Bringing Textual Descriptions to Life: Semantic Parsing of Requirements Documents into Executable Scenarios

Ilia Pogrebezky
IDC Herzliya

Smadar Szekely
Weizmann Institute

Reut Tsarfaty
The Open University

David Harel
Weizmann Institute

Abstract

We present an end-to-end framework for translating natural language (NL) requirements into executable systems. Specifically, we implement semantic parsers for two sorts of input: (i) requirements documents written in a controlled natural language (CNL), and (ii) requirements documents written in NL. For each requirements document, we output the *system model* (SM) architecture along with a set of *live sequence charts* (LSCs) that capture the dynamic behavior of the specified system. Our parsers are embedded in *PlayGo*, a development environment for specifying (*playing-in*) requirements as LSCs and executing (*playing-out*) the resulting behavior. Thus, our output systems may be depicted visually, executed interactively, or provided as a standalone Java executable. PlayGo further allows for post-editing the predicted output via a friendly GUI, thus enabling the rapid development of parallel text:code data for *statistical natural language programming*.

1 Introduction

Ever since the early days of computer science, researchers have been fascinated by the idea of automatically turning natural language (NL) requirements into executable programs. This challenge, termed *natural language programming*, lies at the intersection of two CS disciplines: *software engineering* (SE) on the one hand and *natural language processing* (NLP) on the other hand.

In SE, programming relies on formal languages that enjoy strict interpretation rules. NL requirements involve far more complex utterances, which suffer from ambiguities at all levels of structure — syntax, senses, co-references — making the mapping between meaning and form a complex matter.

Early attempts to approach NL programming in SE resorted to the development of formal specification languages that resemble English. These languages are called *control natural languages* (CNL), which are in fact subsets of English with strict grammars and limited expressivity. The main drawback of CNLs is that it is not trivial nor intuitive for humans to describe system in it. When expanding CNLs to be more intuitive, expressive and natural, painful ambiguities arise.

In recent years, NLP researchers working on *semantic parsing* successfully addressed partial aspects of the NL programming challenge, by offering algorithms that assign a single formal interpretation to an input (potentially ambiguous) NL utterance. Relevant studies include the translation of NL commands into database queries (Wong and Mooney, 2007; Zettlemoyer and Collins, 2009; Poon and Domingos, 2009; Liang et al., 2011) mapping NL instructions into Windows command sequences (Branavan et al., 2009; Branavan et al., 2010), creating input parsers based on NL definitions (Barzilay et al., 2013), and translating NL descriptions into regular expressions (Kushman and Barzilay, 2013). Most of these studies focus on domain-specific languages, and perform semantic interpretation on a sentence-by-sentence basis. In all cases, the presented output does not go all the way towards an executable system.

Here we present an end-to-end system for automatically translating a requirements document consisting of a sequence of NL requirements into a complete, executable, code-base that is represented by a *system model* architecture and a set of *live sequence charts* (LSCs) (Damm and Harel, 2001; Harel and Marelly, 2003) — formal, unambiguous, multimodal charts that capture the dynamic system behavior. Our system is embedded in PlayGo (Harel et al., 2010), an interactive development environment that allows for *playing-in* (specifying) requirements by specifying *live se-*

quence charts and *playing out* (executing) the behavior captured by the LSCs. Through PlayGo, the predicted system can be visualized graphically, executed interactively, or delivered as a standalone Java implementation.

Our parser for CNL requirements documents relies on the joint sentence-discourse decoder of Tsarfaty et al. (2014). The parser for NL requirements extends the joint dependency parser and semantic role labelers of Roth et al. (2014) with further linguistic reasoning that sorts out binding (should the expression refer to an existing entity or create a new one?), verbal aspect (should the action bring about to a change of state?) modalities (is the action mandatory or optional?) and the linear ordering of events depicted in the text.

We showcase natural language programming capacity by parsing the requirements data provided by Roth et al. (2014). These data contain system descriptions written by humans, and we parse them into executable systems. Our framework further supports *post-editing* of output interpretations via a visual GUI that can alter the LSC depiction of the predicted system model and LSCs. This post-editing capacity may be critical for the rapid development of gold standards for statistical text-to-code translation.

Our contribution is hence manifold. We present a semantic representation that supports an effective alignment and translation of texts into code. We present a system for automatically parsing requirements documents — provided in a CNL form or in reasonably rich English – into a complete, executable, scenario-based system. We further provide a tool for rapidly creating LSCs, annotating requirements, and enhancing the predicted output, in support of the development of sophisticated statistical parsers for natural language programming.

2 Formal Preliminaries

2.1 The Semantic Representation

The first contribution of this paper is our proposal that the LSC formal language (Damm and Harel, 2001), previously proposed for manual scenario-based programming, makes for a viable semantic representation for automatic (and statistical) text-to-code translation. We introduce this formal representation via two concepts: a *system model*, representing the static (architectural) aspects of system design and *live sequence charts*, representing the dynamic (behavioral) aspects of system flow.

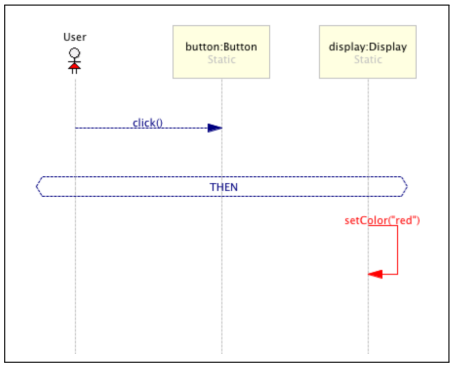

Figure 1: The LSC graphical depiction of the scenario: "When the user clicks the button, the display color must change to red."

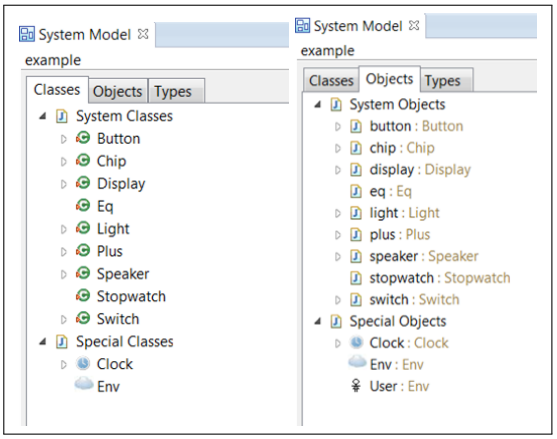

Figure 2: A System Model Representing the System Architecture: *Classes* and *Objects* Views.

A **System Model** (SM) represents the static architecture of a proposed system. It consists of classes (types), objects (instances), methods, and properties, along with their default and actual values. Figure 2 provides a screenshot of the *Classes* and *Objects* views of a specified system. These views also allow to expand a class or an object down to its properties and methods. Every system to be developed using our framework is assumed to also have special classes and objects of one of the following types: **Env** that represents the environment, **Clock** that accounts for time operations, and **User** that simulates an interactive user.

A **Live Sequence Chart** (LSC) is a formal diagram that describes the possible, forbidden or necessary interactions between the entities (classes and objects) that make up the system. Entities in LSC diagrams are represented as vertical lines called *lifelines*, and interactions between entities are represented by horizontal arrows between lifelines (or a self-pointing arrow) called *messages*.

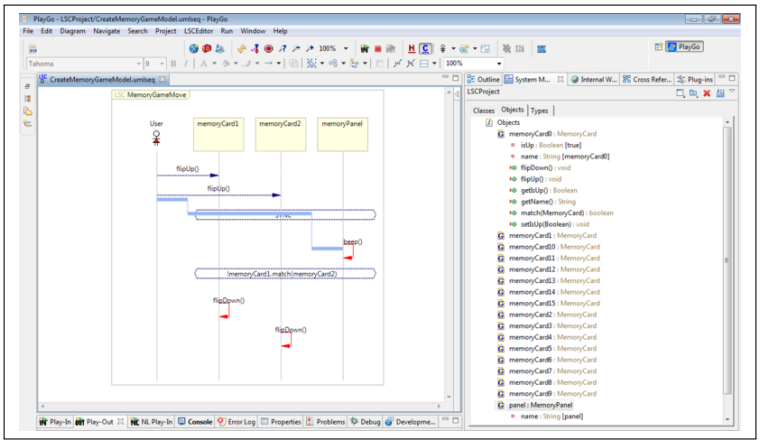

Figure 3: An LSC and a System Model (SM) during play-out. Note the blue traces manifesting the system behavior on the LSC in real time, and the real-time properties of the Objects in the SM.

Messages connect the sender and the receiver, where the head of the arrow points to the receiver. Time in LSCs proceeds from top to bottom on each lifeline, imposing a partial order on the execution of all messages. An LSC message can be "hot" (obligatory), represented by a red color, or "cold" (optional), colored in blue. Every message has an execution status: solid arrows represent methods to be executed, and dash arrows represent methods to be monitored. The LSC specification language also contains control structures such as if conditions, switches and bounded loops. The negation of flows can designate forbidden scenarios. To illustrate, Figure 1 shows the LSC for a scenario that defines a simple dynamic aspect involving two objects from the architecture presented in Figure 2.

2.2 The PlayGo Tool

From a practical point of view, our algorithms for translating textual requirements into the aforementioned semantic representation are integrated into and executed through *PlayGo*, a comprehensive tool for behavioral, scenario-based, programming (Harel et al., 2010). The PlayGo environment supports two basic processes: *play-in* and *play-out* (Harel and Marelly, 2003).

Playing-in a requirement means demonstrating a desired behavioral scenario of the desired system. PlayGo standardly supports manual play-in via (i) editing of the LSC diagrams by dragging and dropping the relevant LSC elements, (ii) interactive play-in, where a user demonstrates the desired behavior by "playing it in" on a GUI, or (iii) a CNL interface, based on subset of English, for specifying scenarios, prompting for manual disambiguation when ambiguities arise.

The complement process of playing in requirements is **playing** them **out**. Play-out refers to the execution of the system behavior described by the LSCs as a scenario-based program. Every execution of an operation is considered a step. Following a user action, the system executes a superstep — a sequence of the steps that follows the user action — which terminates either at a stable situation or when the entire execution terminates (Harel and Marelly, 2003). The execution can be visually depicted as a set of traces, which can be observed to verify the desired behavior. In Figure 3 we show a PlayGo screenshot during play-out, where execution traces are marked in blue on the LSC, and the system model shows real-time objects.

2.3 NL Play-In

Our departure point in this paper is the work of Gordon and Harel (2009), which defines a CNL and a respective grammar for turning utterances into LSCs. That CNL *play-in* interface of PlayGo requires users to manually enter CNL requirements, on a sentence-by-sentence basis, prompting for manual interpretations whenever an ambiguity arises. In this work we lift two fundamental restrictions from this mode of play-in. First, we discard the focus on *sentences* and provide interpretations of complete requirements *documents*, where local disambiguation decisions take global document context into account. Furthermore, we make a significant step towards lifting the *controlled* restriction, and parse NL requirements via a similar *play-in* interface.

3 Programming in a Controlled Natural Language (CNL)

To overcome the need for manual, tedious, sentence-by-sentence disambiguation efforts, we implemented a statistical parser that can accept a complete requirements document, written in the ambiguous CNL of Gordon and Harel (2009), as input and return the disambiguated SM and corresponding LSCs as output. Based on this parser, our newly added operation *import description* now allows PlayGo users to play in requirements by automatically uploading a whole sequence of requirements describing a single system at once.

Our statistical parser is based on the joint sentence-discourse model of Tsarfaty et al. (2014), that takes the form of a Hidden Markov Model (HMM). In this model, emission probabilities reflect the grammaticality of individual requirements and transition probabilities model the process of creating a global system model (SM) out of local SM snapshots associated with the individual requirements. Using Viterbi decoding, the parser searches for the best sequence of SM snaphots that has most likely generated the document.

In order to learn the emission probabilities, we assume a generative probabilistic grammar that assigns probability mass to all (and only) sentences generated by the CNL context-free grammar (CFG) of Gordon and Harel (2009). In order to train the CFG model parameters, for which hardly any real-world parallel text-to-code data exist, we generated over 10000 sentences by sampling rules from the CNL CFG and generating valid parse trees (no syntactic priors are assumed).

With about 10100 annotated sentences (10000 automatically generated and about five case studies provided by Tsarfaty et al. (2014)) we induced a PCFG that can parse CNL utterances to trees that have a direct translation into an LSC on the one hand, and into a system model snapshot on the other hand. We use simple maximum likelihood estimates, smooth lexical probabilities via basic unknown-words distribution, and smooth syntactic distributions via interpolating the seed parameters with parameters learned from the sampled distributions.

Our experiments with the framework contribute two important insights. First, interpolating the small seed of annotated real-world requirements we have at our disposal with a large set of sampled trees boosts accuracy results significantly. Secondly, context consistently matters. As we increase the number of alternative analyses that our model can assign to an individual requirements, context-based modeling successfully alters the LSC disambiguation choice and our accuracy results improve further.[1]

4 Programming in Rich Natural Language

Having lifted the *manual, sentence-by-sentence* restriction on play in, we are ready to tackle a greater question: can we lift the restriction to use a *controlled* fragment of NL? That is, can we automatically interpret requirements documents written in reasonably rich and natural English? Here, we implemented a pipeline parsing system that can accept a requirements document written in unrestricted English as input and provide its SM-and-LSCs interpretation as output.

Our departure point is the joint dependency parser and semantic-roles labeler of Roth and Klein (2015). The representation this parser delivers .is an extension of the CoNLL 2009 dependency format which also identifies for each parse tree the actors, objects , actions and properties[2] mentioned therein.[3]

We designed a rule-based algorithm that constructs, for each requirement, a set of feature structures that captures the lifelines (objects), actions (methods) and themes (method arguments) which are required for executing the scenario. For each of these feature-structures we extract attributed:value pairs that add additional dimensions of semantic interpretation, as we specify shortly. Based on these feature-structures and attribute:value pairs, our algorithm constructs an LSC that represents the dynamic flow of the requirement, and, as side effect, takes the additional static information that is discovered via this requirement to expand the global system model for the entire document.

The attribute:value pairs we extract include the semantic value (i.e., the reference) of each feature structure, the type of binding of each reference (should we use an existing instance or create a new one?) the role of each argument (should the

[1] A technical reference omitted here for anonymity reasons.

[2] where 'properties' is a cover term for various sorts of verbal and nominal modification.

[3] The ontology of semantic roles that are provided by this parser is described in details in (Roth et al., 2014).

```
Lifeline(
        type: actor
        value: < user - 2 , NN >
        binding: < a - 1 , DT >
)
Lifeline(
        type: object
        value: < account - 10 , NN >
        modifier: < user - 9 , NN >
        binding: < a - 8 , DT >
)
Action(
        value: < create - 7 , VB >
        HAS_ACTOR: < user - 2 , NN >
        ACTS_ON: < account - 10 , NN >
        modal: < must - 3 , MD >
        modifier_by : < provide - 12 , VBG >
)
Action(
        value: < provide - 12 , VBG >
        HAS_ACTOR: < user - 2 , NN >
        ACTS_ON: < username - 14 , NN >
        ACTS_ON: < password - 17 , NN >
        modal: N/A
)
Theme(
        value: < password - 17 , NN >
        binding: < a - 16 , DT >
)
Theme(
        value: < username - 14 , NN >
        binding: < a - 13 , DT >
)
```

Figure 4: A feature structure for the NL requirement: "A user must be able to create a user account by providing a username and a password."

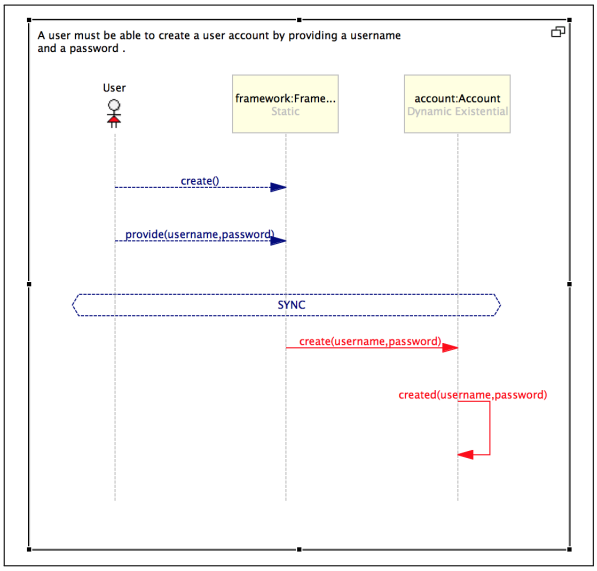

Figure 5: An LSC scenario: "A user must be able to create a user account by providing a username and a password."

argument be a property or a theme?), the modality of the different actions (whether they can, may, or must happen) and the implied linear ordering of actions (by default this order coincides with the order of the verbs, but this is no necessarily so — see Figure 5). We further use a co-reference component to determine the particular values of the property "ownership". For example: in the requirement "A user must be able to login to his account by providing his username and password." , "his" points to "user", and so we can determine the value of the property "owner". The designated feature structure and the corresponding LSC for the first requirement in Roth et al. (2014) are provided in Figures 4 and 5.

As a critical step towards unrestricted NL programming, we now focus on gathering data for training and testing different statistical models for the task. The data set of Roth et al. (2014) contains 320 syntactically parsed and SR-labeled requirements. In order to exploit context within coherent requirements documents, we divided this data into disjoint specified systems and created 25 distinct documents with an average of 12 requirements per system. We used our algorithm to create a parallel corpus of NL requirements aligned with LSC/SM representations. This corpus is intended facilitate the development of statistical models for translating unrestricted NL requirements directly into LSC/SM representations (and hence to executable code).

5 Related Work

Natural language programming research intersects two disciplines in computer science, *software engineering* (SE) and *NL processing* (NLP).

In earlier SE days, Abbott (1983) extracted data types, objects, variables and operators from informal English texts based on a rule-based algorithm. Later, Booch (1986) extended Abbott's approach to object-oriented terms. Saeki et al. (1989) automatically extracted nouns and verbs for identifying model entities, and observed that in order to attain diagrams of reasonable quality, human intervention is required. Mich (1996) later employed a full NLP pipeline that contains a semantic analysis module, dispensing with additional post-processing. More recenty Harmain and Gaizauskas (2003) and Kof (2004) relied on a combination of NLP tools and human interaction.

Recently, NL programming has also been attended to by NLP researchers. Roth et al. (2014) proposed to utilize annotated data to learn semantic parsers for requirements automatically, assigning ontology-based semantic roles to them. Others aimed to go beyond assigning a linguistic representation. Examples include analyzing API documents to infer API library specifications (Zhong et al., 2009), and generating parser programs from input format descriptions (Barzilay et al., 2013).

Here we automatically assigns to each requirement document a dynamic semantic representation that is based on the LSC visual programming lan-

guage. In contrast with the aforementioned studies, our system exploits and benefits from discourse context, rather than interpreting one sentence at a time. Moreover, our output goes beyond a formal linguistic representation and returns an executable system as output, which can be visualized, played out, or post-edited via the PlayGo interface.

6 Conclusion

We present an end-to-end system for NL programming extending the PlayGo architecture with advanced statistical semantic parsing algorithms. We showcased the NL programming capacity for two types of input: CNL requirements, and reasonably rich, uncontrolled, NL requirements. In both cases, the output is a system model accompanied with all relevant LSCs. In the future we intend to develop more sophisticated models for statistical natural language programming. We conjecture that the representation, tool, and data we provide herein will prove immensely valuable for future successful implementations.

References

R. Abbott. 1983. Program design by informal english descriptions. In *Commun. ACM 26(11):882-894 (1983)*.

R. Barzilay, M. Rinard, T. Lei, and F. Long. 2013. From natural language specifications to program input parsers. *Proceedings of ACL*, pages 1294–1303.

G. Booch. 1986. Object oriented development. In *IEEE Transactions on Software Engineering, (2):211–221.*

S.R.K. Branavan, H. Harr Chen, L. Zettlemoyer, and R. Barzilay. 2009. Reinforcement learning for mapping instructions to actions. *Proceedings of ACL-AFNLP*, 1:82–90.

S.R.K. Branavan, L. Zettlemoyer, and R. Barzilay. 2010. Reading between the lines: learning to map high-level instructions to commands. *Proceeding ACL Pages 1268-1277*.

W. Damm and D. Harel. 2001. Lscs: Breathing life into message sequence charts. *Methods Syst. Des., 19(1):4580.*

M Gordon and D. Harel. 2009. Generating executable scenarios from natural language. *Proceedings of CICLing 2009: 456-467*.

D. Harel and R. Marelly. 2003. *Come, Lets Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer.

D. Harel, S. Maoz, S. Szekely, and D. Barkan. 2010. Playgo: towards a comprehensive tool for scenario based programming. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 359–360.

H. M. Harmain and R. Gaizauskas. 2003. CM-Builder: a natural language-based CASE tool. *Journal of Automated Software Engineering*, 10:157–181.

L. Kof. 2004. Natural language processing for requirements engineering: Applicability to large re- quirements documents. In *Conference on Automated Software Engineering*.

N. Kushman and R. Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of NAACL-HLT*, pages 826–836.

P. Liang, M. Jordan, and D. Klein. 2011. Learning dependency-based compositional semantics. *Proceeding of HLT*, pages 590–599.

L Mich. 1996. Nl-oops: From natural language to object oriented requirements using the natural language processing system lolita. In *Natural Language Engineering , 2(2):161–187*.

H. Poon and P. Domingos. 2009. Unsupervised semantic parsing. *Proceeding EMNLP*, pages 1–10.

M. Roth and E. Klein. 2015. Parsing software requirements with an ontology-based semantic role labeler. In *Proceedings of the IWCS Workshop Language and Ontologies 2015*. To appear.

M. Roth, T. Diamantopoulos, E. Klein, and A. Symeonidis. 2014. Software requirements: A new domain for semantic parsers. *Proceedings of the ACL 2014 Workshop on Semantic Parsing (SP14)*.

M. Saeki, H. Horai, and H Enomoto. 1989. Software development process from natural language specification. *Proceeding ICSE '89 Pages 64-73*.

R. Tsarfaty, I. Pogrebezky, G. Weiss, Y. Natan, S. Szekely, and D. Harel. 2014. Semantic parsing using content and context: A case study from requirements elicitation. In *Proceedings of EMNLP*, pages 1296–1307.

Y.W. Wong and R. Mooney. 2007. Learning synchronous grammars for semantic parsing with lambda calculus. In *Proceedings of ACL*.

L. Zettlemoyer and M. Collins. 2009. Learning context-dependent mappings from sentences to logical form. *Proceeding ACL*, pages 976–984.

H. Zhong, L. Zhang, T. Xie, and H. Mei. 2009. Inferring resource specifications from natural language api documentation. In *Proceedings of ASE*, pages 307–318.