Evaluating a Natural Language Interface for Behavioral Programming

Abstract—**In behavioral programming, scenarios are used to program the behavior of reactive systems. Behavioral programming originated in the language of live sequence charts (LSC), a visual formalism based on multi-modal scenarios, and supported by a mechanism for directly executing a system described by a set of LSCs. In an exploratory experiment, we compare programming using LSCs with procedural programming using Java, and seek the best interface for creating the visual artifact of LSCs. Several interfaces for creating LSCs were tested, among them a novel interactive natural language interface (NL). Our preliminary results indicate that even experts in procedural programming preferred the LSCs NL interface over the Java alternative, and their implementation times were comparable to those of the other interfaces tested. The results indicate that the NL interface, combined with the scenario-based essence of LSCs, may be a viable alternative to conventional programming.**

I. Introduction

The language of *live sequence charts* (LSCs) [1] is part of a grand challenge to create a new paradigm for programming that would allow more people to define system behavior, by making programming closer to how they think [2]. In the new paradigm of *behavioral programming* the user specifies the system behavior in an incremental way by specifying independent scenarios. The visual language of LSCs allows specifying scenarios of what may happen, what must happen and what must not happen. These scenarios are based on classical sequence diagrams with the additional modalities of must/may/forbid and they can be executed directly using methods from validation and model-checking [3], [4].

Recent research in this developing programming paradigm has focused on execution, debugging, and visualizations. An attempt has also been made to understand the how previous programming experience affects the learnability of the language by interviewing students learning the LSC language [5]. Yet, the claim that the new paradigm may be useful to programmers, and perhaps even to non-programmers, needs to be evaluated too. Since LSCs as a behavioral programming paradigm is conceptually different from procedural languages, the usefulness of the language to "procedural" programmers is still questionable. Additionally, because LSCs are visual in nature, there are many ways to create them: (i) drawing the diagram by dragging and dropping elements; (ii) playing-in the scenario with a graphical user interface (GUI) of the system or with a model thereof [3], [6]; (iii) typing the scenario in a controlled natural language [7] and; (iv) a combination of the last two, a method we call *Show&Tell* [8]. Figure 1 shows a sample LSC scenario and some of the toolbars and views for creating it.

In the current work we evaluate the LSCs language and the available interfaces to create LSCs in an exploratory experiment. Our research questions include (i) Is the natural language interface quickly learnable and how do the various interfaces to the LSC language compare? (ii) How does the LSC language compare with Java (as an example of a common procedural language) in programming times and when considering user preferences?

Recent years have yielded much research comparing programming languages; this comparison can focus on different aspects, ranging from the language features and capabilities, the type of applications the language is useful for, to assessing the human factor criteria as we do [9], [10]. This is done by posing the question of how usable and learnable the language is.

In the current work we focus on the scenario-based properties of the language that also allow the use of a natural language interface, rather than only the visual aspect. Since the LSC language is very different from procedural languages, evaluation based on feature comparison, as is done for Fortran or C [9], is less relevant. Another aspect is that the tool we use for our evaluation, PlayGo [11], is still under development and there are not many programmers who have adequate expertise in using it. Therefore, evaluating the language using the *cognitive dimensions of notation* framework suggested by Green et al. [12] is worthwhile, but difficult for the time being. Historically, claims of new languages being natural have been made, and they are usually hard to prove [13]. In this sense, the current research is preliminary and exploratory in nature. One objective is to collect initial data for the available user interfaces and use the results to improve the finer interfaces, and another is to explore how naturalness or usefulness of the LSC language.

II. LSCs user interfaces

LSCs are based on sequence diagrams and include a set of vertical lines called lifelines that represent the objects in the scenario, and horizontal arrows called messages that represent the interactions between the objects in the scenario; see Figure 1. Time flows from top to bottom, and there is a partial order between the messages. Additional elements, such as synchronization or alternative constructs can be added (see [1], [3] for a more thorough description of the language).

The fact that LSCs are both visual and scenario-based results in multiple ways of creating them, each with its own advantages. We elaborate on the interfaces evaluated in the experiment.

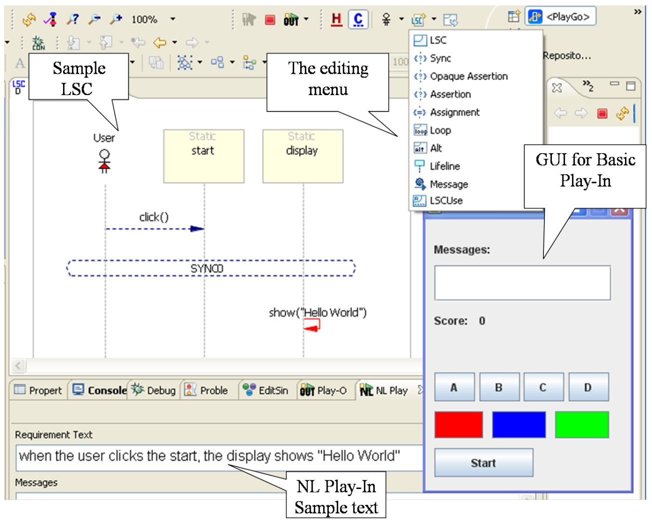
Editing. Since LSCs are visual, they can be created, like many other diagram tools, by adding elements from a menu or dragging and dropping elements from a toolbar as in UML2Tools [14]. LSCs include more information than sequence diagrams; e.g., they include modalities of whether a message may happen or must occur (cold or hot, respectively). This means the user creating the messages must also set the modalities. It also requires the user to tell the system when the monitoring part ends and the execution starts for each scenario (called prechart and main chart, respectively [1], [3]). We call this first interface *Editing* and the main menu for it is shown at the right-hand part of Figure 1.

Basic Play-In. A second way of creating LSCs is the *Basic Play-In*, first defined in [3], [6]. It permits the user to play with the non-behaving system or a mock-up thereof to create the LSC, similar to programming by example (PBE) [15]. For example, to add a message of "click" from the user lifeline to the button lifeline, the user can demonstrate the operation by simply clicking the button. The *Basic Play-In* method is very natural and is made possible due to the scenario-based nature of the LSC language; "demonstrate the scenario to create the requirements". However, it lacks the ability to demonstrate what is cold or hot and additional non-interactive constructs, e.g., conditions, which have to be specified in more standard ways by menu selection. Play-In is different from most PBE systems in that it is domain general and is used to specify rules explicitly and not to infer rules from an example.

Natural Language Play-In (NL-Play-In). Recently, we suggested a natural language play-in interface for LSCs *(NL-Play-In)* [7]. This interface uses a context free grammar to create a controlled natural language for LSCs. Clearly, natural language may include multiple ways to specify the same semantics, therefore the interface prompts the user to resolve ambiguities when they exist. *NL-Play-In* combined with the scenario-based nature of LSCs creates the possibility to "program" by writing separate requirement sentences in (controlled) English. It can also be spoken rather than written, however, the motivation is different than the motivation of languages such as *spoken Java* [16] developed to help programmers with repetitive strain injuries. While in *spoken Java* it is necessary for the user to speak a programming language, in *NL-Play-In* the user writes behavioral requirements rather than a program. For example, to create the LSC in Figure 1, one can write *"when the user clicks the start, the display shows "Hello World""*.

The *NL-Play-In* parser helps the person writing the requirements (who may not even be a programmer) connect between the different requirements by making sure she refers to existing objects and methods or realizes she is adding new ones.

The process includes a stage of grammatical parsing, with the addition of asking the user to resolve any grammatical ambiguities. This is followed by the analysis of the requirement, using the model that serves as a knowledge base and assists in helping the writer make the connection between the different scenarios. The modalities (may/must), the prechart/mainchart indication and the conditions, are added automatically by *NL-*



when the user clicks the start, the display shows "Hello World"

Fig. 1. PlayGo Environment, a sample LSC and the natural language that created it. Also visible on the right is the experiment GUI and on the top the editing menu.

Play-In based on the sentence, avoiding the need to handle them explicitly as in the simpler interfaces. For additional examples of the type of sentences accepted by the *NL-Play-In* and elaborations on refer to [17].

Show&Tell (S&T). An additional method recently developed is *Show&Tell (S&T)* [8]. This method is a combination of *Basic Play-In* and *NL-Play-In*. It is more than a naive combination; rather, the play-in interaction is interpreted based on the textual context. A similar combination of voice and gestures has been used for managing graphical spaces with "put-that-there" [18]. *Show&Tell* integrates text and GUI manipulation to assist in the creation of system requirements. The user can enter her requirements textually but also use the advantages of play-in to interact with the system, in the midst of the requirement specification process, and create parts of the sentence (and later the respective diagram) by interaction without explicitly writing object names or actions.

The interaction is interpreted depending on the current parse of the text. For example, if the text entered so far (prefix text) is `when` and the interaction is <*clicking the button*>, the suggested texts would include `when` <*the user clicks the button*>. However when the prefix text is `when the user`, the same interaction will add the suggestion of <*clicks the button*> or <*clicks*>. Using the grammar parse state and the interaction possibilities, the system will not suggest to add an unreasonable addition that will not make sense grammatically.

III. Experiment

The experiment we carried out was meant to test which of the LSC interfaces is preferable and hence more natural to programmers. Our hypothesis was that *NL-Play-In* would be preferable to *Editing* and *Basic Play-In*, and that the combination of the two in *(S&T)* would be even better.

Since the language of LSCs and the scenario-based approach to programming is new to most programmers, we want to also compare LSCs and a procedural language like Java. In scenario-based programming the programmers think

TABLE I
PARTICIPANTS PREVIOUS EXPERIENCE

	Java Exp.	LSC exp.		Java Exp.	LSC exp.
1	> 5 years	2-5 projects	7	> 5 years	Read only
2	1-2 years	2-5 projects	8	None	1 course project
3	> 5 years	2-5 projects	9	1-2 years	1 course project
4	> 5 years	2-5 projects	10	> 5 years	1 project
5	1-2 years	Read only	11	1-2 years	1 course project
6	> 5 years	> 5 projects	12	C/C++	Pen and paper LSCs

about each scenario separately and the execution mechanism is responsible for handling the connection between the scenarios (see [3], [4] for details regarding the execution). We believe that this fact will make programming simpler and hypothesize that LSCs will be easier than Java, especially when the GUI and the model are given, and the task is to program only the behavior of the system and not its objects.

A. Experiment Design

Participants. Our preliminary experiment involved 12 programmers. All the participants except three were familiar with the LSC language, as they attended the 2009 graduate course on executable visual languages described in [19] or a similar course given two years later, which included the LSC language. Those who did not attend the course were familiar with LSCs by researching or working on some aspect of them. Some had some experience with the PlayGo tool, but none were experienced with the *NL-Play-In* method or the *(S&T)*. Table I summarizes the participants' previous experience.

Tool and Tasks. The experiment was designed to test the objectives using the PlayGo tool, an Eclipse based product that implements the LSCs approach over Java classes using UML and AspectJ [11]. Java was chosen as the procedural language to test, since the tasks were performed in the same IDE and using the exact same GUI and classes, while only behavior among the objects was implemented in LSCs or Java.

The experiment included a part in LSCs and a part in Java (three participants started with the Java part first and the others started with the LSCs tasks first). Both parts included adding unknown behavior to a simple game for teaching letters and colors. An example of the required behavior for the tasks is shown in Table II and the provided GUI is in Figure 1. Tasks T1 and T2 were used in the LSC part only (except for two participants in a pilot experiment that helped establish that programming these in Java was unnecessary). The objectives of the first two tasks were to teach all four interfaces to the participants by requiring them to use all the interfaces and later to compare the interfaces. In task T3 the participants chose their preferred interface method in the LSC part, and implemented a comparable requirement in Java.

During the experiment, participants were provided with tutorials on all the interfaces and examples of natural language sentences for a different example system. They were encouraged to ask questions when they could not complete a task or had problems understanding or using the interfaces. They were also asked to explain the difficulty they encountered when spending a long time on some task or part of it.

TABLE II
SAMPLE TASKS

T1	The LSC in Figure 1.
T3a	When the game starts, the display shows a random letter and displays a random color, the player is expected to click on the button with the displayed letter, and on a button with the displayed color. When he's successful, the display show "Success".
T3b	When the user succeeds the score is increased, if he fails the score is decreased.
T3c	The game is won when the score reaches 5, at which point the display shows "You Win" with a yellow background.

Evaluation. To evaluate the interfaces, we asked the participants to time each of the tasks. In addition, we asked the participants to answer questionnaires following each task.

B. Results

Task Times. When comparing times for creating the same LSCs using the different methods in T1, *Editing* took the longest time (7.6 ± 2.9 minutes), and the other methods had no significant difference (*Play-In* 5.3 ± 3.5, *NL-Play-In* 5.25 ± 3.6 and *(S&T)* 4 ± 1.7 minutes). A similar effect was found in T2 for those participants who completed all four interfaces (five out of the twelve, due to time constraints of the experiment).

The Java T3 task took comparable time (29.6 ± 8.8 minutes) to the equivalent LSC T3 tasks (25.7 ± 5.5 minutes), for the seven participants who completed all tasks in both Java and LSCs. For those who started with Java, the time to completion was longer, but not significantly different than those who started with LSCs; this is reasonable considering that the LSC-equivalent tasks were preceded by the two teaching tasks, T1 and T2 that introduced the system.

Considering that all programmers were experienced with Java, and less so with LSCs, this suggests that the new interfaces and language are natural and easily learnable.

Subjective Questions. When asked what their preferred method was for creating LSCs for the third task, nine out of twelve participants chose the *NL-Play-In*, and said it was the quickest. Of those who chose a different method, one chose the *(S&T)* for T1 and *NL-Play-In* for T2, and the other two preferred *Editing* and *Basic Play-In*, while one mentioned she did not understand what was expected from her, she did not figure out the NL-Play-In, did not ask the experimenter for help and gave up on the Java part almost completely.

According to the verbal interview the *NL-Play-In* felt quickest to almost all participants and did not require changing the medium of entering data; i.e. they did not have to move their hands away from the keyboard.

Regarding the LSC language in comparison to Java, ten out of the twelve who completed almost all of the Java task wrote that the LSC language was easier for the given task than Java. One participant could not decide which was better, and another chose Java as the easiest. The latter participant did not use the *NL-Play-In* for the LSC T3 task, but rather the *Editing* and *Basic Play-In*.

Several participants mentioned that the *Editing* method gives rise to more typing errors, and two mentioned that *(S&T)* could be useful to avoid typing an object name and to avoid typos. Several mentioned that auto-completion in the *NL-Play-In* would have made the task simpler for them. The information for the auto-completion exists in part in the classes methods and properties.

Analyzing the answers of the two participants who did not prefer the *NL-Play-In* shows that typing natural text that translates automatically into LSCs felt uncomfortable because of "uncertainty how to phrase the sentences", and both mentioned that sentence templates or additional practice might have made the task easier. They also mentioned that error-fixing suggestions for *NL-Play-In* were insufficient. The other participants learned pretty quickly the suggested templates and were able to resolve most errors in T3. A representative participant mentioned "I was getting confused with NL" in T2, explained later his choice of using NL: "When you get used to its English, it's quite fast to use".

Additional Observations. It seems that programmers who are used to creating code by typing text appreciate a similar interface even when creating diagrams. Second, switching between the mouse and keyboard is not so convenient for experienced programmers. Entering the diagram in edit mode, selecting elements and then typing in element names or messages was more time-consuming, and required much switching between interfaces.

Basic Play-In avoids some of the diagram clicking, but still requires clicking on the GUI and in other cases editing. *(S&T)*, which we thought would benefit from the advantages of both *Basic Play-In* and *NL-Play-In*, actually suffered from the need to switch between them. Some of this may also have been due to some performance difficulties of PlayGo during the experiment. Most participants thought that *NL-Play-In* was quickest and simplest for them, since it provided a means of creating the entire diagram by a single action, many also mentioned it was "fun". *Editing* and *Basic Play-In* required more specialization in LSCs by directly setting the modalities and synchronization objects.

One of the key features of LSCs is that the order of events matters in execution. In T3a, the order between the user selecting a color and a letter was not mentioned specifically in the requirements, which caused participants to avoid thinking about it in the NL task, and thereby set a single order that was accepted. In Java, many lines of code were required to check that the player clicks on the two options, and the question of order was discussed explicitly by three of the participants. In the final implementation the order in the Java game did matter for all but one participants. This we believe is linked to the fact that LSCs can be underspecified, and allow the programmer to avoid considering such issues unless explicitly required.

IV. Conclusion and Future Work

This exploratory experiment demonstrates that the Natural Language interface for LSCs is viable, quickly learnable and most favorable to programmers than other interfaces. It also confirms that the language of LSCs is comparable to Java in ease of programming, for tasks similar to the ones given, especially those requiring multiple GUI listeners.

One question we would like to test in the future is whether the *(S&T)* method indeed suffered from the necessity to stop typing in order to point and perform actions or rather that it will never be quicker than typing for programmers who blind-type, but it may be the best choice for non-programmers.

In the future it would be interesting to test the ease of use of LSCs in more complex tasks, and for non-programmers.

Acknowledgment

The first-listed author would like to thank Jacob Kiwkowitz and Shahar Maoz for their constructive comments. This research was funded by an Advanced Research Grant from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). In addition, part of this research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

References

[1] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.
[2] D. Harel, "Can Programming Be Liberated, Period?" *IEEE Computer*, vol. 41, no. 1, pp. 28–37, 2008.
[3] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
[4] S. Maoz and D. Harel, "From Multi-Modal Scenarios to Code: Compiling LSCs into AspectJ," in *SIGSOFT FSE*, 2006, pp. 219–230.
[5] G. Alexandron, M. Armoni, and D. Harel, "Programming with the User in Mind," in *Proc. of Psychology of Programming Interest Group Annual Conf. (PPIG)*, 2011.
[6] D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach," *Software and Systems Modeling*, vol. 2, no. 2, pp. 82–107, 2003.
[7] M. Gordon and D. Harel, "Generating Executable Scenarios from Natural Language," vol. 5449, 2009, pp. 456–467.
[8] ——, "Show-&-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior," in *Proc. IADIS Interfaces and Human Computer Interaction*, 2011, pp. 360–364.
[9] N. M. Holtz and W. J. Rasdorf, "An Evaluation of Programming Languages and Language Features for Engineering Software Development," *Engineering with Computers*, vol. 3, pp. 183–199, 1988.
[10] J. Howatt, "A Project-Based Approach to Programming Language Evaluation," *SIGPLAN Not.*, vol. 30, pp. 37–40, July 1995.
[11] D. Harel, S. Maoz, S. Szekely, and D. Barkan, "PlayGo: Towards a Comprehensive Tool for Scenario-Based Programming," in *Proc. of the IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2010, pp. 359–360.
[12] A. F. Blackwell and T. R. G. Green, "A Cognitive Dimensions Questionnaire Optimised for Users," in *Proc. of the 12th Annual Meeting of the Psychology of Programming Interest Group*, 2000, pp. 137–152.
[13] S. Markstrum, "Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress," in *Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '10, 2010, pp. 1–5.
[14] "Eclipse UML2 tools," http://www.eclipse.org/modeling/mdt/?project=uml2tools.
[15] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, Eds., *Watch What I Do: Programming by Demonstration*. Cambridge, MA, USA: MIT Press, 1993.
[16] A. Begel and S. Graham, "Spoken programs," in *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2005, pp. 99 – 106.
[17] "Natural language example in playgo," http://www.weizmann.ac.il/mediawiki/playgo/.
[18] R. A. Bolt, "'put-that-there": Voice and gesture at the graphics interface," *SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 262–270, Jul. 1980.
[19] D. Harel and M. Gordon-Kiwkowitz, "On Teaching Visual Formalisms," *IEEE Software*, vol. 26, pp. 87–95, 2009.