Evaluating Live Sequence Charts as a Programming Technique for Non-programmers

Michal Gordon
Weizmann Institute of Science
Rehovot, Israel
michal.gordon@weizmann.ac.il

David Harel
Weizmann Institute of Science
Rehovot, Israel
dharel@weizmann.ac.il

***Abstract*—Behavioral programming is a recent programming paradigm that uses independent scenarios to program the behavior of reactive systems. Live sequence charts (LSC) is a visual formalism that implements the approach of behavioral programming. The approach attempts to liberate programming by allowing the user to program the behavior of reactive systems by scenarios. We would like to evaluate the approach and seek the naturalness of the best interface for creating the visual artifact of LSCs. Several such interfaces, among which is a novel interactive natural language (NL) interface, exist. Initial testing indicates that the LSCs' NL interface may be preferred by programmers to procedural programming and that in certain tasks LSCs may be a viable and more natural alternative to conventional programming. Many challenges exist in trying to prove the intuitive and natural nature of a new programming paradigm, which differs from others not only in syntax but in many other respects. We describe these challenges in this proposal.**

I. Introduction

In the new paradigm of *behavioral programming* the user specifies system behavior in an incremental manner by specifying scenarios. The language of *live sequence charts* (LSCs) [1] was the basis of this paradigm, and is part of the grand challenge to create a new approach to programming that would allow more people to define system behavior easily, by making programming closer to how they think [2]. The visual language of LSCs allows specifying scenarios of what may happen, what must happen and what must never happen. These scenarios, based on classical sequence diagrams with additional modalities, can be executed directly [3], [4].

The idea that the new paradigm may be useful to non-programmers needs to be evaluated. 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 a model thereof [3], [5]; (iii) typing or speaking the scenario in a controlled natural language [6] and; (iv) a combination of the last two methods, a method we call *Show & Tell* [7].

In the current proposal we would like to evaluate the LSC language and the available interfaces to create LSCs by non-programmers. Our objectives include (i) comparing the various interfaces for creating LSCs and (ii) evaluating whether LSCs can indeed provide a natural way to program; can they liberate programming from the need to write down symbolic artifacts, from the need to specify requirements separately from the program and from the need to structure the behavior according to the system's structure, as discussed in [2].

Recent years have yielded much research comparing programming languages; this comparison has focused on various aspects, ranging from the language features and capabilities, the type of applications for which the language is useful, to assessing the human factor criteria which is what we would like to do [8], [9]. The latter aspect can be addressed by posing the question of how usable and learnable the language is.

In the proposed research, we would like to 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. We cannot evaluate the ideas by comparing to programming languages based on feature comparison, as is done for Fortran or C [8]. Rather, we have to deal with evaluating a language that is implemented in *PlayGo* [10], an Eclipse-based tool still under development that is not yet familiar to non-programmers. Also, we need to develop a method to teach the language to non-programmers in an intuitive way. It is therefore necessary to either design an experiment that will include also a teaching phase but will not be too long, or consider case studies on specific persons and projects. We also describe an exploratory experiment conducted on programmers familiar with the language that compared between the interfaces and between the LSC language and the procedural language of Java.

II. LSC Interfaces

LSCs [1], [3] 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

978-1-4673-1859-4/12/$31.00 © 2012 IEEE

USER 2012, Zurich, Switzerland

Authorized licensed use limited to: Weizmann Institute of Science. Downloaded on August 21,2022 at 13:19:06 UTC from IEEE Xplore. Restrictions apply.

elements, such as synchronization or alternative constructs can be added too (For a more thorough description of the language refer to [3]).

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 creation interfaces below.

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 [11]. LSCs are a lot richer 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 consider and 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*.

Basic Play-In. A second way of creating LSCs is *Basic Play-In*, first defined in [3], [5], and similar to programming by example (PBE) approaches [12]. *Basic Play-In* permits the user to play with the non-behaving system or a mock-up thereof through a GUI to create the LSC. The demonstration on the GUI is used to create parts of the formal rules in the form of an LSC. It is not generalized as in PBE systems, and it is domain-general relative to the GUI of the system, but specific since it creates LSC constructs. 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, to demonstrate what is cold or hot and to specify additional non-interactive constructs, e.g., conditions, one has to use more standard ways of menu selection.

Natural Language Play-In (NL-Play-In). Recently, we suggested a natural language play-in interface for LSC *(NL-Play-In)* [6]. This interface uses a context free grammar to create a controlled natural language for LSC. NL-Play-In is similar to using natural language to create code as is done in *spoken Java* [13], which was developed to help programmers with repetitive strain injuries create Java code by using a speech interface and a grammar built for Java. However, the language is more natural since it translates to scenarios rather than code and does not require knowledge of Java.

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 LSC, creates the possibility to "program" by writing requirement sentences in (controlled) English. For example, to automatically create the LSC in Figure 1, one can write *"when the user clicks the c-button, the light state changes to on and if the display mode is not time, the display mode must change to time"*.

The *NL-Play-In* parser helps the person writing the requirements (who may not be a programmer) to connect the different requirements by making sure she/he refers to existing objects and methods or realizes she/he 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, which serves as a knowledge base and helps the user make the connection between the different scenarios. The modalities (may/must), the prechart/main-chart indication and the conditions, are added automatically by *NL-Play-In* based on the sentence, thus avoiding the need to handle them explicitly as in *Editing* or *Basic Play-In*.

Show&Tell (S&T). An additional method recently developed is *Show&Tell (S&T)* [7]. This method is a subtle combination of *Basic Play-In* and *NL-Play-In*. The play-in interaction is interpreted based on the textual context. The user can enter her/his requirements textually, but can also use the advantages of play-in to interact with the system in the midst of the requirement specification process, creating parts of the sentence (and later the corresponding diagram) by interaction without explicitly writing object names or actions. A similar combination of voice and gestures (rather than text and play-in) has been used for managing graphical spaces with "put-that-there" [14]. Voice pronouns such as "that" and "there" were used to allow the integration of gesture information instead of the voice reference. For example, saying "that" and pointing to an object would integrate the object into the command. *Show&Tell* integrates text and GUI manipulation to assist in the creation of system requirements, rather than preform commands. However, it may also benefit from voice integration.

The interaction is interpreted depending on the current parse of the text, there is no need for textual placeholders, although they could be integrated in a later version. 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 suggest to add only reasonable additions that will make sense grammatically.

III. Empirical Evaluation Proposal

The experiment we want to perform will test which of the LSC interfaces is preferable and hence more natural to non-programmers, and try to prove that describing system behavior in LSC is feasible, natural and intuitive, even for the non-programmers. Our hypothesis is that *NL-Play-In* would be preferable to *Editing* and *Basic Play-In*, and that the process is indeed natural, since scenarios are close to how humans describe and consider behavioral requirements.

Authorized licensed use limited to: Weizmann Institute of Science. Downloaded on August 21,2022 at 13:19:06 UTC from IEEE Xplore. Restrictions apply.

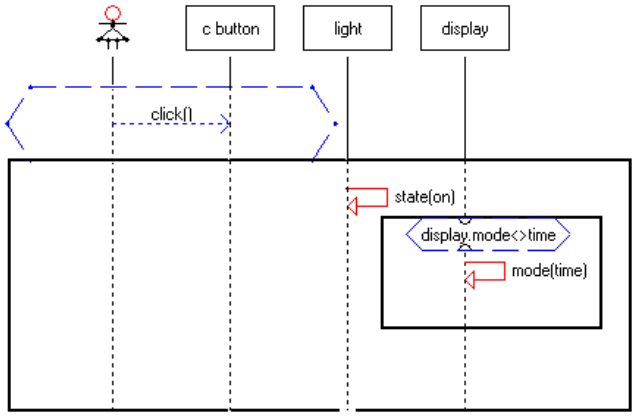

Figure 1. A sample LSC. Time goes from top to bottom; each vertical lifeline represents an object and the horizontal arrows are the messages between the objects. The prechart (blue dashed hexagon) specifies that if the messages in it occur in the correct order, the main chart (solid black rectangle) must be executed.

IV. Past Evaluations

In an exploratory experiment with non-programmers, we compared these interfaces for creating LSCs and also compared programming simple behavioral requirements in LSC to programming the same requirements in Java. The system GUI and low level methods were provided in both cases in advance, and the programmers had only to add the additional behavioral requirements. These included some simple game logic. Since the language of LSC and the scenario-based approach to programming is relatively new, we compared LSC to Java as a sample procedural language.

Participants. The experiment was performed on 10 programmers with experience of between 2-10 years with Java, and understanding of LSC as taught in a course given on visual languages, similar to the course described in [15]. Of the interfaces, *NL-Play-In* was preferred by almost all the participants. Those who did not prefer it mentioned technical problems and personal preference of exact and completely known syntax.

During the experiment, we encountered a necessity for personal assistance to the participants, especially during the learning phase of the *NL-Play-In* interface, and we believe this is a point that must be addressed in any experiment with non-programmers. All the programmers started out with knowledge of LSC at a level taught in a course. This prerequisite shortened the learning phase in the experiment. The question of how much knowledge of LSC is required to complete the task remains to be evaluated and pertains to the learnability of the language. It will affect any experimental protocol with non-programmers, since the preparation for the experiment should reflect this lack of knowledge.

Tool and Tasks. This exploratory experiment was preformed in *PlayGo*, an Eclipse-based product that implements the LSC approach over Java classes using UML and AspectJ [10]. Most of the programmers were familiar with the Eclipse IDE and therefore with the conventional interface. For non-programmers, this prior knowledge may not be available and any such deficiency should be reflected in the available tutorials and the learning phase in the experiment. The task we created for the LSC and Java experiment was as small as possible and included some tasks used for learning.

The complete experiment took over three hours, with at least half of the time devoted to LSC. Not all the participants were able to complete the LSC and the Java, and since we believe non-programmers with no LSC understanding may need more time, it is essential to plan an experiment with reasonable tasks and length. Tasks should be as small as possible to allow cleaner analysis, but complex enough to allow testing the incremental, modular and effective nature of the LSC language.

Evaluation. In our experiment we compared task times and found that *NL-Play-In* and *Play-In* were quicker than the *Editing*, and that *S&T* suffered from implementation bugs and was considered inconvenient, since it required dispersed use of both the keyboard and the mouse. This may be different for programmers who type blindly and for non-programmers who may type slower and prefer mouse shortcuts. We also found that the Java task implementation time was comparable to the implementation time in LSC. In answers to questions during and at the end of the experiment, most of the programmers preferred the *NL-Play-In* interface over the other interfaces and also over Java and had a subjective feeling that *NL-Play-In* was quicker.

Errors were hard to account for, mainly because we did not ask the participants to execute their artifact due to time constraints. In a full experiment with known tasks, small enough, we can collect data on time, errors, and the participants' subjective feeling of intuitive and natural programming. Moreover, we can use Green's cognitive dimensions questionnaire [16], especially on viscosity, provisionality, diffuseness, hard mental operations, and closeness of mapping. Some issues of significance have to do with: how to rate errors, success, and how to assess the intuitive nature of the language and the interfaces.

V. Future Directions

We believe that questions regarding how intuitive and natural a language is are tough [17], but understanding them can help create a better programming language. This seems to be definitely worthwhile, considering the dream of liberating programming.

We would like to plan an experiment with quantitative and convincing results that will support the learnability of the language. We would also like to test the use of LSC in tasks for non-programmers. Such experiments will require a protocol that teaches the language and its interfaces and also tests the usability in planned or in open-ended tasks chosen

Authorized licensed use limited to: Weizmann Institute of Science. Downloaded on August 21,2022 at 13:19:06 UTC from IEEE Xplore. Restrictions apply.

by the users. Valid methods for assessing the outputs of such experiments need to be developed. We would like to create tasks that will enable us to assess the success of the users, as well as how 'fun' and usable programming with the language felt to them. Also, any such experiment needs to consider that programming in LSC should not be considered only as a method for creating visual executable diagrams but rather also as a way to test the ideas of behavioral programming and the naturalness of thinking in separate but incremental scenarios. We would like to be able to test and assess the language in tasks that require incremental system behavior and compare them to incremental development in other programming environments.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their productive 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, *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] 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.

[6] M. Gordon and D. Harel, "Generating Executable Scenarios from Natural Language," vol. 5449, 2009, pp. 456–467.

[7] ——, "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.

[8] 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.

[9] J. Howatt, "A Project-Based Approach to Programming Language Evaluation," *SIGPLAN Not.*, vol. 30, pp. 37–40, July 1995.

[10] 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.

[11] "Eclipse UML2 tools," http://www.eclipse.org/modeling/mdt/?project=uml2tools.

[12] 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.

[13] A. Begel and S. Graham, "Spoken programs," in *IEEE Symp. on Visual Languages and Human-Centric Computing*, 2005, pp. 99 – 106.

[14] 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.

[15] D. Harel and M. Gordon-Kiwkowitz, "On Teaching Visual Formalisms," *IEEE Software*, vol. 26, pp. 87–95, 2009.

[16] 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 (PPIG 2000)*, 2000, pp. 137–152.

[17] 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*, 2010.

Authorized licensed use limited to: Weizmann Institute of Science. Downloaded on August 21,2022 at 13:19:06 UTC from IEEE Xplore. Restrictions apply.