

Smart Play-Out *

David Harel, Hillel Kugler, Rami Marelly and Amir Pnueli

{dharel,kugler,rami,amir}@wisdom.weizmann.ac.il

Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

ABSTRACT

We describe “smart play-out”, a new method for executing and analyzing scenario based behavior, which is part of the Play-In/Play-Out methodology and the Play-Engine tool. Behavior is “played in” directly from the system’s GUI, and as this is being done the Play-Engine continuously constructs Live Sequence Charts (LSCs), a powerful extension of sequence diagrams. Later, behavior can be “played out” freely from the GUI, and the tool executes the LSCs directly, thus driving the system’s behavior. An inherent difficulty in constructing a “play-out” mechanism is how to resolve the nondeterminism allowed by the LSC specification in order to obtain an executable model. Smart play-out, is a recent strengthening of the play-out mechanism, which addresses this problem by using powerful verification methods, mainly model-checking, to execute and analyze the LSCs, helping the execution to avoid deadlocks and violations. Thus, smart play-out utilizes verification techniques to run programs, rather than to verify a program with respect to given requirements, as in traditional verification approaches. The ideas appear to be relevant in various stages of system development, including requirements specification and analysis, implementation and testing.

Keywords

Specification techniques, Verification, Model checking, Requirements Engineering, System Modeling and Execution, Scenarios, Object-Oriented Analysis and Design, UML, LSCs, Play-Out.

1. INTRODUCTION

1.1 Scenario based requirements and LSCs

Understanding system and software behavior by looking at various “stories” or scenarios seems a promising approach, and it has focused intensive research efforts in the last few years. One of the most widely used languages for spec-

*This research was supported in part by the John von Neumann Minerva Center for the Verification of Reactive Systems and by the European Commission project OMEGA (IST-2001-33522).

ifying scenario-based requirements is that of message sequence charts (MSCs), adopted long ago by the ITU [10], or its UML variant, sequence diagrams [9]. Sequence charts (whether MSCs or their UML variant) possess a rather weak partial-order semantics that does not make it possible to capture many kinds of behavioral requirements of a system.

To address this, while remaining within the general spirit of scenario-based visual formalisms, a broad extension of MSCs has been proposed, called live sequence charts (LSCs) [2]. LSCs distinguish between behaviors that may happen in the system (existential) from those that must happen (universal). A universal chart contains a *prechart*, which specifies the scenario which, if successfully executed, forces the system to satisfy the scenario given in the actual chart body. The distinction between mandatory (hot) and provisional (cold) applies also to other LSC constructs, e.g., conditions and locations, thus creating a rich and powerful language, which among many other things can express forbidden behavior (‘anti-scenarios’).

1.2 The Play-In Play-Out Approach

In [5] a methodology for specifying and validating requirements, termed “play-in/play-out approach” is described. According to this approach, requirements are captured by the user playing in scenarios using a graphical interface of the system to be developed or using an object model diagram. The user “plays” the GUI by clicking buttons, rotating knobs and sending messages (calling functions) to objects in an intuitive manner. By similarly playing the GUI, the user describes the desired reactions of the system and the conditions that may or must hold. As this is being done, the supporting tool, called the Play-Engine, constructs a formal version of the requirements in the form of LSCs.

Play-out is a complementary idea to play-in, which, rather surprisingly, makes it possible to execute the requirements directly. In play-out, the user simply plays the GUI application as he/she would have done when executing a system model, or the final system implementation, but limiting him/herself to “end-user” and external environment actions only. While doing this, the Play-Engine keeps track of the actions and causes other actions and events to occur as dictated by the universal charts in the specification. Here too, the engine interacts with the GUI application and uses it to reflect the system state at any given moment. This process of the user operating the GUI application and the Play-Engine causing it to react according to the specification has the effect of working with an executable model, but with no intra-object model having to be built or synthesized.

Play-out is actually an iterative process, where after each step taken by the user, the Play-Engine computes a super-step, which is a sequence of events carried out by the system as its response to the event input by the user. The play-out mechanism of [5] is rather naive when faced with nondeterminism, and makes essentially an arbitrary choice among the possible responses. This choice may later cause a violation of the requirements, whereas by making a different choice the requirements could have been satisfied.

2. SMART PLAY-OUT

Smart play-out was introduced in [4] and since then has been extended to cover a larger set of the LSC language features and to deal more efficiently with larger models. We have also gained experience in applying the method to several applications and case studies.

2.1 Motivation

The motivation is to try to resolve the nondeterminism allowed by the LSC specification in a smarter way. From a conceptual point of view, the potential of many possible sequences of reactions to a user event is due to the fact that LSCs is a declarative, inter-object language, and as such it enables formulating high level behavior in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. Technically, the two sources of this nondeterminism are the partial order semantics among events in each chart, and different charts containing scenarios that do not have enough information about their inter-relationship. These features are very useful in early requirement stages, but can cause undesired under-specification when one attempts to consider them as the system's executable behavior. Smart play-out attempts to remove these sources of nondeterminism during execution. It should be noted that smart play-out, at least as it stands today, looks one super-step ahead, while full synthesis algorithms [3] perform a complete analysis. However, the latter are still impractical, due to computational complexity problems.

2.2 The Approach

The approach we use is to formulate the play-out task as a verification problem, and to use the counterexample provided by a model-checking [1] algorithm as the desired super-step. The system on which we perform model-checking is constructed according to the universal charts in the LSC specification. We define a transition relation generated from the LSC specification, which is designed to allow progress of active universal charts, but prevents any violations. The system is initialized to reflect the status of the application just after the last external event occurred, including the current values of object properties, information on the universal charts that were activated as a result of the most recent external events, and the progress in all precharts. The model-checker is then given a property claiming that always at least one of the universal charts is active. This is really the negation of what we want since in order to falsify the property, the model-checker searches for a run in which eventually none of the universal charts is active; i.e., all active universal charts complete successfully, and by the definition of the transition relation no violations occurred in the process. Such a counter-example is the desired super-step. If the model-checker verifies the property then no correct super-step exists but if it doesn't, the counter-example is exactly

what we seek. For more details see [4].

2.3 Satisfying Existential Charts

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

3. APPLICATIONS

In the demo session we will show and explain the tool, illustrating the approach as applied to several applications we have studied. These include a phone network [8], a machine for manufacturing smart-cards [6] and a model of a biological system [7].

* * *

As a long-term vision, we believe that for certain kinds of systems the play-out methodology, enhanced by formal verification techniques, could serve as the final implementation too, with play-out being all that is needed for running the system itself.

4. REFERENCES

- [1] E.C. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [2] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99).
- [3] D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. *Int. J. of Foundations of Computer Science (IJFCS)*, 13(1):5–51, February 2002. (Also, Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000), LNCS 2088, Springer-Verlag, 2000.).
- [4] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *Proc. 4th Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, volume 2517 of *Lect. Notes in Comp. Sci.*, pages 378–398, 2002.
- [5] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
- [6] H. Kugler and G. Weiss. Modeling a Smart-Card Personalization Machine with LSCs. Technical report, Weizmann Institute, 2003.
- [7] N. Kam, D. Harel, H. Kugler, R. Marelly, A. Pnueli, E.J.A. Hubbard, and M.J. Stern. Formal Modeling of C. elegans Development: A Scenario-Based Approach. Proc. Int. Workshop on Computational Methods in Systems Biology (CMSB 2003), 2003.
- [8] R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, pages 83–100, Seattle, WA, 2002.
- [9] UML. Documentation of the unified modeling language (UML). Available from OMG, <http://www.omg.org>.
- [10] Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.