Steps Towards Scenario-Based Programming with a Natural Language Interface

Michal Gordon and David Harel

Weizmann Institute of Science, Rehovot Israel

Abstract. Programming, i.e., the act of creating a runnable artifact applicable to multiple inputs/tasks, is an art that requires substantial knowledge of programming languages and development techniques. As the use of software is becoming far more prevalent in all aspects of life, programming has changed and the need to program has become relevant to a much broader community. In the interest of broadening the pool of potential programmers, we believe that a natural language interface to an intuitive programming language may have a major role to play. In this paper, we discuss recent work on carrying out scenario-based programming directly in a controlled natural language, and sketch possible future directions.

1 Introduction

Imagine a future home, with a slew of smart home devices installed, such that the various parts of the controlling software can communicate. Now imagine the owners have a new idea about the desired behavior of the system, something that none of the vendors had considered, and which is therefore not a new *configuration.* They would like the shades to be lowered, whenever the TV is turned on and the light outside is too bright. However, if the kids are home, they do not want to dim the living room. Will they have to contact the professional designers of their smart home, or the vendors? We hope not. In fact, we would like to believe that in the future more people will be able to *program*, or enhance existing programs by adding requirements, on their own. In fact, throughout the paper, when we refer to the "programmer", we mean the person (or team of people) who creates a program, not necessarily a professional programmer in the usual sense of the word, perhaps more like a system's engineer.

The art of programming is already available to a broad community, with the open architecture of mobile phone applications, languages and toolkits that let children program games, robots and more [27,37], and methods that allow programming by demonstration and end-user programming [7]. Here we discuss an approach, whereby, the computer is able to derive an executable program directly from the human's language, or something very close thereto.

Computerized understanding of natural language is an extremely complicated and broad problem, and has been studied widely (see, e.g., [41,38,7]). It is a central facet of intelligence, and its difficulty is thus closely related to tackling the Turing test. We do not attempt to solve this problem, nor is it the focus of our discussion. Rather, we concentrate on *programming in controlled natural language* (PiCNL). Subsets of natural language have been used to allow intuitive yet formal interfaces for various tasks. They are often called *simplified* or *controlled* language [1,7]. We review these in Section 2.1. Our main question here is whether we can use controlled language to 'explain' to the computer what we want our system to do under all possible circumstances (very much like the way we would have explained it to a person), and cause it to generate a fully executable program.

The reader may claim that we can already talk to computers: For example, we can ask our smart-phone, to call a particular friend, schedule dinner and make a two-person reservation in our favorite restaurant. These activities, however, are not programming. Such natural language interactions are often called *command and control* [38,12], and can be carried out using voice or natural text interfaces.

In Section 2 we discuss the quest for programming in natural language, controlled natural language and various interfaces that we distinguish from programming. In Section 3 we review several efforts at programming in natural language. Finally, in Section 4 we discuss in some detail a relatively new scenario-based programming paradigm, called *behavioral programming* [22], and show how we use natural language to program reactive systems, and how it can be enriched with GUI-based play-in [21].

2 The Quest for Programming in Natural Language

The concept of programming in controlled natural language, i.e., PiCNL, has been suggested in the past, however it has not become prevalent as a programming method. In this section we discuss controlled natural language, we define programming in natural language, and distinguish PiCNL from various natural language interfaces. We review how PiCNL has developed over the years, including the obstacles encountered and proposed solutions.

The simplest form of using natural language with a computer is *dictation.* The text can be entered to the computer by typing or by speech, and all the computer does is the parsing into words and sentences. There is no requirement that the text be analyzed or understood in any way. Thus, if typing is used, this task is trivial. If the text is spoken we get into the realm of speech recognition, which we will not discuss here. For our purposes in this paper, entering the text can be done either way, depending on the preference of the user.

2.1 Using Controlled Natural Language

Controlled natural language (CNL) is created by restricting the grammar of a natural language to reduce ambiguity and complexity. It can be made more formal than unrestricted NL, well-structured and better amenable to semantics, since due to its restrictions, relevant constructs can be semantically annotated. CNLs enjoy some of the advantages of natural language; they are easy to use and understand, have a quick learning curve, and are close to the application domain. They also avoid much of the ambiguity and vagueness of natural language, and thus also enjoy some of the advantages of formal languages. However, their use is mostly in the technical arena. You probably wouldn't want to try to write "controlled poetry"....

Controlled languages can be learned, and then easily written, with the aid of appropriate editors and parsers. Due to their relative simplicity, they support automation of tasks, and are used in automatic translation, formal document writing, reasoning, robot-controllers, specifications, and ontologies [1,11,29].

Simplified English, as controlled English was originally called, was developed for the aerospace industry's maintenance manuals. It was also referred to as *plain language* or in its trademarked version as simplified technical English (STE) [1]. The idea was to restrict the lengths of sentences and paragraphs, avoid slang and jargon, use active voice, etc. The STE dictionary includes approved words that can be used only according to their specific meaning. For example, the verb *close* can be used in the sense of "close a door" but not as the adjective, "stand close to the landing gear". An alternative word is often suggested — here it was *near*, as in "stand near the landing gear".

2.2 Programming vs. Command & Control

A more advanced form of using natural language with a computer involves the variety of operations that employ familiar natural language for controlling or manipulating systems. These activities fall under the general term of *command & control*, or *natural language interactions*; see, e.g., [38].

Natural language is used for search, for personal assistance, and in general as an interface for commanding other applications [25,38]. A user can say to her smart-phone "call Martin Jones". The word "call" will be recognized semantically, the contacts will be searched for the person in question, and if successful, the command will be performed, i.e., the call will be placed. Command & control systems can retrieve answers that may be given in natural language, or sometimes in more appropriate forms. For example, a user can ask a mobile phone "how far am I from the Fairview Green restaurant?", and the answer may be given as "1.7 miles". Smarter applications can take into account the moving speed of the mobile phone and provide a more relevant answer, like "a fifteen minute walk" or "a three minute drive".

Natural language has been used as a conversational user interface to describe and manipulate visualizations of data [39]. Thus, in a specific setting the user can ask (taken verbatim from [39]): "What is the correlation between the depth and water's temperature", and receive a graph plotting the two. He/she can then add a command "please color by pH", and obtain a color code layered atop the graph. This kind of interface is more advanced and permits complex connections between various commands.

Although the interfaces we described so far are quite elaborate, none of them is considered programming. In *programming*, the directions/recipes are to be applicable in the future to many inputs, most often to infinitely many of them.

Thus, one may say that a central characteristic of a program (as opposed to a command & control system) is its being *reusable* when different inputs arrive from the environment.

The difference is similar to that between giving one-time directions for someone to do something, and teaching that person a skill to use when applicable. For example, showing a child how to wash some specific dishes is different from explaining when and how you decide whether to do the dishes, and how to do so in general. In the latter case, the child will know how to handle a sink full of dishes, an empty sink, or a sink with a single dish. Untreated cases, such as a blocked sink, would require additional directions, or an extension of the "program". Similarly, command & control is telling the car radio to turn on and to tune to channel Z-100. A program is when I tell the car radio that *whenever* I turn on the radio and switch to Z-100, if the channel is airing conversation, rather than music for more than one full minute, switch to another channel.

Although programming has to do with multiple runs of the same program, this is not the same as creating recurrent behavior. Anyone can easily set a recurrent event in an electronic calendar, like a family dinner every Saturday night. Indeed, advance interfaces, such as Google Calendar's *Quick Add* feature, permit setting such recurrent scheduling in natural language. Entering "Family dinner every Saturday" will set the recurring event, with a weekly reminder. However, this is not programming, as it does not depend on varying inputs.

In programming there are multiple different inputs to the same general set of commands and instructions, and the program can deal with all of them, even those that have not been considered explicitly by the programmer. Programming is not something that can be achieved merely by using different menus, or by configuring the system. An example is "whenever it rains, and my calendar shows a soccer game the same day, cancel the event and notify all participants". It is not only the addition of the notification action that makes this programming, but the constant monitoring of rain as an external event. This is a lot more than a command. It is a program rule; a kind of program snippet. And this raises the harder question of how to deal with multiple program snippets, which constitute a full program. For example, how to manage several of these program-rules that interact, and perhaps even contradict one another? We discuss this further in Section 4.

2.3 Programming in Natural Language

Natural language has been used in software engineering for tasks that are highly related to programming. For example, some database queries can be specified by natural language and may be considered a type of programming [42]. Viewing and displaying data can be done with natural language, e.g., with the Articulate system [39]. Natural language has also been used in computer aided software engineering (CASE) tools, to help the process of modeling or creating software engineering artifacts. For example, in [13], natural language is used to help create better use cases for system development, and other papers produce other UML documents, etc.

Methods, that use statistical NLP, combine learning techniques with NLP to analyze natural language and automatically create partial code. For example, transforming English specifications of input file format (with additional information from sample input files) to automatically generate C++ code for input parsers [30], or analyzing API documents to infer API library specifications [43].

As to full programming in natural language, Dijkstra claimed in 1978 [10]:

> "In order to make machines significantly easier to use, it has been proposed (to try) to design machines that we could instruct in our native tongues. This would, admittedly, make the machines much more complicated, but, it was argued, by letting the machine carry a larger share of the burden, life would become easier for us. It sounds sensible provided you blame the obligation to use a formal symbolism as the source of your difficulties. But is the argument valid? I doubt. [...] Instead of regarding the obligation to use formal symbols as a burden, we should regard the convenience of using them as a privilege: thanks to them, schoolchildren can learn to do what in earlier days only genius could achieve."

In our opinion, programming with *controlled* natural language, should be also viewed as a privilege. Those willing to express their system requirements in a CNL and disambiguate their input to the computer when it is not clear enough, can gain the benefits of instantly executable programs.

A 2004 study by Liu and Lieberman [31] discussed how natural language descriptions could be used to make human-machine communications more natural. Those authors state that "several developments might now make programming in natural language feasible".

Sloppy programming [7] initially used unstructured text, yet later found that the unstructured approach caused too many false interpretations. Sloppy programming uses a specific grammar, based on an existing set of scripts, to allow the user to enter something simple and natural. The essence of sloppy programming is to interpret and make sense of the 'sloppy' input. Controlled natural language is similar, in that it is simple and natural enough, yet it restricts the grammar, and requires the user to learn the restrictions from examples and by feedback.

Chickenfoot and CoScripter [7] allow users to write web-customization scripts using simplified Java script commands. They use a domain specific vocabulary to allow performing various programming tasks in the web domain. End-user programming and scenario-based programming are similar in that they aim to create a more natural means of authoring behavioral fragments.

In [34], the authors show how some of the more subtle aspects of procedural programming — steps and loops — can be handled effectively, and express their believe that advances in natural language processing can contribute to the task of natural language programming, for descriptive and procedural programming paradigms.

Indeed, one large community that can benefit from PiCNL are children, too young to acquire formal education of programming. Several advances in natural interfaces to programming for children have been made, most notably with languages like Scratch [37], which enable children to formally describe their system requirements via visual blocks that help them overcome syntax problems. Scratch is available in multiple languages, and allows children the feel of programming naturally. Although, not precisely PiCNL, Scratch allows formal programming with blocks containing natural English text.

In [10], Dijkstra also remarks that

> "there is a sharp decline in people's mastery of their own language [...], and many people are no longer able to use their own native tongues effectively."

He says that this "New Illiteracy"

> "should discourage those believers in natural language programming that lack the technical insight needed to predict its failure."

Despite Dijkstra's gloomy statements, we feel that the time is ripe for major efforts to program in natural language.[**] Dijkstra's pessimism can be overcome by a careful choice of the limited CNL to be used, and the appropriate programming paradigm into which it will be translated. PiCNL will be suitable only for those willing to master CNL, disambiguate problems, and understand how faults may occur. In Section 4, we shall discuss how such a paradigm can reduce some of the technicalities that make natural language programming difficult.

3 Approaches to Programming in Natural Language

Several research efforts have led to languages that can create executable code from a CNL. Each one defines its own CNL style and translates the CNL into a different notation, each with its own merits and areas of applicability. We describe some of these, focusing not on methods that support computing, but rather on those that create executable artifacts.

In [6], use-case templates, written a CNL, are translated into *process algebra* (in the CSP notation). This method was implemented in a Microsoft Word^{TM} plug-in that checks adherence of use-case specifications to a CNL grammar and translates them into process algebra. It then allows carrying out system property verification. This technique supports user-view use-cases, which can be used to specify user operation and expected system responses, and component-view use-cases, with one component that invokes an action and another that provides the service. After automatic translation to the CSP notation, a model checker is used to check refinement between user and component views.

The CNL in [6] is used to write imperative sentences, which describe actor actions, and affirmative sentences, which describe system characteristics. Requirements are written in tabular form, in numbered steps. For the user-view use-case, each step includes a user action, a system state, and the system's response. The CNL reflects the selected domain. Besides automatically generating

[**] The second-listed author's work on visual languages has given him an earlier reason to believe that Dijkstra's pessimism need not always be taken too seriously.

formal models, the use of the CNL in [6] prevents the introduction of ambiguous formatted sentences in the use-case specification, thus helping to increase document quality.

Attempto controlled english (ACE) [11] is an example of a CNL that was designed to serve as a knowledge representation language, and its output is fully executable. ACE accepts a sequence of anaphorically interrelated sentences. This means that references to objects mentioned in previous sentences are acceptable, creating a coherent text of linked sentences. These can include coordination, subordination, quantification and negation. One can describe something that is the case — a fact, an event, a state. The interpretation of the sentences is deterministic, and a paraphrase reflects the interpretation to the programmer.

The lexicon can be modified by the programmer for domain specific content. Questions can be written in CNL and are translated into Prolog queries, which are then answered by logical inference. The knowledge can be executed for simulation or prototyping. Execution involves adding statements that would start the simulation, e.g., "customer1 is a customer", "card1 is a card", etc.

ACE is used in a variety of applications: as an NL interface in database query languages and robot controllers, in planning medical reports, for the semantic web (translation to and from web-languages) for protein ontologies, and as a reasoner that performs deductions [29].

Two-level-grammar (TLG) [5], is an object-oriented requirements 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, allowing services and functions to be referred to and called upon. This formalism allows one to describe object-oriented behavior naturally, and each function definition is composed of logical rules executed in the order they are given.

TLG is natural-language-like in style, but is sufficiently formal to be automatically translated into object-oriented formal specifications.

In *spoken Java* [3], programmers can describe their Java program orally in natural language. 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.

The efforts in references [5,11], are general. However, domain specific applications also exist; e.g., for robot controllers. In [28] *linear temporal logic mission planning toolkit* (LTLMoP) is used for writing specifications in structured English. The language is used to specify safety and liveness properties. The implementation is through a grammar that translates into LTL.

The MOOIDE system [7], based on the Metaphor system tests the idea of describing behavior with stories in the domain of a virtual reality storytelling game. The game itself is a reactive system. The interface in MOOIDE takes the form of a dialog in natural language about a growing set of terms that are added to the world, and it uses common sense semantics. The MOOIDE system, although domain specific, has many elements similar to the scenario-based programming approach that we describe in the next section.

4 Behavioral Programming in Controlled Natural Language

Behavioral programming (BP) is a recently proposed programming paradigm in which system behavior is described in scenarios, similar to the way people naturally specify behavior [22]. This naturalness appears to be a crucial component of the quest for *liberating programming* [19]. We have developed a natural language input interface for BP [14], in which scenarios are described with CNL and are transformed automatically into a BP formalism called *live sequence charts* (LSC) [9]. These, in turn can be executed, using play-out [20], planning algorithms or synthesis [32].

We focus on two of the main concepts underlying behavioral programming, namely, inter-object programming and unification.

In the *inter-object* approach a behavior is usually described as a "story" that considers the operations that occur between objects, rather than focusing on the operations within objects, as is the case in the *intra-object* style of object-oriented programming. Although in both cases each object has unique operations and properties, in intra-object behavior the programming process focuses on the objects, whereas in inter-object programming, the focus is on the interaction *between* the objects. Shifting the focus to the between-objects behavior, allows for a far more natural and "liberated" style of programming. See [19,21].

Here are examples of inter-object specifications: "If the alarm of a watch is set, then whenever the current time reaches the alarm time, the beeper turns on". "Whenever the beeper is on, it beeps every two seconds".

These scenarios may be easy to describe and follow, but they cannot be executed together as a single system, unless the idea of *unification* is introduced. Unification means that events of the same type between the same objects, represent the same event. Since in the specifier's mind the operation of sending a text message (which is also an event) is the same in both scenarios, in order to execute what the programmer meant, these two events should be unified.

4.1 Live sequence charts

Live sequence charts, constitute a visual formalism for specifying multi-modal scenarios. An LSC can assert mandatory behavior (termed "hot"), possible behavior (termed "cold"), as well as forbidden behaviors and their combinations. The LSC language [9,21] extends message sequence charts (MSC) [26] (termed sequence diagrams in UML [40]). In an LSC, objects are represented by vertical lines, called lifelines, and messages between objects are represented by horizontal arrows between objects. Time advances along the vertical axis and messages entail an obvious partial ordering.

A cold monitored event (dashed blue arrows) is monitored, and if it occurs the next event in the partial order should be monitored or executed. A hot executed event (solid red arrows) means that the system should perform the event eventually. The LSC language also includes conditions, assertions, loops, switch cases, time, symbolic instances, and several additional constructs. Figure 1 shows a typical LSC.

A set of LSCs can be executed using the *play-out* mechanism [21], which monitors at all times what must be done, what may be done and what is forbidden, and proceeds accordingly. This results in a full execution of the LSC specification using a naïve strategy, considering the current state and progressing by choosing arbitrarily from all possible next events to be triggered.

Since different fragmented scenarios are combined into a single functional executable system, there is a risk of contradictory requirements that can produce violations during execution.

Contradictions can subtly, arise from multiple scenarios. Finding an execution order that makes it possible to execute without violations requires considering future states when choosing an event. Techniques that use model-checking, planning and synthesis, have been developed, to look ahead and choose an execution order in a smarter fashion [18,32]. Synthesis can often be used to verify that the specification is valid or to exhibit inconsistencies [33].

4.2 Natural language play-in

In [14], we describe a natural language interface to the LSC formalism, named *NL-play-in*. The programmer can write in a controlled English, using terms, e.g., nouns, verbs, adjectives, that are relevant to the system being described, and reusing them in further requirements to allow unification during execution. The terms used become part of a growing *system model* that includes the system's objects, and their methods and properties.

The interface consists of a context-free grammar (CFG) bottom-up parser and a dialog system that help the programmer create both a system model and a set of LSC scenarios. The resulting system is fully executable. The controlled natural language accepts declarative requirement sentences. The parser includes semantic information for creating the LSCs, adding loops and conditions, and specifying which events should be monitored and which should be executed. The grammar is general: it analyzes all terms with the help of the WordNet dictionary [35], in order to determine whether a word is a noun, a verb, or an adjective and whether it is meant as an object, a method or a persistent property.

When a sentence is analyzed, terms that are not completely understood by the system are disambiguated using a quick-fix interface to the programmer. The word in question is marked with a squiggly line, and hovering over it with the mouse provides the programmer with additional information, and a list of possible solutions. Disambiguation includes resolving grammar problems and semantic issues.

Grammar problems include incomplete sentences, sentences without a verb, or sentences that are not part of the grammar. Semantic problems include phrases that can be either a target object or a parameter for a method. Semantic problems are more prevalent at early stages. As the requirements accumulate, and the programmer resolves problems, the information becomes part of the model and is used to resolve further ambiguities automatically.

The process of developing a system and its requirements is intermixed. Sometimes the programmer knows what the system should do, and only then considers what the system model will be, while often it is the other way around. The process continues throughout development, adding requirements and extending the model. Our method supports both development directions: creating the model as it becomes necessary when adding requirements, or adding the requirements for an existing model.

In one development direction, when a requirement is parsed, non-existing model parts, e.g., objects, classes, methods, and properties, are verified with the programmer as necessary, and the model is augmented with new model parts. Any addition of model parts is explicit, to verify that new parts are introduced only when they cannot be unified with existing parts. We call this process *model disambiguation*. Only after the model is complete, a new LSC is created that captures the requirement. Viewing the LSC allows the programmer to verify that the requirement was parsed correctly. Finally, the system created can be executed at any stage with the existing model and the LSCs.

In the other development direction, when a system model exists, it is possible for the programmer to specify requirements, and any references to the model parts are immediately understood, and require no additional user interaction. Many times the model is actually a non-behaving graphical user interface (GUI) of the final system. In this case, the model will be created automatically from the GUI objects.

When objects and methods are created automatically, they can be later replaced by graphical entities, or augmented with low-level code. For example, a button object may have a method *click* that is referenced in a scenario. The same *click* can later be implemented, to show and accept clicks from the user.

For a thorough guide we refer the reader to [14], in which an example of a wristwatch is described (also available in `http://www.weizmann.ac.il/mediawiki/playgo/index.php/Wristwatch_Example`).

The following CNL demonstrates the style of programming with NL-play-in:

> 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 state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may not change.
> When the user clicks any button, the beeper turns to off.

The fully executable diagrams that result automatically from these NL requirements are shown in Figures 1, 3 and 2.

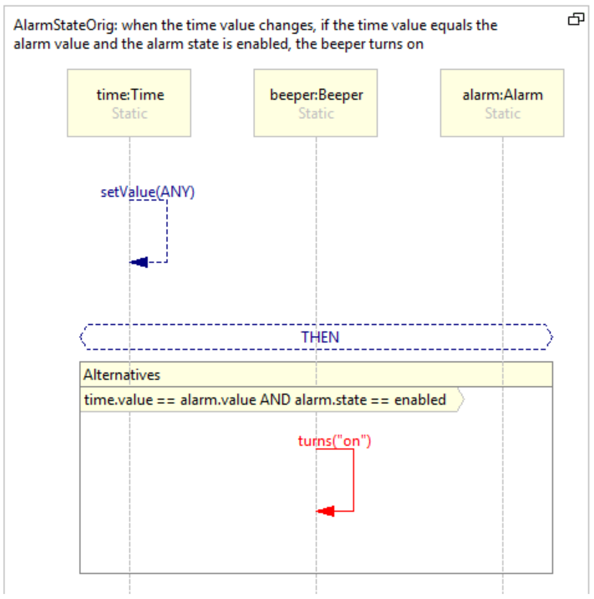
AlarmStateOrig: when the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on
time:Time
Static
beeper:Beeper
Static
alarm:Alarm
Static
setValue(ANY)
THEN
Alternatives
time.value == alarm.value AND alarm.state == enabled
turns("on")

Fig. 1. A simple LSC created for the sentence "when the time value changes, if the time value equals the alarm value and the alarm state is enabled, the beeper turns on".

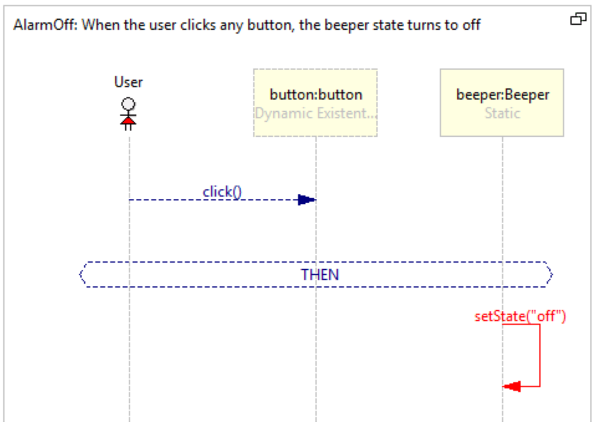

Fig. 2. The LSC created for the requirement "When the user clicks any button, the beeper turns to off."

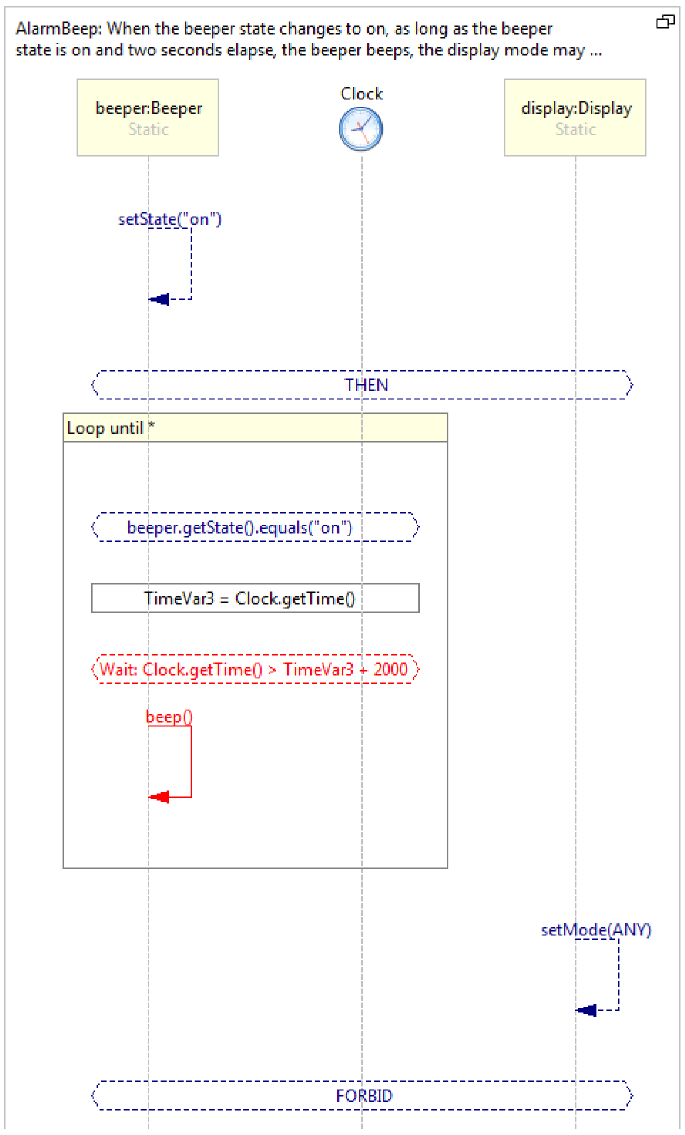
AlarmBeep: When the beeper state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may ...
beeper:Beeper
Static
Clock
display:Display
Static
setState("on")
THEN
Loop until *
beeper.getState().equals("on")
TimeVar3 = Clock.getTime()
Wait: Clock.getTime() > TimeVar3 + 2000
beep()
setMode(ANY)
FORBID

Fig. 3. The LSC created for the requirement "When the beeper state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may not change."

It is possible to extend the model disambiguation to suggest connections between different terms according to word similarity or synonyms. For example, if “opening the radio”, and “unlocking the radio”, both appear, the parser can suggest to the programmer to make the connection between the methods, causing unification between these terms during execution.

4.3 Show & tell

In [15] we describe an extension of the NL-play-in interface for LSCs, which combines it with the *play-in* method [17], by interweaving CNL and user interaction. In play-in, the programmer specifies scenarios by playing-them-in directly from a graphical user interface (GUI) of the system being developed, similar to programming by demonstration [8]. *Show & tell* means that the programmer can combine writing in natural language with actual showing. Some parts of a scenario, for example, the `when-then`, or the `if`, are easier to write than to show, while other parts, like the `click` of a button, are easier to show. Show & tell also helps avoid typos and the necessity to specify the names or the operations with their exact terms.

While play-in is similar to programming by demonstration [8], show & tell is similar to the *put-that-there* method [4], and other multi-modal user interfaces, see `http://www.wisdom.weizmann.ac.il/~michalk/Projects/SaT/` for a demo. Experiments we have carried out [16] show that when the interface is a mouse and keyboard, show & tell combinations may not be more convenient for people who type quickly. Our experiments also expose the learnability of the NL-play-in approach.

4.4 Limitations and Future Work

NL-play-in meant for the high-level *programming* of reactive systems — dynamic systems that respond to events, depending on their current state [23]. The method, as part of the BP paradigm, supports incremental development of systems by continuously adding requirements. NL-play-in can help bridge the gap between the requirement engineering process and the development of the final system, and allow a shorter life cycle. NL-play-in is suitable for interweaving fragmented requirements, including negative requirements, and can be combined with other programming styles, e.g., statecharts [2]. The systems we have already created include a wristwatch, a chess game, a baby monitor, and parts of an ATM machine.

The programmer’s identity can range from the professional to the end-user, and the vocabulary and level-of-detail can change according to the programmer’s needs. When programming the behavior of a robot, pertaining to where it heads, what it sees, or what directions it receives, the terms will be different from the case of programming at the level of the robot moving body parts.

For modifying existing systems, e.g., augmenting existing behavior with additional requirements or forbidden behaviors, the programmer should be familiar with the details of the model. However, even if the model is less known, show & tell can help the programmer refer directly to relevant objects and terms.

The natural language interface can be improved substantially, with, e.g., reference resolutions, verbal shortcuts and the use of synonyms, all of which can make the writing more friendly, as is the case with ACE [11] or MOOIDE [7].

Another challenge in using natural language for programming is its assimilation. For non-programmers this requires developing teaching methods. The BPJ library [22] lets expert programmers use BP concepts in their own programming environment, supporting a gradual transition from procedural programming to BP and later to natural language programming.

When broader groups of people will program, software engineering activities will probably broaden too, and will require better visualization and navigation methods; some research on these approaches in the content of LSCs has already started [24], and this work can be adopted to the NL interface too.

Another consideration is requirement coverage. The one responsibility of the programmer is to enter the requirements. However, since requirements need to cover multiple possibilities and many system states, he may require help in considering all possibilities. Such support could include supplying many views that will help him understand the system. For example, complex systems may benefit from requirements analysis by other formats than natural language, e.g. tabular visualization, that will help the programmer see the bigger picture and uncover gaps in the specification. Precise documentation in software engineering [36] becomes extremely relevant when programming in natural language because in a way the documentation becomes the final program.

It is not only the understanding of how to program, but also the need to program that is still elusive. What will new programmers want to program? Prior to the introduction of smartphones, few people thought of creating their own applications. However, at present there is an astonishing variety of applications, and their number is growing rapidly. We hypothesize that as technology comes to play a much larger role in people's lives, the need to program or re-program such systems will increase dramatically. This, in fact, constitutes a major part of the motivation for PiCNL.

5 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. AECMA Official Site. http://www.simplifiedenglish-aecma.org/Simplified_English.htm.
2. D. Barak, D. Harel, and R. Marelly. InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming. *IEEE Trans. Soft. Eng.*, 32(7):467–485, 2006.

3. A. Begel and S. Graham. Spoken programs. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, (VL/HCC'05), pages 99–106, 2005.
4. R. A. Bolt. "Put-that-there": Voice and Gesture at the Graphics Interface. *SIGGRAPH Comput. Graph.*, 14(3):262–270, 1980.
5. 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), pages 280–289, 2002.
6. G. Cabral and A. Sampaio. Formal Specification Generation from Requirement Documents. *Electron. Notes Theor. Comput. Sci.*, 195:171–188, Jan. 2008.
7. A. Cypher, M. Dontcheva, T. Lau, and J. Nichols. *No Code Required: Giving Users Tools to Transform the Web.* Morgan Kaufmann Publishers Inc., 2010.
8. 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.
9. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
10. E. W. Dijkstra. On the Foolishness of "Natural Language Programming". In *Program Construction, Int. Summer School, Marktoberdorf, Germany*, volume 69 of *Lecture Notes in Computer Science*, pages 51–53. Springer, 1978.
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. T. Geller. Talking to Machines. *Commun. ACM*, 55(4):14–16, 2012.
13. R. T. Giganto. A Three-Level Algorithm for Generating Use Case Specifications. In *Proc. Software Innovation and Engineering New Zealand Workshop*, (SIENZ'07), 2007.
14. M. Gordon and D. Harel. Generating executable scenarios from natural language. In *Proc. 10th Int. Conf. on Computational Linguistics and Intelligent Text Processing*, (CICLing'09), pages 456–467. Springer-Verlag, 2009.
15. 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.
16. M. Gordon and D. Harel. Evaluating a Natural Language Interface for Behavioral Programming. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, (VL/HCC'12), pages 17–20, 2012.
17. D. Harel. From Play-In Scenarios To Code: An Achievable Dream. *Computer*, 34(1):53–60, 2001.
18. 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.
19. D. Harel. Can Programming be Liberated, Period? *Computer*, 41(1):28–37, 2008.
20. 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.
21. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine.* Springer-Verlag, 2003. (See also paper in *Software and System Modeling*, 2(2);82-107, 2003).
22. D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, 2012.
23. D. Harel and A. Pnueli. Logics and Models of Concurrent Systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, 1985.

24. D. Harel and I. Segall. Visualizing Inter-Dependencies between Scenarios. In *Proc. 4th ACM symp. on Software visualization*, (SoftVis'08), pages 145–153, 2008.
25. M. A. Hearst. "Natural" Search User Interfaces. *Commun. ACM*, 54(11):60–67, 2011.
26. ITU: International Telecommunication Union. Recommendation Z.120: Message Sequence Chart (MSC). Technical report, 1996.
27. S.-H. Kim and J. W. Jeon. Programming LEGO Mindstorms NXT with Visual Programming. In *Proc. Int. Conf. on Control, Automation and Systems*, (ICCAS'07), pages 2468–2472, 2007.
28. 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.
29. T. Kuhn and N. E. Fuchs, editors. *Proc. 3rd Int. Workshop on Controlled Natural Language (CNL)*, volume 7427 of *Lecture Notes in Computer Science*. Springer, 2012.
30. T. Lei, F. Long, R. Barzilay, and M. Rinard. From Natural Language Specifications to Program Input Parsers. In *Proc. Annual Meeting Assoc. for Computational Linguistics*, (ACL'13), 2013.
31. H. Liu and H. Lieberman. Toward a Programmatic Semantics of Natural Language. In *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing*, (VL/HCC'04), pages 281–282, 2004.
32. 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.
33. S. Maoz and Y. Sa'ar. Two-way Traceability and Conflict Debugging for AspectLTL Programs. In *Proc. 11th Int. Conf. on Aspect-oriented Software Development*, (AOSD'12), pages 35–46, 2012.
34. R. Mihalcea, H. Liu, and H. Lieberman. NLP (Natural Language Processing) for NLP (Natural Language Programming). In *Proc. 7th Int. Conf. Computational Linguistics and Intelligent Text Processing*, (CICLing'06), pages 319–330, 2006.
35. 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.
36. D. L. Parnas. Precise Documentation: The key to Better Software. In *The Future of Software Engineering*, pages 125–148. Springer, 2011.
37. M. Resnick et al. Scratch: Programming for All. *Comm. of the ACM*, 52(11):60–67, 2009.
38. B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman, 1986.
39. Y. Sun, J. Leigh, A. Johnson, and S. Lee. Articulate: a Semi-automated Model for Translating Natural Language Queries into Meaningful Visualizations. In *Proc. 10th Int. Conf. on Smart Graphics*, (SG'10), pages 184–195, 2010.
40. UML. Unified Modeling Language Superstructure, v2.1.1. Technical Report formal/2007-02-03, Object Management Group, 2007.
41. T. Winograd. Understanding Natural Language. *Cognitive Psychology*, 3(1):1 – 191, 1972.
42. Y. W. Wong and R. J. Mooney. Learning Synchronous Grammars for Semantic Parsing with Lambda Calculus. In *Proc. 45th Annual Meeting of the Assoc. for Computational Linguistics*, (ACL'07), 2007.
43. H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring Resource Specifications from Natural Language API Documentation. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering*, (ASE'09), pages 307–318, 2009.