Synthesizing Reactive Systems from LSC Requirements using the Play-Engine

Hillel Kugler
Microsoft Research
Cambridge, United Kingdom
hkugler@microsoft.com

Cory Plock
New York University
New York, NY, United States
plock@cs.nyu.edu

Amir Pnueli
New York University
New York, NY, United States
amir@cs.nyu.edu

Abstract

Live Sequence Charts (LSCs) is a scenario-based language for modeling object-based reactive systems with liveness properties. A tool called the Play-Engine allows users to create LSC requirements using a point-and-click interface and generate executable traces using features called play-out and smart play-out. Finite executable trace fragments called super-steps are generated by smart play-out in response to user inputs. Each super-step is guaranteed not to violate the LSC requirements, provided one exists. However, non-violation is not guaranteed beyond each individual super-step. In this work, we demonstrate a powerful extension to smart play-out which produces only traces that are guaranteed not to violate the LSC requirements, provided the requirements are realizable. Using this method, we may synthesize correct executable programs directly from LSC requirements.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/specifications—languages, tools; D.2.2 [*Software Engineering*]: Design tools and techniques—object-oriented design methods; D.2.4 [*Software Engineering*]: Software/program verification—formal methods.

General Terms Design, languages, verification.

1. Introduction

Live Sequence Charts (LSCs) [2] is a formal requirements language that is well-suited to the design of liveness-enriched reactive systems. LSCs provides a visual method for representing reactive behavior, which can enhance both readability and human understanding. A tool called the Play-Engine [5] provides the ability to create and execute LSC requirements in software. A method called the *play-in/play-out approach* was also described in [5]. Play-in provides an intuitive method to capture requirements using a graphical representation of the system, while play-out executes the scenarios in a way that gives a feeling of running an implementation of the system.

A Play-Engine feature called smart play-out is introduced in [4]. The user plays the role of the environment by injecting an input event and then smart play-out discovers a non-violating *super-step* (i.e., series of output events), provided one exists. Smart play-out is naive in choosing among a set of correct super-step candidates, and furthermore does not analyze beyond one super-step. A poor super-step choice can lead to a state from which no further super-step exists—a violation of the LSC requirements.

We have created an algorithm for deciding the *realizability* of LSC requirements (i.e., deciding the existence of a system which can guarantee non-violation.) If answered in the affirmative, we can construct a strategy for achieving only non-violating behaviors. These results may be used to synthesize non-violating executable systems directly from LSC requirements. We have implemented these results as an extension to the Play-Engine using a rich subset of the LSC language already supported by smart play-out [4].

2. Main Results

Our extension to smart play-out works in the following way. The user first creates LSC requirements using play-in. The requirements are then transformed automatically into a finite discrete transition system called a *game structure* [6, 3]. A realizability decision procedure determines if the system can satisfy the LSC requirements regardless of environment inputs. If the requirements are realizable, a controller is extracted [6, 1]. From this point, the user interacts with the Play-Engine in the same way as smart play-out—by injecting input events and observing the generated responses. Instead of choosing a response for a given input arbitrarily among the correct super-steps like smart play-out, we utilize the synthesized controller to identify responses which are guaranteed not to result in future violations.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

Realizability checking and controller extraction follow a game-theoretic approach in which two players, an environment and a system, compete to satisfy opposite goals. The system attempts to satisfy a *winning condition* (given as a temporal logic formula), which characterizes the LSC liveness properties—namely, that the main chart of every universal LSC in the requirements is inactive (or, closed) infinitely often. The environment's goal is to steer the game away from the winning condition, which amounts finding paths through which any main chart remains open forever. The system *wins* the game if it can satisfy the winning condition regardless of the environment's actions.

3. Example

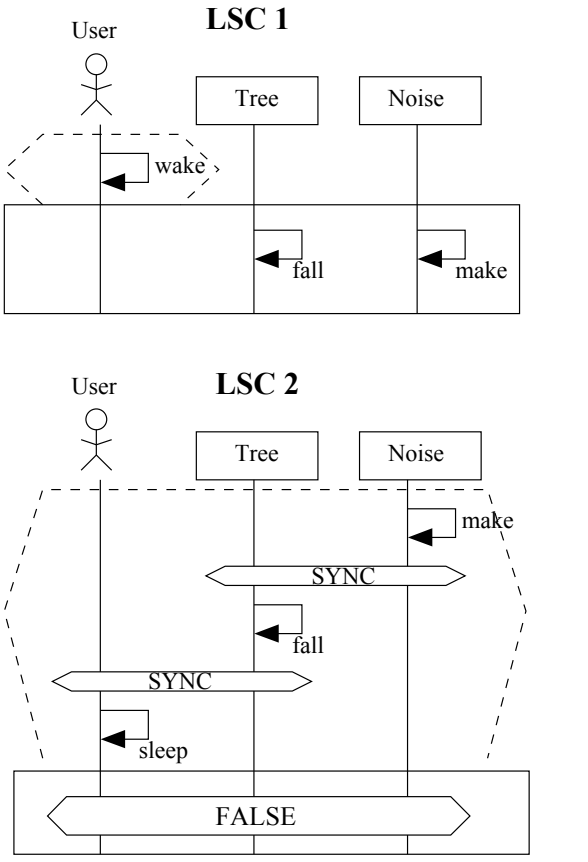

Figure 1. LSC Requirements

The LSC requirements of Fig. 1 consists of two universal LSCs in which an environment (i.e., `User`) interacts with a system (i.e., instances `Tree`, `Noise`). **LSC1** states that whenever the user wakes, a tree must fall and a noise must be made, in any order. **LSC2** is an anti-scenario which specifies that the noise followed by the tree falling followed by the `User` sleeping, is not an allowable behavior. While not particularly interesting by themselves, these LSCs when interacting together produce somewhat more interesting behavior.

We wish to determine if these requirements are realizable. Consider the executable trace prefix, `wake`, `fall`, `make`. The prechart of **LSC1** is satisfied when `wake` occurs and the occurrence of `fall` and `make` satisfy the main chart, causing it to close. The same sequence never satisfies the prechart **LSC2**. Subsequently, the main chart of **LSC2** remains closed throughout the execution of the trace prefix. After this sequence, the super-step is complete. If the environment were to execute either `wake` or `sleep` as the next input, no violation would occur.

Now consider the trace prefix `wake`, `make`, `fall`. As can be seen, this trace once again satisfies **LSC1**. The same prefix will cause the prechart of **LSC2** to progress midway and await the environment message `sleep`. After this sequence, the super-step is complete and the system is ready to accept new inputs. However, if the environment now chooses to execute `sleep`, the LSC requirements are violated since the execution will remain forever in the false condition of **LSC2**.

We observe that the above specification is realizable: the system can avoid violation by always choosing the super-step `fall`, `make` in response to the environment input `wake`. Our smart play-out extension affirmatively decides the realizability and constructs a controller which disallows the super-step sequence `make`, `fall` from occurring in response to `wake` by eliminating game structure transitions.

References

[1] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[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 specifi cations. In *Proc. 5th Inf. Conf. on Implementation and Application of Automata (CIAA'00)*, volume 2088 of *Lect. Notes in Comp. Sci.*, pages 1–33. Springer-Verlag, July 2000.

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

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

[6] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lect. Notes in Comp. Sci.*, pages 364–380. Springer-Verlag, 2006.