PlayGo: Towards a Comprehensive Tool for Scenario Based Programming

David Harel[1], Shahar Maoz[2], Smadar Szekely[1], and Daniel Barkan[1]
[1]Dept. of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel
[2]Dept. of Computer Science 3, Software Engineering, RWTH Aachen University, Germany

ABSTRACT

We present *PlayGo*, a comprehensive tool for scenario-based programming, built around the language of live sequence charts and the play-in/play-out approach [7], which includes a compiler into AspectJ code and means for debugging the execution. PlayGo is intended to be a full IDE that addresses major parts of the vision of *Liberating Programming* [3]. This paper presents the first version of PlayGo, which already includes several of the intended capabilities.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms

Design, Languages

Keywords

Live sequence charts, reactive systems, IDE, scenario-based programming

1. INTRODUCTION

System design and programming is a complex and challenging task. Many languages, tools, and methodologies have been developed over the years to make programmers more successful in producing software and hardware systems that meet their expectations. In a recent paper [3], the first-listed author presented a concept/dream, that calls for liberating programming from the need to explicitly construct the program as a symbolic, textual, or graphical artifact; from the need to specify requirements (*the what*) separately from the actual executable program (*the how*), with the consequential need to pit one against the other; and from the need to structure behavior according to the system's structure, necessarily having to provide each piece or object with its full behavior.

In this paper we present the first version of *PlayGo*, which will hopefully become a comprehensive IDE that will support this dream. The present version brings modest manifestations of these ideas to life by implementing and integrating the *play-in/play-out* approach proposed for scenario-based programming in [7], using the language of *live sequence charts* (LSC) [1, 6].

ASE'10, September 20–24, 2010, Antwerp, Belgium.
ACM 978-1-4503-0116-9/10/09.

LSC is a visual language that extends classical message sequence charts with *modalities* and *modularity*, thus enabling intuitive and incremental specification of reactive systems. Play-in is a user-friendly high-level way of entering the scenario-based behavior and automatically generating the specification formally in LSCs. The user specifies behavior directly, by example, on a mock or real GUI of the system under development. Play-out is a method for executing a set of modal scenarios directly; the subtle issues around executability are related (among other things) to the fact that the language can specify allowed, forbidden and mandatory behavior and the fact that scenarios can be symbolic, with the classes and methods having to be unified at runtime. See [7].

PlayGo may be viewed as a vastly extended and broader elaboration of the initial (and experimental) *Play-Engine* tool developed in our group almost a decade ago to support LSCs and play-in/out [7]. In contrast to the Play-Engine, PlayGo, even in its initial version, adheres to the UML standard, and has an open architecture that enables extension and integrations with other tools. Moreover, execution is carried out by compilation and not by an interpreter.

Below we give an overview of PlayGo. Additional details, including screenshots and demos, are available from http://www.wisdom.weizmann.ac.il/~playgo/

2. OVERVIEW OF PLAYGO

2.1 Play-In

Play-in is the process of creating (specifying) a scenario by directly manipulating objects in the GUI of the system under development [7]. The GUI (which obviously will be different for the different models specified, and whose construction is not part of the current version of PlayGo) is assumed to already exist, but it may also evolve as the system specification evolves. The user enters events by actually causing them to happen on the GUI; for example, by clicking buttons or entering text in display fields. Each such GUI operation is immediately and automatically reflected in the LSC, which is generated on the fly, and in the continuously accumulating underlying model.

As part of the play-in process, instances and operations can be added to the LSC. An instance can represent a concrete GUI object or can stand for a set of GUI objects that participate in the scenario. A special type of instance represents the user, or the environment. Operations represent

method calls between the participating objects. In addition, LSCs may contain conditions, assignments and structural constructs (e.g., loops).

Technically, PlayGo enables play-in on top of a user-defined GUI application, which in the current version is prepared outside PlayGo and is fed into the tool before starting the play-in process. Currently PlayGo supports GWT-based web applications [14]. We plan to support additional technologies in the future. Additionally, the generated LSCs can be manipulated in the graphical editor, by changing the values of properties, either from a special properties view or programmatically, by using the LSC Generic Infrastructure API provided as part of PlayGo.

2.2 Play-Out

Play-out is the method introduced in [7] for executing LSC specifications. Every execution of an operation is considered a *step*. Following a user action, the system executes a *superstep* – a sequence consisting of the steps that follow the user action, and which terminates either at a stable situation, where the next event is a user action, or when the entire execution terminates.

During play-out, the current state of each LSC is represented by a *cut*. Given the scenario-based nature of the specification, a cut may induce multiple events to be simultaneously enabled for execution. In this case, the play-out mechanism chooses which event to execute based on some strategy. In the "naïve" play-out technique of [7], these choices are made nondeterministically, and could lead to violations. In the smarter play-out algorithms of [5, 9], the next step is carefully computed, using model checking or planning, in order to help avoid violations.

The current version of PlayGo implements naïve play-out only. However, this is not done interpreter-like (as in the Play-Engine), but by a variant of the S2A compiler of [11], which transforms LSCs into AspectJ code. During compilation, each LSC is statically analyzed and is translated into a scenario aspect that simulates an automaton whose states represent cuts and whose transitions are triggered by aspect pointcuts. Scenario aspects are locally responsible for listening out for relevant events and advancing the cut state accordingly. The compilation scheme generates a coordinator, implemented as a separate aspect, which observes the cut state changes of all active scenario aspects, chooses a method and proceeds to execute it. S2A follows the "strategy" design pattern to enable integration of various play-out strategies.

PlayGo provides unique debugging capabilities specially tailored for scenario-based execution, which include built-in step and superstep modes. Breakpoints are defined at the model level, visually, on the charts themselves. The debugger uses cut state information collected from the generated scenario aspects to display the LSC cuts. This model-level debugging is integrated with the standard code level debugging, allowing the user work interchangeably on both levels of abstraction.

During execution, PlayGo can generate model-based traces [10], which can be used as input for the Tracer tool of [12].

3. IMPLEMENTATION

PlayGo is implemented as a set of Eclipse plugins and is packaged as an Eclipse product. The decision to choose Eclipse as the core infrastructure of PlayGo stems from Eclipse's strong architecture and its adaptation of existing standards (such as UML). Eclipse architecture dictates openness, standard ways to implement extensions and to integrate with other software modules defined as Eclipse plugins. Furthermore, we believe that the Eclipse approach to building IDEs, and the set of tools it provides, fit the vision and principles underlying PlayGo.

PlayGo uses the Eclipse UML2Tools plugin [13] as its UML library, and extends it with an infrastructure library that provides an intuitive and generic mechanism for extending UML2 elements. This generic infrastructure hides the complexity of UML profiles from the developer, and is used in PlayGo for defining the UML-compliant variant of LSCs, which is the source language for our compiler [6]. Most of the work is done by interface declarations, so the code that the developer is required to produce is minimal.

Play-in is implemented on top of a user-defined GWT-based web application. Thus, the resulting target system can be shared via the internet.

PlayGo is still experimental, and we are working on many different topics to strengthen the vision of [3] and to broaden the power and applicability of the tool. Examples include a natural language interface for play-in [2], synthesis from scenarios to state-machines [4], implementing a version of smart play-out [5] and integrating our recent Java library for scenarios, which would enable one to program the scenarios directly in Java [8].

4. ACKNOWLEDGMENTS

We thank Guy Weiss for his contributions to the development of PlayGo. This research was supported by an Advanced Research Grant from the European Research Council (ERC), under the EU's 7th Framework.

5. REFERENCES

[1] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[2] M. Gordon and D. Harel. Generating executable scenarios from natural language. In *CICLing*, 2009.

[3] D. Harel. Can programming be liberated, period? *IEEE Computer*, 41(1):28–37, 2008.

[4] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *Int. J. of Foundations of Computer Science*, 13(1):5–51, 2002.

[5] D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *FMCAD*, 2002.

[6] 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.

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

[8] D. Harel, A. Marron, and G. Weiss. Programming coordinated scenarios in Java. In *ECOOP*, 2010.

[9] D. Harel and I. Segall. Planned and traversable play-out: A flexible method for executing scenario-based programs. In *TACAS*, 2007.

[10] S. Maoz. Model-based traces. In *MoDELS 2008 Workshops*, 2009.

[11] S. Maoz and D. Harel. From multi-modal scenarios to code: Compiling LSCs into AspectJ. In *ACM SIGSOFT FSE*, 2006.

[12] S. Maoz and D. Harel. On tracing reactive systems. *Software and Systems Modeling (SoSyM)*, 2010. DOI:10.1007/s10270-010-0151-2

[13] Eclipse UML2Tools. http://www.eclipse.org/modeling/mdt/?project=uml2tools.

[14] Google Web Toolkit. http://code.google.com/webtoolkit/.