Thesis for the degree
Doctor of Philosophy

Submitted to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

עבודת גמר (תזה) לתואר
דוקטור לפילוסופיה

מוגשת למועצה המדעית של
מכון ויצמן למדע
רחובות, ישראל

By
Michal Gordon (Kiwkowitz)

מאת
מיכל גורדון (קיבקוביץ')

ממשקים טבעיים ואינטראקטיביים לתכנות התנהגותי

Natural and Interactive Interfaces for Behavioral Programming

Advisor:
Prof. David Harel

מנחה:
פרופ' דוד הראל

December 2012

טבת תשע"ג

Acknowledgments

I would like to thank my advisor Prof. David Harel, who has been an ideal teacher. He allowed me to explore for myself, while guiding me wisely on the crossroads. I have learned so much from him during this PhD, how to write, how to select which ideas to pursue, how to approach the harder problems, and so much more. I feel privileged to have worked with him, and I thank him for sharing his time, patience and wisdom.

I would like to thank colleagues and group members for collaborations and discussions over the years: Assaf Marron, Shahar Maoz, Naamah Bloch, Guy Katz, Avital Sadot, Dana Sherman, Yaki Setty, Yaniv Sa'ar, Guy Weiner, Smadar Szekely, Yaarit Natan, Daniel Barkan, and Guy Weiss. Special thanks to Shahar Maoz and Assaf Marron.

I thank the members of my committee, Prof. Amiram Yehudai and Prof. David Peleg, for fruitful comments and discussions. Special thanks to Amiram for helpful comments and suggestions over the years and for his enthusiasm for my research.

I would like to thank to thank my parents for their encouragement and support. I would like to thank my husband, Goren, my love, my partner, my friend, who always took part in my work and was an inspiration and a motivation. I doubt if I would have been able to interweave a PhD and motherhood if not for his support. To Noga and Doron, I hope they grow in a world with many more possibilities, a world where they can reach the stars and beyond.

Abstract

This thesis describes the development of intelligent interfaces for scenario-based programming, specifically for the language of *live sequence charts* (LSC). The main topic presents *natural language play-in* (NL-play-in), a method that supports creating LSCs by writing structured natural language requirements. The method transforms text into the visual formalism of LSCs using a context-free-grammar and additional information used for disambiguation of the writer's intentions, including possible interaction with the writer. A second topic described, is *show & tell*, a combination of NL-play-in with user interaction using a graphical user interface (GUI) of the system, extending *play-in*. In play-in a GUI of the system being programmed is provided for carrying out the actual act of programming behaviorally. Show & tell merges, naturally and intuitively, this ability with the ability to specify behavior textually in natural language. It interprets the writer's interaction in the context of the textual requirements.

Finally, we also present a preliminary evaluation of the interfaces, and additional methods developed for scenario-based programming and its generalization, *behavioral programming*. These include ideas for navigation and comprehension for scenario-based programs.

On the whole, this thesis deals with creating more natural ways to program using scenario-based programming, making real world programming accessible to a larger community.

Contents

List of Abbreviations

BP	Behavioral programming
CASE	computer aided software engineering
CNL	Controlled natural language
GUI	Graphical user interface
HCI	Human computer interface
LSC	Live sequence chart
MSC	Message sequence chart
NL	Natural language
NLP	Natural language processing
OO	Object oriented
PBD	Programming by demonstration
SD	Sequence diagrams
UML	Unified modeling language

1 Introduction

Behavioral programming (BP) [23] is a programming paradigm in which system behavior is described by independent modules aligned with scenarios. The language of *live sequence charts* (LSC) [7], proposed in 1999, is a visual formalism that extends message sequence charts (MSC) [25] with multimodality, and it constituted the first executable implementation of the paradigm. The present thesis continues along the lines of *play-in* [22, 21], proposed in 2003 for creating LSCs. In play-in, a graphical user interface (GUI) of the system being programmed is provided for carrying out the actual act of programming behaviorally.

In the main part of the thesis we propose advanced interfaces for playing in behavior. First, we introduce a controlled natural language (NL) interface for LSC [12], with a dialog system to resolve natural language ambiguities. The user is requested to clarify and answer questions when the natural language is not clear. The result is a *natural language interface* for specifying fully executable programs, called *NL-play-in*. We then extend this interface with *show & tell* [14], a combination of play-in, and the natural language interface. Show & tell merges, naturally and intuitively, the ability to demonstrate parts of the behavior interactively and the ability to specify behavior textually in natural language. We also report on preliminary user evaluation of these interfaces [15], which test the interfaces on programmers. Additional capabilities are developed to navigate and comprehend scenarios [13, 8] and to address some of the challenges arising from the new behavioral programming paradigm.

This thesis is organized as follows. Sections 2-3 present the core of the thesis: the natural language interface for LSC in Section 2; and show & tell, a combination of the NL interface with the original play-in, in Section 3. Section 4 reports on the comparisons and user evaluations performed on the new interfaces. In Section 5 discusses a semantic navigation method for LSC, developed for comprehension of LSCs. The papers and details of each

topic, appear in Section 6, and Section 7 includes a discussion for each of the topics in the thesis and possible future work. In Section 8 we describe work in progress.

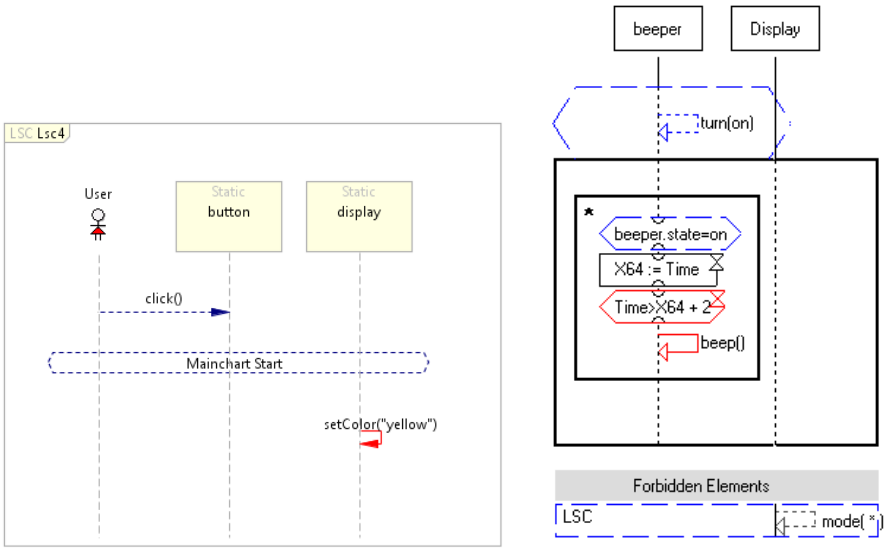
2 Specifying Behavior in Natural Language

The formal language of *live sequence charts* (LSC) [7] is similar to message sequence charts (MSC) [25] or UML sequence diagrams (SD) [34] and is visual in nature. Therefore, traditionally, the charts in the language can be created by drawing or by drag-&-drop of elements. Additionally, charts can be created by *play-in*, a method that allows manipulating a graphical user interface (GUI) of the system being described, in order to demonstrate (or "play-in") events that occur in a scenario [22]. These events are added to the diagram and become part of the programmed behavior. However, the creation process of play-in is not always easy, since it requires finding many buttons, opening dialogs, etc. For this reason, we have created a controlled natural language interface, named *NL-play-in* for generating LSC diagrams, detailed in [12].

The NL interface provides the ability to write a natural language description of the scenario, that is translated automatically into an appropriate LSC. For example, the sentence "when the user clicks the button, the display color changes to yellow" yields the LSC shown in Figure 1 (a). Figure 1 (b) shows an LSC created for a more complex sentence that includes a forbidden event.

The translation algorithm uses a context-free grammar [26] with semantic information for LSCs. It applies an active chart parser [26]. The terminals of the grammar, the words, are static terminals relevant to the LSC language and dynamic terminals relevant to the specific system being described. The dynamic terminals are incremented with each additional requirement. The WordNet dictionary [31] is used to find information about terminal words that are domain specific and add them as the relevant part of speech. Therefore, the language is domain general and the user can describe any system.

The parsing is non-linear and takes $O(n^3)$, for an input with n terms. However, this is compensated for by the fact that n is not too long for a single requirement. When there are ambiguities in interpreting a phrase, a short dialog with the user asks him/her to resolve them. The dialog with the



(a) (b)

Figure 1: (a) A sample PlayGo LSC, created for the sentence "when the user clicks the button, the display color changes to yellow". (b) A Play-Engine LSC, created for the sentence "when the beeper turns on, as long as the beeper state is on, if two seconds elapse the beeper beeps, the display mode cannot change" with a forbidden cold event shown at the bottom of the LSC.

user is performed through a *quick fix* interface [5], in which the questions of the system are displayed when hovering over a squiggly line that marks the questioned text. The user is also presented with a quick list of solutions that the algorithm suggests, and he can quickly resolve the problems by selecting an option from the list.

3 Show & Tell, or Combining NL and Interactive Play-In

The original LSC paradigm suggested in 1999 by Damm and Harel [7] was augmented with the interface of *play-in* in 2003 by Harel and Marelly [22]. Using this interface the user can "play-in" scenarios with the GUI of the system in order to create parts of the LSC. For example, if the user clicks a button, then the message `click` between `user` and `button` is added to the current diagram. The ability to demonstrate behaviors or point to objects has been used in additional interfaces for programming or for applications [6].

Some scenario parts are better described than shown, and others are quicker to show. Therefore, we have created the *show & tell* interface that extends our natural language interface with the play-in idea. It is described in [14]. Show & tell allows alternating between the writing (or the uttering, when applying speech recognition methods) and the showing, according to the preference of the user. The algorithm interprets the interaction based on the current state of the text. It calculates the interaction's intention based on the current text, and suggests whether the interaction was meant to refer to the operation, to an object name or to a full part of the scenario.

For example, the interpretation of an interaction like *the user click of a button*, depends on the text entered at the interaction time. If the textual requirement is "`when the user`" the interaction will automatically add the additional text of "`clicks the button`". However, if the textual requirement entered was "`when the user clicks the`", the same operation — *clicking of the button*, would only add "`button`" to the text.

When a GUI of the system exists, it is necessary to refer to the exact object name or operation, in order for the executable requirements to operate correctly. Show & tell adds the ability to point to objects or demonstrate operations (e.g., sliding, selecting) to get their exact name as given during

GUI development.

The algorithm that interprets the interaction, extends the natural language parser with on-line capabilities. Each interaction adds a set of possible grammar edges with the relevant object and operation names. These edges are processed by the parser with respect to its state at the time of the interaction. The active chart parser has been adopted to deal with multiple possible inputs and it selects those edges that complete the current parse. A completion is defined as a grammar edge that advances the current parse and is correct grammatically. The best completion is defined as the one that adds the longest additional text to the current text. The user is provided with the best completion as a suggestion, and he can view and select other possible interpretations of the interaction. After he approves the additional text, it is added to the current requirement and later becomes part of the LSC. Details are provided in [14].

4 Evaluation of the Various Interfaces for LSC

We developed the natural language interface to render programming in LSC learnable, usable, and natural, in the hope it will be a step towards the vision of liberating programming [19]; e.g., making the process of specifying to a computer what to do more intuitive, natural and fun. Accordingly, this section describes research we have performed on user evaluation of the LSC language and the new interfaces for it: the natural language interface and the show & tell. Language comparisons and claims about the naturalness of a programming language are difficult to prove [29]. Nevertheless, we have developed an experimental setup to test the following research questions: (i) Is the natural language interface quickly learnable and how do the various interfaces of 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?

The preliminary study involved 12 programmers familiar with the LSC language that programmed 3 simple tasks using the different LSC interfaces: using diagram tools, using the original play-in, using the natural language interface and using the show & tell. We found evidence that the natural language interface was quickly learnable by the majority of the programmers, and the majority reported that using the natural language interfaces felt to them quicker than the other interfaces. Additionally, some of the programmers reported the NL interface as fun. The experiment pointed-out some interesting issues: the fact that there are multiple ways to describe behavior in NL (rather than an exact syntax) can be hard for some programmers; especially for those that wish to learn an exact syntax. We believe it would be interesting to collect additional parameters and also evaluate the approach for non-programmers.

We also saw that for programmers who are used to typing, the show & tell interface did not offer an advantage. Participants mentioned the fact that it required them to switch between keyboard and mouse, which slowed them

down. We believe additional evaluation can be used to show whether this is also true for non-programmers, who may be slower at typing. Additionally, we hypothesize that when the *tell* part is replaced by a speech recognition engine, the advantages of the *show* part will become more prominent.

Additional tasks were used to compare these interfaces with programming in a different language, namely, Java. In the tasks we designed, the majority of the expert Java developers preferred LSCs with the NL interface over Java. They explained the advantage of the LSC language, in the given task, in how simple it was to specify in LSCs that multiple events should occur before the system should perform an action.

Figure 2 shows the PlayGo [20] tool during the experiment. More details on the evaluation can be found in [15].

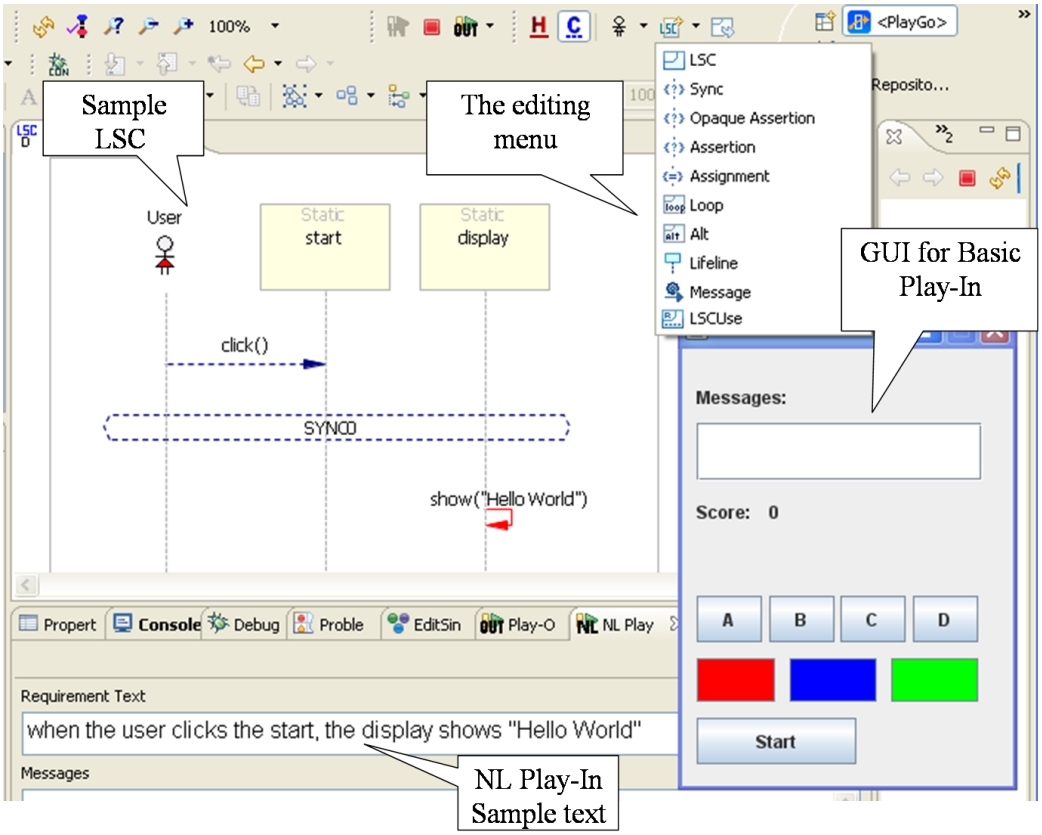


Figure 2: PlayGo environment, with a sample LSC and the natural language that created it. The GUI used in the experiment appears on the right and the editing menu is above it.

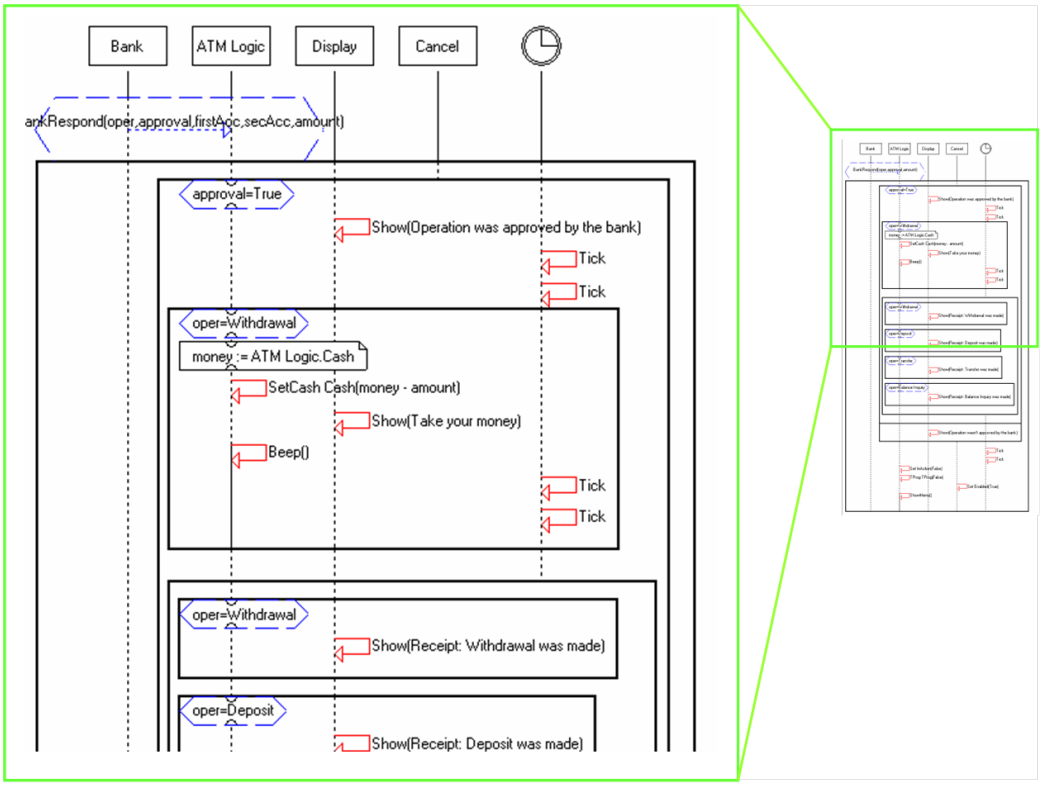


Figure 3: A large LSC, only the half part is displayed in order for the messages to be readable.

5 Semantic Navigation of LSC

Another contribution of our research to the visual language of LSC, which is also relevant to sequence diagrams (SD) [34], is in the field of diagram comprehension — navigating large LSCs. In a visual programming language as LSC, the navigation and ability to comprehend depend on the available tools. We have created a semantic navigation algorithm that allows zooming in or out of large diagrams, focusing on events or messages that are more relevant to a given task, and leaving context hints for full diagram comprehension. Figure 3 shows an example of a large LSC.

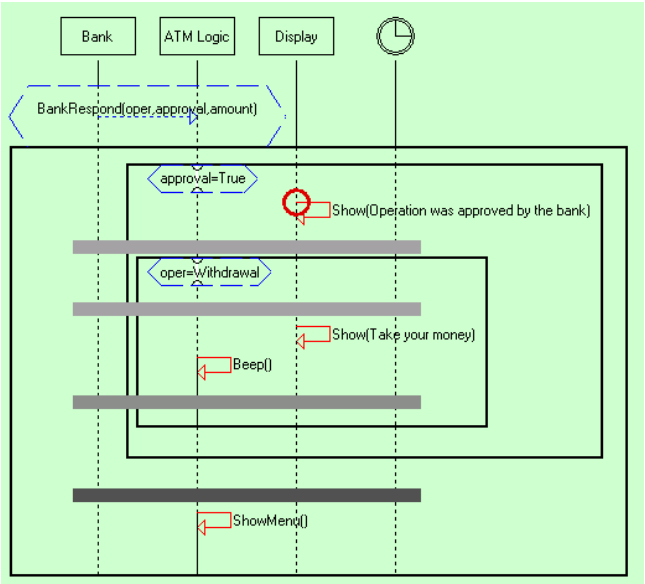


Figure 4: The zoomed LSC, with placeholders.

Our semantic zoom algorithm, described in [13], uses semantic weights to show/hide different elements in the diagram, while hinting at those hidden using a *placeholder*. Figure 4 shows a zoomed LSC. The placeholders are depicted as gray rectangles. The placeholder weights are coded by the level of the grayscale color, thus hinting at the amount of information being hidden. Moreover, the various placeholders are merged when possible to save space, but are used also to show the structure of the diagram. Additional details and the various ways to calculate useful weights can be found in [13].

6 Papers

This section includes copies of the following published peer-reviewed papers:

1. M. Gordon and D. Harel. Generating Executable Scenarios from Natural Language", *In Proc. 10th International Conference on Computational Linguistics and Intelligent Text Processing, CICLing'09*, Lecture Notes In Computer Science, vol. 5449. Springer-Verlag, 456-467, 2009.

2. M. Gordon and D. Harel. Show-and-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior. *In Proc. IADIS Interfaces and Human Computer Interaction, IHCI'11*, pages 360-364, 2011.

3. M. Gordon and D. Harel. Evaluating a Natural Language Interface for Behavioral Programming. *In Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing, VLHCC'12*, pages 17-20, 2012.

4. M. Gordon and D. Harel. Semantic Navigation Strategies for Scenario-Based Programming, *In Proc. IEEE Symposium on Visual Languages and Human-Centric Computing, VLHCC'10*, pp. 219-226, 2010.

Additionally, we include a broad review paper covering the core of the thesis, namely, the natural language interface and the the show & tell method, which has yet to be published. The paper is titled "Programming in Natural Language" and it subsumes papers 1, 2 and 3 from the above list.

Generating Executable Scenarios from Natural Language

Michal Gordon and David Harel

The Weizmann Institute of Science, Rehovot, 76100, Israel
{michal.gordon,dharel}@weizmann.ac.il

Abstract. Bridging the gap between the specification of software requirements and actual execution of the behavior of the specified system has been the target of much research in recent years. We have created a natural language interface, which, for a useful class of systems, yields the automatic production of executable code from structured requirements. In this paper we describe how our method uses static and dynamic grammar for generating live sequence charts (LSCs), that constitute a powerful executable extension of sequence diagrams for reactive systems. We have implemented an automatic translation from controlled natural language requirements into LSCs, and we demonstrate it on two sample reactive systems.

1 Introduction

Live Sequence Charts are a visual formalism that describes natural "pieces" of behavior and are similar to telling someone what they may and may not do, and under what conditions. The question we want to address here is this: can we capture the requirements for a dynamic system in a far more natural style than is common? We want a style that is intuitive and less formal, and which can also serve as the system's executable behavioral description [1].

To be able to specify behavior in a natural style, one would require a simple way to specify pieces of requirements for complex behavior, without having to explicitly, and manually, integrate the requirements into a coherent design. In [2], the mechanism of *play-in* was suggested as a means for making programming practical for lay-people. In this approach, the user 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 system, still a 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). As a result, the system generates automatically, and on the fly, live sequence charts (LSCs) [3], a variant of UML sequence diagrams [4] that capture the behavior and interaction between the environment and the system or between the system's parts. In the current work we present an initial natural language interface that generates LSCs from structured English requirements.

An LSC describes inter-object behavior, behavior between objects, capturing some part of the interaction between the system's objects, or between the system and its environment. LSCs distinguish the possible behavior from the necessary behavior (i.e., liveness, which is where the term "live" comes from), and can also express forbidden behavior — scenarios that are not allowed, and more. Furthermore, LSCs are fully executable using the *play-out* mechanism developed for LSCs in [2], and its more powerful variants [5, 6]. To execute LSCs the play-out mechanism monitors at all times what must be done, what may be done and what cannot be done, and proceeds accordingly. Although the execution does not result in an optimal code, nor is the executed artifact deterministic (since LSC are under-specified) it is nevertheless a complete execution of the LSC specification. The execution details are outside the scope of this paper, but are described in detail in [5, 2].

By its nature, the LSC language comes close to the way one would specify dynamic requirements in a natural language. We suggest to take advantage of this similarity, and to translate natural language requirements directly into LSCs, and then render them fully executable. One interesting facet of this idea is rooted in the fact that the natural and intuitive way to describe behavioral requirements will generate fragmented multi-modal pieces of behavior which is also the main underling philosophy of LSCs. The play-out mechanisms are able to consider all the fragmented pieces together as an integrated whole, yielding a fully executable artifact. Thus, our translation into LSCs can be viewed as a method for executing natural language requirements for reactive systems.

As to related work (discussed more fully later), we should say here that natural language processing (NLP) has been used in computer-aided software engineering (CASE) tools to assist human analysis of the requirements. One use is in extracting the system classes, objects, methods or connections from the natural language description [7, 8]. NLP has been applied to use case description in order to create simple sequence diagrams with messages between objects [9], or to assist in initial design [10]. NLP has also been used to parse requirements and to extract executable code [11] by generating object-oriented models. However, it is important to realize, that the resulting code is intra-object — describes the behavior of each object separately under the various conditions, and it is usually limited to sequential behavior. The resulting OO artifact is focussed on object-by-object specification, and is not naturally inter-object.

The paper is structured as follows: Section 2 contains some brief preliminaries, Section 3 presents an overview of the translation method, and Section 4 demonstrates the details using an example. Section 5 discusses related work and Section 6 concludes.

2 Preliminaries

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) — as well as possible

behavior — what may happen (with a cold temperature). The LSC language [3] has its roots in message sequence charts (MSC) [12] or its UML variant, sequence diagrams [4], where 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. Figure 1 shows a sample LSC. In this LSC the *prechart* events, those that trigger the scenario, appear in the top blue hexagon; in this case, a cold (dashed blue) click event from the user to the `c` button. If the prechart is satisfied, i.e., its events all occur and in the right order, then the main chart (in the black solid rectangle) must be satisfied too. In the example, there is a hot (solid red) event where the light state changes to `on` and a cold condition, in the blue hexagon, with a hot event in the subchart it creates. The meaning is that if the display mode is not `time`, then it must change to `time`. There is no particular order between the events in the main chart in the example, although in general there will be a partial order between them, derived from the temporal constraints along the vertical lifelines.

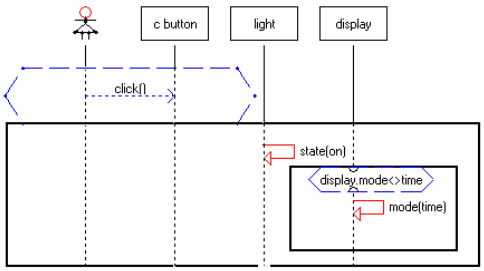


Fig. 1. A simple LSC. The prechart (the blue dashed hexagon) contains the cold event (blue dash arrow) "user clicks the `c` button", while the main chart (the black solid rectangle) shows two hot events (red solid arrow): one shows the light state changing to `on` and the other is a hot event with a cold condition (blue dashed hexagon) that specifies that if the mode is not `time` then it must change to `time`.

The basic LSC language also includes conditions, loops and switch cases. In [2], it has been significantly enriched to include time, scoped forbidden elements, and symbolic instances that allow reference to non-specific instances of a class.

Later, we will be describing a context-free grammar for behavioral requirements that will serve as our controlled English language. To recall, a context-free grammar (CFG) is a tuple $G = (T, N, S, R)$, where T is the finite set of terminals of the language, N is the set of non-terminals, that represent phrases in a sentence, $S \in N$ is the start variable used to represent a full sentence in the language, and R is the set of production rules from N to $(N \cup T)^*$. In the LSC grammar, parts of the grammar are static T_S and other parts are dynamic T_D.

3 Overview of LSC Grammar

Requirements are a way of describing scenarios that must happen, those that can happen, and those that are not allowed to happen. The static terminals describe the flow of the scenario; e.g., "*when* something happens *then* another thing should happen", or "*if* a certain condition holds *then* something *cannot* occur". The dynamic terminals refer to the model, the objects and their behaviors.

The static terminal symbols are *if, then, must, may* etc. They are relevant for inferring the semantics of LSCs. The dynamic terminals are all unrecognized terminals processed by a dictionary and transformed from part of speech to possible parts of the model. They are grouped into `objects`, `properties`, `methods` and `property values` which are not mutually exclusive.

For example in: "The user presses the button", `user` and `button` are both objects. Similarly, `presses` is a verb that is added to the methods terminal list. Other types of terminals are properties and property values. These can be identified as in the following example: "the display color changes to red", where the noun `color`, which is part of the noun phrase, is a property of the display object and the adjective `red` is a possible property value. Property values may also include possible variables for methods.

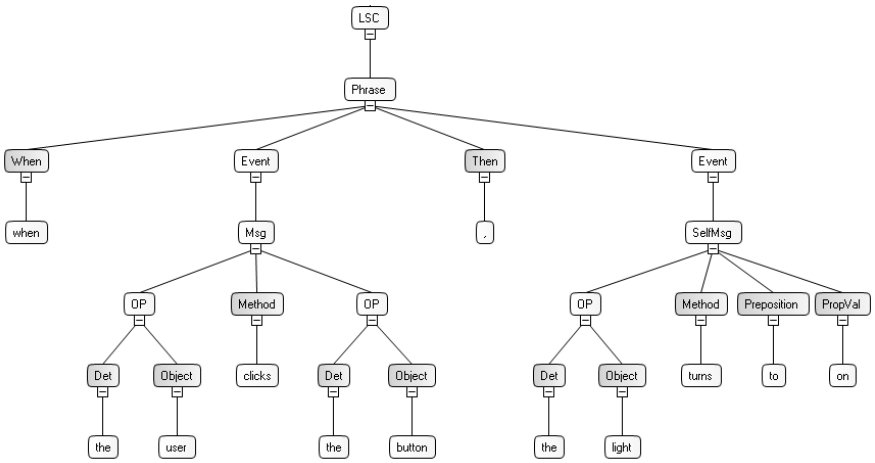


Fig. 2. The parse tree for the sentence "when the user clicks the button, the light turns on". The parts of the LSC grammar detected are shown. There is one message *Msg* which is a message from object phrase (*OP*) `user` to object phrase `button`, and another self message *SelfMsg* of object `light` with method `turn` and argument `on`.

Figure 2 displays the parse tree for the requirement: "when the user clicks the `b` button, the light turns to on". When analyzing the parse tree, the *when* and *then* hint to where the prechart ends and the main chart begins, the messages

added are `click` from the user to the button in the prechart and `turn` with a parameter `on` in the main chart, as seen in Fig. 3(a).

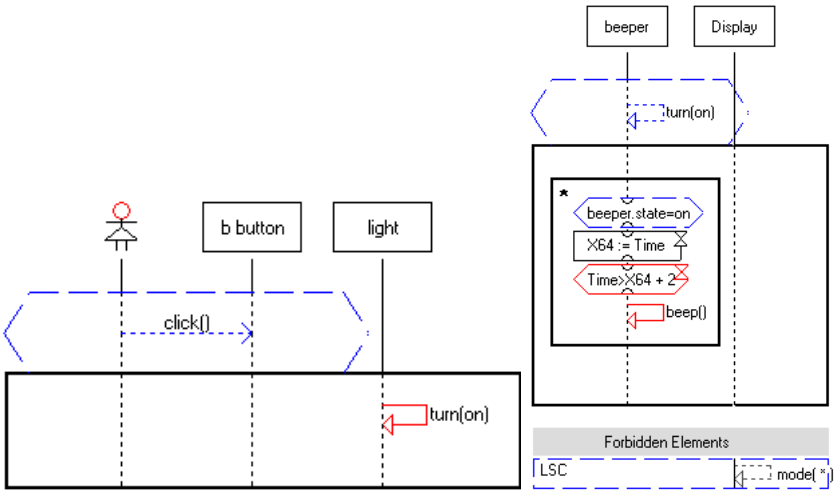


Fig. 3. Sample LSCs. (a) A simple LSC created for the sentence: "when the user clicks the `b` button, the light turns on". (b) A more complex LSC created for the sentence: "when the beeper turns on, as long as the beeper state is on, if two seconds have elapsed, the beeper beeps and the display mode cannot change".

The grammar is inherently ambiguous, due to use of dictionary terminals. The same word could be used for noun, object or property value. We therefore parse each sentence separately and update the grammar as the user resolves ambiguities relevant to the model. Our parser is an active chart parser, bottom-up with top-down prediction [13]. We detect errors and provide hints for resolving them using the longest top-down edge with a meaningful LSC construct. For example a message or a conditional expression that have been partially recognized provide the user with meaningful information.

4 LSC Grammar Constructs

4.1 Example Requirements Translation

We now describe the main parts of our method for automatically translating structured requirements into LSCs. We demonstrate the main language phrases by constructing a simplified version of a digital watch described in [14]. There, the watch behavior was described using statecharts formalism. Here, we describe the same system in natural language and then automatically transform it into

LSCs. Generally, the watch displays the time and can switch between different displays that show (and allow changes to) the alarm, date, time and stopwatch. It has an option to turn on a light, and it has an alarm that beeps when the set time arrives.

An example, taken verbatim from [14] is this: "[The watch] has an alarm that can also be enabled or disabled, and it beeps for 2 seconds when the time in the alarm is reached unless any one of the buttons is pressed earlier". This requirement is ambiguous and unclear for our purposes: when a button is pressed should the alarm time be cancelled or should the beeping stop? Basic user knowledge of the system helps us infer that the beeper should stop. Also, the fact that the alarm beeps only when it is enabled is deduced by common knowledge, as it is not explicit in the text. The structured requirements for these will be: "when the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on"; "when the beeper turns on, if two minutes have elapsed, the beeper turns to off"; "when the user presses any button, the beeper shall turn off". Although the original requirement is fragmented and separated into several requirements, the combined effect of these requirements will achieve the same goal.

4.2 Translating Constructs

In this section we show how our initial grammar translates controlled natural language to LSCs. The grammar is structured and required rigid and clear requirements, however they are natural to understand and compose. Since we allow multiple generations of similar constructs we hope to enlarge the possible specifications. We shall describe how the basic structures — messages and property changes, and some of the less trivial ideas that include parsing temperature, conditions, loops and symbolic objects. Few advanced ideas such as asserts and synchronization are not supported at the current time, nevertheless, the current grammar allows implementing executable systems and has been tested on the digital watch example and on an ATM machine example.

Messages. The simplest language construct in LSCs is the message between objects, or from an object to itself. Messages can be method calls or property changes. In the case of methods, the verb specifies the method to call. For example "the `c` button is clicked" is mapped into a *self message* from the `c` button to itself. Messages can also be specified between objects as in "the user presses the `c` button". Parameters can also be used as in: "the light turns to on", in which case the `turn` method of the `light` is invoked with a value of `on` as a parameter. When a sentence can be fully parsed into more than one basic structure, the user is notified of the location and selects the terminal to use for the word. For example is the button an argument for press or an object with the method press. The user selection is integrated into the dictionary using weights which effectively cause the button in the rest of the text to be an object, unless specified differently.

Temperature. LSCs allow the user to specify whether something may happen, for which we use a *cold* temperature (depicted in dashed blue lines), or what must happen, which is *hot* (depicted by solid red lines). The grammar allows the user to specify the temperature explicitly by using the English language constructs *may* or *must* and some of their synonyms. If the user does not explicitly specify the temperature of the event, it is inferred from the sentence structure. For example, the *when* part is cold and the *then* part is hot. In English it is obvious that the *when* part may or may not happen, but that if it does then the *then* part must happen. See Fig. 4 for an example.

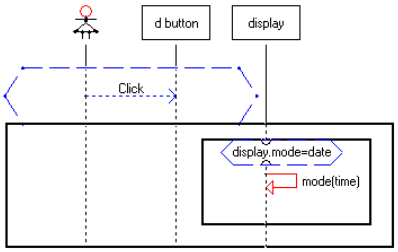


Fig. 4. The LSC created for the sentence "when the user presses the `d` button, if the display mode is date, the display mode changes to time". The message in the *when* part is cold (dashed blue arrow), while the messages in the *then* part are hot (solid red arrows).

Conditions. Conditions, that are frequent in system requirements are readily translated into conditions in the LSC formalism. The grammar accepts expressions that query an object's property values, such as "if the display mode is time". The condition is implemented in the LSC as a cold condition, and all phrases that occur in the *then* part of the phrase appear in the subchart of the condition. The dangling-else ambiguity that appears frequently in programming languages is resolved similar to most parsers by choosing the 'else' that complete the most recent 'if', which is reasonable also in natural text. We allow the user to manipulate the hierarchical structure of the sentence using commas and conjunctions, see, for example, Fig. 5.

Symbolic Objects. In English, definite or indefinite *determiners* are used to specify a specific object or a non-specific object respectively. The determiners are part of the static terminals that differentiate between objects and symbolic objects. Consider the sentence "when the user presses any button, the beeper shall turn to off". The requirement is translated into the LSC of Fig. 6, where the `button` is symbolic (drawn with a dashed borderline) and can be any of the buttons. The LSC semantics also requires that a symbolic object becomes bound

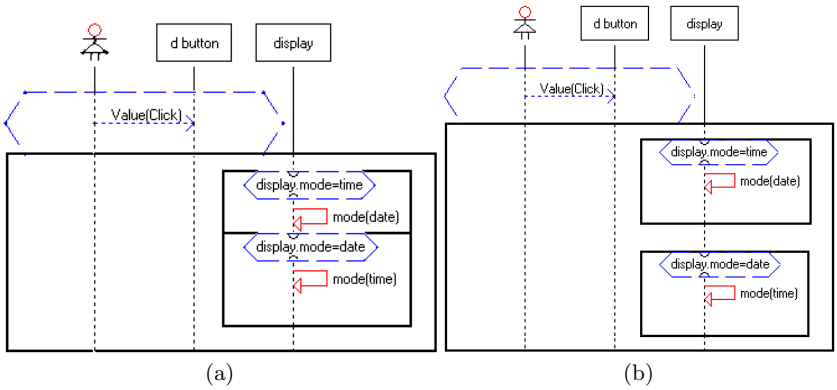


Fig. 5. Conditions in LSCs. (a) The LSC created for the sentence "when the user presses the `d` button, if the display mode is time, the display mode changes to date, otherwise if the display mode is date, the display mode changes to time". (b) Shows what would happen if the *otherwise* would be replaced by an *and*. The second condition is not an alternative to the first, and the behavior would not be as expected. Consider, for example, what would happen if the `display mode` is `time`: the execution would enter both conditions and nothing would happen to the display mode. This behavior could also be avoided by separating the single requirement into two different requirements, resulting in two separate LSCs.

using an interaction with another object or a property. Thus, the sentence "when the user presses a button, a display turns on" is not valid, since the `display` is not bound at all and is supposedly symbolic. It is clear that the sentence is ambiguous also to an English reader, and the user is prompt to resolve the problem.

Forbidden Elements. Our grammar also supports forbidden elements when using negation of messages. For example, "the display mode cannot change" would result in a forbidden element. The scope of forbidden elements is important to the semantics of LSCs; i.e., to what parts of the LSC they are relevant. We use the syntax tree and the location of the forbidden statement in it to resolve the scope, conjunction can be used to verify that a forbidden phrase is inside a subchart. See Fig. 3 (b) for an example.

Forbidden Scenarios. In addition to specifying negative events as forbidden elements, one can also specify forbidden scenarios — scenarios that cannot happen. These are specified using language phrases such as "the following can never happen", prefixing the scenario that is to be forbidden. In the LSC, the scenario described is created in the prechart with a hot false condition in the main chart, which entails a violation if the prechart is completed. To separate the 'when' from the 'then' parts of the scenario, we add a synchronization of all the objects

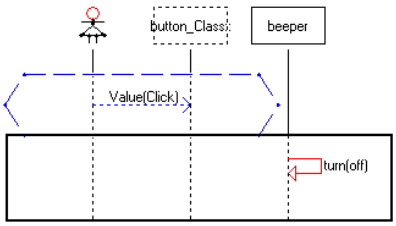


Fig. 6. The LSC created for the sentence "when the user presses any button, the beeper shall turn to off". The `button` object referred to by the user is a non-specific object and is therefore translated as a symbolic object of the button class, shown using a dashed box.

referenced in the scenario at the end of the 'when' part as extracted from the syntax tree.

Additional Constructs. The grammar supports translation into additional LSC constructs, such as local variables, time constraints, loops and non-determinism. It is currently of preliminary nature and is being extended to deal with additional ways of specifying new and existing constructs to make it more natural to users. The fact that sentences are parsed separately allows the use of the ambiguous grammar. Resolution of ambiguity is achieved by interaction with the user to obtain information about the model and by propagating model information between different sentences.

4.3 Implementation and Execution

Once the requirements are parsed and the model is known, the objects and their basic methods are implemented separately with the names extracted from the text. We use the dictionary to extract word stems and we also support word phrases for methods or objects by concatenating the words with a hyphen. We implemented the watch's simple interface with the Play-Engine GUIEdit tool described in [2]. In the final implementation, logical objects that have properties or methods, that do not effect the system visually and do not need additional implementation, are created automatically in the Play-Engine.

The GUI was set up to include the objects low level behavior (e.g., the button's `click`, the light's `turn on`, the time's `increase`). In the future we plan to attempt to connect directly to an existing model by extracting the object names and methods by using reflection on the model and matching them to the specification using synonyms [15].

Requirements were written to describe all aspects of the watch's behavior depicted in the statechart of the watch. A demonstration of the implemented watch is available in [16]. We also implemented another system — an ATM — to test the grammar. Since currently the grammar requires explicit repetition of objects and often needs the user to specify the behavior using a particular

sentence, we would like to extend the grammar and also integrate some form of reference resolution.

5 Related Work

NLP has been used to aid software engineering in many ways. In [17] controlled natural language use case templates are translated into specifications in CSP process algebra that may be used for validating the specified use cases. Use cases are specified in a table containing different steps of user action, system state and system response. Our approach allows inputting information of multiple steps in a single sentence more naturally and integrating different requirements. Our LSCs can also be validated or run (see smart play-out [5]) using model checkers.

There are approaches that generate executable object oriented code from natural language. The approach in [8] uses two-level-grammar (TLG) to first extract the objects and methods (a scheme that may be used for our initial phase as well) and it then extracts classes, hierarchies and methods. In [11], TLG is used to output UML class diagrams and Java code. The methods are described in natural language as a sequence of intra-object behaviors. (In contrast, our approach connects inter-object requirements and appears to be more fitting for reactive systems.)

Attempto Controlled English (ACE) [18, 19] is a user-friendly language, based on first-order logic with rich English syntax, for translating NL into Prolog. It can by used for basic reasoning and queries but not for reactive systems.

Other works assist UML modeling and the design procedures with support tools that help extract the main objects and message sequences from natural language [20, 21], thus making the transition from a NL specification to design less prone to errors. In [21] the scenarios in use cases are parsed to extract a tight representation of the classes and objects for the class diagram.

By and large, we have not encountered a translation that can create a reactive system from fragmented requirements.

6 Conclusions and Future Work

Creating complex reactive systems is not a simple task and neither is understanding natural language requirements. We have presented a method that allows one to translate controlled NL requirements into LSCs, with which a reactive system can be specified. The implementation of the system is thus a set of fragmented yet structured requirements — namely the LSCs, which are both natural and fully executable.

The current situation regarding the execution of LSCs is not without its limitations. For example, LSCs do not always result in a deterministic execution and the execution is also not always optimal. However, there is progress in many directions regarding the execution of LSCs; e.g., using an AI planning algorithm [6] can help the user choose one deterministic and complete path for system execution.

The ability to translate a controlled language into LSCs is a step in the right direction. The translation we suggest is tailored for the LSC language. However, it needs to be extended in order to support more of the rich language that humans normally use.

We would like to extend our scheme so that it becomes reasonably robust to errors, more user-friendly and so that it includes also dialogues that will help users understand how to write controlled requirements. We have yet to test the system on naive subjects.

We would like to add more abilities that will improve the natural language interface with the user. For example allowing specification using language "short-cuts", e.g. using the word *toggles* for changing between a few properties. We would like to add reference resolution, allowing the user to refer to objects previously mentioned as *it*. We would like to integrate NLP tools that resolve aliases for methods and properties, using dictionaries and common sense systems, this would allow the system to understand that different words refer to the same method or property, for example that *click* and *press* are the same method.

Another direction we would like to pursue is to include tools for transforming NL requirements to LSCs and back in a round-trip fashion, to enable easy project modification.

We believe the LSCs and the inter-object approach are naturally close to NL requirements. We hope the work presented here constitutes a small step towards improving the process of engineering reactive systems using natural language tools.

7 Acknowledgments

The authors would like to thank Shahar Maoz, Itai Segall and Dan Barak for helpful discussions and technical assistance. We would also like to thank the reviewers of an earlier version of this work for their helpful comments.

References

1. Harel, D.: Can Programming be Liberated, Period? Computer **41**(1) (2008) 28–37
2. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer-Verlag (2003)
3. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design **19**(1) (2001) 45–80
4. UML: Unified Modeling Language Superstructure, v2.1.1. Technical Report formal/2007-02-03, Object Management Group (2007)
5. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements. Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02), Springer-Verlag (2002) 378–398
6. Harel, D., Segall, I.: Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). (2007) 485–499

7. Mich, L.: NL-OOPS: From Natural Language to Object Oriented Requirements Using the Natural Language Processing System LOLITA. Natural Language Engineering **2**(2) (1996) 161–187
8. Bryant, B.: Object-Oriented Natural Language Requirements Specification. Proc. 23rd Australian Computer Science Conference (ACSC). (2000)
9. Segundo, L.M., Herrera, R.R., Herrera, K.Y.P.: UML Sequence Diagram Generator System from Use Case Description Using Natural Language. Electronics, Robotics and Automotive Mechanics Conference (CERMA'07) **0** (2007) 360–363
10. Drazan, J., Mencl, V.: Improved Processing of Textual Use Cases: Deriving Behavior Specifications. Proc. 33rd Int. Conf. on Trends in Theory and Practice of Computer Science (SOFSEM'07). (2007) 856–868
11. Bryant, B.R., Lee, B.S.: Two-Level Grammar as an Object-Oriented Requirements Specification Language. Proc. 35th Annual Hawaii Int. Conf. on System Sciences (HICSS'02). (2002) 280
12. ITU: International Telecommunication Union: Recommendation Z.120: Message Sequence Chart (MSC). Technical report (1996)
13. Jurafsky, D., Martin, J.H.: Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Prentice-Hall (2008)
14. Harel, D.: On Visual Formalisms. Commun. ACM **31**(5) (1988) 514–530
15. Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.: Introduction to WordNet: An On-line Lexical Database. http://wordnet.princeton.edu/ (1993)
16. Requirements to LSCs Website: http://www.wisdom.weizmann.ac.il/ michalk/reqtolscs/
17. Cabral, G., Sampaio, A.: Formal Specification Generation from Requirement Documents. Brazilian Symposium on Formal Methods (SBMF). (2006)
18. Fuchs, N.E., Schwitter., R.: Attempto: Controlled natural language for requirements specifications. Proc. Seventh Intl. Logic Programming Symp. Workshop Logic Programming Environments. (1995)
19. Fuchs, N.E., Schwitter, R.: Attempto Controlled English (ACE). Proc. 1st Int. Workshop on Controlled Language Applications. (1996) 124–136
20. Takahashi, M., Takahashi, S., Fujita, Y.: A Proposal of Adequate and Efficient Designing of UML Documents for Beginners. Knowledge-Based Intelligent Information and Engineering Systems. (2007) 1331–1338
21. Giganto, R.T.: A Three Level Algorithm for Generating Use Case Specifications. Proceedings of Software Innovation and Engineering New Zealand Workshop 2007 (SIENZ07). (2007)

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.

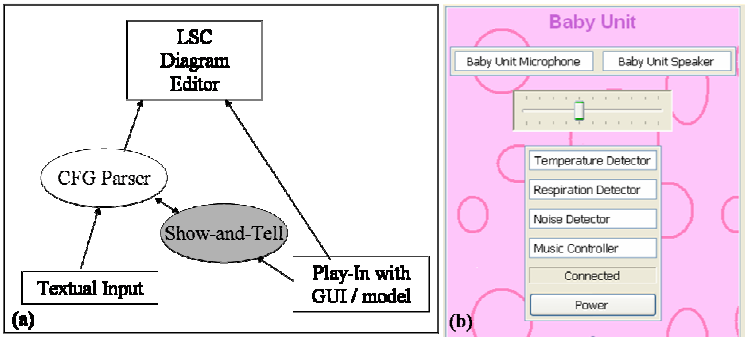


Figure 1. (a) The system architecture. (b) Part of the baby monitor sample application GUI.

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.

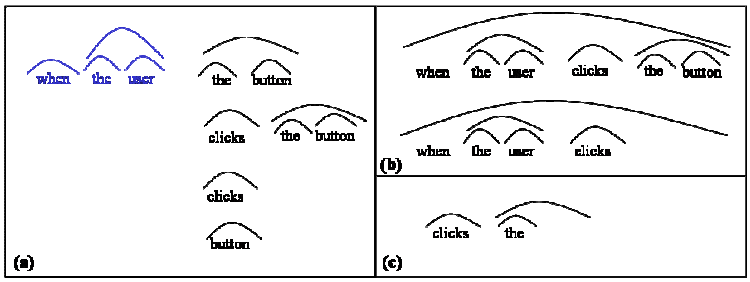


Figure 3. Parse sample for the sentence "when the user" and an interaction of [clicking a button].

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/.

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-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].

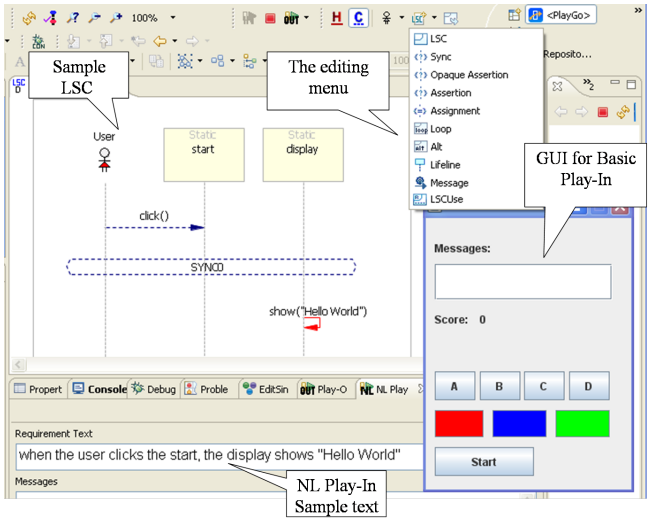


Fig. 1. PlayGo Environment, a sample LSC and the natural language that created it. Also visible on the right is the experiment GUI and on the top the editing menu.

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.

Semantic Navigation Strategies for Scenario-Based Programming

Michal Gordon and David Harel
Dept. of Computer Science and Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel
Email: {michal.gordon, dharel}@weizmann.ac.il

***Abstract*—The scenario-based approach to specification and programming uses powerful extensions of sequence diagrams, such as *LSCs* (live sequence charts), to model system behavior. Previous work in this area presents interesting challenges related to the scalability of the approach and to better tool support for analysis, execution, and comprehension. Here we suggest new semantic-rich ways of viewing sequence diagrams and LSCs for better comprehension of both a single large chart and a full multi-chart specification, in a variety of software engineering tasks. Our method uses weighted messages to create a semantic order that enables semantic zooming and scrolling of different parts of a chart, providing visual hints about context.**

***Keywords*-Live Sequence Charts; Sequence Diagrams; Semantic Zoom; Program Navigation; Program Comprehension;**

I. Introduction

LSCs (*live sequence charts*), a new scenario-based programming language, and sequence diagrams, a popular scenario-based specification language, are both visual formalisms that present major challenges to usability. In the case of LSCs, these challenges are made even more acute due to the multi-modal features of the language and the dependencies between different charts. These features are also found in some other programming paradigms. The challenges include scalability of the presentation, and navigation strategies for comprehension and debug. In this paper, we suggest ways to address some of these challenges by applying a new method of *semantic navigation* to LSCs. Specifically, we define new zooming and scrolling methods for LSCs based on custom weights for diagram elements. Our methods are described for LSCs but can also apply to sequence diagrams, and they can be extended to textual code.

Our custom weights are generated automatically, depending on the task at hand, or manually by the user, and they end up creating a semantic order on the elements of an LSC. This order is different from the spatial order of the elements in the chart, and we define it to assign higher weights to elements that are semantically "more relevant" to the current task. For example, in comprehension, elements that appear only once in the chart may provide more information than ones that repeat.

The order allows semantic zooming and panning on a diagram, while some additional definitions we provide maintain the context, abstracting unnecessary information to assist comprehension in the specific task. More generally, our work can be viewed as the application of the concept of *semantic zoom*, adapted from information visualization and user interface design, to the programming domain in general. We demonstrate the approach and show the new visual notations for the scenario-based visual formalism of LSCs implemented in the *play-engine* tool [1].

II. Background

Live sequence charts, LSCs, are used for specifying multi-modal scenario-based behavior [2], [1]. An LSC describes inter-object behavior, i.e., behavior between objects, capturing part of the interaction between the system's objects, or between the system and its environment. LSCs are an enrichment of message sequence charts (MSCs) [3] and the UML sequence diagrams [4], in which objects are represented by vertical lifelines and messages between the objects are visualized as horizontal arrows, with time advancing from top to bottom, as in Figure 1. Additions include subcharts, which can contain alternative statements and messages, allowing one to specify different behavior under different conditions, as well as loops and synchronization points along the lifelines.

LSCs add modalities of behavior to MSCs and sequence diagrams by, e.g., distinguishing possible (cold) from necessary (hot) behavior (the latter is where the term "live" comes from), and can also express forbidden behavior, i.e., scenarios that are not allowed to occur. A prechart fragment, represented as a blue dashed hexagon as in Figure 1, at the beginning includes cold messages, whose occurrence triggers the main part, depicted as a solid black rectangle as in Figure 1, of the LSC. To execute LSCs, the *play-out* mechanism or its more powerful variants [5], [6] monitors at all times what must be done, what may be done and what cannot be done, and proceeds accordingly. Although the execution does not result in optimal code, nor is the executed artifact deterministic (since LSC can yield an under-specification) it, nevertheless, leads to a complete consistent execution of the LSC specification, if one exists. The details of play-out are outside the scope of this paper, and are described in detail in [5], [1].

LSC is an example of a visual formalism that is sufficiently novel so as to render scalability of presentation and

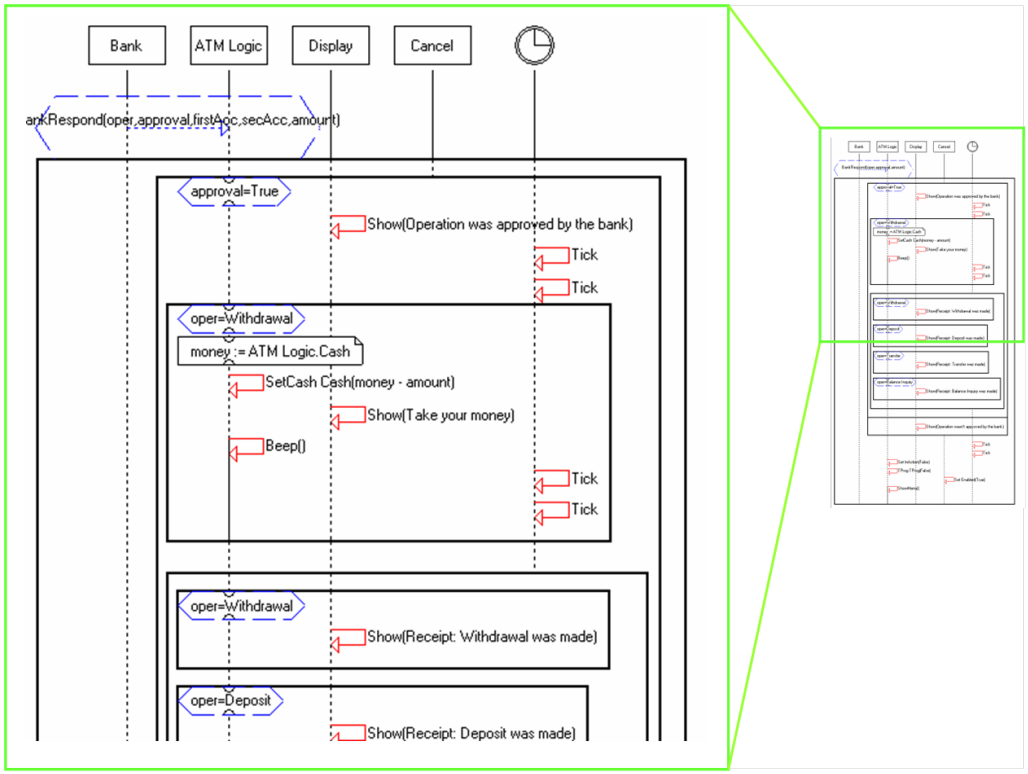


Figure 1. The top half of the full LSC, with readable messages, part of a larger LSC

tool support for analysis a challenge that has not yet been fully addressed. Several pieces if work have addressed the issue of viewing large sequence diagrams (SDs) [7], [8], but they do not address issues of various tasks of comprehension, nor do they apply to the considerably more semantically-complex case of a *set* of diagrams, as when dealing with a full LSC specification. They also lack the ability to deal with the multi-modality of LSCs and the semantical issues these raise. These approaches also seem to require much intervention on the user's side, and little to assist him/her in filtering the information wisely, as would be expected from a truly semantic method of zooming.

Semantic zoom has been used in many fields of information processing. The idea is to balance details and context when displaying information. However, when working with LSCs, there is no clear definition of what the important details are. Here we propose a method for using custom weights, which make it possible to navigate between level of detail in LSCs. The weights are generated to correspond to different tasks, and visualization methods are suggested to allow semantic zoom and navigation.

The current paper centers around LSCs, yet many of the ideas can be used to also contribute to MSCs and UML sequence diagrams.

III. Defining Semantic Order

When setting up a system for carrying out semantic zoom, some form of the relevant details has to be carefully defined, to allow for navigation between different levels. Many programming languages, and LSCs among them, lack explicit definitions of such information. In our case, we found that the additional information would best come in the form of custom weights on the elements of the chart, thus creating a semantic order. When the language supports hierarchy, this information, which induces an order on element-sets, can be integrated with the custom weights. However for navigation to be continuous we require an element-wise semantic order that includes all the elements.

Most programs, including LSCs, can set as the default semantic order the textual/spatial order of the code. However, more meaningful orders, which do not necessarily coincide with the natural order in the program or diagram, can be generated by the user or computed automatically, depending on the task at hand. We shall discuss some suggestions for semantic orders in section VI, and we refer to an LSC with a semantic order as a *weighted LSC*.

More formally, a custom weight $w(m_i)$ is defined for each message m_i of the LSC. Since some elements of an LSC contain others, as in, e.g., subcharts (fragments in UML terminology), the message weight induces a weight for each subchart as the sum of the weights of the messages in the

subchart, $w(s_i) = \sum\{w(m_i)|m_i \in s_i\}$, and a weight on the lifelines as the sum of the weights of messages connected to the lifeline, $w(l_i) = \sum\{w(m_i)|m_i \in cover(l_i)\}$. The custom weights define a semantic order on the elements of the LSC, and ties (equal weights) can be resolved by reverting to the spatial order in the chart.

Consider the LSC in Figure 1. First, the vertical order induces higher weights for messages higher in the vertical dimension, so that message *SetCash* has higher weight than message *Beep*. Second, the horizontal order induces higher weights for messages that are left of other messages (higher in the horizontal dimension) and on the same vertical line, hence message *SetCash*, which is to the left of message *Show(take your money)*, has higher weight. (In the figure they are not on the same vertical line due to tool limitations but they could be on the same vertical line, since there is no order between them semantics-wise.) However, message *Show(take your money)* is also higher than *Beep* and therefore we get the order $w(SetCash) > w(Show) > w(Beep)$. Section VI describes how the weight of an element can be chosen to reflect the information it provides relevent to the task at hand. Thus, the semantic order will allow a viewer to focus on more relevant elements while ignoring others.

IV. Visual Notations

A. The placeholder

Given a particular semantic order to be used (which does not necessarily depend on the "geography" of the diagram), we have to find ways to support the kinds of navigation we want, such as zoom in, zoom out, and pan/scroll at a specific zoom level. We do this by hiding some of the elements, and in LSCs this will apply directly to a message, a subchart, or a lifeline. Hiding an element can be performed anywhere, and since it means removal of information, we add a *placeholder* instead to provide context information and to hint at the fact that data has been hidden. A specific "look" for the placeholder must be devised, which should indicate the location of the missing elements and include some coding that means for capturing the sum of weights of the elements it replaces. In LSCs, the placeholder for a removed message is depicted as horizontal gray lines at the appropriate location. A vertical gray line holds the place for a removed lifeline. The weights are coded by the level of the grayscale color of the placeholder, thus hinting at the amount of information being hidden. When an element is hidden and its weight is added to a specific placeholder, we refer to the element as being consumed by the placeholder. Adjacent placeholders are placeholders that are immediately next to each other, with no interfering elements between them. If, in the process of adding a placeholder at a particular location it turns out that there is already one in an adjacent location, the two are merged into one, and the weights are summed

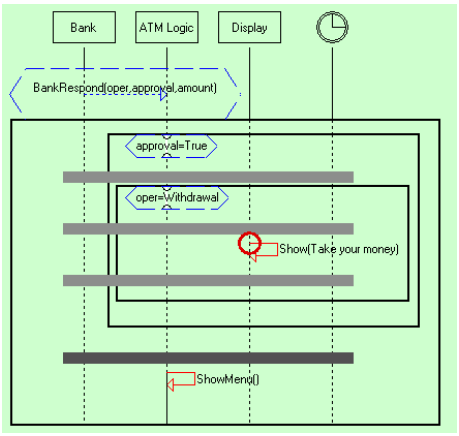


Figure 2. Placeholders not merged due to subchart separation

(and the darkened color will reflect the added weight). Structural containers impose some limitation on the placeholder merge algorithm. Subcharts in LSCs, for example, cannot be merged in a naïve way, since doing so would result in removing structural information. A placeholder inside a subchart can replace the subchart completely after it has assumed the weight of all elements in the subchart. Only then can the placeholder merge with placeholders outside the boundaries of the subchart, thus leaving the structural information intact for as long as necessary. This can be seen in the example in Figures 2 and 3 that show the LSC before and after the hiding operation of a single message that causes a merge operation cascade for four placeholders. The algorithm for hiding elements and merging placeholders is given in the next section.

B. Last change marker

Our navigation allows continuous zooming and scrolling using the scroll wheel (the former also requires holding down the ctrl button). This means that consecutive zoom / scroll steps may be taken but since the semantic order does not depend on 'geography', there is no guarantee that the elements being shown or hidden are in adjacent spatial locations. We therefore mark the last change at each step, using a special *last change marker* and focus on it at each step. In LSCs, we use a red circle; see Figure 2. When the last step added an elements, the last element added is marked as in Figure 2. When last step hid an element, the placeholder that consumed the element, is marked; see Figure 3. When scrolling we mark the added element, or the placeholder that changed due to the hidden element, depending on the direction of the scrolling.

We have also found that it is helpful to provide a clear indication to the viewer when the code/diagram is shown at some zoom level that is not the regular full-detail one. In LSCs we use a light green background for all zoomed diagrams, rather than the normal white.

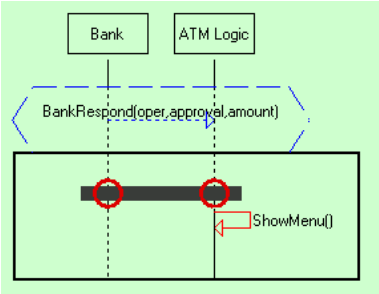


Figure 3. A merged placeholder

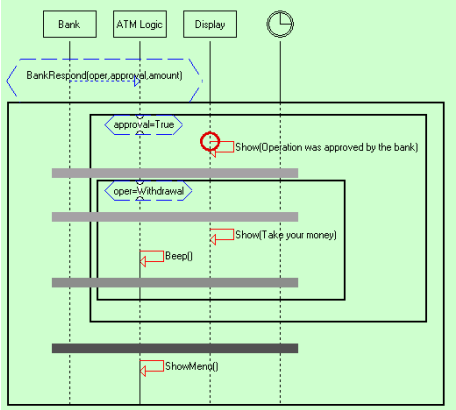


Figure 4. The full LSC zoomed

C. Semantic navigation

1) Zooming: During zoom-out, less information is displayed at each step and the level of detail decreases. Therefore, the element with the least weight is hidden and is replaced by a placeholder. This, in effect, leaves the most informative elements on the diagram, and they will be the last to be hidden.

During zoom-in, more information is displayed at each step and the level of detail increases. Therefore, at each step the hidden element with the largest weight is added and its placeholder is updated accordingly.

This principle can be applied to vertical elements or horizontal ones (i.e., lifelines) when appropriate. In our implementation for LSCs, the zoom is applied to the vertical elements.

In applications that have a limited area available for drawing the diagram, the principle can be used to find the right level of zoom for the area. The area can induce the maximal number of vertical or horizontal elements that can be displayed. We start by hiding all elements and then adding the most informative elements one after another, counting also the number of placeholders that are required at each step to find the optimal zoom level.

2) Scrolling: When viewing a chart at a specific zoom level, there are cases (e.g., when the chart fills the entire available "canvas") that call for allowing the user to scroll and see adjacent details presented according to the semantic order.

In such cases, we allow the user to scroll with a fixed-sized window over the ordered set of elements. For each scroll operation, the elements in the window are shown and all others are hidden and are replaced by placeholders.

3) Filtering: We can easily allow the application of a filter of a specific weight threshold. All elements with custom weight below the threshold will be hidden (and replaced by appropriate placeholders). This will allow the user to focus on the more relevant information.

The filtering operation is essentially setting an exact zoom level, but is carried out without having to go through the continuous changing of the level.

V. The Navigation Algorithm

We now describe the navigation algorithm for hiding weights and merging placeholders. It can be applied independent of the calculations for the custom weights, and has been implemented in the *play-engine* tool. The next section discusses different ways to calculate the weights.

Here we view an LSC as a structure consisting of a tree T of vertical elements taken from the following two sets: a set of *subcharts* S and a set of *messages* M listed by their 'geographical' order. The LSC also contains a list of *lifelines* L. For simplicity, no two vertical elements can be at the same vertical location and therefore each element can be replaced by a single placeholder. Subcharts can have $child$ elements of type subcharts and messages, while messages cannot have child elements (thus, $child(e) = null$ for $e \in M$). Each message m or subchart s has a $parent$ element in the tree. Each message m and subchart s has a non empty $cover$ set of lifelines from L that it is connected to. A message is connected to at most two lifelines.

Since the basic element of an LSC is the message, the navigation is in the vertical dimension and it affects the horizontal elements; e.g. if all the elements connected to a lifeline have been removed, the lifeline will also be removed.

In the zoomed LSC, placeholders from a set P can be added as leaves in the tree. A placeholder p has a list of $contain$ elements from $S \cup M$ that it consumed. Placeholders are added only in the vertical dimension but could also have been added for removed lifelines.

The input to the algorithm is an LSC $\{T, L\}$ with weights calculated as in section III, and the output is a zoomed LSC $\{T', L'\}$. Let E be a list of the messages in T sorted in increasing order by weight $(m_1, ..., m_n)$, ; thus, $w(m_i) \leq w(m_{i+1})$.

For navigation, a copy of the original LSC is created, and at each step the tree structure is updated, and then drawn. The order of child elements in the tree determines their vertical order. If two elements i and j have the same parent

```
procedure: MergePlaceholders(element)
foreach c ∈ child(element) do
  MergePlaceholders(c)
for i ←0 to length(child(element)) - 2 do
  e0 ←child(element,i) //gets i'th child of element
  if e0 is a placeholder
    e1 ←child(element,i+1)
    if e1 is a placeholder  //merge placeholders
      weight(e0) ←weight(e0) + weight(e1)
      contain(e0) ←contain(e0) ∪ contain(e1)
      remove e1 from element //decreases number of elements
      i ←i − 1 //loop increases i, decrease since element removed
  elseif e0 is a subchart and length(child(e0)) = 1
      and child(e0,0) is a placeholder
    e0 ←child(e0) //replace subchart e0 by its child placeholder
    i ←i − 1
```

Table I
THE MERGE PLACEHOLDER ALGORITHM

and i is listed before j, then i will be drawn above j in the vertical dimension.

For zoom-out, we create a copy $\{T', L'\}$ of the original LSC and at each zoom-out step $i = 1, ..., n$, the procedure *RemoveMessage*(m_i) is called. In *RemoveMessage*(m_i), the message m_i is removed from T' and is replaced by a placeholder p with $parent(p) \leftarrow parent(m_i)$, $w(p) \leftarrow w(m_i)$. $w(l)$ for $l \in cover(m_i)$ is decreased by $w(m_i)$ and if $w(l) = 0$, the lifeline l is removed from L' and is not drawn. Then the procedure *MergePlacholders*(T') (see Table I) is called recursively, first on all child elements and than on their parent elements, guaranteeing that subchart elements will also be merged.

For zoom-in, if the zoom-out step is some $i > 1$ then *AddMessage*(m_i) is called for $i \leftarrow i - 1$. In *AddMessage*(m_i), let $p = \{p \in P | m_i \in contain(p)\}$, m_i is added to $parent(p)$ before p, the placeholder's weight is updated $w(p) = w(p) - w(m_i)$ and if $w(p) = 0$ then p is removed from T'. For all lifelines $l \in cover(m_i)$, if $l \notin L'$ then l is added to L', and $w(l) \leftarrow w(l) + w(m_i)$. If m_i is a child of a subchart in T that is contained in p, then m_i is not added directly; rather, the subchart is added and m_i is added as its child.

At a specific zoom-out level, for scrolling up (respectively, down), at each step the message with the smallest index m_i is removed (resp., added) and the message with the largest index m_j is added (resp., removed) using the *RemoveMessage* and *AddMessage* procedures, and then *MergePlaceholders* is called.

VI. MEANINGFUL SEMANTIC ORDERS

The semantic order is determined by weights that are calculated automatically before navigation as a consequence of the task at hand. These weights can additionally be modified manually by the user, but in most cases it is preferable to calculate them automatically. The computation may take time but it is performed offline, and therefore does not affect the user's experience of navigation. This section discusses some possible semantic orders and their applicability.

The spatial order is inherently part of the diagram. It has a clear temporal meaning, which induces a default semantic order for navigation.

Another order mentioned earlier is that created between groups of messages using lifeline composition, as suggested in [9], or class hierarchy, as described in [10]. In composition, a lifeline is composed of additional lifelines defined in a separate diagram. In hierarchy, a lifeline can also be composed of other lifelines (but the semantics is different, of course). Ties within element-sets can be resolved using the spatial order.

The other orders we suggest are heuristic, and depend on the task at hand.

A. Interdependencies for comprehension

One heuristic semantic order connected to program comprehension is related to the dependencies between messages. This order is the result of applying some function on statistics of the elements to calculate custom weights automatically. For statistics, the definition of unification is used. Roughly, when there exists two messages with the same method and connecting the same objects, but in different LSCs, they can be unified. The formal definition can be found in [1] and can be applied also to messages in the same LSC.

For a message m, we define the following local and global statistics. Local statistics depend on the single LSC: let $U_l(m)$ be the number of messages unifiable with the message local to the LSC. Let $C_l(m)$ be the *causal weight*, which is the fraction it constitutes of the prechart. A message that does not appear in the prechart has weight 0, and that of a message that is the only one in the prechart (i.e., its occurrence alone triggers the LSC's main chart) is 1.

Global statistics depend on the entire specification; that is, the full set of LSCs. Let $C_g(m)$ be the number of messages that can either be caused by m (e.g., m is in a main chart and the unifiable messages are in a prechart), or are causal to m (e.g. m is in a prechart, and the unifiable messages are in a different main chart), $U_g(m)$ the number of messages unifiable with m that are not causal or caused by m.

Our current implementation in the *play-engine* tool supports only this semantic order, and we have used equal weights for the local and global components. These weights, $w_l = 0.5$ and $w_g = 0.5$ are parameters that may be changed according to user preferences.

Using the aforementioned statistics, the final custom weight for a message m is the real number: $w(m) = w_l(\frac{1}{1+U_l(m)} + C_l(m)) + w_g(\frac{1}{1+U_g(m)} + C_g(m))$.

The $U_l(m)$ and $U_g(m)$ components make the weight of messages that appear once higher than those that repeat, both locally and globally. The causal components, both local

$C_l(m)$ and global $C_g(m)$ add to the weight of messages that cause changes in the same LSC or between LSCs. In a way this is counterintuitive to indexing methods that give higher weights to repetitive elements. However, it makes sense when the task at hand is comprehension, or, more specifically, comprehending the interdependencies between different LSCs.

Testing this heuristic on sample specifications shows that messages that are used frequently, such as clock ticks or enabling and disabling of buttons, tend to have lower weights, and are indeed less interesting to the reader. See, for example, part of a large LSC from an ATM sample specification in Figure 1 and how when zoomed, the tick messages are hidden before other messages in Figure 4.

B. Semantic order for debug

Another semantic order that plays a significant role in software analysis and debugging is the order of execution. In languages such as LSCs this is not identical to the textual/spatial order. LSCs are of inherent potential nondeterminism, with a partial order existing between messages in a single LSC and subtle behavioral and temporal dependencies between multiple LSCs, with their enabled and forbidden events [2]. And in smart play-out, the execution mechanism plans a series of steps ahead of time [5], so that the notion of order of execution is not a trivial matter that can be read from the text/chart.

When the task is debugging, and the focus of the user is on the recently executed message, the semantic order can change with each debug step to show previously executed messages and future enabled messages with higher weights. This will allow the user to watch a smaller window of proximal messages not necessarily in the spatial sense but in the executable sense. The ability to watch a partial LSC in a small window can clear an area for displaying other relevant LSCs, thus allowing the user to see how the LSCs interact. We have not implemented semantic zoom for debug and we leave the details for future work.

C. Generation order

Another order, which requires additional information that is available during diagram generation, is the order the user chose to add the elements during programming. This order has the value of displaying the user's cognitive process, and how certain elements were added later than others. In many cases, the later additions are the low-level details, or sometimes 'patches' to fix holes in the specification. This semantic order is relevant for comprehension and can be accumulated during the programming process (play-in with LSCs, for example).

D. User selection

A user may want to directly affect the semantic order, for example, in LSCs assigning more weights to some messages, and less to others. As in other interactive works [7], allowing the user to interact with the diagram and specify the details he/she is interested in helps navigation, this order can be combined with a default order to avoid requiring the user to assign weights to all the elements and to help the user avoid excessive interaction. Using the placeholder element for interaction, the user can also choose to expand all elements consumed by a placeholder by double-clicking it, or to extract the single highest weighted element, depending on the tool implementation.

VII. Applications to conventional programming

Reverse engineered sequence diagrams are widely available and are automatically generated by commercial and research tools. In many cases these diagrams are very large and hard to read and navigate. Applying a semantic order and using the suggested algorithms for zooming and scrolling can improve the usability of these diagrams.

Our approach may also be useful in navigating textual code. Although textual code is spatially ordered by lines, there is much information filtering that can be applied to lines of code. Most editors today allow the ability to collapse lines of code that are part of the same function or class. However, code lines have similar dependencies to those of LSC messages. If we replace unification by calls to the same function, we can create a weight function for each line that will provide information on how much this line is repetitive within a code project or a class. This information may be valuable when comprehending code and debugging.

Once the information exists, one can even debug only lines with an information level higher than a certain threshold, hiding other code lines using the suggested placeholder algorithm. For example, one can hide code lines that call a logger or deal with a database, that often repeat throughout the code, although they do not appear in consecutive lines.

VIII. Related Work

The idea of semantic zoom and zoomable user interfaces is fundamental to navigating large information spaces [11], and has been addressed also in the domain of structured textual code [12] and in model engineering [13], [7], [14]. In [12] a *degree of interest* (DOI) is defined, to allow fisheye view of information, a view that distorts information in order to allow focusing on some details rather than others. The idea is applied to textual code based on its tree structure.

Some of the ideas discussed here have been previously researched for textual code. Structure of textual code has been used in [15] for better comprehension and for navigation between components. Different indexing strategies, such as statistical measures for code parts [15] or social tagging by experts [16], have been used for better navigation in large textual code projects.

The navigation problem becomes more difficult when dealing with complex graphical models that present layout.

Challenges that do not exist when reading sequential text [14].

Many navigation solutions exist for class diagrams [13], which have been researched more extensively than LSCs or SDs, and include hierarchies that are exploited for navigation.

More recent work also address navigating and zooming for the full set of UML diagrams, [8], [14]. In [8], various diagrams are connected by special arrows for quick navigation and additionally, semantic zoom has been suggested for interrelationships between elements from different diagrams and for displaying the coarser details of a single diagram. The work in [8] also discusses sequence diagrams briefly, mentioning for example focusing on selected lifeline titles. A similar work that focuses on multiple UML diagrams [14] acknowledges that in UML multi-diagram models are loosely coupled and are therefore hard to navigate. Novel ways have been suggested to integrate different modeling aspects (such as structure, data and behavior) into a coherent model that allows definitions for navigations. In these works, sequence diagrams are treated as one among the many diagrams available in UML.

Recently, sequence diagrams have been acknowledged as important in reverse engineering [7] and novel ways have been suggested to view large diagrams using interactive zooming. They include interactively focusing on parts of the diagram while the context is displayed as small low resolution image and as collapsed fragments in the zoomed diagram. These methods can also apply to live sequence charts, yet they require extensive user interaction.

In the current paper, LSCs are considered as interconnected scenarios in an executable specification and semantic zoom is discussed for navigation and comprehension. New methods for displaying missing information and context are suggested, and less interaction is required from the user when navigating. The formulation of custom weights enables the creation of new detailed orders that are not part of the single LSC, but can provide additional information for different tasks.

IX. Conclusions and Future Work

The main contribution of the current work is to allow semantic navigation in LSCs, a form of visual programming language that does not have a trivial level of details for zooming or navigation.

Nevertheless, some of our ideas can extend beyond the domain of LSCs. Specifically, the idea of creating a semantic order that provides information not found in the original order of the artifacts might be of more general use. Also the idea of creating some form of placeholder for abstracted information, that can hint at the amount of abstracted information for non-continuous regions and merge with other placeholders depending on the 'geography', may be useful for other environments. For example, it is possible to use the notion of placeholder to abstract states in a statechart, if a different visual notation, such as a dot placeholder, is used, and the rule to merge placeholders is adjusted so that there would be a straight line connecting two placeholders, in order for them to merge.

We believe our work can also contribute to the dependency graph between LSCs, when navigating a large specification, as described in [17], and that it can assist in viewing connected LSCs side by side for simulation and debug. Although it is hard to assess how much the new method helps in navigation, we did run a cognitive walk through a large specification of an ATM that also included some large LSCs (Figure 1) to support our claims that the method can assist navigating.

When encountering a large LSC, it is necessary to scan it, sometimes completely and all the way to its end, in order to understand what it 'says'. In the context of LSCs, it is often necessary to scan multiple LSCs to understand how they interact. Our method provides zoom and scrolling that are widely used when reading information that is too large to fit on a screen. The user receives feedback that he/she is viewing parts of the full diagram from the placeholders, and also has knowledge about where the missing information is hidden so that he/she can form a mental model of the sequence of events that occurred, rather than having to scan the full document.

We have implemented the current ideas for the interdependencies semantic order. We plan to create a tool that will work also for UML SDs and will allow interaction and user defined weights. We would also like to evaluate our method in visualizing debug and simulation runs, operations that do not scale well for large LSCs or for a large specification, at the current time.

Acknowledgment

The authors would like to thank Shahar Maoz for preliminary discussions on some of the ideas in this work and for his helpful comments. We thank Liat Nakar for contributing her large LSC specification of an ATM system. We also appreciate the assistance and helpful suggestions of Itai Segall and Smadar Szekely. The first-listed author would like to thank Goren Gordon for his support, general and specific.

References

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

[2] 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. [Online]. Available: citeseer.ist.psu.edu/damm01lscs.html

[3] ITU: International Telecommunication Union, "Recommendation Z.120: Message Sequence Chart (MSC)," Technical report, 1996.

[4] UML, "Unified Modeling Language Superstructure, v2.1.1," Object Management Group, Tech. Rep. formal/2007-02-03, 2007.

[5] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart Play-Out of Behavioral Requirements," in *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD'02)*. Springer-Verlag, 2002, pp. 378–398.

[6] D. Harel and I. Segall, "Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs," in *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, 2007, pp. 485–499.

[7] R. Sharp and A. Rountev, "Interactive exploration of uml sequence diagrams," in *Proc. of the 3rd IEEE international Workshop on Visualizing Software For Understanding and Analysis (VISSOFT'05)*, 2005.

[8] M. Frisch, R. Dachselt, and T. Brückmann, "Towards seamless semantic zooming techniques for uml diagrams," in *Proc. of the 4th ACM symposium on Software visualization (SoftVis'08)*, 2008, pp. 207–208.

[9] Y. Atir, D. Harel, A. Kleinbort, and S. Maoz, "Object composition in scenario-based programming," in *Proc. Fundamental Approaches to Software Engineering, 11th International Conference, (FASE'08)*, 2008, pp. 301–316.

[10] D. Lo and S. Maoz, "Mining hierarchical scenario-based specifications," in *24th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'09)*, 2009.

[11] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot, "Context and interaction in zoomable user interfaces," in *Proc. of the working conference on Advanced visual interfaces (AVI'00)*, 2000, pp. 227–231.

[12] G. W. Furnas, "Generalized fisheye views," *SIGCHI Bull.*, vol. 17, no. 4, pp. 16–23, 1986.

[13] A. Egyed, "Semantic abstraction rules for class diagrams," in *Proc. of the 15th IEEE International Conference of Automated Software Engineering (ASE'00)*, 2000, pp. 301–304.

[14] T. Reinhard, S. Meier, R. Stoiber, C. Cramer, and M. Glinz, "Tool support for the navigation in graphical models," in *Proc. of the 30th international conference on Software engineering (ICSE'08)*, 2008, pp. 823–826.

[15] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proc. of the 23rd International Conference on Software Engineering (ICSE'01)*, 2001, pp. 103–112.

[16] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Waypointing and social tagging to support program navigation," in *Extended abstracts on Human factors in computing systems (CHI'06)*, 2006, pp. 1367–1372.

[17] D. Harel and I. Segall, "Visualizing inter-dependencies between scenarios," in *Proc. ACM Symposium on Software Visualization (SOFTVIS'08)*, 2008, pp. 145–153.

Programming in Natural Language *

Michal Gordon and David Harel

The Weizmann Institute of Science, Rehovot, 76100, Israel
{michal.gordon,dharel}@weizmann.ac.il

Abstract. We describe the idea of programming by writing natural language requirements in order to bridge the gap between system requirements and a final executable system. We claim that formal structured natural language requirements can serve as the mean and the end to programming the behavior of reactive systems, using fully executable languages such as *live sequence charts* (LSC). We use natural language processing methods and a dialog system to resolve and disambiguate English requirements and translate them into the language of LSC. Furthermore, we describe *show & tell*, where the user can specify the behavior by using natural language interspersed with *play-in*, the method of constructing LSCs by interacting with the final system or a mock-up thereof. The method is domain-general and uses a dynamically growing grammar, assists in building the underlying model of the system and leads to direct execution. We demonstrate the approach on various reactive systems, and have also evaluated it as a new programming method.

1 Introduction

The language of *live sequence charts (LSCs)* [8] triggered the grand challenge of "liberating programming", in which more people will be able to define system behavior easily by making programming more similar to how people think [20]. 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. 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 written form of requirements attempts to maximize clarity and reduce ambiguity in order to avoid back and forth changes during the development cycle. Therefore, analyzing the natural language (NL) of requirements and assuming they can be written more formally

* An early abridged version appeared in Proc. of the 10th Int. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing'09), Springer-Verlag, 2009, pp. 456-467.

with the help of tools is a reasonable direction to pursue. In order to simplify this process, the method proposed, termed *natural language play-in (NL-play-in)* assists the technical writers of the requirements (e.g. requirement engineers), using a parser and a dialog system that guide them in writing structured NL requirements. While adding the behavioral requirements, the method also helps the engineer build the system model: the objects, the classes, and the methods, that are relevant to the system. As we shall see, the result is fully executable.

In the past, natural language processing (NLP) has been used in computer-aided software engineering (CASE) tools to assist human analysis of the requirements. One use is in extracting the system classes, objects, methods or connections from the natural language description [39, 3]. Another is applying NLP to use case description in order to create simple sequence diagrams with messages between objects [43], or to assist in initial design [9]. NLP has also been used to parse requirements and to extract executable code by generating object-oriented models using two-level grammar (TLG) [4]. Additionally, controlled natural language (CNL) translations have been used by the Attempto language for reasoning [10, 11]. However, it is important to realize, that 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 object-oriented (OO) artifact is focused on object-by-object specification rather than being directly aligned with the requirements, as are LSCs.

To be able to specify behavior in a natural style, a simple way to specify pieces of requirements for complex behavior is needed, without having to explicitly, and manually, integrate the requirements into a coherent design. This natural style is the basis of behavioral programming and the LSC language.

In [18, 26], the mechanism of *play-in* and *play-out* were described as means for making programming more practical for lay-people. In the play-in approach, the user 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). As a result, LSCs are generates automatically, and on the fly. The play-in method allows to demonstrate parts of the behavior, and some facets thereof are similar to other systems for programming by demonstration (PBD) [7].

In the play-out mechanism, multiple scenarios, or behavior fragments, are integrated into a fully executable artifact. The original play-out algorithm is described in [26]. Since the 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 the set of LSC scenarios have been developed, employing synthesis [30], planning algorithms [19] and model-checking [22] to help alleviate this problem. Here we ignore the issue of how to execute the resulting LSCs, focusing instead on the methods to create LSCs.

Our work describes NL-play-in, a natural language interface to LSCs, based on the classical play-in method, augmented by natural language processing of the specifications. By its nature, the LSC language is closer to the way one would specify dynamic requirements in a natural language. We suggest to take advantage of this similarity, translating natural language requirements directly into LSCs, thus rendering them fully executable. Accordingly, our translation into LSCs can be viewed as a method for executing natural language requirements for reactive systems, or, to put it more succinctly, it enables programming in natural language. Figure 19 shows a screenshot of the PlayGo development environment [25] that implements the idea; it will be described in Section 3.8.

We then proceed to propose a novel integration between NL-play-in and the original play-in method, termed *show & tell (S&T)* [13]. The idea is more than a naïve combination; rather, play-in interaction with the GUI is interpreted intelligently based on the NL text, in which it is interspersed, helping the system guess precisely what the user meant. Furthermore, we believe that our show & tell ideas can be extended to combine interaction and speech interfaces in other domains.

NL programming raises many questions: are the writers of such executable requirements considered programmers? How familiar should they be with the target language (in our case, LSCs)? How much knowledge of logic should they have, and how competent should they be in identifying inconsistencies in the specification, which may later result in bugs or unwanted behavior? We address some of these important issues in later sections.

Section 2 describes the LSC language and the play-in method of creating LSCs. Section 3 discusses the NL interface for LSCs, and provides a detailed example. Section 4 describes show & tell and its implications. Section 5 describes a preliminary experiment with the approach and Sections 6 and 7 discuss related work and future directions, respectively.

2 LSC Language Overview

In its basic form, an LSC specifies a multi-modal piece of behavior, stating what can be done, what must be done and what is forbidden. The behavior is described as a partial ordered sequence of message interactions between object instances. Thus, it can assert mandatory behavior - what must happen (denoted by the "hot temperature"), possible behavior - what can, or may happen (denoted by the "cold temperature"), as well as forbidden behaviors and other possibilities and combinations. The LSC language [8, 26] extends message sequence charts (MSC) [32], which in the UML are termed sequence diagrams [45]. 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.

LSCs extend MSCs with the additional modalities of hot and cold and with a prechart (dashed blue hexagon) and main chart (solid black rectangle) fragments that specify the "if" and "then" parts of the chart, respectively. A sample LSC

is seen in Figure 1. The prechart contains the events that trigger the scenario and if they indeed occur and in the right order, the main chart must occur too. The LSC language also includes conditions, assertions, loops and switch cases. Additionally, in [23, 26] 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.

A variant of the LSC language is shown in Figure 2. This variant, described in [24], is UML compliant and does not require the prechart and main chart fragments, but rather adds the modality of "execute" (as solid arrows) or "monitored" (as dashed arrows) to each message. Our implementations of NL and S&T translate into both variants, with some differences between them, detailed in the following sections.

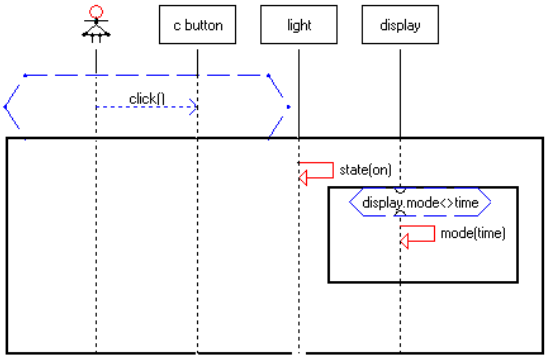


Fig. 1. A simple LSC. The prechart (blue dashed hexagon) contains the cold event (blue dashed arrow) "user clicks the `c` button", while the main chart (black solid rectangle) shows two hot events (red solid arrow): the light state changing to `on` and a hot event with a cold condition (blue dashed hexagon), specifying that if the mode is not `time` then it must change to `time`

.

A set of LSCs is executed using the *play-out* mechanism [26, 27] . *Play-out* monitors at all times what must be done, what may be done and what is forbidden 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 to look-ahead and choose an execution order in a smarter fashion. Details can be found in [26, 22].

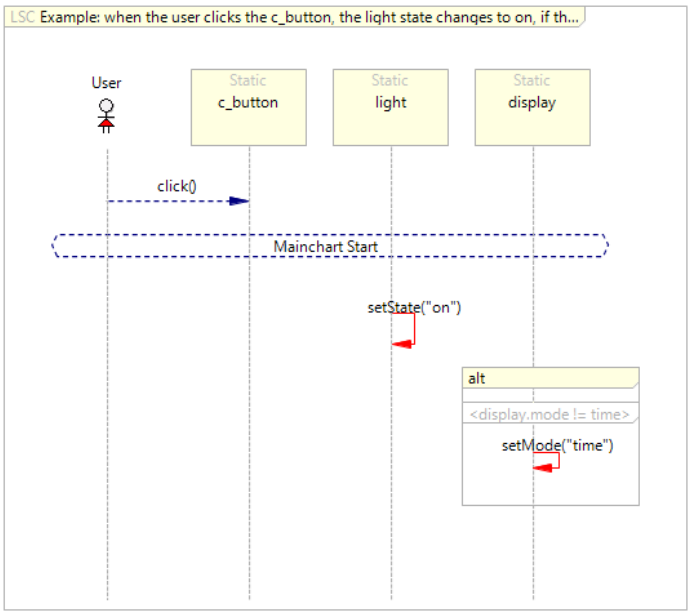


Fig. 2. The same LSC seen in Figure 1 but in the UML compliant form, the main difference is that the prechart does not exist and a synchronization invariant seen with the label 'Mainchart Start' can be added to synchronize events to occur only after the prechart events.

2.1 Basic play-in

In [18], the mechanism of play-in was suggested as a means for programming in LSCs, and at the same time trying to make programming more convenient for lay-people. In this approach, the user specifies scenarios by playing-them-in directly from a graphical user interface (GUI) of the system being developed. The method is related to programming by demonstration (PBD) [7], where programs are generalized from user actions. PBD end-user-programming is meant for customizing software applications using preferences, macro recordings, scripts, etc. In play-in, the demonstration is used to specify formal rules explicitly in order to describe the system's behavior; no generalization is attempted.

During play-in, the developer interacts with the GUI that represents the objects in the system, still a behavior-less one, 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). This method works well for some operations, mostly those that involve GUI, but is a little more cumbersome for other operations or properties, e.g., selection from menus. More complex operations, such as the "if-then" or "when some X then some Y" are less convenient to specify by demonstration, and are done by tools and menus. The NL interface for LSC, which we describe in the next section, attempts to alleviate these difficulties and take a further step towards liberating programming for lay-people. It also does not require a pre-constructed GUI but can connect to an existing one; it can also be used to build a new GUI from the behavior described.

3 Natural Language LSCs

The natural language interface for LSCs consists of a context free grammar (CFG) parser [33], a dialog system, and a knowledge base that also serves as the system model. Figure 3 describes the architecture and the flow. The parser uses a CFG adapted for LSCs and is an extension of the active chart parser of Kay [34]. The grammar is domain general and is composed from a set of rules and terminal symbols. The terminal symbols include static terminals that are used as language directives such as *if, then, must, may,* etc. and dynamic terminals that are relevant to the model and are processed by a dictionary or taken from the working system model.

3.1 NL requirements processing

To recall, a context-free grammar (CFG) is a tuple $G = (T, N, S, R)$, where T is the finite set of terminals of the language, N is the set of non-terminals that represent phrases in a sentence, $S \in N$ is the start variable used to represent a full sentence in the language, and R is the set of production rules of the form $N \rightarrow (N \cup T)^*$. The LSC grammar is defined similarly, with the addition of $T = T_s \cup T_d$, where T_s is a finite set of static terminals with semantics specific to the LSC language, T_d is a dynamic finite set of terminals that is created for each system, and $T_s \cap T_d = \emptyset$.

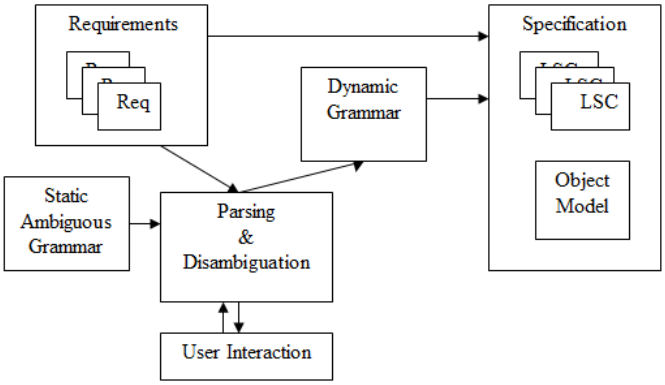


Fig. 3. The architecture of our NL system for LSCs and the data flow therein. The user can affect the parsing process and in turn also the dynamic grammar parts.

Requirements are a way of describing scenarios that must happen, those that can happen, and those that are not allowed to happen. The static terminals in our grammar are used to help describe the flow of the scenario; e.g., "*when* something happens *then* another thing should happen", or "*if* a certain condition holds *then* something *cannot* occur". The dynamic terminals refer to the model, its objects and their behaviors.

The static terminal symbols are *if, then, must, may,* etc. They are important for inferring the semantics of LSCs. The dynamic terminals are processed by a dictionary and transformed from parts of speech to possible parts of the model. They are grouped into `objects`, `properties`, `methods` and `property values`, which need not be mutually exclusive. The dictionary is used to check for word stems and transform words, such as `clicks` or `clicked`, into the stem `click`. It is also used to assign each unknown word to the most probable group according to its dictionary meaning. The parsing process is augmented with a dialog system, in which the user resolves ambiguities or inconsistencies in the entered requirements. Furthermore, the user can choose from a menu the group a particular word should belong to.

For example, in "the user presses the button", `user` and `button` are both objects. Similarly, `press` is a verb that is added to the methods terminal group. Other types of terminals are properties and property values. In the example "the display color changes to red", the noun `color`, which is part of the noun phrase, is a property of the display object and the adjective `red` is a possible property value. Property values may also be used as possible variables for methods.

Figure 4 displays the parse tree for the requirement: "when the user clicks the `b` button, the light turns to on". When analyzing the parse tree, the *when* and *then* determine where the prechart of the LSC ends and its main chart begins, the messages added are `click` from the user to the button in the prechart and `turn` with a parameter `on` in the main chart, as seen in Fig. 5(a).

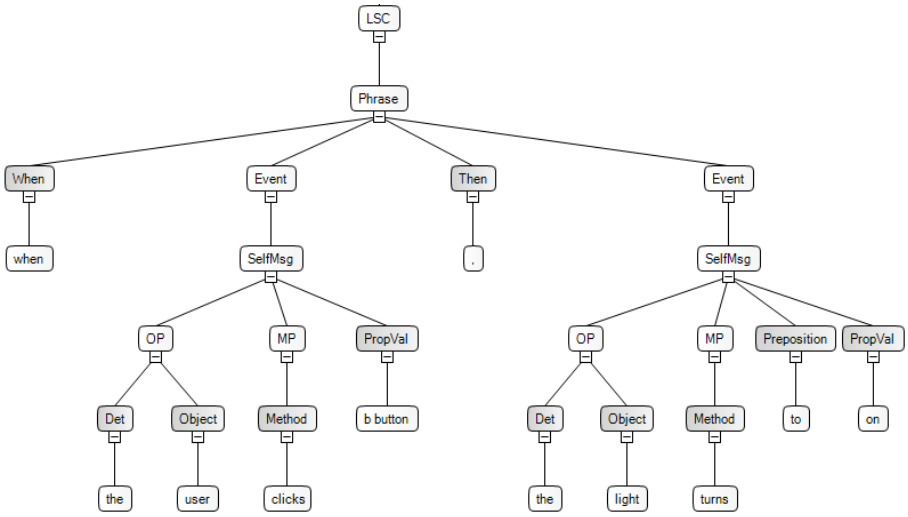
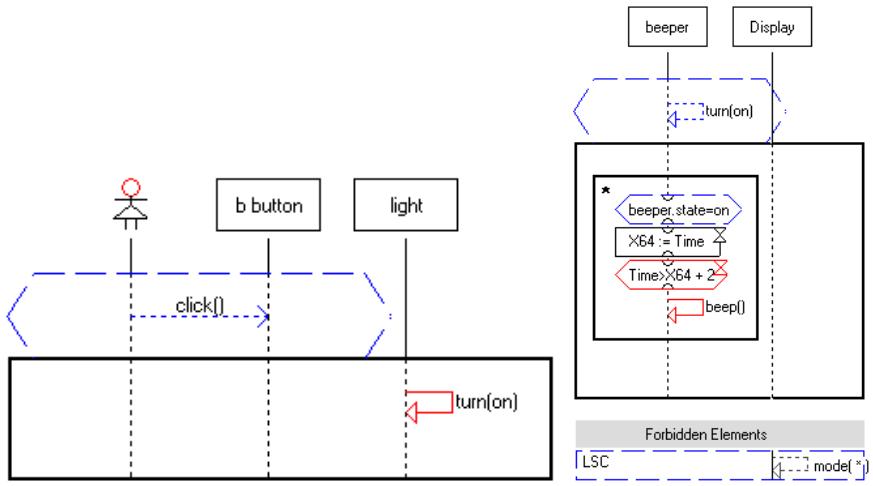


Fig. 4. The parse tree for the sentence "when the user clicks the button, the light turns on". The parts of the LSC grammar are shown in the nodes. There is one message, *Msg*, from object phrase (*OP*) `user` to object phrase `button`, and another self message, *SelfMsg*, of object `light` with method `turn` and argument `on`.



(a) (b)

Fig. 5. Sample LSCs created automatically by our NL system. (a) A simple LSC created for the sentence "when the user clicks the `b` button, the light turns on". (b) A more complex LSC created for the sentence "when the beeper turns on, as long as the beeper state is on, if two seconds have elapsed, the beeper beeps and the display mode cannot change".

The grammar is inherently ambiguous, due to use of dictionary terminals: the same word can be used for noun, object or property value. However, this is no real obstacle, since we parse each sentence separately and update the grammar as the user resolves ambiguities. In some cases, if a model is available as code or as a GUI, some symbols may be resolved by referring to the model. For example, if the model has an object named `beeper` then chances are that the word `beeper` will be an object.

Our parser is an active chart parser, carrying out bottom-up parsing with top-down prediction [33]. We detect errors and provide hints for resolving them using the longest top-down edge with a meaningful LSC construct. For example, a message or a conditional expression that has been partially recognized provides the user with meaningful information. In Section 3.4, we describe additional methods for optimizing the dialog with the user and the interface for resolving ambiguities.

3.2 The digital wristwatch system

We now describe the main parts of our method for automatically translating structured requirements into LSCs. We demonstrate the main language phrases by constructing a simplified version of the digital wristwatch described in [17]. There, the wristwatch behavior was described using the statecharts formalism. Here, we describe the same system in our controlled natural language and then automatically transform it into LSCs. Generally, the watch displays the time and can switch between different displays that show (and allow changes to) the alarm, date, time and stopwatch. It has an option for turning on the light, and an alarm that beeps when the set time is reached.

An example, taken verbatim from [17] is this: "[The wristwatch] has an alarm that can also be enabled or disabled, and it beeps for 2 seconds when the time in the alarm is reached unless any one of the buttons is pressed earlier". This requirement is ambiguous and unclear for our purposes: when a button is pressed should the alarm time be canceled or is the intention merely that the beeping stop? Basic user knowledge of the system helps us infer that the beeper should stop. Furthermore, the fact that the alarm beeps only when it is enabled is deduced by common knowledge, as it is not explicit in the text. The structured requirements for this example is: "when the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on"; "when the beeper turns on, if two minutes have elapsed, the beeper turns to off"; "when the user presses any button, the beeper shall turn off". Although the original requirement is fragmented and separated into several simpler ones, the combined effect of these will achieve the same goal.

3.3 LSC grammar constructs

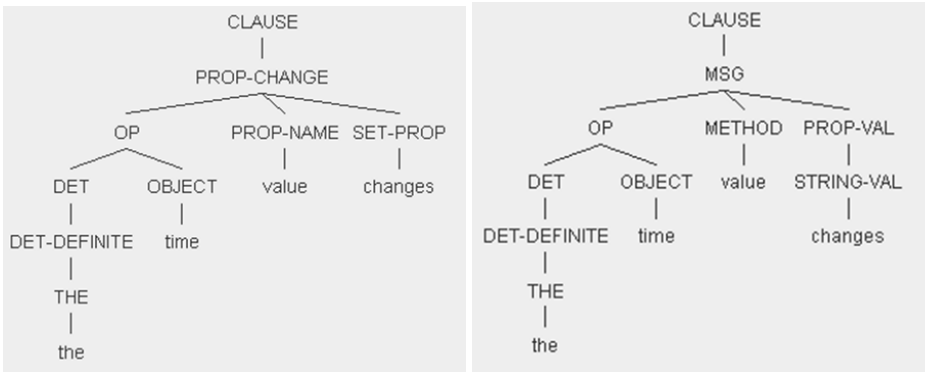
We now show how our grammar translates controlled natural language into LSCs. Since we allow a CFG we can increment the grammar with additional rules that allow various ways of generating similar constructs. We can thus increase the

set of accepted specifications by augmenting the grammar. However, ambiguity may grow as the grammar grows which would require the user to explicitly disambiguate his intentions in too many cases for the process to be friendly. The fact that sentences are parsed separately allows resolution of ambiguity in various ways, as described Section 3.4. We shall describe how the basic structures — messages, property changes, and some of the less trivial ideas that include temperature, conditions, loops and symbolic objects — are parsed. A number of advanced ideas, such as asserts and synchronization, are not yet supported in the current implementation. Nevertheless, the current grammar allows implementing fully executable systems, and has been tested, among other examples, on the digital wristwatch, on an automatic teller machine, and a baby monitor system.

Messages. The simplest language construct in LSCs is the message between objects, or from an object to itself. Messages are of two types: method calls or property changes. In the first type, the verb specifies the method to call. For example "the `c`-button is clicked" is mapped into a *self message* from the `c`-button to itself. A message between objects, like "the user presses the `c`-button", results in a message between the user and the c-button. Parameters can also be used, as in "the light turns to on", in which case the `turn` method of the `light` is invoked with a value of `on` as a parameter. In the second type, the message is some change in a property value, and the property is used as an adjective in the sentence. For example, in "the `display color` changes to yellow", color is a property of display and the message sets the property value to yellow.

When a sentence can be fully parsed into more than one basic structure, there is a grammatical ambiguity for at least one word. The user is notified of the ambiguous word location and he/she may select the correct word sense from a list of possible terminal symbols. For example, if nothing is known about the dynamic terminals of the system, the sentence "the time value changes" can have two complete parse options, both valid. Is `value` meant as a noun —- "a numerical quantity" — or is it used in the sentence as a verb — "to estimate or assign worth of"? In the first case `value` can be a property of `time`, and in the second, a method. Figure 6 shows the two possible parses. In our system, the user is prompted disambiguate between the possibilities, by selecting one of the options and his/her selection changes the dynamic dictionary terminals. The selection effectively causes `value` in the subsequent text to be interpreted as a one interpretation rather than another, avoiding additional disambiguation.

Temperature. LSCs allow the user to specify whether something may happen, captured by a *cold* temperature (depicted using dashed blue lines), or what must happen, which is *hot* (depicted using solid red lines). The grammar allows the user to specify the temperature explicitly by using the English language constructs *may* or *must* and some of their synonyms. If the user does not explicitly specify the temperature of the event, it is inferred from the sentence structure. For example, the *when* part of a sentence is cold and the *then* part is hot. In



(a) (b)

Fig. 6. Two parses for the same sentence. (a) `Value` is used as a noun and interpreted as a property of time. (b) `Value` is used as a verb and is interpreted as a method with the argument `changes`.

English it is implicit that the *when* part may or may not happen, but that if it does then the *then* part must happen. See Fig. 7 for an example.

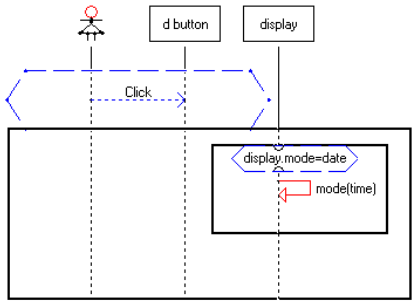


Fig. 7. The LSC created for the sentence "when the user presses the `d` button, if the display mode is date, the display mode changes to time". The message in the *when* part is cold (dashed blue arrow), while the messages in the *then* part are hot (solid red arrows).

Conditions. Conditions, frequent in system requirements, are readily translated into conditions in the LSC formalism. The grammar accepts expressions that query an object's property values, such as "if the display mode is time". The condition is implemented in the LSC as a cold condition, and all phrases that occur in the *then* part of the phrase appear in the subchart of the condition. The dangling-else ambiguity that appears frequently in programming languages

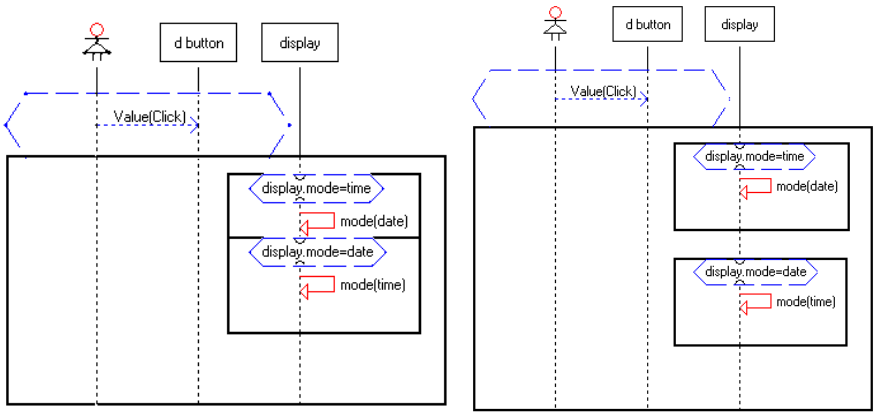


Fig. 8. Conditions in LSCs. (a) The LSC created for the sentence "when the user presses the `d` button, if the display mode is time, the display mode changes to date, otherwise if the display mode is date, the display mode changes to time". (b) Shows what would happen if the *otherwise* were replaced by an *and*. Clearly, these two give rise to very different behaviors.

is resolved in a way similar to what is done in most parsers, by choosing the 'else' that completes the most recent 'if', which is reasonable also in natural text. We allow the user to manipulate the hierarchical structure of the sentence using commas and conjunctions; see, for example, Fig. 8.

Symbolic objects. In English, definite or indefinite *determiners* are used to specify a specific object or a non-specific object, respectively. The determiners are part of the static terminals that differentiate between objects and symbolic objects. Consider the sentence "when the user presses any button, the beeper shall turn to off". The requirement is translated into the LSC of Fig. 9, where the `button` is symbolic (drawn with a dashed borderline) and can stand for (and during execution will be instantiated by) any of the buttons. The LSC semantics in fact require that during execution a symbolic object become bound using an interaction with another object or a property. Thus, the sentence "when the user presses a button, a display turns on" is not valid, since the `display` is not bound at all and is supposedly symbolic. It is clear that the sentence is ambiguous also to an English reader, and the user is prompted to resolve the problem.

Forbidden elements. Our grammar supports forbidden elements, by using the negation of messages. For example, "the display mode cannot change" would result in a forbidden element. In LSCs, the scope of a forbidden elements is crucial, of course, specifying the parts of the LSC to which they are relevant [26]. We use the syntax tree and the location of the forbidden statement within

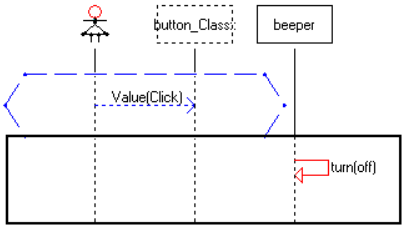


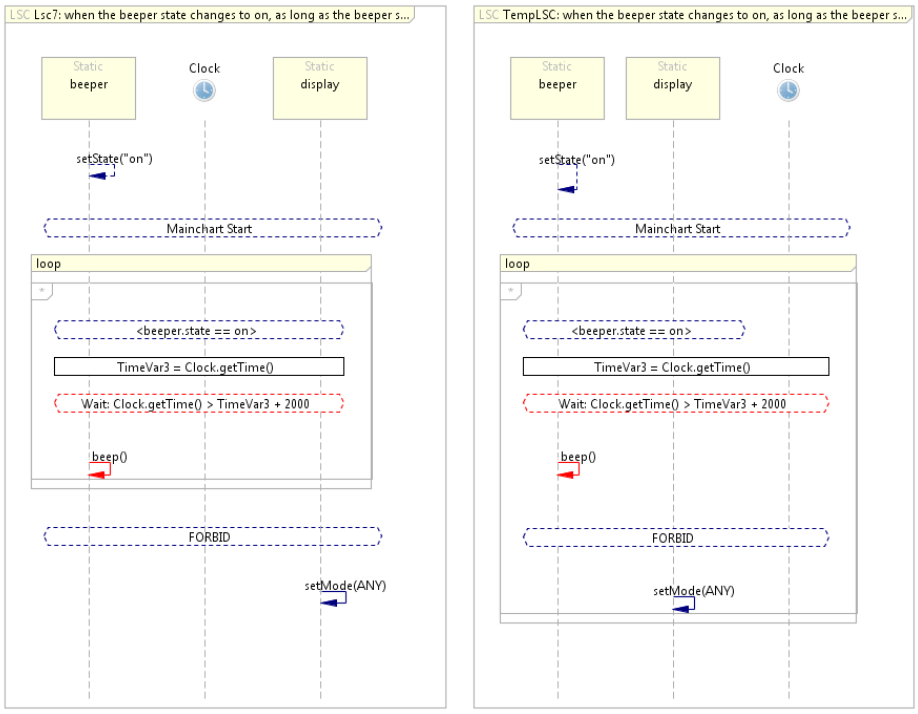
Fig. 9. The LSC created for the sentence "when the user presses any button, the beeper shall turn to off". The `button` object referred to by the user is a non-specific object and is therefore translated into a symbolic object of the `button` class, shown using a dashed box.

it to resolve the scope. Conjunction can be used to verify that a forbidden phrase is inside a subchart, as demonstrated in Figure 10. A forbidden element scope is by default the LSC chart.

In Play-Engine, the forbidden elements are displayed in a separate section at the bottom of the LSC chart, and the scope can be set to any of the LSC subcharts, for example, in Fig. 5 (b) the mode change of display appears in the bottom area. The scope is a property of the element and is not displayed directly on the chart but rather through interaction with the chart. In PlayGo forbidden elements appear in the chart after a `forbid` assertion. The scope is the parent chart of the assertion and the forbidden element, see Figure 10.

Forbidden scenarios. In addition to specifying negative events as forbidden elements, one can also specify forbidden scenarios — entire scenarios that cannot happen. These are specified using language phrases such as "the following can never happen", prefixing the scenario that is to be forbidden. In the LSC, this can be done by putting the scenario in the prechart with a hot false condition in the main chart, which entails a violation if the prechart is completed. To separate the 'when' from the 'then' parts of the scenario, we add a synchronization of all the objects referenced in the scenario at the end of the 'when' part, as it is extracted from the syntax tree.

Additional constructs. Our grammar supports translation into additional LSC constructs, i.e., loops, time constraints, local variables, and non-determinism. For example, the following requirement translates into an LSC with a loop, "when the beeper state changes to on, as long as the beeper state is on, if two seconds have elapsed, the beeper beeps". The loop created is unbounded, controlled by a test of whether the `beeper state` is `on`, using a cold condition, see Figure 5(b). When the condition evaluates to false, the loop will exit. The LSC created also refers to the passing of time using `two seconds have elapsed`. This part translates to saving the current time in a local variable and then adding a condition that *waits* until two seconds elapse from the saved time; see Figure



(a) (b)

Fig. 10. An UML compliant LSC created in PlayGo with a cold forbidden event of display mode changing. The forbidden message `SetMode(ANY)` appears below a forbid assertion. In (a), the scope of the forbidden message is the chart, and in (b) it is the loop subchart that contains the forbid assertion and the forbidden message.

5(b). The grammar includes static terminals, that support time references; e.g., seconds, minutes, hours, etc.

Another possibility, for advanced users, is to refer to variables. Variables in LSCs are locally defined inside a specific LSC. In the sentence appearing in Figure 11, some variables are defined and are later referred to in the controlled English.

Some additional terminal symbols augment the grammar with the ability to specify nondeterministic or probabilistic choices. In Figure 12, the example adds a feature of a psychedelic light that uses this ability.

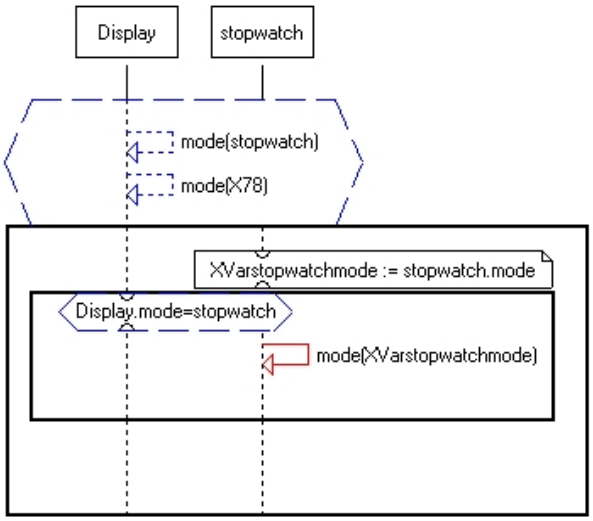


Fig. 11. The LSC created for the sentence "when the display mode changes to stopwatch and the display mode changes, store the current stopwatch mode, and then if the display mode is stopwatch, set the stopwatch mode to the last stopwatch mode". The `store` and `last` terminal symbols allow refering to the local variable. When the display mode is stopwatch and then it changes to some other mode, the last stopwatch mode is saved and restored when the display mode changes again to stopwatch.

3.4 Resolving problems with the help of the user

To resolve imprecise and ambiguous requirments with the help of the user, we have implemented a dialog system using an interface similar to the *quick fix* interface of programming environments like Eclipse IDE [6]. The quick fix interface is quick and comfortable; it draws a squiggly underline for words or sentence parts that have a problem, notifying the user that a problem exists and at which location. Hovering over the problem location with the mouse provides an explanation of the problem. When available, resolve options are displayed in

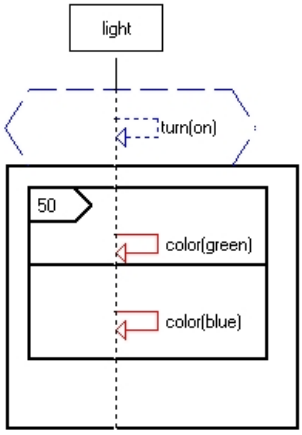


Fig. 12. The LSC created for the sentence "When the light turns on, sometimes the light color changes to green and other times the light color changes to blue". Here the choice is governed by a 50-50 possibility split, but other probabilities can be used explicitly.

the pop-up dialog that describes the problem. The user 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 13 shows a sample problem and available solutions. The quick fix interface displays the problem on the text itself — the focus of the user's attention when entering requirements through text — allowing him/her to fix it with a single click.

Our NL tool works in three stages:

1. Grammatical analysis.
2. LSC disambiguation.
3. Connecting elements to the model.

The first phase checks that the sentence conforms to the LSC grammar and parses it. In the second phase,if more than one full parse for the input sentence exists, the first point of difference between the leaves of the two trees is found and the difference in interpretation is displayed on the relevant word. For example, in the partial sentence "the controller turns the knob", when no information about model elements exists, one interpretation is a message `turns` from object `controller` to `knob` and another is a self message `turn` from object `controller` to itself with the message parameter of `the knob`. The first point of difference in the text is `the knob`, which is an object in the first interpretation and a value in the second. Therefore, the `the knob` text will be displayed with a squiggly line and the two resolve options will be object and value.

In the third phase, we first check that the terms the user refers to appear in the *system model*. The system model is a tree structure containing all the

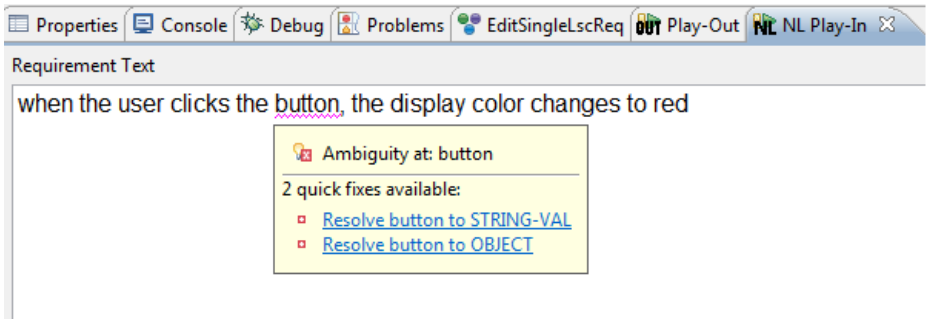


Fig. 13. A sample of quick fix options for resolving an ambiguity for `button`. The problem is indicated by the squiggly line and the quick box dialog that appears when hovering with the mouse over the problem displays the problem and possible solutions' when available. Selecting a solution performs the necessary changes.

objects in the specification, each object with its properties and methods, with the relevant signature. It thus represents the classes and object instances relevant to the system specification. Figure 14 displays the system model of the wristwatch example. Some objects may be represented in the GUI, while others may not. The system model also requires that each object is of a specific class type, and hierarchy between classes is allowed.

The system model serves as the knowledge base for the system and allows verifying that requirements refer to existing objects, asking the user if he/she would like to create new objects, classes, properties, etc. This type of analysis helps the user make the necessary connection between various requirements and the GUI or the model of the system. For example, if the user wrote about the `click` of the `D-button`, the method will now check that a `D-button` object exists as part of the system model. If it does not exist, but an object with a similar name does (similarity here is tested using the Levenshtein string distance [37]), one of the possible resolve options will be to change the object to the existing object.

Another type of similarity that can be tested is using the dictionary to find word synonyms. In other words, a user writing `click` in one place and `press` in another, while referring to the same object, is probably referring to the same operation.

When no similar method/object is found, the user can decide to use the quick-fix menu to add a new one. This way, the NL requirements are automatically connected to the model, to the GUI (if one exists), and among themselves, since the system helps the user refer to the existing terms when possible.

3.5 The NL algorithm

We now describe the algorithm that translates the NL requirements into LSCs and resolves errors with the help of the user. The parsing is performed in a

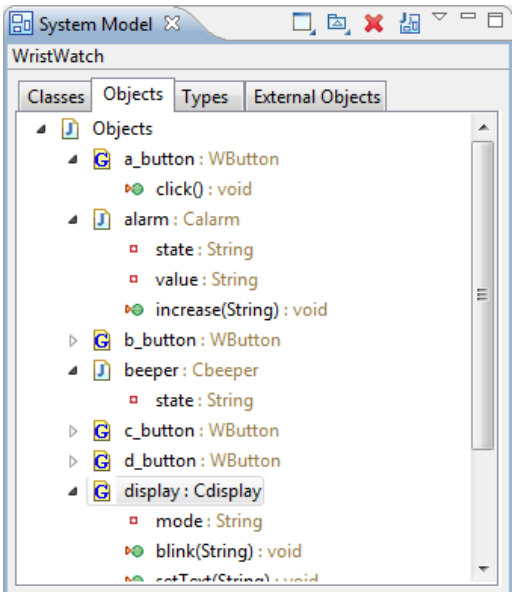
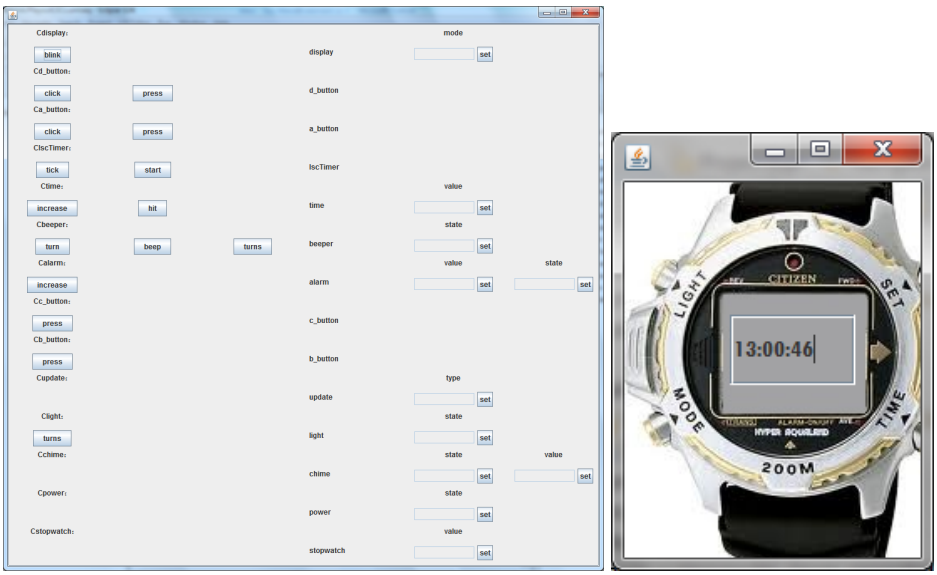


Fig. 14. The system model view with a model of the wristwatch. The view contains different tabs. The main tabs are the classes tab with the classes, their methods and properties, and the objects tab shown here, with the objects, their methods and properties. Objects that are created from a GUI application are marked with a `G`,



(a) (b)

Fig. 15. The wristwatch GUI. (a) The system GUI generated automatically from the model. (b) A graphical GUI created manually, connected to the LSCs

bottom-up fashion, online, in an incremental manner, using an active chart parser [33]. The grammar's definitions include semantic information for creating an abstract-syntax-tree from the parse output, as described in [33]. Table 1 lists some of the grammar rules and the respective semantic rules used to create the LSC parts. See Figure 16 for details on the algorithm.

Once a full parse is found, the following actions are performed in order. If more than one full parse is found, LSC disambiguation is performed, s described in Section 3.4. Following this phase, an LSC is created using the semantic information extracted from the parse tree. The parse tree is analyzed recursively until all edges are processed using the semantic rules of the grammar. Each text part is mapped to the grammar rule that parsed it and to the LSC portion it will generate. The LSCs induce also the model: objects, methods and properties. The algorithm checks that the model parts exist, and if not it displays a problem, allowing the user to select from the quick fix options, e.g., adding new elements to the model, or changing to existing elements. When there are no more problems, the user saves the text and the LSC is created automatically.

Error information helps the user complete the sentence assuming the longest edge in the active chart parser is the most relevant, and displaying the next expected symbol. While this may not yield a correct parse, it constitutes a heuristic guess on completing an incomplete sentence. Example problems in this case are: "you need to add a `then` part" or "you need to add a `property name`", etc. This process helps the writer understand the types of sentences accepted by the language. As explained in Section 5, an initial evaluation shows that this is something we can indeed expect from the user.

3.6 Writing rigorous requirements

The NL-play-in process helps the writer to be more rigorous by, e.g., requiring him/her to refer to terms already in the model clearly, and using the same naming conventions. If, for example, in one requirement there is a reference to a user *clicking* a button, and in another requirement to *pressing* a button, the connection between click and press needs to be determined. This interaction creates awareness that a new method is added, which can be reused in subsequent requirements. It also makes the writer more aware of the objects, methods and properties, thus better connecting new requirements to existing ones.

When the model system (with or without a GUI) has been designed in advance, requirement writing is easier, since the system helps the user refer to already existing terms, and there is no need to resolve ambiguities that can be clarified by what already exists.

An additional possibility when writing rigorous requirements is to help check that the requirements have some effect, by being referenced in other requirements, or by having an effect in the code that augments the LSCs. For example, a light turning on, or a communication channel opening, can be implemented in low-level code, triggered through the LSCs. When a writer feels he/she has a complete set of requirements, we could, in principle, check how "connected" the requirements are and obtain a list of methods or properties that are not

Rule	Semantics
LSC → MainPreClause	Create LSC
LSC → Forbid Then Clause	Create forbidden LSC
MainPreClause → When Clause-Cold Then Clause-Hot	Add prechart and mainchart
Clause-Cold → Clause	Create cold clause
Clause-Hot → Clause	Create hot clause
Clause → Clause Connect Clause	Add two clauses
Clause → Cond-Clause	Conditional clause
Clause → Loop-Clause	Loop clause
Clause → Msg	Method call clause
Clause → Prop-Chage	Property change clause
Clause → Time-Change	Time change clause
Loop-Clause → While Expression Then Clause	Create a loop with condition
Msg → Op [Temp] Method Op [Prop-Val]	Create a method call message
Msg → Op [Temp] Method [Prop-Val]	Create a method call message
Msg → Op Method Op Prop-Name	Create method call with a property as argument
Prop-Change → Op [Temp] [Set-Prop] Op Prop-Name [Prop-Val]	Create a property change message
Prop-Change → Op Prop-Name [Temp] Set-Prop [Prop-Val]	Create a property change message
Expression → Op Prop-Name Compare Prop-Val	Create Condition Expression
Forbid → the following can never happen	
Op → Det Object	Create object
Op → Det Object with Prop-Name Prop-Val	Create object with condition
Connect → Then	exit scope
Connect → And After That	add sync
Connect → And Only Then	add sync
Connect → And	add
Det → Det-Indefinite	indefinite determinant
Det → Det-Definite	Definite determinant
Det-Indefinite → a | an | any | all | some | other | another	symbolic object
Det-Definite → the	instance of an object
Temp-Hot → must | shall | should | will	hot element
Temp-Cold → may | could | can | does	cold element
Temp-Hot-Not → Temp-Hot not	hot forbidden element
Temp-Cold-Not → Temp-Cold not | cannot	cold forbidden element
Set-Prop → turn | change | set | turns | changes | is set | sets	
Proposition → to | from | by | on | in | of	
When → when | whenever	
Then → then | , | do	exit scope

Table 1. LSC Grammar Rules: the semantics for each rule appear in curly brackets some of them enumerations and other function calls. Symbols in square brackets are optional, and rules with a right hand side with | refer to different possibilities for the right hand side.

connected to other LSCs. This can help verify that the methods or properties have some low-level effect, or alternatively, prompt fixing the requirements.

Consider, for example, a blinking state of some light. Either the blinking state should be used somewhere else in the requirements, or it should trigger some change in a hardware/low-level behavior of the system being described. A connectivity analysis can help remind the writer that he/she wanted to add a description of what blinking means, or prompt him/her to implement it as a low-level operation. This addition is analogous to analyzing code and showing a programmer unreferenced functions, as is done by dependency analysis tools.

3.7 The chicken and egg dilemma

In [26, 27], there is an assumption that a behavior-less GUI is given, with which the user can carry out play-in. However, this is not always so. Rather, the GUI requirements often become clear only during the process of describing the requirements. Dually, when playing with the GUI, additional behavioral requirements emerge, which, in turn, require additional GUI requirements. And so on, in a cyclic fashion. Thus, in many cases the behavior and the GUI are developed side by side; initial behavior defines the initial system, and as the GUI interface comes to life it triggers additional behavior, which requires changes to the interface and the model, etc.

With this in mind, we have designed NL-play-in to support both directions. If a GUI/model exists as a behavior-less application, its terms will be integrated into the grammar's dynamic terminals and the requirements will be easily connected to them. On the other hand, if requirements define non-existent objects or methods, these can be added to the model by a single click in the quick-fix interface, and later the GUI can be enhanced accordingly.

Low-level operations can also become part of the model when entering the behavioral requirements. For example, the turning on of a light may require a change in the final engineered system, but this is not necessary at the LSC programming stage.

An additional capability for playing with a GUI-less system includes automatically producing a simple GUI from the LSC behavior. Since the requirements produced induce a system model, we generate a simple GUI from it, called *system GUI*, that displays all the objects, allows viewing and changing the properties, and activating the possible operations. In our case studies we found that this generated system GUI is a good solution for non-programmers or for programmers with no GUI experience.

Figure 15 shows the system GUI produced for the wristwatch, side by side with the graphical GUI created manually. In Section 7, we discuss automatically generating a better GUI by extending the grammatical analysis of the language requirements to find graphical objects directives.

3.8 Implementation and execution

To implement the wristwatch, we used PlayGo [25], an Eclipse based IDE that supports the UML compliant version of LSC from [38]. PlayGo was augmented with a plug-in that implements the NL-play-in, using the WordNet dictionary [41] for the dynamic terminals, via the RiTa Java library [31].

In the wristwatch example, the GUI was set up to include the objects' low-level behavior (e.g., the button's `click`, the light's `turn on`, the time's `increase`). PlayGo can support extracting the GUI object names and methods directly by using reflection on the model.

Requirements were written to describe all aspects of the wristwatch's behavior depicted in its statechart specification [17]. A demonstration of the implemented wristwatch is available in [46]. Other example systems and their NL requirements appear in [], including an ATM system and a baby monitor system.

4 Show & Tell

Show & tell integrates the NL interface with play-in. There are obvious benefits to writing (*telling*) requirements in a natural language, but playing-in with the GUI in order to *show* objects and operations also has advantages. Therefore, the idea is for the user to make the best use of both. The user can enter requirements textually but can also create parts of the sentence automatically by interaction with the GUI, without explicitly writing object names or actions.

The term *show & tell* is used to denote the process of showing the audience something and telling them about it. When a GUI object has a graphical representation, of a slider for example, then showing the dragging action in the midst of creating a textual requirement may be more convenient than describing it textually. The same goes for pointing a button on a toolbar.

Show & tell combines the interaction with the text depending on the context. For example, the writer enters the text: *"when the user"*, and then clicks the `A button`, in this case adding *"clicks the A button"* automatically to the text. The algorithm interprets the interaction based on the current state of the text and finds the best completion. The user can also just enter the *"when"* and click the same button, in which case the algorithm adds *"the user clicks the A button"*. The grammar generates possible and reasonable complete suggestions for the text. The longest one is added to the text, and the quick fix interface provides additional possibilities.

To explain how the combination is carried out, we use some definitions from chart parsing algorithms, and refer the reader to [33]. A given action of the user creates a list of possible parse edges. Each of these includes a grammar rule with input text that can generate it, a location, and the words that create it. The edge is of the form $r, [i, j, k]$, where $r \in R$ is a grammar rule, i and j are the start and end indexes of the text added by the edge, and k is the dot index that specifies which right hand symbols of the rule have been found for each edge. If all symbols have been found, then the dot index equals the size of the rule's right

```
public void ChartParse(chart, agenda) {
while(agenda){
      ProcessEdge(agenda.pop());
} }

private void ProcessEdge(edge) {
   AddToChart(Edge edge) //if not in chart already
   if(edge.isComplete()) {
      ForwardProcess(edge);
   }
   else //complete edge {
      BackwardProcess(edge);
      BottomUpPredic(edge);
} }

private void ForwardProcess(A → α • Bβ, [i, j]) {
   foreach(Edge B → γ•, [j, k] in chart)  {
      AddToAgenda(A → αB • β, [i, k]) //if not in agenda
} }

private void BackwardProcess(B → γ•, [j, k]) {
   foreach(Edge A → α • Bβ, [i, j] in chart)  {
      AddToAgenda(A → αB • β, [i, k]); //if not in agenda
} }

private void BottomUpPredict(B → γ•, [i, j]) {
   foreach(Rule A → Bβ in grammar)  {
            AddToAgenda(A → B • β, [i, j]); //if not in agenda
}  }

public void Initialize() {
   foreach(rule r in grammar) {
      if(rule.getLeftSymbol == startSymbol) {
         AddToAgenda(new Edge(rule));
}  }  }

public void ProcessNewWord(word, i)  //online parsing  {
   AddToAgenda(W → word•, [i, i + 1] // where W is the rule for the word
   ChartParse(chart, agenda);
}

public void ProcessUserInput(possibleEdgeList, inputLength) {
   duplicateChart = chart;
   duplicateAgenda = agenda;
   foreach(Edge edge in possibleEdgeList) {
      AddToAgenda(edge);
   }
   ChartParse(chart, agenda);
   possibleTexts = FindBestParse(chart, inputLength);
   foreach(Text text in possibleTexts) {
      bestText = MaxLength(text, bestText); //longest text better
   }
chart = duplicateChart;
agenda = duplicateAgenda;
}

private ListOfTexts FindBestParse(chart, inputLength) {
   possibleTexts = null;
   foreach(Edge edge in chart) {
      if(edge.getStartIndex() == 0 and
         edge.dotIndex > inputLength) {
            text = getWords(edge); returns word terminals of an edge
            possibleTexts += text;
}  }  }
```

Fig. 16. The show & tell algorithm: (a) contains the general active chart parsing 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 finding it in the input. 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 $[i, j]$ indicate where the edge begins on the input and where it ends, to allow combining edges. The online parser is initialized and then calls `ProcessNewWord` for each word entered. (b) shows the `ProcessUserInput` that is called when a user action is performed, and initiates the search for the interaction's meaning.

side list of symbols and the edge is *complete*, otherwise, it is *incomplete*. The parsing algorithm we use is an online version of an active chart parser [34] with similar definitions. Our interaction edge is of the form $r, [i, j, k], (w_1, ..., w_n)$, with the addition of $w_1, ..., w_n$, the words that the edge may add as a result of the interaction. The ability to specify end and start indexes means that an action can add not only a single edge following the parse text, but also a sequence of edges. The interaction actions add only complete edges, assuming a user performing an action never refers to an incomplete edge, since edges are aligned with rules and thus with semantic meanings; see, for example, Figure 17(c).

The edges created by an action are added to a temporary copy of the current parse chart, and are marked as interaction edges; they are processed using bottom-up prediction and new edges are inserted to the parse temporary chart. The algorithm, then searches for the best possible parses. These have to be edges with end index j, longer than the input index before the interaction, and start index $i = 0$. Thus, they are a parse of the entire input. The longest such edge is selected and its words are displayed to the user. Additional edges that meet these conditions are displayed as alternatives for the interaction using quick fix. A more formal description of the algorithm is provided in Figure 16 reprodeced from [13], Figure 18 shows how show & tell is connected to the NL parsing architecture.

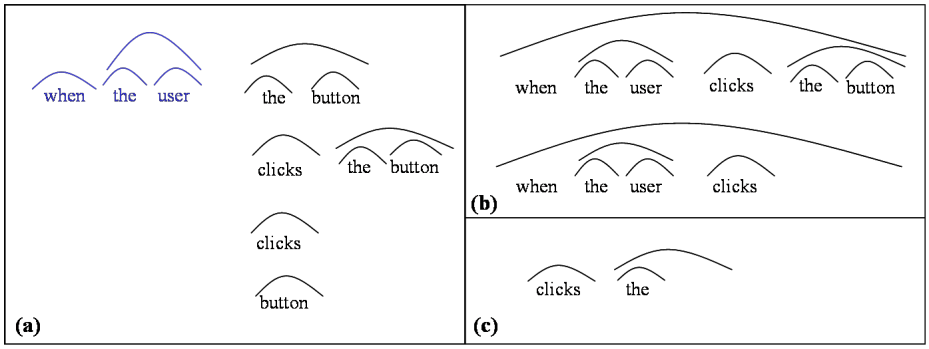


Fig. 17. Possible additions for an interaction: (a) the prefix text and some possible interaction outputs; (b) two possible edges from the combination; (c) a possible incomplete addition but incomplete edge, as discussed in Section 4, which is not a reasonable suggestion of the algorithm.

Although the current implementation of show & tell requires showing and writing, it can easily be extended by using speech recognition to actually "tell". This requires using speech engines with low error rate, which are becoming feasible in recent years and will probably become more abundant in the future. We have tested an implementation of NL-play-in using speech with the Microsoft Speech API (SAPI 5.1) engine [40] and obtained good results with native English speakers. See the empirical evaluation in Section 5.

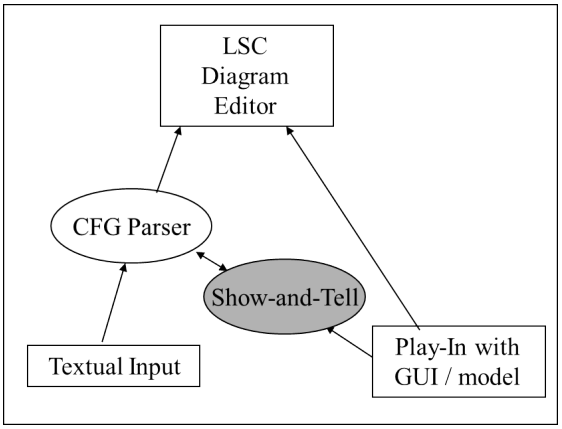


Fig. 18. The connection between show & tell and the other system components.

A similar combination of voice and gestures (as opposed to writing and showing) has been used for managing graphical spaces with "put-that-there" [2]. In that work, the user uses voice commands to request adding, moving and manipulating objects on a screen, but can also use the voice pronoun reference of `this, that, here, there`, and point a hand to show the referenced objects or locations. Show & tell integrates text and GUI manipulation to assist in the creation of system requirements, in a domain generic application.

The idea of show & tell has been described here in the context of programming. However, combining spoken or written language with actions or gestures, controlled by a context-free grammar, can be generalized to many kinds of interactions. In fact, show & tell can be used as a generic architecture for combining text and gestures, and we plan to substantiate this claim in future work.

5 Preliminary Evaluation

We have conducted an exploratory experiment comparing play-in, NL-play-in and show & tell, as well as comparing programming in LSCs and Java, see [14].

Our main questions were: (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 duration, and when considering user preferences? The experiment was conducted with twelve participants who had basic knowledge of the LSC language, as taught in the course on visual languages described in [21]. Some also had experience as Java programmers. Table 2 provides details of participants' previous experience.

Table 2. 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	None	Pen and paper LSCs

5.1 Methods

To deal with the first question, we asked participants to program various tasks using four methods: basic play-in, NL-play-in, show & tell, and direct construction of the charts. The latter method entailed creating the diagram by drawing it, dragging and dropping elements. For the second question, we asked the participants to program very similar tasks once in LSCs and once in Java. Half started with LSC and the other half started in Java to counterbalance effect of programming a similar second task. All participants worked in the Eclipse IDE with PlayGo. This allowed them to start with a given GUI, (see Figure 19), and add the behavior in LSCs or in Java. The participants were provided with a tutorial on each method for creating LSCs.

The tasks we provided for the LSC and Java experiment were small, and included implementing an unknown game that teaches letters and colors. Table 3 shows some sample tasks.

Table 3. Sample tasks

T1	The LSC in Figure 19.
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 shows "Success".
T3b	When the user succeeds the score is increased, and 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.

The participants noted the begin and end times of each task. They also answered questions in writing after each task, and were interviewed at the end of the entire experiment. In tasks $T1$ and $T2$ they were asked to program the required behavior using all four methods, and then write which method felt quickest and which of them they preferred. The order between the four methods changed between participants to counterbalance effects of programming the same behavior repetitively. In task $T3$ they were asked to program the required

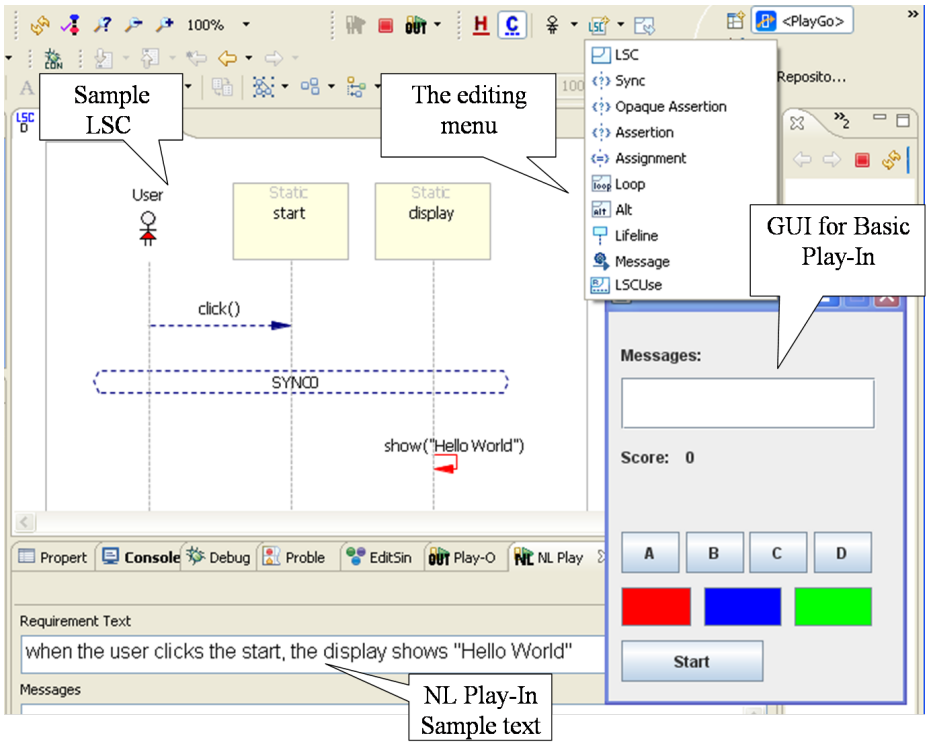


Fig. 19. PlayGo environment, with a sample LSC and the natural language that created it. The GUI used in the experiment appears on the right and the editing menu is above it.

behavior with their preferred method and explain why they preferred it. Seven out of the twelve participants completed the third task also in Java, the order was counterbalanced, so that three participants started with the Java task.

5.2 Results

Of the four interfaces, NL-play-in was preferred by nine out of twelve participants. Those who did not prefer it mentioned technical problems and personal preference for exact and well-defined syntax. They also suggested that for the learning phase, templates or more examples would have made it easier for them.

We compared task times using a paired two-tailed T-test and found that NL-play-in (5.25 ± 3.6 minutes) and play-in (5.3 ± 3.5 minutes) were quicker than direct chart creation (7.6 ± 2.9 minutes). We found that show & tell suffered from implementation bugs and was considered inconvenient, since it required interspersed 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 are also certain that replacing typing by speech recognition will, to a large extent, eliminate this problem. We also found that the Java task implementation time (29.6 ± 8.8 minutes) was comparable to the implementation time in LSC (25.7 ± 5.5 minutes).

In answers to questions during and at the end of the experiment, most of the participants preferred the NL-play-in interface to the other interfaces, and to Java. It is interesting to note that most participants reported that NL-play-in felt quickest to them even when it was not quickest considering implementation duration. Some of the comments collected were that entering requirements in English, resolving ambiguities, and then seeing the final generated LSC was fun and rewarding. Errors were hard to account for, mainly because we did not ask the participants to execute their artifact due to time constraints (the entire experiment took approximately three hours).

Regarding the learnability of the LSC language, many of the participants had a tough time completing the NL tasks at the beginning of the session, and expressed some frustration with the language. However, about an hour later, they exhibited satisfaction when using the NL interface and getting quick results. For example, when interviewed, one participant who said "I was getting confused with NL" in T2, later explained his choice of using NL in T3 as follows: "When you get used to its English, it's quite fast to use". When asked in a later interview regarding the NL interface learnability, the same participant answered: "The beginning was tough, but I got used to it pretty quickly. Not sure the start can be eased, it was pretty quick". This sequence of responses is representative and was consistent for most of the participants.

6 Related Work

NLP has been used to aid software engineering in many ways. Some methods help create models and support the design process [44, 12] without yielding a

complete executable artifact. These methods can extract from the text objects and message sequences, and help reduce errors in the design process. For example, in [44], use case scenarios are parsed to extract a representation of the classes and objects for the class diagram.

In the remainder of this section, we review various methods that translate natural language (or a subset thereof) into an executable artifact. These methods have similar motivations to ours, and each comes with its own style and benefits. One should remember, however, that these are all NL in full and none offer the ability to combine NL with GUI-based actions, as in show & tell.

In [5], use case templates, written in controlled natural language (CNL), are translated into *process algebra* (CSP). A Microsoft Word plug-in checks adherence of the use case specification to the CNL grammar. The models allow checking refinements between different model views and system property validation. They support user view use cases, which specify user operation and expected system responses, and component view use cases with one component that invokes an action and another one that provides the service. The translation into CSP is similar to ours. However, in [5] the CNL is defined as table entries in use cases, rather than more natural sentences with commas and conjunctions as we do.

In [35], a *behavior tree* (BT) notation is used, and a state machine is synthesized, rendering the BT executable. Although the authors refer to their process as execution of natural language requirements, the execution is actually of the BT and there is no automatic translation of the natural language requirements into BT. Quoting from [35]: "Behavior Tree models are developed directly from natural-language system functional requirements by a stepwise process of first translating the behavior expressed by individual requirements into a partial tree and then integrating the fragments together to form a complete tree". In our work, the translation from the natural language is automatic and the LSCs created are directly traceable, and can then be directly executed.

There are additional approaches that generate executable object-oriented code from natural language. *Two-level-grammar* (TLG), [3, 4], is an object-oriented requirement specification language with a natural language style. It is sufficiently formal to allow automatic transformations into UML class diagrams and into object-oriented code, such as Java. The methods are described in natural language as a sequence of behaviors. The language includes sequences of events and separation into services/functions that are later referred and called. It is a way of describing object-oriented code and operations textually using function definitions. Each function definition is composed of logical rules executed in the order they are given.

Table 4 contains an ATM example from [4]. It shows the writing style of TLG in the left column, and the right column describes similar behavior written in our input NL. Our approach interleaves fragmented requirements at execution and appears to require less design from the writer's side.

In *spoken Java* [1] by Begel and Graham, programmers can describe their Java program orally in natural language, and the relevant code is automatically

Table 4. Textual requirements for an ATM. The left column is taken verbatim from [4]. the right column describes similar behavior in NL-play-in.

TLG	NL for LSCs
The bank keeps the list of accounts. Each account has thre integer data fields: ID, PIN, and balance. The ATM machine has 3 service types: withdraw, deposit and balance check. For each service first it verifies ID and PIN from the bank.	When a user withdraws an amount, if the account state is verified and the amount is less than or equal to the account balance, then the account balance is decreased by amount.
Withdraw service withdraws an amount from the account of ID with PIN in the bank in the following sequence: First it gets the balance of the account of ID from the bank, if the amount is less than or equal to the balance then it decreases the balance by Amount, updates the balance of the account of ID in the bank and then outputs the new balance.	When a user enters an ID and the user enters a PIN, the account with ID is checked, if the account verifies PIN, the account state changes to verified.

created. The method was developed for programmers who suffer from repetitive strain injuries, and therefore the natural language is very similar to Java and programming knowledge is a prerequisite.

Attempto controlled English (ACE) [10, 11] is a textual language for writing functional requirement specifications. It is based on first-order logic with a rich English syntax. ACE uses declarative sentences that can be combined to form powerful composite sentences. It also includes some forms of anaphora to make the language concise and natural. Example sentences in ACE for an ATM machine, called SimpleMat, are:

> The customer enters a card and a numeric personal code that SimpleMat checks. If the personal code is not valid then SM rejects the card.

ACE is translated into first order logic clauses that are collected to form a knowledge base. The user can then submit queries to the knowledge base, and the answers are given in ACE. ACE is used mainly for reasoning on the web and for querying a knowledge base, and in [11] it has been translated into Prolog. Still, we believe that with minor modifications ACE could be translated into LSCs. In fact, NL-play-in can benefit from many features already implemented in ACE, such as anaphoric reference resolution. We indeed plan to pursue this line of work. We should add that ACE is tailored for the reasoning domain and not for reactive system programming.

In [36] a domain specific structured English interface is created for robot controllers and motion planning. It is template-based and maps directly to linear temporal logic, supporting safety and liveness properties. The implementation is through a simple grammar that translates into LTL. Some sample behavior text from [36] is: 'Environment starts with false.', 'Always not Dummy', 'Robot

starts in r1 with false', 'Activate Beep if and only if you are in r9 or r12 or r17 or r23', 'Go to r1', 'Go to r3', etc. Since LSCs can specify safety and liveness too, we believe that NL-play-in can be applied to the robot controllers domain with minor adaptations.

7 Future Work

Our work can be extended significantly. The NL tools can be better implemented as a mature development environment for non-programmers. For example, the transformation process can be extended to transform NL requirements to LSCs and back in a round-trip fashion; this will enable friendly project modification in natural language. Additional capabilities include showing the connection between different textual requirements and allowing intelligent navigation between them. Such an extension could be made possible with the aid of navigation and visualization techniques for viewing multiple LSCs and their inter-connections; see [29].

Learning algorithms can be applied to extend the source language according to user preferences. The system could track repetitive errors that a user makes and accept them as his/her personal way of writing. For example, in our experiments we noticed that many non-native English speakers forget to add the *"the"* determinant before object references. They create sentences such as "when user clicks button". These repeated errors could be learned and integrated into a more convenient user-specific grammar.

We can extend the grammar to include behavior "shortcuts" for certain systematic behaviors, for example using the word *toggles* for changing between properties. For example, instead of "when x happens, if the display color is red, the display color changes to yellow, if the display color is yellow, the display color changes to red", we could write "when x happens, toggle display color between red and yellow. Such shortcuts are valuable when writing the requirements for large reactive systems in specific domains.

Our language processing tools can be extended to include spell-checking, anaphoric reference resolution, and additional NLP capabilities [16, 33], allowing the user to refer to objects previously mentioned in the sentence by, e.g., *it*. Similarly to the voice references of "Put-That-There" [2], textual pronouns and anaphoric references could be resolved. This would allow sentences such as "when the user clicks the button, its color changes to red", relaxing the need for explicit naming, similar to what is done in ACE [10]. Tools from NLP can furthermore be integrated to resolve aliases or synonyms for methods and properties, using dictionaries and ontology systems.

We can integrate learning methods to expand our source language according to new specifications added by the user. This could, in some cases, be used to add grammar rules dynamically; e.g., when a user wrote a sentence that was not part of the language and then created the LSC chart directly. Although automatically extending a grammar from a single bad example is a hard problem, the suggested

process is simpler since additional information is available from the final LSC and the model.

It should be possible to exploit the NL input to enrich the system model or GUI automatically. For example, if a requirement discusses the click of some unknown button, a button can be created and added to the GUI. The current implementation supports automatic construction of a GUI but does so without considering the nature of objects and methods, providing a uniform rendition of them all. Creating a 'smarter' GUI would require handling the initial layout of the objects and should provide the user with the ability to edit the automatic suggestion.

To further liberate programming and grant non-programmers the ability to program, we would like to add shortcuts and simplifications to NL-play-in. This includes integrating concepts and ideas from simple and widespread programming languages meant for a larger community. See *Scratch* [42] for example, which includes sharing, tinkering, and a web learning platform. Scratch makes programming easy: users avoid syntax issues since they program by connecting existing blocks of code while learning mathematical and computational ideas. Remarkably, the Scratch environment allows combining different stories into a program [15], similar to in LSCs, and more generally, behavioral programming. Scratch exposed the world of programming to a large community of children, and we have hopes that our work will help expose another community of writers to the programming world.

Future work should extend the way people program with other natural interfaces, such as, speech recognition, which is becoming abundant, and gesture interfaces with devices like Microsoft's Kinect. These interfaces, combined with telling a story in natural language should allow simple creation and modification of programs in the spirit of show & tell but smarter.

Consider, for example, a smart home system in the future. Advanced interfaces detect the home-owner's spoken commands in every room and analyze his/her gestures to find the objects or locations referred to in the commands. The future smart home should obviously allow the home-owner to boil water, fill a bath, or turn on the air-conditioning by oral commands. However, a programmable smart home should also allow the home-owner to modify the smart home behavior using scenarios/rules that are personally suited for him/her. Behavioral programming would work behind the scenes to verify that a connection exists between the various rules, and that they can indeed operate together. For example, the home-owner could request that "If the temperature is below 12 degrees Celsius and my car enters the garage, the air-conditioning should start heating the living room". Adding references to locations, a user could specify: "when no one has been in this room for over ten minutes, close this and that" pointing to the air-conditioning unit and to the living room light, respectively, referring by "this room" to the room he is standing in.

The potential of our work is in generating scenarios/rules using combinations of natural language and gestures, and in recognizing inconsistencies between scenarios and informing the user. For example, if at some point the user requested

that "As long as there is no one in the house, the living-room heating should never be turned on". This requirement would contradict the first requirement, and the user should get an indication.

8 Conclusions

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, with which a reactive system can be both specified and executed. Moreover, we have expanded the power of this NL interface by combining it with play-in pointing, to yield the hybrid show & tell method. The implementation of the system is thus a set of fragmented yet structured requirements — namely the LSCs, that 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. Some of the key features include formal rules, fragmented and scenario-based descriptions in the spirit of behavioral programming, executable and verifiable rule system with an intelligent natural interface. These idioms come closer to 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 in the execution methods; see for example, [19, 22, 23, 30, 28]. 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.

9 Acknowledgments

The 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 to DH from the European Research Council (ERC) under the European Community's FP7 Programme.

References

1. A. Begel and S. Graham. Spoken programs. In *Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing*, VL/HCC'05, pages 99–106, 2005.
2. R. A. Bolt. "Put-that-there": Voice and Gesture at the Graphics Interface. *SIGGRAPH Comput. Graph.*, 14(3):262–270, July 1980.
3. B. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proc. 23rd Australian Computer Science Conference*, ACSC'00, 2000.

4. B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In *Proc. 35th Annual Hawaii Int. Conf. on System Sciences (HICSS'02)*, page 280, 2002.
5. G. Cabral and A. Sampaio. Formal Specification Generation from Requirement Documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, Jan. 2008.
6. D. Carlson. *Eclipse Distilled*. Addison-Wesley, 2005.
7. A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
8. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
9. J. Drazan and V. Mencl. Improved Processing of Textual Use Cases: Deriving Behavior Specifications. In *Proc. 33rd Int. Conf. on Trends in Theory and Practice of Computer Science*, SOFSEM'07, pages 856–868, 2007.
10. N. E. Fuchs and R. Schwitter. Attempto: Controlled natural language for requirements specifications. In *Proc. 7th ILPS Workshop on Logic Programming Environments*, 1995.
11. N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). In *Proc. 1st Int. Workshop on Controlled Language Applications*, pages 124–136, 1996.
12. R. T. Giganto. A Three Level Algorithm for Generating Use Case Specifications. In *Proc. of Software Innovation and Engineering New Zealand Workshop*, SIENZ07, 2007.
13. M. Gordon and D. Harel. Show-and-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior. In *Proc. IADIS Interfaces and Human Computer Interaction*, IHCI'11, pages 360–364, 2011.
14. M. Gordon and D. Harel. Evaluating a Natural Language Interface for Behavioral Programming. In *Proc. of IEEE Symp. on Visual Languages and Human-Centric Computing*, VL/HCC'12, pages 17–20, 2012.
15. M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on Naturalness of Scenario-Based Programming. In *Proc. 17th Annual Conf. on Innovation and Technology in Computer Science Education*, ITiCSE'12, pages 198–203, 2012.
16. A. Haghighi and D. Klein. Simple Coreference Resolution with Rich Syntactic and Semantic Features. In *Proc. 2009 Conf. on Empirical Methods in Natural Language Processing*, EMNLP'09, pages 1152–1161, 2009.
17. D. Harel. On Visual Formalisms. *Commun. ACM*, 31(5):514–530, 1988.
18. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *Computer*, 34(1):53–60, 2001.
19. D. Harel. Playing with Verification, Planning and Aspects: Unusual Methods for Running Scenario-Based Programs. In *Proc. 18th Int. Conf. on Computer Aided Verification*, CAD'06, pages 3–4, 2006.
20. D. Harel. Can Programming be Liberated, Period? *Computer*, 41(1):28–37, 2008.
21. D. Harel and M. Gordon-Kiwkowitz. On Teaching Visual Formalisms. *IEEE Software*, 26:87–95, 2009.
22. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart Play-Out of Behavioral Requirements. In *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design*, FMCAD'02, pages 378–398, 2002.
23. D. Harel, H. Kugler, and A. Pnueli. Smart play-out extended: Time and forbidden elements. In *Proc. 4th Int. Conf. on Quality Software*, QSIC'04, pages 2–10, 2004.
24. D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

25. D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: Towards a Comprehensive Tool for Scenario Based Programming. In *Proc. 25th Int. Conf. on Automated Software Engineering*, ASE'10, pages 359–360, 2010.
26. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag, 2003.
27. D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: the Play-In/Play-Out Approach. *Software and System Modeling*, 2(2):82–107, 2003.
28. D. Harel and I. Segall. Planned and Traversable Play-Out: A Flexible Method for Executing Scenario-Based Programs. In *Proc. 13th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 485–499, 2007.
29. D. Harel and I. Segall. Visualizing Inter-Dependencies between Scenarios. In *Proc. 4th ACM symp. on Software visualization*, SoftVis'08, pages 145–153, 2008.
30. D. Harel and I. Segall. Synthesis from scenario-based specifications. *Journal of Computer and System Sciences*, 78(3):970–980, 2012.
31. D. C. Howe. RiTa: Creativity Support for Computational Literature. In *Proc. 7th ACM conf. on Creativity and Cognition*, pages 205–210, 2009.
32. ITU: International Telecommunication Union. Recommendation Z.120: Message Sequence Chart (MSC). Technical report, 1996.
33. D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, 2008.
34. M. Kay. Readings in Natural Language Processing. chapter Algorithm Schemata and Data Structures in Syntactic Processing, pages 35–70. 1986.
35. S.-K. Kim, T. Myers, M.-F. Wendland, and P. A. Lindsay. Execution of Natural Language Requirements using State Machines Synthesised from Behavior Trees. *Journal of Systems and Software*, 85(11):2652–2664, 2012.
36. H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating Structured English to Robot Controllers. *Advanced Robotics Special Issue on Selected Papers from IROS 2007*, 22(12):1343–1359, 2008.
37. V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707–710, 1966.
38. S. Maoz, D. Harel, and A. Kleinbort. A Compiler for Multimodal Scenarios: Transforming LSCs into AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 20(4):18, 2011.
39. L. Mich. NL-OOPS: From Natural Language to Object Oriented Requirements Using the Natural Language Processing System LOLITA. *Natural Language Engineering*, 2(2):161–187, 1996.
40. Microsoft. Microsoft speech api 5.1, http://www.microsoft.com/speech/download/old/sapi5.asp.
41. G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Introduction to WordNet: An On-line Lexical Database. http://wordnet.princeton.edu/, 1993.
42. M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.
43. L. M. Segundo, R. R. Herrera, and K. Y. P. Herrera. UML Sequence Diagram Generator System from Use Case Description Using Natural Language. *Electronics, Robotics and Automotive Mechanics Conference*, 0:360–363, 2007.
44. M. Takahashi, S. Takahashi, and Y. Fujita. A Proposal of Adequate and Efficient Designing of UML Documents for Beginners. In *Proc. Knowledge-Based Intelligent Information and Engineering Systems*, KES'07, pages 1331–1338, 2007.
45. UML. Unified Modeling Language Superstructure, v2.1.1. Technical Report formal/2007-02-03, Object Management Group, 2007.

46. P. W. with NL examples. www.playgo.co. website, 2012.

7 Discussion

7.1 Specifying behavior in natural language

By its nature, the LSC language is closer to the way one would specify dynamic requirements in a natural language. This thesis suggests to take advantage of this similarity, translating natural language requirements directly into LSCs, thus rendering them fully executable. Accordingly, our translation into LSCs can be viewed as a method for executing natural language requirements for reactive systems, or, to put it more succinctly, it enables programming in natural language.

7.1.1 Requirements engineering

NL has been used previously in computer aided software engineering (CASE) tools. In [33, 11], NLP supports the design process, and is used to create class diagrams, or use cases, reducing the errors in the design. Our NL requirements generate a class and object model, namely the system model, however the NL method additionally allows executing the requirements. It can also be used to assist in the development of the model objects and classes. Our NL algorithm helps create the connections between the existing objects and new objects when adding new requirements, hence helping in the design [16].

7.1.2 Related executable langauges

There are additional methods with similar motivation to ours, which translate CNL into more formal executable languages. In [4], use case templates are translated into process algebra, which is executable. Two-level-grammar (TLG) [2, 3] translates object-oriented (OO) specifications written in a natural language style into OO code. Our method uses behavioral programming ideas to integrate the requirements during execution, requiring less design

effort from the writer.

Attempto [9, 10] is a textual language for writing functional requirement specifications, based on first order logic and a rich English syntax. It uses declarative sentences, translates into Prolog and is used for reasoning on the web and for querying databases.

NL specifications have been used in specific domains, e.g., for robot controllers [27].

In general, we may say that these other efforts lack some of the main benefits of BP, the ability to interweave separate requirements and the ability to specify not only liveness properties but also forbidden events and safety requirements.

7.1.3 Preference learning

Our NL play-in can benefit from extending the source language according to user preferences. The system could track repetitive errors that a user makes and accept them as his/her personal way of writing. For example, in our experiments we noticed that many non-native English speakers forget to add the *"the"* determinant before object references. Such repeated errors could be learned and integrated into a more convenient user-specific grammar.

7.1.4 Extending the grammar

The LSC grammar can be extended to include behavior "shortcuts" for certain systematic behaviors, for example using the word *toggles* for changing between properties. Such shortcuts are valuable when writing the requirements for large reactive systems in specific domains. Additionally, the grammar can be improved by adding existing methods for reference resolution and spell-checking. Better use can be made of the dictionary, to connect synonyms and allow different references for the same object or operation. Another possibility is to extend the semantics of objects and automatically

create a GUI with meaningful common objects; e.g., sliders, buttons, text boxes, since they may be specified in the text.

7.2 Show & tell

We have described show & tell, an interface that interweaves describing behavior textually with demonstrating behavior on a GUI.

7.2.1 Related work

The basic play-in method is similar in some ways to programming by demonstration (PBD) [6]. These methods record demonstrations and use them for repeating tasks, many times generalizing the demonstration. A similar combination of voice and gestures (as opposed to writing and showing) has been used for managing graphical spaces with "put-that-there" [1]. In that work, the user uses voice commands to request adding, moving and manipulating objects on a screen, but can also use the voice pronoun reference of `this, that, here, there`, and point a hand to show the referenced objects or locations. Show & tell integrates text and GUI manipulation to assist in the creation of system requirements, in a domain generic application, and can also benefit from references to objects and locations.

7.2.2 Extending the interfaces

In show & tell, the demonstration is performed by pointing at objects or demonstrating operations on a GUI. However, the ideas can be extended with tracking hands or gestures using devices like touch screens or Microsoft's Kinect. This can make the *show* part connect to real systems, locations and gestures.

7.2.3 Speech recognition

Although the current implementation of show & tell requires showing and writing, it can easily be extended by using speech recognition to actually "tell". This requires using speech engines with low error rate, which are becoming feasible in recent years and will probably become more abundant in the future. We have tested an implementation of NL-play-in using speech with the Microsoft Speech API (SAPI 5.1) engine [30] and obtained good results with native English speakers. Since our evaluation found that using the hands for both typing and showing lessens the user experience, we also believe show & tell may benefit from seamless transition between speech and gestures, as is the case when talking and showing.

7.2.4 Show & tell as a general interface

The idea of show & tell has been described here in the context of programming. However, it can also be viewed as a new type of human-computer interface (HCI). Combining spoken or written language with actions or gestures, controlled by a context-free grammar, can be generalized to many kinds of interactions; for example: gaming, robot controllers, avionics and more.

7.3 Evaluation

The user evaluation experiment we describe in [15] is a preliminary one. Additional evaluation (partly in progress) includes case studies of building larger systems using NL-play-in, analyzing the learning requirements in order to work with NL-play-in, and documenting the limitations of the interface.

To further liberate programming and grant non-programmers the ability to program, future work could add shortcuts and simplifications to NL-play-in. This includes integrating concepts and ideas from simple and widespread programming languages meant for a larger community. Consider *Scratch*,

[32] for example, which includes sharing, tinkering, and a web learning platform. Scratch makes programming easy: users avoid syntax issues since they program by connecting existing blocks of code while learning mathematical and computational ideas. Remarkably, the Scratch environment allows combining different stories into a program [17], similar to the underlying idea of LSCs and, more generally, behavioral programming. Scratch exposed a large community of children to the world of programming and we have hopes that our work will help expose another community of writers to the programming world.

7.4 Semantic navigation of LSC

The ideas of semantic navigation of LSCs can be extended to other forms of visual diagrams, such as class diagrams, activity diagrams, etc. While modeling tools are developing, and systems are becoming more complex, tools and methods are necessary to navigate and comprehend the connections between these elements. Semantic navigation is one of several methods that can help this process. As large volumes of data become part of software engineering, in code and models artifacts, methods like semantic navigation are necessary and should be further developed.

Furthermore, semantic navigation for LSCs, is part of a growing set of tools for comprehension of behavioral programs [8, 24, 28] . A new emerging paradigm requires tools that will help different users work with the language, understand it or modify it, and a lot has yet to be done. From our latest case studies, it appears that for complex systems, better understanding of system states is required from the writer. The states, which are hidden in the BP approach, may need to be exposed in some cases. This may be resolved by providing different perspectives and views for BP, and specifically for LSCs. Additionally, a combination of different paradigms may be in order; for example, combining BP with statecharts [18] or other formalisms.

8 Work in Progress

8.1 Auto generation of interaction fragments

One limitation of the current show & tell algorithm is that the possible edges that an interaction can add are created manually, using knowledge of the grammar rules. This means that if the grammar is modified, the possible edges may need to be modified, and furthermore, the interaction may not provide all the relevant possibilities found in the grammar. Also, to generalize the ability to connect interaction input with a grammar, we extend show & tell by generating automatically from the grammar the possible edges an interaction can create.

The idea is a *template-generation* algorithm that statically analyzes the grammar and generates possible template sentences with some of the dynamic terminals used as *interaction placeholders* . These IntPs will be filled at run-time by the user interactions, and templates that have no IntPs can be fed to the previously described show & tell algorithm for further processing. Such a generalization may also be useful in a different setting. For example, a smart phone with a grammar for command and control. If the phone uses a natural language interface based on a context free grammar, and is endowed with a way to input additional data through interaction, the template-generation can help combine the two.

The algorithm requires some definitions that include: *interaction input*, *interaction output*, and *interaction filters*. The set of interaction input is the set of terminal symbols generated by the interaction. The set of interaction output is the set of non-terminal symbols that are considered interesting as the output of an interaction. The latter also provides a halting condition for template generation. The algorithm does not generate the infinite set of sentences the language accepts, but rather small clauses determined by the output non-terminal set. To demonstrate: in the LSC grammar, the symbol `DET` for a determinant (like `a` or `the`), is not considered a complete

or interesting output by itself and will not be added to the output non-terminals. Therefore, the generation will not output the non-terminal `DET`. However, a `MESSAGE` is considered an interesting interaction in the domain of LSC, therefore, the generation will stop when the templates create a `MESSAGE`; it will not continue to generate more complex templates of which `MESSAGE` is part of.

To get a succinct and relevant set of templates, the search removes semantically identical rules. Moreover, the definitions include the set of interaction filters; symbols that filter out templates that are too detailed. For example, in the LSC grammar we support the following kinds of sentences: "the user *must* click the button", "the user *may* click the button", and "the user *cannot* click the button". All these, share a common symbol of `TEMPERATURE` that can get one of several terminals (must/may/cannot), with different semantic meanings. To filter out such sentences, which will burden the user, the `TEMPERATURE` symbol can be added to the interaction filters symbols.

This idea needs to be tested on several grammars to demonstrate how interaction can be combined naturally with different grammars. We would like our examples to include systems that work with a command-and-control grammar using a speech interface and that can be combined with a mouse our touchscreen interaction.

References

[1] R. A. Bolt. 'Put-That-There": Voice and Gesture at the Graphics Interface. *SIGGRAPH Comput. Graph.*, 14(3):262–270, July 1980.

[2] B. Bryant. Object-Oriented Natural Language Requirements Specification. In *Proc. 23rd Australian Computer Science Conference*, ACSC'00, 2000.

[3] B. R. Bryant and B.-S. Lee. Two-Level Grammar as an Object-Oriented Requirements Specification Language. In *Proc. 35th Annual Hawaii Int. Conf. on System Sciences*, HICSS'02, page 280, 2002.

[4] G. Cabral and A. Sampaio. Formal Specification Generation from Requirement Documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, Jan. 2008.

[5] D. Carlson. *Eclipse Distilled*. Eclipse series. Addison-Wesley, 2005.

[6] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[7] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In *Proc. 3rd Int. Conf. on Formal Methods for Open Object-Based Distributed Systems*, FMOODS, page 451, 1999.

[8] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On Visualization and Comprehension of Scenario-Based Programs. In *Proc. 19th IEEE Int. Conf. on Program Comprehension*, ICPC'11, pages 189–192, 2011.

[9] N. E. Fuchs and R. Schwitter. Attempto: Controlled natural language for requirements specifications. In *Proc. 7th Intl. Logic Programming Symp. Workshop Logic Programming Environments*, 1995.

[10] N. E. Fuchs and R. Schwitter. Attempto Controlled English (ACE). In *Proc. 1st Int. Workshop on Controlled Language Applications*, pages 124–136, 1996.

[11] R. T. Giganto. A Three Level Algorithm for Generating Use Case Specifications. In *Proc. of Software Innovation and Engineering New Zealand Workshop 2007*, SIENZ07, 2007.

[12] M. Gordon and D. Harel. Generating Executable Scenarios from Natural Language. In *Proc. of the 10th International Conference on Computational Linguistics and Intelligent Text Processing*, CICLing'09, pages 456–467. Springer-Verlag, 2009.

[13] M. Gordon and D. Harel. Semantic Navigation Strategies for Scenario-Based Programming. In *Proc. of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC'10, pages 219–226. IEEE Computer Society, 2010.

[14] M. Gordon and D. Harel. Show-and-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior. In *Proc. IADIS Interfaces and Human Computer Interaction*, IHCI'11, pages 360–364, 2011.

[15] M. Gordon and D. Harel. Evaluating a Natural Language Interface for Behavioral Programming. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, VLHCC'12, pages 167–170, 2012.

[16] M. Gordon and D. Harel. Programming in Natural Language. 2012. unpublished.

[17] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on Naturalness of Scenario-Based Programming. In *Proc. of the 17th Annual Conf. on Innovation and Technology in Computer Science Education*, ITICSE'12, 2012. To Appear.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

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

[20] D. Harel, S. Maoz, S. Szekely, and D. Barkan. PlayGo: Towards a Comprehensive Tool for Scenario Based Programming. In *25th Int. Conf. on Automated Software Engineering*, ASE'10, pages 359–360, 2010.

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

[22] D. Harel and R. Marelly. Specifying and Executing Behavioral Requirements: the Play-In/Play-Out Approach. *Software and System Modeling*, 2(2):82–107, 2003.

[23] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *Commun. ACM*, 55(7):90–100, 2012.

[24] D. Harel and I. Segall. Visualizing Inter-Dependencies between Scenarios. In *Proc. of the 4th ACM symp. on Software visualization*, SoftVis '08, pages 145–153. ACM, 2008.

[25] ITU: International Telecommunication Union. Recommendation Z.120: Message Sequence Chart (MSC). Technical report, 1996.

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

[27] H. Kress-Gazit, G. E. Fainekos, and G. J. Pappas. Translating Structured English to Robot Controllers. *Advanced Robotics Special Issue on Selected Papers from IROS 2007*, 22(12):1343–1359, 2008.

[28] S. Maoz and D. Harel. On Tracing Reactive Systems. *Software and Systems Modeling (SoSyM)*, 10(4):447–468, 2011.

[29] 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*, PLATEAU '10, pages 1–5, 2010.

[30] Microsoft. Microsoft speech api 5.1, http://www.microsoft.com/speech/download/old/sapi5.asp.

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

[32] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.

[33] M. Takahashi, S. Takahashi, and Y. Fujita. A Proposal of Adequate and Efficient Designing of UML Documents for Beginners. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 1331–1338, 2007.

[34] UML. Unified Modeling Language Superstructure, v2.1.1. Technical Report formal/2007-02-03, Object Management Group, 2007.

Statement About Independent Collaboration

I hereby state that the work presented in this thesis was done by me, and with my supervisor, who was obviously part of the research.

List of Core PhD Papers:

[GH09] M. Gordon and D. Harel. Generating Executable Scenarios from Natural Language. *In Proc. of the 10th International Conference on Computational Linguistics and Intelligent Text Processing* ,CICLing'09, pages 456-467, 2009.

[GH 10] M. Gordon and D. Harel. Semantic Navigation Strategies for Scenario-Based Programming. *In Proc. of the 2010 IEEE Symp. on Visual Languages and Human-Centric Computing,* VLHCC '10, pages 219-226, 2010.

[GH 11] M. Gordon and D. Harel. Show-and-Tell Play-In: Combining Natural Language with User Interaction for Specifying Behavior. *In Proc. IADIS Interfaces and Human Computer Interaction*, IHCI'11, pages 360-364, 2011.

[GH 12] M. Gordon and D. Harel. Evaluating a Natural Language Interface for Behavioral Programming. *In Proc. IEEE Symp. On Visual Languages and Human-Centric Computing,* VLHCC'12, pages 167-170, 2012.

List of Additional Papers:

[HG09] D. Harel and M. Gordon-Kiwkowitz. On Teaching Visual Formalisms. *IEEE Software*, 26:87-95, 2009.

[EGHMW11] N. Eitan, M. Gordon, D. Harel, A. Marron, and G. Weiss. On visualization and comprehension of scenario-based programs. *In Proc. 19th IEEE Int. Conf, on Program Comprehension*, ICPC '11, pages 189-192, 2011.

[GMM12] M. Gordon, A. Marron, and O. Meerbaum-Salant. Spaghetti for the Main Course? Observations on Naturalness of Scenario-Based Programming. *In Proc. of the 17th Annual Conf. on Innovation and Technology in Computer Science Education,* ITICSE'12, *2012*.

[AAGH] G. Alexandron, M. Armony, M. Gordon, and D. Harel. The Effect of Previous Programming Experience on the Learning of Scenario-Based Programming. *In Proc. 12th Koli Int. Conf. on Computing Education Research*, 2012.

תמצית

תזה זו מתארת ממשקים חכמים לתכנות מבוסס-תרחישים, ובמיוחד לשפת ה- live sequence charts, או בקיצור LSC. הנושא העיקרי מתאר את ממשק ה NL-play-in, הכנסת דרישות בשפה טבעית, שיטה שמאפשרת יצירה של LSCs ע"י כתיבה של דרישות בשפה טבעית מובנית. השיטה ממירה טקסט לפורמליזם החזותי של LSCs ע"י שימוש בדקדוק חסר-הקשר ואינטראקציה עם הכותב, המשמשת להבהרת רב-משמעות בכוונותיו. נושא שני הנכלל בתזה הוא show & tell, *תאר והדגם*, שילוב של הכנסת דרישות בשפה טבעית עם אינטראקציה של המשתמש עם ממשק משתמש גרפי (GUI) של המערכת המתוארת. זו הרחבה של שיטת ה-play-in.

ב play-in, GUI של המערכת המתוכנתת משמש לביצוע בפועל של תכנות התנהגותי. תאר והדגם ממזג בצורה טבעית ואינטואיטיבית את היכולת הנ"ל עם היכולת לתאר התנהגות בטקסט בשפה טבעית. הוא מפענח את האינטראקציה של הכותב בהקשר של הדרישות הכתובות.

לבסוף, אנו מציגים גם הערכות ראשוניות של הממשקים, ושיטות נוספות שפותחו לתכנות מבוסס-תרחישים, ולהרחבתה, הנקראת תכנות התנהגותי. שיטות אלא כוללות רעיונות לניווט והבנה של תוכנות מבוססות-תסריטים.

ככלל, עוסקת תזה זו ביצירת דרכים טבעיות יותר לתכנת באמצעות תכנות מבוסס-תרחישים, במטרה להפוך תכנות בעולם האמתי נגיש לציבור רחב יותר.