. If the *Logger.On* is false, then being a cold condition, the chart exits gracefully and the `Ack` is not sent. The LSC `NotifyHangup`, in Fig. 1(e), specifies the notification to the operator that the hangup has completed. Finally, the LSC `CorrectLogging`, given in Fig. 1(f), is an anti-scenario; it states that the scenario in which the three messages, `StartHangup` from the phone to the logger, `EndHangup` from the phone to the logger, and `Ack` from the logger to the phone, are sent in this order is forbidden.

Throughout the paper, we consider a generalization of this example specification, for n phones, in which the six LSCs are replicated n times. In replica i, the object Tel1 is replaced by Tel$i$. Note that LSCs, in general, support symbolic instances (lifelines that represent entire classes rather than concrete objects), with



(a) The `HangupAllLowPri` LSC

(b) The `CheckPri` LSC

(c) The `Hangup` LSC

(d) The `Logger` LSC

(e) The `NotifyHangup` LSC

(f) The `CorrectLogging` LSC

Fig. 1. A 6-LSC specification for a simple phone system in which the operator may decide to hang up low priority calls.

which this replication could have been avoided [4]. However, since as of now none of the currently known smart play-out implementations supports symbolic instances, we avoid using them in this paper.

2.2 Play-out

An operational semantics and an execution technique termed *play-out* were defined for the LSC language in [4]. Play-out remembers at each point in time the set of *active* LSCs (those for which the prechart has already completed, but the main chart hasn't), and for each such LSC it holds the current *cut* (listing what has already happened, and what has not). At each step, the play-out mechanism chooses one message that is *enabled* in some LSC (i.e., it appears directly after the current cut), and does not violate any other chart (a message is violating if it appears in an active chart but is not enabled in it), and executes it.

The original play-out mechanism of [4] is naïve, in the sense that there is no look-ahead when selecting the action to be executed. Thus, non-determinism is solved ad-hoc without considering the long-term consequences of the choice. Two "smart" techniques have been suggested thus far to partly address this issue: (1) model-checking based play-out, termed *smart play-out* [7], and (2) AI planning-based play-out, termed *planned play-out* [11]. In this paper we use the term *smart play-out* to refer to the general idea of smart look-ahead execution of scenario-based programs, and not only to the specific model-checking based implementation thereof.

While naïve play-out chooses its steps one by one, smart play-out reacts to an external event by seeking a sequence of system events that drive the specification to completion. The problem of smart play-out can be defined as follows: given a specification and a current configuration (the set of current cuts, together with the current state of all objects and variables), find a sequence of legal steps that lead the system to a stable state, i.e., one in which no main charts remain active. This sequence of steps is termed a *superstep.* Both known smart play-out implementations solve this by reduction: they translate a given specification and configuration into a model, and then use powerful computational methods (model-checking or planning) in order to find an appropriate path in this model. When found, such a path is translated back into a superstep in the original specification. The model created by both algorithms is proportional in size to that of the LSC specification, which refers to the number of lifeline locations in it. Thus, any reduction in the size of the LSC specification fed to a smart play-out algorithm will yield a smaller model created by them, which in turn may result in better running time.

3 Accelerating Smart Play-Out

We now show how to reduce the size of the specification before smart play-out is computed, by exploiting the special structure of LSCs, and details of its operational semantics. Our algorithm identifies constructs that are either irrelevant to the current superstep, or are unnecessary input for smart play-out. These constructs are then temporarily removed from the specification.

The algorithm consists of four steps. The first three are performed iteratively, until a fixpoint is reached, and then the last one is applied; see Fig. 2. Intuitively, *Activation Closure* detects charts that cannot participate in the superstep, *Early Evaluation* pre-evaluates conditions and assignments whenever possible, *Unreachable Elimination* removes superfluous unreachable constructs, and finally *Construct Elimination* eliminates constructs for which no reasoning is needed to order them correctly during execution of the superstep. Note that each step acts on the result of the previous steps, which will, in general, be a smaller specification. Thus, for example, a construct that is reachable in the original specification may become unreachable by some early evaluation, and will be removed by later steps.

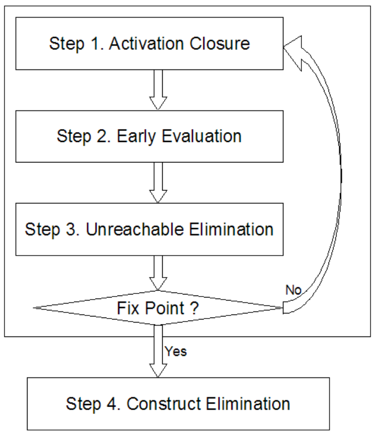

Step 1. Activation Closure
Step 2. Early Evaluation
Step 3. Unreachable Elimination
Fix Point ?
No
Yes
Step 4. Construct Elimination

Fig. 2. An overview of the acceleration algorithm.

3.1 The example

We refer here to the example from Fig. 1, expanded to support three phones by replicating all six LSCs three times, and replacing the Tel1 object with Tel1, Tel2, and Tel3 in each replica. We denote the copy number of each LSC by a subscript (e.g., $\texttt{Hangup}_1$ denotes the Tel1 copy of LSC `Hangup`). Also, we denote the event of object o_1 sending message `msg` to object o_2 by $o_1 \xrightarrow{\texttt{msg}} o_2$.

We consider the following initial configuration: Phone 1 is in a low priority call (i.e., the message `InCall` was sent from Tel1 to Operator, and $Tel1.LowPri$ is true). Phone 2 is in a high priority call (i.e., the message `InCall` was sent from Tel2 to Operator, and $Tel2.LowPri$ is false). Phone 3 is not in a call (i.e., the message `InCall` was never sent from Tel3 to the Operator).

Now, suppose the operator decides it must hang up all low priority calls, i.e., the message `HangupLowPri` is sent from the operator to itself. At this point some main charts become active, and smart play-out starts. The initial configuration is as follows: two main charts are active – $\texttt{HangupAllLowPri}_1$ and $\texttt{HangupAllLowPri}_2$ – and all other charts are closed. Recall that $Tel1.LowPri$ is true and $Tel2.LowPri$ is false. Also assume that $Logger.on$ is true.

3.2 Activation Closure

Consider the message $Operator \xrightarrow{\texttt{HangupLowPri}} Operator$ in the example. It does not appear in any main chart, and therefore will never be sent again throughout the superstep. Since LSC $\texttt{HangupAllLowPri}_3$ is not active, we can conclude that it will never become active in the superstep (as one of its prechart messages will not be sent). Therefore this LSC can be safely removed from the specification. Now we know that $Operator \xrightarrow{\texttt{HangupLowPri}} Tel3$ will not be sent, since it appears in no main chart of the remaining specification, and hence $\texttt{CheckPri}_3$ can be removed.

The *Activation Closure* step removes from the specification LSCs that can not become active in the superstep by computing the least fixpoint of LSCs, such that for every LSC in the activation closure, each message in its prechart appears in some main chart in the closure (regardless of their order in the prechart).

We ignore here the case where advanced constructs such as `if-then-else`, appear in the prechart. To adapt the method to the more general case, one needs to add an LSC to the activation closure not only if its entire prechart is contained in the set of possible messages, but even if some set of messages that could satisfy the prechart is contained in it.

3.3 Early Evaluation

Now consider phones 1 and 2. Their value of *LowPri* cannot change throughout the superstep (there is no main chart message that changes them in the specification). Therefore, the condition of the `if-then-else` construct in $\texttt{CheckPri}_1$ and $\texttt{CheckPri}_2$ can be evaluated in advance. This will be carried out by the *Early Evaluation* step.

More generally, the *Early Evaluation* step locates properties that will not be changed during any superstep, and pre-evaluates all conditions and assignments that use their value. This step does not affect the set of legal supersteps.

3.4 Unreachable Elimination

Following the early evaluation of conditions, some parts of the LSC may become unreachable. In our example, the "else" part in $\texttt{CheckPri}_1$ and the "if" part in $\texttt{CheckPri}_2$ are both unreachable now, and can be removed. This will be done by the *Unreachable Elimination* step.

Note that even if a message is unreachable in one LSC, it may be executed by another chart, so that unreachable messages can still cause chart violations if executed when not enabled, and one needs to take extra care when eliminating messages. Therefore, we eliminate a message only if all its appearances in main charts are unreachable. To avoid changing the partial order of the LSC, eliminated constructs are replaced by an appropriate synchronization construct (a constant true condition covering the relevant lifelines). The *Unreachable Elimination* step does not affect the set of legal supersteps.

3.5 Repeating Steps 1-3

Applying the steps on the example as above may lead to the conclusion that $Tel2 \xrightarrow{\texttt{Hangup}} Tel2$ will not be sent. Therefore by rerunning *Activation Closure* we can conclude that $\texttt{Hangup}_2$, for example, can now also be removed. This illustrates why the first three steps are executed repeatedly until a fixpoint is reached.

In general, the set of messages appearing in main charts of the *Activation Closure* dictates which properties may be modified in the superstep, thus affecting the *Early Evaluation* and *Unreachable Elimination* steps. In turn, those steps affect the *Activation Closure* computation, by removing messages from the charts. Therefore, the three steps need to be computed iteratively until a (greatest) fixpoint is reached.

Since each step removes only elements (or entire LSCs) that will never take part in any superstep and does not change the set of legal supersteps, the same applies to the repeated execution.

Note that in our example, *Activation Closure* is exact; all LSCs in the final specification will take part in the superstep. In the general case, this is not necessarily true. *Activation Closure* calculates a safe approximation of the set of LSCs that may participate in the execution, and not necessarily the exact set.

3.6 Construct Elimination

All steps mentioned so far eliminate constructs that cannot participate in the superstep. The purpose of the *Construct Elimination* step is to identify (and eliminate) constructs that may participate in the superstep, but for which the exact timing is not important.

For example, consider the message $Logger \xrightarrow{\texttt{LogHangup}} Logger$. It appears in one main chart only, $\texttt{Hangup}_1$ (we have already removed $\texttt{Hangup}_2$ and $\texttt{Hangup}_3$), and does not change any object property. Therefore, there is no real need for smart evaluation in determining when to send it; sending it whenever it is enabled is fine. This is the purpose of the last step, *Construct Elimination*, which identifies constructs for which no smart evaluation is needed and removes them.

The most important part of this step is identifying constructs that are redundant in terms of the smart play-out computation. For example, a message that changes no properties and appears only once in the (already reduced) specification can be sent whenever it is enabled, without the need for any smart ordering. These messages are removed by the *Construct Elimination* step. Similarly, this step identifies redundant conditions, subcharts and entire LSCs, and removes them.

3.7 Superstep Reconstruction

The output of our algorithm is a new specification and initial configuration, which can then be given as input to any smart play-out method. However, in general, a superstep found by this combined method, though legal in the modified specification, is not necessarily legal in the original one.

Consider the result of applying the algorithm to the example, as shown in Section 3.1. As we saw, the *Activation Closure* step removed the LSCs related to phone 3. These LSCs will never become active in this superstep, therefore there is nothing to do regarding them in the superstep reconstruction. Similarly, the modifications performed by *Early Evaluation* and *Unreachable Elimination* do not affect the set of legal supersteps, and their modifications need not be taken into account in the superstep reconstruction.

However, the last step, *Construct Elimination*, does affect the set of supersteps and we must take its modifications into account when constructing a legal superstep for the original specification. In the example above we saw that $Logger \xrightarrow{\texttt{LogHangup}} Logger$ is removed from the specification by this step. As opposed to previous steps, this is not because it will not participate in any superstep but because it is easy to decide when to execute it: it can be executed whenever enabled. For example, consider the LSC $\texttt{Hangup}_1$, and suppose the message $Logger \xrightarrow{\texttt{LogHangup}} Logger$ is the only one removed from it. Now consider a superstep found by applying smart play-out to the modified specification. This superstep may activate this LSC and then send the message $Tel1 \xrightarrow{\texttt{StartHangup}} Logger$. As a result, the message $Logger \xrightarrow{\texttt{LogHangup}} Logger$ becomes enabled (in the original specification); since it was removed by *Construct Elimination*, we know it should be executed (naïvely) whenever enabled, therefore we should now execute it and advance the cut accordingly.

This example is representative of the general rule that constructs removed by *Construct Elimination* should be executed whenever enabled. Therefore, in order to reconstruct a legal superstep in the original specification, one merely needs to start executing the superstep found for the new one. Following each step, the list of eliminated constructs should be checked. Any construct that was eliminated and is now enabled can, and should, be safely executed.

3.8 Complexity

It is easy to see that all steps of our algorithm take time polynomial in the size of their input (the LSC specification). Moreover, the number of times they are performed is linear in the size of the specification (each iteration must remove at least one construct in order for a fixpoint not to be reached). Therefore, the entire algorithm is in PTIME.

Clearly, the algorithm is heuristic. On some specifications it may work very well, while on others it might not change the specification at all. Section 4 shows cases for which the algorithm yields a significant improvement in running time, as well as a case for which no improvement is achieved.

4 Experimental Results

Consider a parameterized generalization of the example introduced in Section 2, containing n phones, and an initial configuration where half of the phones are in call, and half of those are low priority. Fig. 3 plots the running time (log scale) of model-checking-based smart play-out on a standard PC, as a function of n, for four different specifications: (1) The original specification with no acceleration, (2) the specification after applying *Activation Closure* only, (3) the specification with the first three steps applied iteratively until a fixpoint is reached, and (4) the specification with the complete acceleration algorithm applied. It is evident from the figure that each step adds a significant improvement to the running time, and allows a better scale-up of the size of the specification.

Interestingly, by a slight modification of the initial configuration, in which we set *logger.On* to be false (instead of true as in the previous run), the acceleration algorithm ends up with an empty specification: it removes all constructs, as it finds that non of them is crucial for the smart play-out algorithm. This means that all supersteps from the initial state are correct, and any naive play-out would succeed in this case. We are thus able to completely avoid running the model-checker; smart play-out computation time reduces to zero.

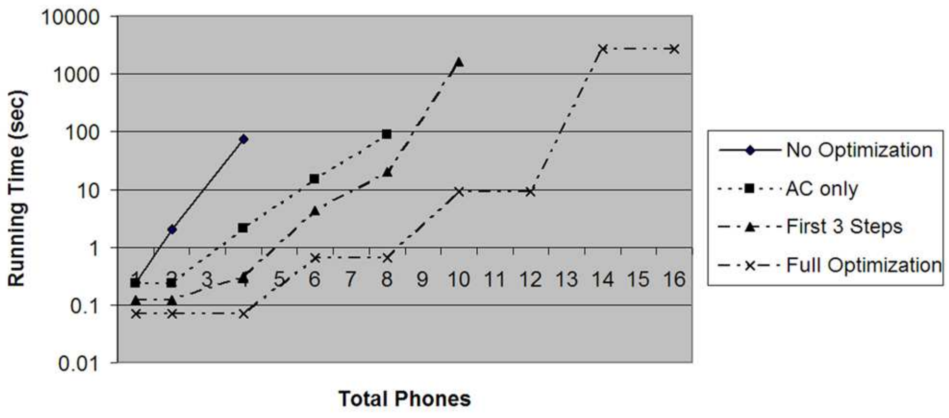


Fig. 3. Running time as a function of the number of phones, for different setups. Smart play-out runs that did not terminate within one hour were aborted, thus the maximum number of phones that could be handled within this time frame are 4 (no optimization), 8 (*Activation Closure* only), 10 (first 3 steps) and 16 (full optimization).

We have also applied our algorithm to two previously published specifications: (1) The Depannage telecommunication system described in [12]. Two subsets of the specification were executed. For the first, the acceleration saved 36% of the running time (21 seconds instead of 33), and for the second, the algorithm ended with an empty specification (smart play-out computation time reduces to zero). (2) The elevator example presented in [11]. The algorithm made no changes to the specification, thus no improvement was achieved in running time.

In all experiments, the running time of the algorithm itself was negligible.

5 Related and Future Work

In model-checking, the cone of influence method [9] attempts to locate only those variables that affect variables referred to in the specification, and remove all other variables. Thus, parts of our method may be viewed as a variant of the cone of influence method.

The *Early Evaluation* step can be viewed as a special case of constant propagation [13], which is used in compiler optimization to pre-evaluate expressions for which the value is known in advance.

Our work may also be viewed as a mix of static and dynamic forward program slicing [14], but applied to models rather than to code. In this sense, the reduced specification represents a safe approximation of the minimal forward model slice required for a correct superstep computation.

In this paper we focus on execution of LSCs, an extension of MSC. We believe some of the ideas presented here may be adapted to accelerate execution and simulation of other variants of MSCs, see, e.g., [1, 15].

Our algorithm uses ideas from static analysis of code, such as early evaluation of conditions [16]. Additional ideas could probably be adapted to our needs. For example, we could probably conduct better data-flow and control-flow analysis to identify which messages are dependent on which others, and thus gain better knowledge for the acceleration process. Clearly, there is a trade-off between the power of the acceleration method and its own running time. The goal is to find the best possible approximation of the minimal specification needed for the smart play-out method. The better the approximation, the less running time will be needed for the smart play-out itself.

Some limitations of our approach are worth noting. In the *Activation Closure* step we do not consider the order of messages but only whether they are included in the LSCs or not. This results in an over approximation; in the worst case the fixpoint may include all LSCs in the specification, even though only a small subset of them may participate in one of the possible supersteps. Other limitations relate to data; e.g., we ignore the value assigned in property set messages so we may fail to identify some conditions that can be evaluated early.

In model-checking, partial order reduction [9] reduces the state-space to be explored by identifying transitions that result in the same state when executed in different orders. Similarly, some steps of scenario-based specifications, when executed in different order, may result in the same global system state. Therefore, methods and ideas similar to those used in partial order reduction may help in improving smart play-out as well. Note that our *Construct Elimination* step may have the effect of a partial order reduction.

Our method reduces the size of the LSC specification, which in turn reduces the size of the model given to the smart play-out techniques. However, smart play-out efficiency may be improved also by adding constraints, such as anti-scenarios or forbidden elements. This would make the LSC specification larger but could induce a smaller state-space for applying smart play-out.

As mentioned above, our algorithm computes a safe approximation of the minimal forward model slice required for a correct superstep computation, for the purpose of smart play-out acceleration. However, this model slice can also be used for model comprehension, since presenting it to the user may aid in focusing on the more important LSCs, modulo a given configuration. Developing techniques for presenting scenario-based model slices to the user, textually or visually, is an interesting direction for future work.

Acknowledgments We thank Moshe Vardi for his advice on this work. We thank Andrew Phillips for comments on a draft of this paper.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. IEEE Trans. Software Eng. **29**(7) (2003) 623–633
2. Broy, M.: A semantic and methodological essence of message sequence charts. Sci. Comput. Program. **54**(2-3) (2005) 213–256
3. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. J. on Form. Meth. in Sys. Design **19**(1) (2001) 45–80
4. Harel, D., Marelly, R.: Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag (2003)
5. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Software Eng. **29**(2) (2003) 99–115
6. ITU: International Telecommunication Union Recommendation Z.120: Message Sequence Charts. Technical report (1996)
7. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-out of Behavioral Requirements. In: Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02). Volume 2517 of LNCS., Springer (2002) 378–398
8. Harel, D., Kugler, H., Maoz, S., Segall, I.: How Hard is Smart Play-Out? On the Complexity of Verification-Driven Execution. In: Perspectives in Concurrency Theory (Festschrift for P.S. Thiagarajan), University Press (India) (2009) 208–230
9. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
10. Harel, D., Kugler, H., Maoz, S., Segall, I.: Accelerating smart play-out. Technical report, Weizmann Institute of Science (2009)
11. Harel, D., Segall, I.: Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In: Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). Volume 4424 of LNCS., Springer (2007) 485–499
12. Combes, P., Harel, D., Kugler, H.: Modeling and Verification of a Telecommunication Application using Live Sequence Charts and the Play-Engine Tool. Int. J. Soft. Sys. Mod. (SoSyM) **7**(2) (2008) 157–175
13. Callahan, D., Cooper, K.D., Kennedy, K., Torczon, L.: Interprocedural Constant Propagation. In: Proc. SIGPLAN Symp. on Compiler Construction (CC'86), ACM (1986) 152–161
14. Korel, B., Yalamanchili, S.: Forward Computation of Dynamic Program Slices. In: Proc. ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA'94), ACM (1994) 66–79
15. Krüger, I.: Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs. In: Proc. 6th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'03). Volume 2621 of LNCS., Springer (2003) 387–402
16. Pezzé, M., Young, M.: Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons (2008)