The Play-in/Play-out Approach and Tool: Specifying and Executing Behavioral Requirements

David Harel Hillel Kugler Rami Marelly
The Weizmann Institute of Science

1. Introduction

Two kinds of behavior in object-oriented analysis and design are identified and discussed in [1,2]: *inter-object behavior*, which describes the interaction between objects per scenario, and *intra-object behavior*, which describes the way a single object behaves under all possible circumstances. For modeling intra-object behavior, most object-oriented modeling approaches adopt *statecharts* [3,4]. For the requirements aspect, one of the most widely used languages is that of *message sequence charts* (MSCs), adopted long ago by the ITU [5], or its UML variant, *sequence diagrams* [6,7].

According to many OO-based methodologies for system development, the user first specifies the system's *use cases* [8], and the different instantiations of each use case are then described using sequence charts. In a later modeling step, the behavior of a class is described by an associated statechart, which prescribes the behavior of each of its instances. Finally, the objects are implemented as code in a specific programming language.

Sequence charts (whether MSCs or their UML variant, sequence diagrams) possess a rather weak partial-order semantics that does not make it possible to capture many kinds of behavioral requirements of a system. This is mostly due to the fact that they do not distinguish between possible and mandatory behavior, and are thus far weaker than, e.g., temporal logic or other formal languages for requirements and constraints.

To address this, while remaining within the general spirit of scenario-based visual formalisms, a broad extension was proposed in 1999, called *live sequence charts* (LSCs)[1]. LSCs distinguish between scenarios that *may* happen in the system (existential) from those that *must* happen (universal). They can also specify messages that *may* be received (cold) and ones that *must* (hot). A condition too can be cold, meaning that it *may* be true (otherwise control moves out of the current block or chart), or hot, meaning that it *must* be true (otherwise the system aborts). Moreover, the progress of instances may be defined to be *hot* thus enforcing the instance to progress, or *cold* thus enabling the instance to remain in its location. Among many other things, LSCs can express forbidden behavior ('anti-scenarios').

Since its expressive power is far greater than that of MSCs, the language of LSCs makes it possible to start looking more seriously at the relationships and possible transitions between the behavioral artifacts of these modeling steps: on the one hand use cases and LSCs, which represent the system's requirements in an inter-object style, and on the other hand statecharts, which represent its implementable model in the intra-object style. Given the discussion in [2], we should strive to be able to verify that the former is true of the latter, but also to synthesize the latter from the former [9].

How should these more expressive requirements be entered? One cannot imagine automatically synthesizing the LSCs or temporal logic from the use cases, since use cases are informal and highly abstract. This leaves us with having to construct the LSCs manually, which is problematic, because despite the visuality, LSCs constitute a formal language, which will not always be appropriate for the people involved in the early stages of requirements capture. Towards the end of [2], this problem was addressed, and a higher-level approach to the problem of specifying scenario-based behavior, termed *play-in scenarios*, was proposed and briefly sketched. The details of this play-in methodology and the *play-engine* tool we have built to support it are given in [10]. The same paper also describes *play-out*, a complementary idea to play-in, which, rather surprisingly, makes it possible to execute the requirements directly without having to build or synthesize an intra-object model of the system. In [11], *smart play-out* is introduced, which strengthens the 'naive' play-out algorithms by using analytic techniques from the field of formal verification, mainly model checking.

In this paper we briefly overview the play-in/play-out methodology and the play-engine tool.

2. Play-In

The main idea of the play-in process is to raise the level of abstraction in requirements engineering, and to work with a look-alike version of the system under development. This enables people who are unfamiliar with LSCs, or who do not want to work with such formal languages directly, to specify the behavioral requirements of systems using a high level, intuitive and user-friendly mechanism. These could include domain experts, application engineers, requirements engineers, and even potential users.

What "play-in" means is that the system's developer first builds the GUI of the system, with no behavior built into it. In systems for which there is a meaning to the layout of hidden objects (e.g., a board of an electrical system), the user may build the graphical representation of these objects as well. In fact, for GUI-less systems, or for sets of internal objects, we simply use the object model diagram as a GUI. In any case, the user "plays" the GUI by clicking buttons, rotating knobs and sending messages (calling functions) to hidden objects in an intuitive drag & drop manner. (With an object model diagram as the interface, the user clicks the objects and/or the methods and the parameters). 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 play-engine continuously constructs LSCs automatically. It queries the application GUI (that was built by the user) for its structure, and interacts with it, thus manipulating the information entered by the user and building and exhibiting the appropriate LSCs. We have made special efforts to enable the user to carry out as much of the play-in as possible by manipulating the GUI directly, without having to deal with the LSCs explicitly.

3. Play-Out

After playing in (a part of) the specification, the natural thing to do is to verify that it reflects what the user intended to say. One way of testing requirements is by constructing a prototype intra-object implementation and using model execution to test it [12]. Instead, we would like to extend the power of our interface-oriented inter-object play methodology, to not only specify the behavior, but to test and validate the requirements as well. And here is where the play-out mechanism enters.

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, 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. This makes it very easy to let all kinds of people participate in the process of debugging the specification, since they do not need to know anything about the specification or the language used. It yields a specification that is well tested and which has a lower probability of errors in later phases, which are a lot more expensive to detect and eliminate.

We should emphasize that the behavior played out need not be the behavior that was played in. The user is not merely tracing scenarios, but is operating the system freely, as he/she sees fit.

Universal charts "drive" the model by their *action/reaction* nature, whereas existential charts can be used as system tests or as examples of required interactions. Rather than serving to drive the play-out, existential charts are *monitored*, that is, the play-engine simply tracks the events in the chart as they occur. When (and if) a traced chart reaches its end, it is highlighted, and the user is informed that it was successfully traced to completion.

The play-engine supports several extensions we have made to LSCs, including variables that can be used to represent several instantiations of the same scenario with different specific values in the messages, symbolic instances that allow for the instances themselves to be symbolic, representing classes rather than actual objects, and a rather powerful means for dealing with real time [14]. The algorithmic mechanism underlying play-out is described in [10,13].

4. Smart Play-Out

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 response to the event input by the user. This process is rather naive: For example, there can be many sequences of events possible as a response to a user event, and some of these may not constitute a "correct" super-step. We consider a super-step to be correct if when it is executed no active universal chart is violated. The multiplicity of possible sequences of reactions to a user event is due to the fact that a declarative inter-object behavior language such as LSCs allows high-level requirements given in pieces (e.g., scenario fragments), leaving open details that may depend on the implementation. The partial order semantics among events in each chart and the ability to separate scenarios in different charts without saying explicitly how they should be composed are very useful in early requirement stages, but can cause under-specification and non-determinism when one attempts to execute them.

Smart play-out focuses on executing the behavioral requirements with the aid of formal analysis methods, mainly model-checking. Our smart play-out uses model-checking to find a correct super-step if one exists, or proves that there isn't one. We do this by formulating the play-out task as a verification problem, in such a way that a counterexample resulting from the model-checking will constitute the desired super-step. The transition relation is defined so that it allows progress of active universal charts but prevents violations. The property to be checked is one that states that always at least one of the universal charts is active. In order to falsify it, the model-checker searches for a run in which eventually none of the universal charts is active; i.e., all active universal charts completed successfully, and by the definition of the transition relation no violations occurred. Such a counter-example is exactly the desired super-step. If the model-checker manages to verify the property then no correct super-step exists.

The other kind of thing smart play-out can do is to find a way to satisfy an existential chart. Here we cannot limit ourselves to a single super-step, since the chart under scrutiny can contain external events, each of which triggers a super-step of the system. Nevertheless, the formulation as a model-checking problem can be used with slight modifications for this task too. Also, when trying to satisfy an existential LSC, we take the approach that assumes the cooperation of the environment. We should add that the method for satisfying existential LSCs can also be used to verify safety properties that take the form of an assertion on the system state. This is done by putting the property's negation in an existential chart and verifying that it cannot be satisfied.

*   *   *

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 the play-out being all that is needed for running the system itself.

References

[1] Damm, W., and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design*, **19**(1), 2001. (Also *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems*, pp. 293-312, 1999.)

[2] Harel, D., "From Play-In Scenarios To Code: An Achievable Dream", *IEEE Computer* **34** (1, 53-60, 2001.

[3] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Prog. 8* (1987), 231-274.

[4] Harel, D., and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer* (1997), 31-42.

[5] Z.120 ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), ITU-TS, Geneva, 1996.

[6] UML documentation, available from the Object Management Group (OMG), http://www.omg.org.

[7] Rumbaugh J., I.Jacobson and G.Booch, *The UML Reference Manual,* Addison-Wesley, 1999.

[8] Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[9] Harel, D., and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications", *Int. J. of Foundations of Computer Science* (IJFCS), 13(1), 5-51, 2002.

[10] Harel, D., and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach", Tech. Report MSC01-15, The Weizmann Institute of Science, 2001.

[11] Harel, D., H. Kugler, R. Marelly, and A. Pnueli "Smart Play-Out of Behavioral Requirements", *Proc. 4th International Conference On Formal Methods in Computer-Aided Design* (FMCAD'02), Portland, Oregon, 2002, to appear. Also available as Tech. Report MSC02-08, The Weizmann Institute of Science, 2002.
[12] M.Fränzel and K.Luth. "Visual Temporal Logic as a Rapid Prototyping Tool". *Vis. Lang. And Comput.*, 2001.
[13] R. Marelly, Harel, D., and H. Kugler, "Multiple Instances and Symbolic variables in Executable Sequence Charts", *Proc. 17th Ann. ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'02), Seattle, WA, 2002, to appear. Also available as Tech. Report MSC02-05, The Weizmann Inst., 2002.
[14] Harel, D., and R. Marelly, "Playing with Time: On the Specification and Execution of Time-Enriched LSCs", Manuscript, May 2002, submitted.

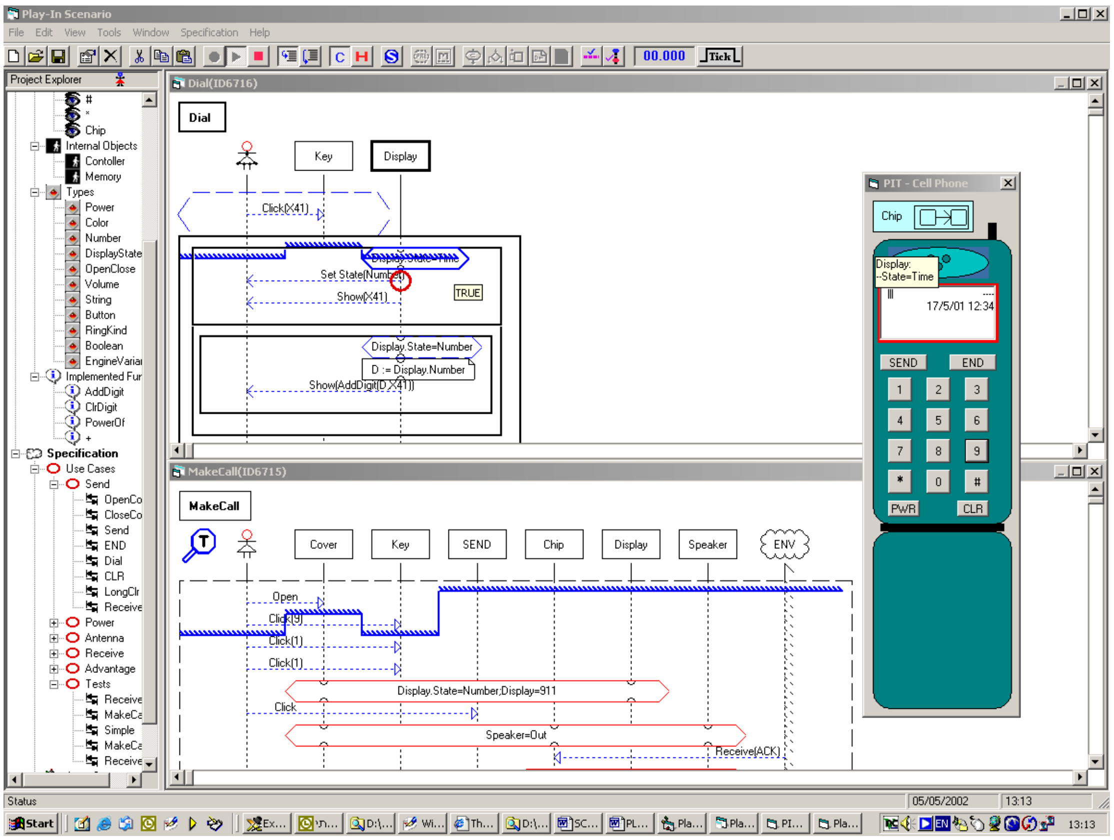

Fig.1: Snapshot of a play-out session of a cellular phone. The next step will be taken from the top universal chart, and is circled in red. The bottom existential chart is being traced (denoted by the 'T' icon). The rectilinear comb-like lines show the current location of each object during execution.