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.

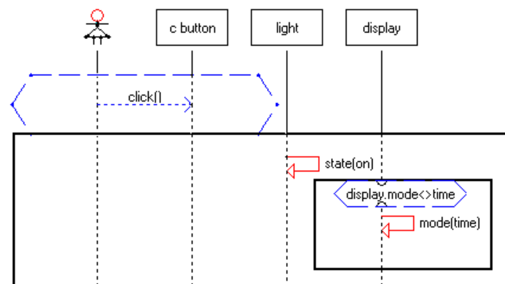

Fig. 1. A simple LSC. The prechart (the blue dashed hexagon) contains the cold event (blue dash arrow) "user clicks the `c` button", while the main chart (the black solid rectangle) shows two hot events (red solid arrow): one shows the light state changing to `on` and the other is a hot event with a 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.

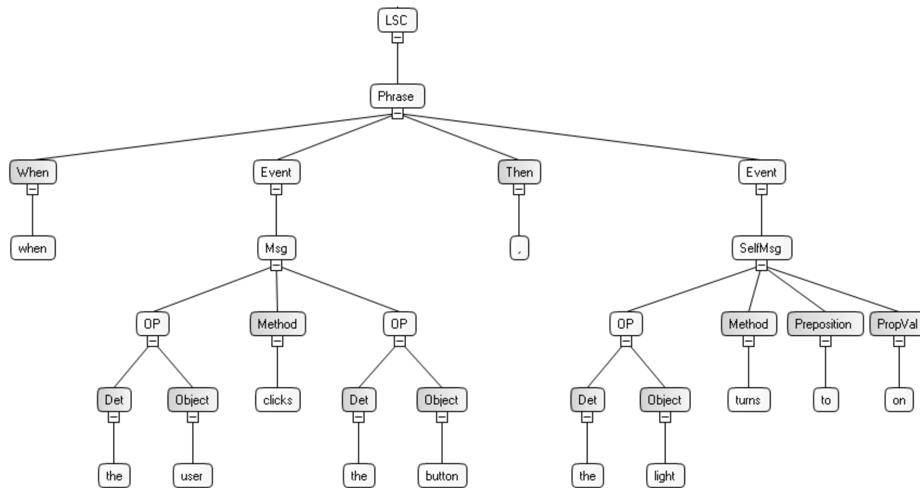

Fig. 2. The parse tree for the sentence "when the user clicks the button, the light turns on". The parts of the LSC grammar detected are shown. There is one message *Msg* which is a message from object phrase (*OP*) `user` to object phrase `button`, and another self message *SelfMsg* of object `light`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

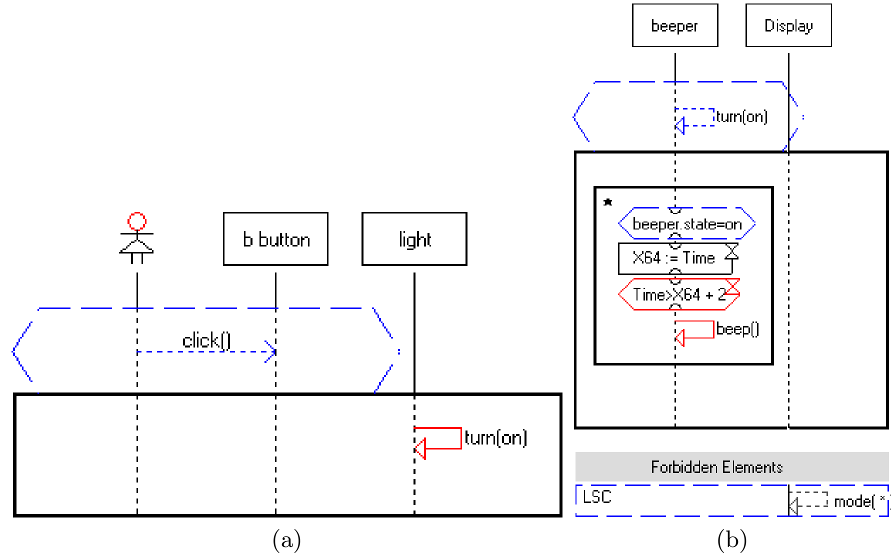

Fig. 3. Sample LSCs. (a) A simple LSC created for the sentence: "when the user clicks the `b` button, the light turns on". (b) A more complex LSC created for the sentence: "when the beeper turns on, as long as the beeper state is on, if two seconds have elapsed, the beeper beeps and the display mode 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.

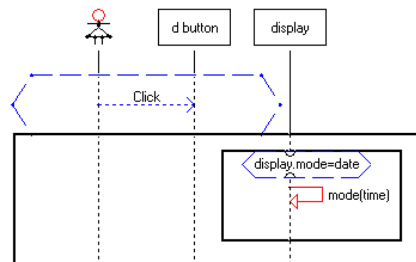

Fig. 4. The LSC created for the sentence "when the user presses the `d` button, if the display mode is date, the display mode changes to time". The message in the *when* part is cold (dashed blue arrow), while the messages in the *then* part are hot (solid red arrows).

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

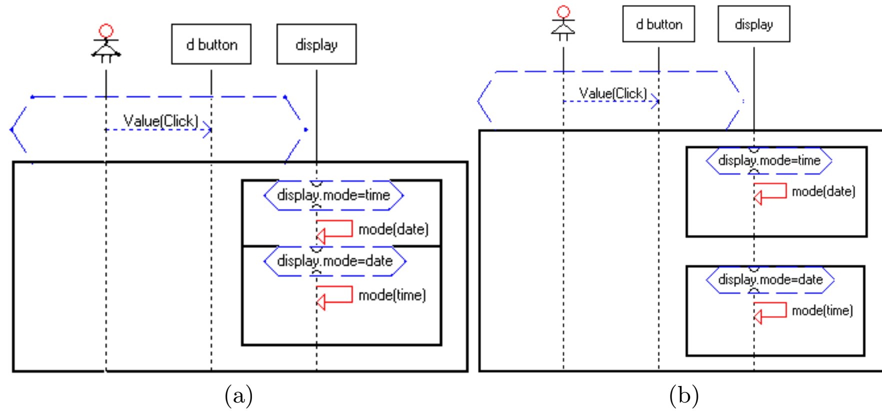

Fig. 5. Conditions in LSCs. (a) The LSC created for the sentence "when the user presses the `d` button, if the display mode is time, the display mode changes to date, otherwise if the display mode is date, the display mode changes to time". (b) Shows what would happen if the *otherwise* would be replaced 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

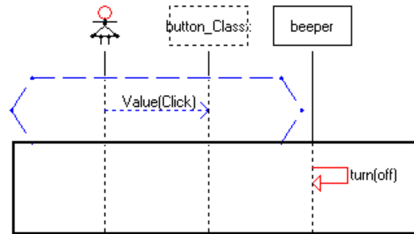

Fig. 6. The LSC created for the sentence "when the user presses any button, the beeper shall turn to off". The `button` object referred to by the user is a non-specific object and is therefore translated as a symbolic object of the button class, shown using a dashed box.

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)