BEHAVIORAL PROGRAMMING WITH A SUBSET OF NATURAL LANGAUGE: AN EVALUATION STUDY

Michal Gordon-Kiwkowitz[1] and David Harel[2]
[1]*Holon Institute of Technology, Holon, Israel*
[2]*Weizmann Institute of Science, Rehovot, Israel*

ABSTRACT

We consider the idea of behavioral programming by writing controlled natural language requirements, which are then compiled directly into executable code. Our motivation is to bridge the gap between system requirements and a final system. We claim that formal structured natural language requirements can serve as the means and the end to programming the behavior of reactive systems, if one utilizes a fully executable language that supports story-like sentences, such as *live sequence charts* (LSC). We use natural language processing methods combined with user interaction in order to understand English requirements, and translate them into LSCs, disambiguating and clarifying intentions with the user when necessary. We test a multi-modal user interface we term *show & tell*, where natural language is interspersed with *play-in*, and model-based co-reference resolution. Our approach is domain general and builds the underlying model of the system leading to direct execution. The paper presents novel evaluation studies that show its potential as an effective tool for natural language programming of reactive system behavior.

KEYWORDS

Live Sequence Charts, Controlled Natural Language, Modeling, Behavioral Programming, Scenario-Based Programming, Human Computer Interfaces

1. INTRODUCTION

The language of *live sequence charts (LSCs)* (Damm & Harel, 2001) and the subsequent challenge of "liberating programming", by which system behavior will be easily definable by making programming more similar to how people think (Harel, 2008), are the motivation for this paper. Many languages and interfaces have been invented in recent years for end-user programming (Cypher et al., 2010; Schlegel et al., 2019). Here we evaluate an interface for reactive systems' programming by engineers. LSCs are an executable visual formalism that describes natural "pieces" of behavior in a manner similar to telling someone what they may and may not do, and under what conditions, using short scenarios. The question we address here is: can we capture the desired behavior of a system in a far more natural style than is common? We want a style that is intuitive and less formal, and which can serve both as the system's behavioral specification and as its final executable artifact, bridging the gap between requirements and implementation.

Most system requirements are short and fragmented, and require further analysis in order to tie them together; in this sense they can be considered appropriate for automatic execution using LSC. The method we propose, termed *controlled natural language play-in (CNLPI)* assists requirement engineers, using a parser and a dialog system that guide them in writing structured controlled natural language (CNL) requirements, with a system model, that can be directly executed.

Natural language processing (NLP) has been used in computer-aided software engineering tools to assist human analysis of the requirements (Zhao et al., 2020). One such use is in extracting the classes, objects, methods or connections from the natural language description (Bryant & Lee, 2002; Mich, 1996). Another is applying NLP to use cases, in order to create simple sequence diagrams with messages between objects (Segundo et al., 2007), or to assist in initial design (Drazan & Mencl, 2007). A tool has been developed for capturing requirements in controlled natural language, backed by a formal requirements analysis engine and to automatically generate a complete set of requirements-based test cases (Moitra et al., 2019). Another tool has

introduced the generation of core requirements from product requirements written in a natural language (Reinhartz-Berger & Kemelman, 2020).

Neural networks have been used to assist modeling elements of use case documents (Madala et al., 2020), and statistical methods have been used to transform natural language specifications into program input parsers (Lei et al., 2013) and into LSCs, using the CNL we describe (Tsarfaty et al., 2014). NLP has also been used to parse requirements and to extract executable code by generating object-oriented models using a two-level grammar (Bryant & Lee, 2002), and for reasoning in the Attempto language (Fuchs & Schwitter, 1995). NLP is widespread in end-user programming, for example, in web programming (Cypher et al., 2010). However, we note that in these efforts the resulting code is not scenario-based ; in most cases it describes the behavior of each object separately under the various conditions, and is usually limited to sequential behavior. The resulting code is rarely aligned with the system's requirements, and neither do these systems allow user interaction for complete system requirements translation.

An end-user interface for programming in Python was also developed and evaluated for programmers and non-programmers (Schlegel et al., 2019). Recently, deep learning algorithms for natural language programming have also been introduced (Desai et al., 2016). However, they require a large corpus of data for training.

In (Harel, 2001; Harel & Marelly, 2003), the mechanisms of *play-in* and *play-out* were described as means for making programming more practical. In play-in, the system engineer specifies scenarios by playing-them-in directly from a graphical user interface (GUI) of the system being developed. The developer interacts with the GUI that represents the objects in the as-of-yet behavior-less system, in order to show, or teach, the scenario-based behavior of the system by example (e.g., by clicking buttons, changing properties or sending messages). Play-in was developed independently, but is similar to other methods like programming by demonstration (PBD) (Cypher et al., 1993, 2010) in that it is a way of demonstrating behavior, and the behavior is generalized into recurring behavior. In play-in, demonstration is used to specify explicit requirements, usually GUI-based, in a scenario-based fashion. Play-in includes negative and possible scenarios, and always results in a full executable. The notion of demonstrating an example and having the system generalize it is but one facet of play-in. This paper focuses on the controlled natural language as an interface for programming, rather than on the idea of generalizing from examples.

In *play-out*, the execution phase, multiple scenarios are integrated into a fully executable artifact (Harel & Marelly, 2003). Since overall system behavior is defined by an incremental set of separate behaviors, it is usually underspecified, and may contain contradictions. Over the past few years, advanced algorithms for executing a set of LSC scenarios have been developed, employing synthesis (Harel & Segall, 2012), planning algorithms (Harel, 2006) and model-checking (Harel et al., 2002) to help alleviate this problem. In the present paper we focus on methods for creating LSCs, rather than executing them.

Thus, the main contribution of the current paper is an evaluation conducted to better understand the advantages and shortcoming of controlled natural language programming with LSCs as compared to other LSC interfaces and the ability to teach them.

The paper is organized as follows: Section 2 describes the LSC language and the CNL method for creating LSCs. Section 3 describes the evaluation studies and Sections. The paper concludes with Sec. 4.

2. THE LANGUAGE OF LSC

An LSC specifies behavior, stating what can be done, what must be done and what is forbidden. The behavior is a sequence of messages between object instances, with additional constructs that can add conditions. Thus, an LSC can assert mandatory behavior - what must happen, possible behavior - what can, or may happen, as well as forbidden behaviors and other possibilities and combinations. The LSC language (Damm & Harel, 2001; Harel & Marelly, 2003a) extends message sequence charts (MSC) (ITU: International Telecommunication Union, 1996), which in the UML are termed sequence diagrams (UML, 2007). In both formalisms, 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.

A sample LSC is seen in Figure 1. The cold monitored events (dashed blue arrows) mean that the events are monitored and if they occur, the next event in the partial order should be monitored or executed. A hot execute event (solid red arrows) means that the system should perform the event eventually. The LSC language also includes conditions, assertions, loops and switch cases. Additionally, in (Harel & Marelly, 2003) the language was enriched to include also time, scoped forbidden elements and symbolic instances that allow reference to a class in general, not only to an instance, with the instantiation being carried out at runtime.

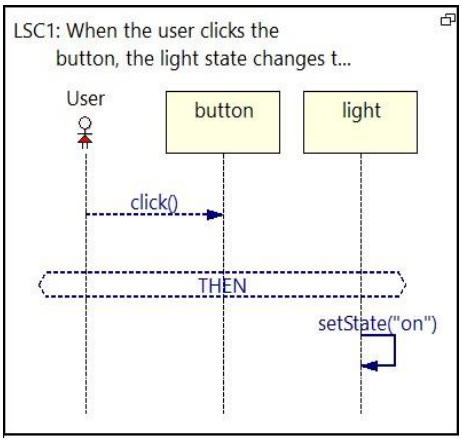

Figure 1. A sample LSC. The monitored message click is shown as a dashed blue arrow. A THEN synchronization invariant synchronizes executed events to occur only after the monitored events. The executed event here setState(''on'') is depicted as a solid red arrow. One possible sentence that creates this LSC is: "When the user clicks the button, the light state changes to on"

A set of LSCs is executed using the *play-out* mechanism (Harel & Marelly, 2003) . *Play-out* monitors at all times what must be done, what may be done and what is den and proceeds accordingly, yielding a complete execution of the LSC specification. The execution can be naïve, considering only the current state and progressing by choosing arbitrarily from all possible next events to be triggered, or use methods from model-checking or planning (Harel et al., 2002), to look-ahead and choose an execution order in a smarter fashion.

In this contribution, we focus on the interactive interface for programming LSCs, the basic one being *Play-in*. While programming by demonstration is meant for programming software applications using preferences, macro recordings, scripts, etc.; play-in uses demonstration to specify formal rules explicitly in order to describe a system's behavior. We believe play-in is more relevant to the requirement's engineering community, than to those making changes to customized software.

The play-in method works well for some operations, mostly those that involve the GUI. More complex operations, such as the "if-then" or "when some X then some Y" are less convenient to demonstrate, and are done, e.g., using menus. The CNL interface attempts to alleviate these difficulties, taking a further step towards liberating programming.

2.1 Controlled Natural Language Interface

The natural language interface for LSCs consists of a context free grammar (CFG) bottom-up parser (Jurafsky & Martin, 2008), a dialog system, and a knowledge base that also serves as the system model. The parser uses a CFG adapted for LSCs and is an extension of the active chart parser of Kay (Kay, 1986). The grammar is domain general and is composed from a set of rules and terminal symbols. The terminal symbols include static terminals / reserved words, that are used as language directives such as *if*, *then*, *must*, *may*, etc. and dynamic terminals are processed by a dictionary or taken from the working system model.

The grammar used in the current study is described in (Gordon & Harel, 2009), it includes the ability to specify in CNL the following elements: messages, condition, forbidden messages, symbolic entities and some additional constructs not tested in this research.

To resolve imprecise and ambiguous requirements with the help of the engineer, we have implemented a dialog system using an interface similar to the *quick fix* interface of programming environments like Eclipse

IDE (Carlson, 2005). The engineer may then select one of the options from the dialog (a quick option that fixes the problem) or make a different change on his/her own once the problem is known. Figure 2 shows a sample problem and available solutions.

The problems encountered may be a result of different types of analysis: a grammatical analysis, an LSC disambiguation or a model connected analysis. In the first type, a word may have two different meanings, e.g., *value* can be used as a verb or a property. In the second type, a sentence can be interpreted differently when creating the LSC. For example, in the partial sentence "the controller toggles the light", when no information about model elements exists, one interpretation is a message toggles from object controller to light and another is a self message toggles from object controller to itself with the message parameter of the light.

In the third type, the analysis checks that the terms the engineer refers to appear in the *system model*. The system model is a tree structure containing all the objects in the specification, each object with its properties and methods. Figure 2 displays the system model of the wristwatch example. Some objects may be represented in the GUI, while others may not.

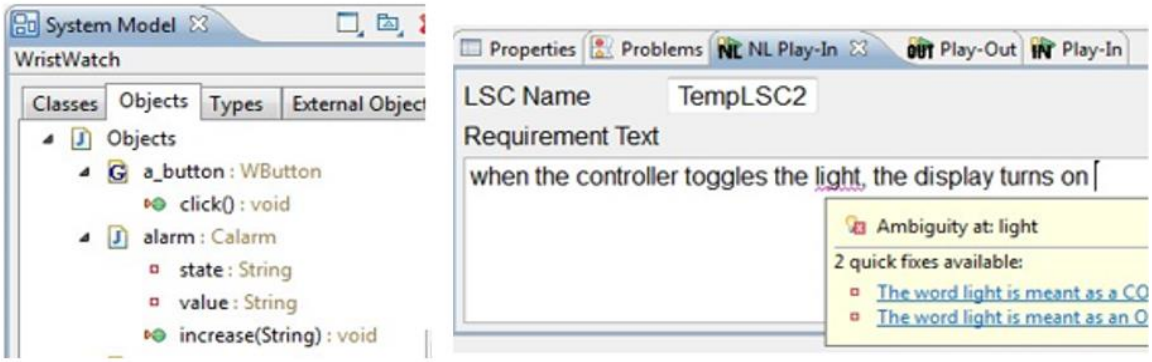

Figure 2. The tool**.** On the left is the system model view with a model of a wristwatch. On the right is a sample of quick fix options for resolving an ambiguity. The problem location is indicated by the squiggly line and the quick box dialog that appears when hovering with the mouse over the problem displays the problem's description and possible solutions, if available. Tool available at http://wiki.weizmann.ac.il/playgo/index.php/PlayGo_Feature_List#Natural_language_play-in

3. EVALUATION STUDIES

We next describe three preliminary evaluation studies which aim to better understand the CNL behavioral approach to development. In study 1 we conducted a user study with a novice programmer, interviewing her while she created 2 full projects using the language. Our main question was: how long would it take a novice programmer to learn the language. Study 2 was more elaborate and included a class of graduate students attempting to comprehend the language in a shorter time frame. The next tree sections report on our preliminary findings.

3.1 Study 1: Novice Programmer Example

We asked the novice programmer to learn the CNL for LSCs on her own, using the PlayGo website [http://wiki.weizmann.ac.il/playgo/index.php/Main_Page] and the tutorial used in (Gordon & Harel, 2012).

As for the novice programmer, after a week, she was able to complete the task from the first experiment and was then able to program systems of her own choice. She created modest examples of a cell phone [http://wiki.weizmann.ac.il/playgo/index.php/Phone_Specification] and a chess game [http://wiki.weizmann.ac.il/playgo/index.php/Chess_Game_Example]. This participant was in her twenties, fresh out of undergraduate school, and a non-native English speaker. Her ability to write in CNL increased and eventually she created systems with 20 requirements. Some syntax choices in the grammar, like the necessity to add a determinant before an object, were not naturally occurring in her English. Her learning of the CNL included learning the CNL syntax, which is similar to learning the syntax of other programming languages and specifically she had to learn to add a determinant before object names. Part of writing the CNL requirements

was dedicated to figuring out what type of behaviors were low-level and require further implementation in Java. For example, in the chess game, calculating the possible location of all pieces was done in Java code. The examples are available [http://wiki.weizmann.ac.il/playgo/index.php/Chess_Game_Example].

Since we concluded that knowledge of LSC is required, we continued to test comprehension and creation of LSC in a shorter time frame.

3.2 Study 2: Quick Comprehension and Creation

In this study, we focused the evaluation on using PlayGo to model a coffee machine system. We concentrated on the following research questions: (i) Is the controlled natural language play-in method sufficient for writing and reading requirements? (ii) Can it be learned and used effectively in two sessions?

Participants. The experiment took place at Tel Aviv University (TAU) and the Academic College of Tel Aviv-Yafo (MTA). Participants were recruited from computer science graduate students, who had taken advanced classes on software modeling. Students were offered a small monetary reward for their time, independent of performance in the experiment. No grades were involved. A single three hour class meeting, conducted a week before the experiment, was dedicated to introducing the ideas of scenario-based programming and to presenting the LSC language and the PlayGo tool. The meeting included a short live demo.

In total, 19 graduate students participated, sixteen male and three female. All had over three years of programming experience, in languages such as Java and C#. Each received a booklet with instructions and questions, as well as a copy of a pre-prepared PlayGo workspace, running on Eclipse in one of the two universities student labs. Each of the participants spent between 2 and 2.5 hours to complete the experiment. sixteen participants completed all parts of the experiment, while three had to leave before completion due to other unrelated scheduled obligations (we left them out of the analysis for the questions they did not answer). Participants signed a consent form and the study was approved by the institutional IRB.

Protocol. We designed an evaluation study with small tasks and questions that attempt to get user preference and abilities in qualitative and quantitative measurements. Participants received a set of tasks divided into five parts, roughly going from basic and easy tasks to more complex and difficult ones. We had the participants add requirements to increase the system's complexity, while also gaining experience using the play-in approach. Part I dealt with LSC comprehension, and parts II and IV dealt with play-in, examining the process of creating an LSC, both in CNL text and graphically. Parts III and V were related to play-out and will be reported upon in a future publication. We gave participants a high-level description of a new requirement and asked them to write it down more formally, first by writing it in general English, then by drawing it, and finally by using the CNLPI interface in PlayGo to define the new LSC.

Each of the three parts included a mix of multiple-choice questions for quantitative analysis and open-ended questions for a more qualitative analysis. We planned to perform the experiment in a 2-3 hour session, allowing participants to advance at their own pace. We estimated times after a test run with two developers from our group.

The LSCs used in the experiment were adapted from several variants of the vending machine specifications described in (Maoz & Sa'ar, 2012, 2013). They consist of 5-7 LSCs, and involve a user, a water heater, a panel, a cashier, and a dispenser. When designing an example application for the experiment, we chose to adapt this previously studied example, in order to eliminate the risk of our own bias (e.g., to create an application that is unfairly simple) .

The questionnaire was designed to test the hypothesis using qualitative and quantitative questions. Table 1 summarizes the type of questions and their goal. The complete questionnaire is available in supporting materials (https://drive.google.com/file/d/1hn2bGmPZBw5fVyiy2454leNyZIOyO3zg/view?usp=sharing).

Part I. The tasks in part I intended to examine the participants understanding of basic LSC concepts and their ability to read and understand a given single LSC and a given specification that consisted of several LSCs, presented both graphically and in controlled natural language text. Given a specification with four LSCs, we asked participants to distinguish system events from environment-controlled events, to describe in their own words what the system does, to select a correct, short high-level description of what the system does (e.g, "serves coffee", "counts coins and returns change"), etc. We then asked the participants to reflect on which of the presentation forms they used more often in order to understand the system's behavior and answer the questions: the diagrams or the textual description in controlled natural language.

Part II. The task in part II was to define a requirement in which the panel shows the word error whenever the user asks for coffee but coffee cannot be served. Participants had to figure out that this situation occurs when the user requests coffee but three coins have not yet been inserted.

Part IV. In part IV, the task was a bit more complicated, in the context of a more complex specification (two new LSCs were introduced). The participants had to add a cancellation functionality that would allow the user of the vending machine to retrieve her coins.

Table 1. Goal and type of questions in the questionnaire. Q_i is the question number, and P_i is the part number, as some parts had an unnumbered open question

Description	Goal	Multiple Choice	Open Ended
Programming background	Establish knowledge level	Q1	Q2
LSC background	SubjectiveLSC understanding	Q3, Q5, Q44	Q4, Q45
LSC Comprehension	Understanding existing specification	Q6, Q7, Q9	P1, Q8
Comprehension preference	Subjectivepreference	Q10	
LSC generation	Creating LSCs in NL, CNL, drawing		Q11-13, Q15, Q16, Q27-29, Q32
Generation preference	Preferred creation method	Q14, Q30	Q15, Q31

Due to the limited time, we did not ask participants to create the LSCs in diagram form using PlayGo, nor did we allow for time to try and run the new specification with the LSC that was added using CNLPI. However, participants were asked to reflect on which of the activities they preferred, drawing or writing natural language text.

3.2.1 Results

Comprehension. All participants were able to complete the comprehension task, which made their results relevant for further analysis. All but four rated their understanding of LSCs after the single lesson as ‘not so good’, two rated their understanding as ‘good’, and two as ‘not good at all’. In the second class we added a question of LSC understanding also at the end of the last task, and for the seven participants who answered the understanding question before and after the experiment there was a significant difference between the two answers (Wilcoxon 1-tail signed-rank test $W(5) = 0$, $p \leq 0.05$). For the comprehension task in part I, we found a significant preference to use the diagram over the text, which is in line with the fact that the text can be ambiguous while the diagram contains complete and formal information ($W(13) = 3.5, p \leq 0.05$).

Play-in. For part II, all participants were able to provide a single LSC that captured the requirement; some of these LSCs were correct while others were almost correct. Most participants entered a requirement that caused an error if there were less than 3 coins, while others showed an error, when the number of coins was different than 3. We did not find a tendency to prefer the CNLPI over the drawing, or the other way around ($W(14) = 49, p > 0.05$).

As one participant mentioned: *“You cannot work only on the text, you must have the diagram in order to verify that the text was correctly understood”* [translated]. We therefore feel that the question might have caused confusion between using CNLPI with or without the diagram as feedback, versus creating the diagram directly. We also believe that since participants drew freely on paper, without getting any feedback on whether the syntax was correct, it was not appropriate to compare to the CNLPI, where they did use the tool and had to adhere to the exact syntax.

No preference was found also in part IV ($W(17) = 52, p > 0.05$). However in task two, we got a larger variety of LSCs; participants used between one and three LSCs. Most participants attempted to handle not only the requirement we asked for, i.e., the cancel functionality, but also the problems that may be caused by cancelling in mid-operation.

Overall, it seems that participants were able to express their requirements in LSCs, both in diagram format, and in CNL. In some cases, the CNL was similar to the general requirement description. For example, participant #3 wrote the requirement as: *“When the user requests coffee from the panel, if the cashier coins are less than 3, the panel shows ‘error’.”*, and the CNL was exactly the same. There were additional participants who chose to use the CNL examples for their description, when they were asked to use the controlled English as example for their own requirements. Thus, we also had participants who wrote text fragments that were replaced later by text that was acceptable by the CNL parser. For example, participant #5 started with: *“When the user requests coffee from the panel, if the dispenser cannot serve coffee, the panel shows ‘error’.”*. This

requirement, which includes a condition on an event (not supported in LSCs), was changed to: *"When the user requests coffee from the panel, if the dispenser status is 'cannot serve coffee', the panel shows an error."*

Participants were able to work with PlayGo, after a very short tutorial session. Participants were able to specify useful requirements in CNLPI mainly by using the examples provided. Some did complain about understanding the syntax errors and commented as an example: *"The parser error are not very informative. You need to hit the correct syntax. If you could draw the LSC while the user is typing it could help a lot"*. However they were ultimately able to solve the problems with little help.

Most participants were able to complete the task, comprehend the system, and extend it with some LSCs, in expected time, with very little tutoring and practice. We believe this hints to the learnability of the language and the PlayGo tool. Those participants that did not complete the second task were all in the second group, which participated in the experiment in the afternoon, and we believe, since it was voluntary, that they preferred to leave due to the late hour. However, this might have been due to the task difficulty.

4. CONCLUSION

In answer to some of the questions raised in the introduction about the knowledge necessary to program LSCs with CNL, it was important in all cases for the participants to have knowledge of the LSC visual language, and to verify that their text was translated as they expected. Natural language even in the controlled form we discussed, is not formal enough for execution. With the help of a system model, the CNL requirements are less ambiguous, but it is only with the LSC chart that the formalism is completely clear. Therefore, the writer verifies her requirements with the visual representation of the LSCs and is expected to understand them. Another question, is how long it takes an engineer to learn the method. Our evaluation has shortcomings, and a better tool and longer evaluation are required, yet we show a possibility of using CNL for LSC after two short sessions allowing comprehension and creation of a demo system.

Creating complex reactive systems is not a simple task and neither is understanding natural language requirements. We have presented a method that translates controlled NL requirements into LSCs, which are fully executable. Moreover, we have expanded the power of this NL interface by combining it with play-in, yielding the hybrid show & tell method. The implementation of the system is thus a set of fragmented, yet structured requirements/ namely the LSCs, which are fully executable.

The ability to translate a controlled natural language into the formal language of LSCs is a step in the direction of making programming more readily available in our developing digital world. The translation we suggest is tailored for the LSC language, but can be extended to support other languages too. It can also be integrated with other non-programmer approaches.

Some of the key features this work offers include formal rules, fragmented and scenario-based descriptions in the spirit of behavioral programming, and an executable and verifiable system with an intelligent natural interface. These ideas take a step towards having the computational tools adapt to human nature than to have humans adapt and learn computational concepts.

The current situation regarding the execution of LSCs is not without its limitations. LSCs do not always result in a deterministic execution and the execution is not always optimal. Nor are LSCs yet scalable to very large systems. However, progress is being made on the execution methods; see for example, (Harel et al., 2002; Harel & Segall, 2012). The work presented here shows how NLP and the LSC formalism, together with the behavioral programming approach, take programming closer to how humans specify requirements.

ACKNOWLEDGEMENT

We would like to thank Shahar Maoz and Smadar Szekely for their help with the evaluation experiment, Shmuel Tishberowitz for assisting in recruiting participants for it, and Jayasree Nair for helping with the checkers example. This work was supported in part by grant number 857/12 to D.H. from the Israel Science Foundation.

REFERENCES

Bryant, B. R., & Lee, B.-S. (2002). Two-Level Grammar as an Object-Oriented Requirements Specification Language. *Proc. 35th Annual Hawaii Int. Conf. on System Sciences (HICSS'02)*, 280–289.

Carlson, D. (2005). *Eclipse Distilled*. Addison-Wesley.

Cypher, A., Dontcheva, M., Lau, T., & Nichols, J. (2010). *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann Publishers Inc.

Cypher, A., Halbert, D. C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B. A., & Turransky, A. (Eds.). (1993). *Watch What I Do: Programming by Demonstration*. MIT Press.

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

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., & Roy, S. (2016). Program Synthesis Using Natural Language. *Proceedings of the 38th International Conference on Software Engineering*, 345–356.

Drazan, J., & Mencl, V. (2007). Improved Processing of Textual Use Cases: Deriving Behavior Specifications. *Proc. 33rd Int. Conf. on Trends in Theory and Practice of Computer Science*, 856–868.

Fuchs, N. E., & Schwitter, R. (1995). Attempto: Controlled natural language for requirements specifications. *Proc. 7th ILPS Workshop on Logic Programming Environments*.

Gordon, M., & Harel, D. (2012). Evaluating a Natural Language Interface for Behavioral Programming. *Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing*, 17–20.

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

Harel, D. (2001). From Play-In Scenarios To Code: An Achievable Dream. *Computer*, *34*(1), 53–60.

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

Harel, D. (2006). Playing with Verification, Planning and Aspects: Unusual Methods for Running Scenario-Based Programs. *Proc. 18th Int. Conf. on Computer Aided Verification*, 3–4.

Harel, D., Kugler, H., Marelly, R., & Pnueli, A. (2002). Smart Play-Out of Behavioral Requirements. *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design*, 378–398.

Harel, D., & Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag.

Harel, D., & Segall, I. (2012). Synthesis from scenario-based specifications. *J. of Comp. and Sys. Sci.*, *78*(3), 970–980.

ITU: International Telecommunication Union. (1996). *Recommendation Z.120: Message Sequence Chart (MSC)*.

Jurafsky, D., & Martin, J. H. (2008). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall.

Kay, M. (1986). *Readings in Natural Lang. Processing* (B. J. Grosz, K. Sparck-Jones, & B. L. Webber, Eds.; pp. 35–70).

Lei, T., Long, F., Barzilay, R., & Rinard, M. (2013). From Natural Language Specifications to Program Input Parsers. *Proc. Annual Meeting Assoc. for Computational Linguistics*.

Madala, K., Piparia, S., Blanco, E., Do, H., & Bryce, R. (2020). Model elements identification using neural networks: A comprehensive study. *Requirements Engineering*. https://doi.org/10.1007/s00766-020-00332-2

Maoz, S., & Sa'ar, Y. (2012). Assume-Guarantee Scenarios: Semantics and Synthesis. *MoDELS*, 335–351.

Maoz, S., & Sa'ar, Y. (2013). Counter Play-out: Executing Unrealizable Scenario-based Specifications. *Proc. 2013 Int. Conf. on Software Engineering*, 242–251.

Mich, L. (1996). NL-OOPS: From Natural Language to Object Oriented Requirements Using the Natural Language Processing System LOLITA. *Natural Language Engineering*, *2*(2), 161–187.

Moitra, A., Siu, K., Crapo, A. W., Durling, M., Li, M., Manolios, P., Meiners, M., & McMillan, C. (2019). Automating requirements analysis and test case generation. *Requirements Engineering*, *24*(3), 341–364.

Reinhartz-Berger, I., & Kemelman, M. (2020). Extracting core requirements for software product lines. *Requirements Engineering*, *25*(1), 47–65. https://doi.org/10.1007/s00766-018-0307-0

Schlegel, V., Lang, B., Handschuh, S., & Freitas, A. (2019). Vajra: Step-by-Step Programming with Natural Language. *Proceedings of the 24th International Conference on Intelligent User Interfaces*, 30–39. https://doi.org/10.1145/3301275.3302267

Segundo, L. M., Herrera, R. R., & Herrera, K. Y. P. (2007). UML Sequence Diagram Generator System from Use Case Description Using Natural Language. *Electronics, Robotics and Automotive Mechanics Conf.*, *0*, 360–363.

Tsarfaty, R., Pogrebezky, I., Weiss, G., Natan, Y., Szekely, S., & Harel, D. (2014). Semantic Parsing Using Content and Context: A Case Study from Requirements Elicitation. *Proc. Int. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 1296–1307.

UML. (2007). *Unified Modeling Language Superstructure, v2.1.1* (formal/2007-02-03; Issue formal/2007-02-03). Object Management Group.

Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E.-V., & Batista-Navarro, R. T. (2020). Natural Language Processing (NLP) for Requirements Engineering: A Systematic Mapping Study. *ArXiv Preprint ArXiv:2004.01099*.