ISBN: 978-972-8939-52-6 © 2011 IADIS

SHOW-AND-TELL PLAY-IN: COMBINING NATURAL LANGUAGE WITH USER INTERACTION FOR SPECIFYING BEHAVIOR

Michal Gordon and David Harel
Weizmann Institute of Science
Rehovot, Israel

ABSTRACT

In search of improving the ways to create meaningful systems from requirements specifications, this paper combines the *showing* and *telling* of how a system should behave. Using scenario-based programming and the language of *live sequence charts*, we suggest how user interaction with the system and user written requirements in natural language can interleave to create specifications through an interface that is both natural and agile.

KEYWORDS

Intelligent interfaces, Requirement engineering, Scenario-based programming, Live sequence charts

1. INTRODUCTION

Scenario-based programming is a method that allows specifying system behavior by describing system scenarios using precise and executable methods. The language of *live sequence charts* (LSC) (Damm and Harel 2001) is one method for these types of descriptions. LSCs add expressive power to earlier sequence-based languages by being multi-modal: an LSC can distinguish what must happen from what may happen, and can specify also what is forbidden from happening. The resulting specification is fully executable.

One of the advantages of LSCs is their use for describing system behavior for reactive systems. The language constitutes a step in the direction of *liberating programming* and making programming more accessible to people who are not programmers, as described in (Harel 2008). The LSC language has been extended with a tool (the *Play-Engine*) that supports intuitive GUI-based methods for capturing the behavior (termed *play-in*) and for executing a set of LSCs (termed *play-out*); see (Harel and Marelly 2003). The present work focuses on introducing an enriched method for play-in, which creates an improved interface for specifying system requirements and for scenario-based programming.

The new method combines natural language parsing methods with user interaction and uses these to create an intelligent user interface. The user specifying the system's behavior can use the method most relevant for the type of behavior he/she is specifying, by *showing* — interacting with the system or by *telling* — describing (parts of) the scenario in a semi-natural language (Gordon and Harel 2009). Any textual requirements thus entered are parsed, so that our *show-and-tell* (*S&T*) play-in algorithm can intelligently guess the user's intention when there are multiple possibilities.

As in real life, a picture is often worth a thousand words and other times a textual description is more appropriate. In analogy, there may be cases when the interaction is simpler to put in words than to demonstrate, or vice-versa. The main contribution of this paper is in combining the two in a natural and semantically meaningful way.

2. THE LANGUAGE OF LSCS AND PLAY-IN

In its basic form, an LSC specifies a multi-modal piece of behavior as a sequence of message interactions between object instances. It can assert mandatory behavior — what must happen (with a *hot* temperature) —

or possible behavior — what may happen (with a *cold* temperature), as well as what is forbidden from happening. In the LSC language, objects are represented by vertical lines, or lifelines, and messages between objects are represented by horizontal arrows between objects. Time advances along the vertical axis and the messages entail an obvious partial ordering. The events that trigger the scenario appear at the top in blue dashed lines; if they are satisfied, i.e., all events occur and in the right order, then the hot events (in red solid lines) must be satisfied too. See (Harel&Marelly 2003).

Play-in is a method for capturing a scenario in an LSC by interacting with a GUI representation of the system. This allows users to operate the final system, or a representation of it, thus 'recording' their behavior by demonstrating it. Although play-in is intuitive and can be easily adopted by users without orientation to programming, it has some drawbacks. First, it requires a pre-prepared GUI of the non-behaving system. Although this is reasonable for some systems (e.g., a general robot with no behavior), in others the specification of the GUI will typically only emerge after considering parts of the system behavior. Another problem is that interactions relevant to the system's behavior often take place among logical objects, for which there can exist only some general representation with possible lower level functions and properties. It is straightforward to interact with a button that initializes a wireless connection, but there is no need to force a graphical representation on the wireless connection when referring to it.

There are also many requirements that are less interactive and more programmatic in their essence, such as specifying a condition or selecting some variable. These are the cases where S&T-play-in becomes relevant: natural language descriptions are quick and simple and can be interleaved with interaction when it is most relevant, as we show below.

Play-in has been extended by controlled natural English (Gordon and Harel 2009). Nouns and verbs are found using the Wordnet dictionary by Miller et. al. (1993) and LSCs are created based on interactions specified in clear sentences, using a specially tailored context free grammar for LSCs. Ambiguities are resolved by the person entering the requirements when the sentences do not translate into meaningful LSCs. The model of the system, its objects and possible low level behaviors, accumulate, and are used to translate further sentences more successfully. In this scheme, no GUI is required at the initial stages of the process, and it can be designed later, based on the system model. This allows the requirements engineer or programmer to concentrate on the system's behavior rather than on its structure and components.

One advantage of the natural language approach is the ability to refer to system objects, conditions, variables and loops in the text in a way that is close to the process performed when the application expert simply writes what he/she wants the system to do.

The natural language play-in of Gordon and Harel (2009) emphasizes writing logical constructs in English, rather than selecting them from menus or dragging them from a graphical toolbar. However, there are cases when using the mouse to point and select is quicker. When a certain knob has a graphical representation and possible low level behaviors, then *showing* the action may be more convenient than *telling* or describing it textually. As when a parent directs a child to return the milk to its proper place in the refrigerator could involve the parent saying 'please return the milk to its place', while pointing to the refrigerator.

3. SHOW-AND-TELL PLAY-IN

The *show-and-tell-play-in* method (*S&T-play-in*) uses online parsing and the state of the current parse to interpret the interaction and integrate it into the scenario-based requirement; i.e., into the LSC that is being constructed on the fly.

An interaction can be interpreted in multiple ways. When an object is selected (from the model or the GUI), either its name or the operation performed on it (e.g., clicked, or turned) may be integrated into the sentence. When an object property is selected, it may be a reference to the property name or to the property value. The parsing is performed bottom-up using an active chart parser similar to that of Kay (1986) and adapted for online parsing as in Jurafsky and Martin (2009), Figure 1a shows the system architecture and Figure 2 provides details of the algorithm.

In each requirement being entered, the indexes represent the locations between the words (as in $_0$ *when* $_1$ *the* $_2$ *user* $_3$). An edge represents a grammar rule and the progress made in recognizing it. We use the common dotted rule, where a dot ('●') within the right-hand side of the edge indicates the progress made in recognizing the rule, and two numbers indicate where the edge begins on the input and where its dot lies.

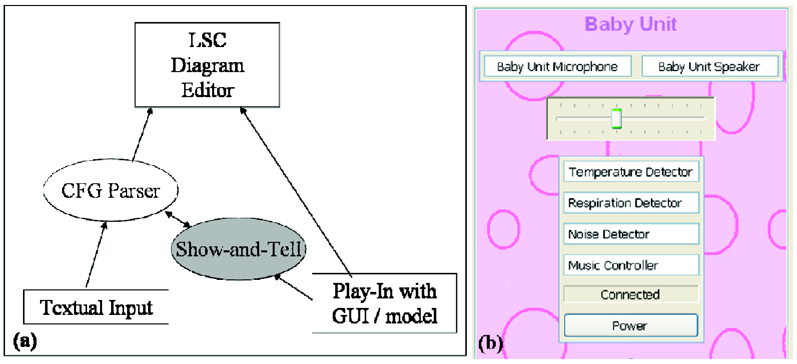

Figure 1. (a) The system architecture. (b) Part of the baby monitor sample application GUI.

Technically, an interaction generates possible edges for the parsing with the object names or operations selected, and the algorithm tests which edges complete the current parse properly. The longest valid completion is selected (Figure 2b), and additional valid possibilities are presented to the user and can be selected by him/her on the fly.

```
procedure ChartParse(chart, agenda)
    while agenda
        current-edge ← Pop(agenda)
        ProcessEdge(current-edge)

Procedure ProcessEdge(edge)
    AddToChart(edge) //if not in it already
    if incomplete(edge)
        ForwardProcess(edge)
    else
        BackwardProcess(edge)
        BottomUpPredict(edge)

procedure ForwardProcess(A → α • B β, [i,j])
    for each(B → γ •, [j,k]) in chart
        //if not in it already
        AddToAgenda(A → α B • β, [i,k])

procedure BackwardProcess(B → γ •, [j,k])
    for each(A → α • B β, [i,j]) in chart
        //if not in it already
        AddToAgenda(A → α B • β, [i,k])

procedure BottomUpPredict (B → γ •, [i,j])
    for each (rule A → B β) in grammar
        AddToAgenda(A → B • β, [i,j])
(a)
```

```
procedure Initialize
    for each rule in grammar
        if rule.leftSymbol is StartSymbol
            Add rule to agenda

procedure ProcessNewWord(word, i) //used for online parsing
    AddToAgenda(W → word •, [i, i+1]) //where W is the rule for word
    ChartParse(chart, agenda)

procedure ProcessUserInput(possibleEdgesList, currentInputLength)
    for each edges in possibleEdgesList
        duplicateChart ← chart
        duplicateAgenda ← agenda
        AddToAgenda(edges)
        ChartParse(chart, agenda)
        possibleTexts ← FindBestParse(chart, currentInputLength)
        bestText ← Max(longest text in possibleTexts list,bestText)
        chart ← duplicateChart
        agenda ← duplicateAgenda

procedure FindBestParse(chart, currentInputLength)
    possibleTexts ← null
    for each edge in chart
        if edge.startIndex is 0 And edge.dotIndex > currentInputLength
        then text ← GetWords(edge) //gets words terminals of an edge
            possibleTexts += text
(b)
```

Figure 2. (a) Parse procedures from Jurafsky and Martin (2009), (b) *ProcessNewWord* is used for online parsing, while *ProcessUserInput* is used to fuse interactions into the parsing.

Consider the example in Fig. 3. The textual requirement at the interaction point is ‘when the user’, so the parse is not complete. However, some edges are already completed in the bottom-up parsing, shown above the sentence part, as can be seen in the left part of the figure, which displays edges as curved lines.

We assume that an interaction creates only complete edges, since when guessing what the user meant, it is reasonable to assume he/she thinks in complete ‘chunks’ of the language; e.g., he/she can refer to a noun, a verb, or parts of sentences that include them. For example, while ’[the button]‘ and ’[clicks] [the button]‘ are reasonable edges and are complete edges in the grammar, ’[clicks] [the]‘ is not, as can be seen in Fig. 3c.

At each step, the algorithm adds one of the edges or a set of edges to the current parse, and tests whether these interaction edges advance or complete any of the parse edges, as shown in Fig. 2 ProcessUserInput.

From the possible completed or advanced edges, the longest one is selected; in Fig. 3b, this would be the top edge.

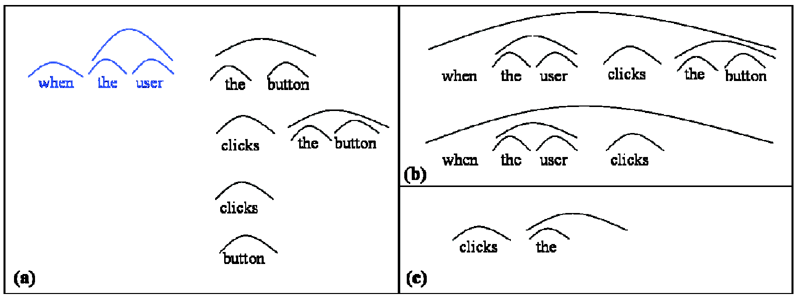

Figure 3. Parse sample for the sentence "when the user" and an interaction of [clicking a button].

4. CASE STUDY: THE BABY MONITOR

We describe some examples from the development process of a baby monitor system, which allows parents to watch over their baby by monitoring respiratory movement and room temperature. We depict interaction outputs in square brackets.

Since the parsing is online and the transformation to LSCs is linear, complete constructs can be directly transformed to their LSC counterparts, which could allow the user to see the LSC created as he/she works. Our current implementation only adds text according to user interaction and generates the visual representation of the LSC when the user completes the requirement and selects the generate LSC option.

The interaction with a GUI (Figure 1b) or with the system model (the list of system objects without their graphical representations) can involve one of three actions: selecting the object, performing an action (e.g., calling a method) or setting/getting a property (attribute) of the object.

One issue that needs to be dealt with when specifying LSC requirements is the question of whether the interaction is meant as a full interaction or just the selection of the object. In the sentence: "*when the user clicks the* [increase-threshold-temperature-button], *the temperature-threshold increases*", the interaction of clicking the button adds only the button name to the already entered text. However, another example that adds a full phrase is: "*when* [the user drags the sensitivity-button], *the sound sensitivity changes to the sensitivity-button value*". Notice that the interaction adds a full phrase of dragging the sensitivity-button and not only the object name because of the different text entered when the interaction occurs.

Another type of information that can be entered is the selection of properties, as in the sentence: "*when the baby-unit temperature changes, if the baby-unit temperature is greater than* [temperature-threshold], *the* [light state changes to blinking]". In the first interaction, only the threshold property itself is added by the interaction, while in the second part, the user changes the state of the light to blinking and the full phrase is added to the sentence,. Sample clips can be found in http://www.wisdom.weizmann.ac.il/~michalk/SaT/.

5. RELATED WORK

The work presented here builds upon the original play-in idea of (Harel and Marelly, 2003), which allows user interaction with a GUI for specifying behavior. User interaction for capturing behavior is also found in many programming-by-example systems, from programming by dragging icons on screen in the Pygmalion, Cocoa or Stagecase environments to constructing grammars with the Grammex system, all described by Cyper et.al. (1993). These systems can be viewed as extensions of macro systems that allow recording a sequence of operations performed by the user and then repeating the sequence while generalizing some aspects of the operations, rather than specifying the full behavior explicitly.

The idea of multimodal interfaces as discussed by Ingebretsen (2010) is already in use in intelligent interfaces for gaming and in smart phones. These modalities include speech, facial expression, body posture, gestures and bio-signals. The question of multimodal synchronization and fusion is interesting for many application areas. Perhaps the initial methods we discuss for requirement engineering synchronizing text (or speech) and user's mouse operations (comparable to gestures) can be useful in other fields too.

Using speech recognition and natural language as an interface for specification has been discussed before. For example, in (Graefe and Bischoff, 1997), interacting with a robot can benefit from a combination, where the context, the knowledge base acquired by the robot at each point, helps to better understand the directions to the robot. Our method currently parses textual requirements, but we have also tested the use of speech recognition dictation with the Microsoft™ Speech API 5.1. In S&T, the text (or speech) is used as the context for understanding the interaction.

6. CONCLUSION AND FUTURE WORK

This paper introduces the idea of combining two interfaces: text and interaction with a GUI into a single intelligent interface that interprets an interaction based on the state of the textual parse to allow generating system requirements in a natural way. We show here only some of the possibilities of combining interaction and text, as the domain of possibilities is fixed by the system interface and the grammar. Other interfaces that have a text/speech interface using a grammar or command-and-control may benefit from a suitably adapted version of the S&T interface.

The method requires further evaluation. One way to do this is to check and compare the time and effort required to create diagrams using only play-in, using only natural language play-in and using S&T-play-in.

One extension we would like to add to the algorithm, is a generic method to create interaction edges using a given grammar and minimal information on the interaction. This may make the method more useful outside the particular LSC-based method for specifying system behavior.

ACKNOWLEDGEMENTS

We would like to thank Smadar Szekely and the other members of our group's software team developing the PlayGo tool used to test the method. This research was supported in part 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 Seventh Framework Programme (FP7/2007-2013).

REFERENCES

Cypher, A. et al, 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

Damm, W. and Harel, D., 2001. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design* 19:1, 45–80.

Gordon, M. and Harel, D. 2009. Generating Executable Scenarios from Natural Language. *Proc. 10th Int. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing'09)*, Mexico, pp. 456–467.

Graefe, V.and Bischoff, R., 1997. A Human Interface for an Intelligent Mobile Robot. *Proc. 6th Int. Workshop on Robot and Human Communication*. Japan, pp. 194–199.

Harel, D. 2008. Can Programming be Liberated, Period?, *IEEE Computer* 41:1, 28–37

Harel, D. and Marelly R., 2003. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach. *Software and System Modeling* 2, 82–107.

Ingebretsen, M. 2010, In the News, *Intelligent Systems, IEEE* 25:4, 4–8

Jurafsky, D. and Martin, J. H. 2009, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition* , Pearson Prentice Hall.

Kay, M., 1986. Algorithm schemata and data structures in syntactic processing. In *Readings in Natural Language Processing*, Morgan Kaufmann, pp. 35–70.

Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D. and Miller, K., 1993. Introduction to WordNet: An On-line Lexical Database. http://wordnet.princeton.edu/.