The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science
M.Sc. program - Research Track

From Natural Language descriptions to executable scenarios

by
Ilia Pogrebezky

M.Sc. dissertation, submitted in partial fulfillment of the requirements
for the M.Sc. degree, research track, School of Computer Science
The Interdisciplinary Center, Herzliya

January 2017

This work was carried out under the supervision of Dr. Reut Tsarfaty from the Open University of Israel and Prof. Shimon Shocken from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.

Abstract

Every software engineering process starts with an idea. Most of these ideas are described and formalized by specification documents written by humans by means of Natural Language. Then, software engineers translate those specifications into executable code.

In previous studies, researchers aimed to automate this translation task. Initially, Abbott, 1983 focused on extracting object models from sentences (a.k.a. Classes, Methods and Properties). Later on, Mich, 1996 defined a semi-English language, a.k.a. a Controlled Natural Language (CNL), for specification documents. CNLs enjoy an easy translation to a formal representation (like other programming languages) but they suffer of non-intuitiveness and they are unnatural to humans.

In this research, we aim to automatically translate requirements documents into executable code using Natural Language processing and machine learning techniques. Our main goal is to go beyond an object model extraction, by capturing the instantiation of objects and the interactions between them. Furthermore, we aim to handle requirements in true Natural Language while not setting any assumptions on the input.

Our method translates requirements documents into executable code via an intermediate formal representation. This representation is a set of Live Sequence Charts (LSCs) — formal, unambiguous multi-modal charts that capture the dynamic system behavior — and a system model (SM) that captures the structure of the system (the static part). This representation has a direct translation into Java (Harel et al., 2010).

Gordon and Harel, 2009, defined a CNL and a semi-automatic parser, that with user help can map a single CNL sentence into an LSC representation. In this thesis, we fully automate the CNL-to-LSC translation process, i.e., remove the need of an interactive user to solve ambiguities. Moreover, we also allow the user to enter a complete requirements document and use document context to help the disambiguation of individual requirements.

Using a small seed of annotated data, we developed a statistical parser that successfully parses requirements in CNL with 95% F-Score on a small benchmark of CNL scenarios created by human experts (for more details see chapter 4 table 4.11), alleviating the need of human disambiguation and speeding up the translation process significantly. We showed empirically that context matters, i.e., parsing a requirements document as a whole, instead of each sentence in isolation, improved parsing results.

As a subsequent step, we aim to extend our input domain to unrestricted Natural Language. Roth et al., 2014 treat the mapping from Natural Language requirements to formal representations as a semantic parsing task and contribute an annotated data set which contains a set of requirements documents from CS students homework assignments annotated with semantic role labels such – *Actor, Owner, etc*. Each document describes a simple application such as: taxi reservation, a library ordering system, and so on.

We developed a rule-based algorithm that integrates statistical tools including Part-of-speech taggers, a semantic role labeler, phrase and dependency parsers in order to translate those specification documents into executable code. Our algorithm exploits the context of the entire document

and translates the requirements into SM and LSCs. We further developed an evaluation method for object/data modeling based on tree edit distance, and showed that the agreement score between our system's output and an expert software engineer is almost the same as the agreement score between two human experts, when designing a model based on the same set of requirements.

Acknowledgements

I would first like to thank my thesis advisor Dr. Reut Tsarfaty of the Computer Science Department at the Open University of Israel. The door to Dr. Tsarfaty's office was always open whenever I ran into a trouble spot or had a question about my research or writing. She consistently allowed this thesis to be my own work, but steered me in the right direction whenever she thought I needed it.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Shimon Shocken and Prof. David Harel, for their encouragement, insightful comments, and hard questions.

I would also like to acknowledge again Prof. David Harel and the Weizmann Institute for hosting parts of the research in the software engineering lab on the Weizmann Premises. Especially, I would also like to thank Smadar Szekely and Guy Weiss from the Weizmann Institute for their technical support during our development in the *PlayGo* tool.

I would like to acknowledge all the anonymous reviewers of our submitted papers to *EMNLP 2014* and *ACL 2015*, and I am gratefully indebted to them for their very valuable comments on parts of this thesis.

Finally, I must express my very profound gratitude to my parents, Rimma and Dima Pogrebezky, and to my fiancée and future wife, Lilach Gold, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Ilia Pogrebezky, Bnei Ayish

Statement of Authorship and Collaboration

Chapter 4 of this thesis was done in collaboration with *Weizmann Institute* and parts of it were published as Reut Tsarfaty, Ilia Pogrebezky, Guy Weiss, Yaarit Natan, Smadar Szekely and David Harel. *"Semantic Parsing Using Content and Context: A Case Study from Requirements Elicitation"*. In: Proceedings of the international meeting on Empirical Methods of Natural Language Processing (EMNLP), October, 2014.

Contents

List of Figures

List of Tables

List of Abbreviations

NLP	Natural Language Processing
RE	Requirements Engineering
SE	Software Engineering
CNL	Controlled Natural Language
POS	Part-of-Speech
PCFG	Probabilistic Context-Free Grammars
HMM	Hidden Markov Model
UML	Unified Modeling Language
LSC	Live Sequence Chart
SM	System Model

Chapter 1

Introduction

1.1 Background

Let us start with a simple question: what is a computer program? Wikipedia describes it as follow: *"A computer program is a collection of instructions that performs a specific task when executed by a computer ... A computer program is usually written by a computer programmer in a programming language."* . This definition misses an important observation, namely, that the specific task should address some desired behavior. Another interesting point in wikipedia's definition is the need of a specialist and a programming language to create a computer program.

Software engineering usually starts with a need and an idea that can address this need. The need and the way to solve it are described and formalized in Natural Language requirements documents, via a process that is called *Requirements Elicitation*. Later, those requirements documents are translated (i.e., implemented) by software engineers into a programming language that can be executed by the machine and fulfill the desired need.

In this thesis, we aim to automatically translate requirements documents into executable code using Natural Language Processing (NLP) methods and techniques. NLP is a field of research concerned with the interactions between computers and humans in Natural Language. Many challenges in NLP involve natural language understanding, enabling computers to derive meanings from natural language input.

1.2 Related research

In previous scientific studies, researchers aimed to automate the text-to-code translation task. At the beginning, Abbott, 1983 focused on extracting object models from sentences (a.k.a. Classes, Methods and Properties). Later, Mich, 1996 defined a semi-English language, a.k.a. a Controlled Natural Language (CNL), and used it for specification documents. CNLs enjoy an easy translation to a formal representation (like other programming languages) but they suffer from non-intuitiveness and they are unnatural to humans.

Recent studies opened the door to translating unrestricted NL into code in domain-specific tasks. Examples include analyzing API documents to infer API library specifications (Zhong et al., 2009), analyzing code comments to detect concurrency bugs (Tan, Zhou, and Padioleau, 2011) and automatically generating parser programs from natural language input format descriptions (Barzilay et al., 2013).

Unlike to the aforementioned studies, we aim to develop a general technique for semantic parsing of requirements documents in the area of reactive systems. Furthermore, we aim to go beyond capturing the static parts (i.e., the objects structure) and also capture the dynamic behavior of the system (i.e., object initiations, inter-object instance communication and interactions, and more).

1.3 Hypothesis, Methodology and Strategy

We aim to tackle the task of translating requirements documents into executable code using an intermediate formal representation called *Live sequence charts* (LSC) which is a formal, unambiguous, representation of multi modal charts that capture the dynamic system behavior. The set of LSCs is accompanied by a *system model* (SM) that captures the structure of the system (the static part).

From the *SE* perspective, the LSC language has been defined for formalizing requirements in a fashion that accommodates the way humans think. Moreover, its constructs are scenarios that can be usually described with a single sentence in natural language. From the *NLP* perspective, LSCs have a very good alignment with linguistic constructs that NLP algorithms aim to recover, such as *entities, events, temporality* and more. Finally, the LSC representation directly fulfills the basic requirement of text-to-code translation, since it has a direct translation into executable java code (Harel and Maoz, 2006, Harel et al., 2010).

Gordon and Harel, 2009, defined a Controlled Natural Language using a context-free grammar and a semi-automatic parser, that with the help of an interactive user can solve ambiguities and map a single CNL sentence into an LSC/SM representation.

Our first challenge is to create a fully automatic parser for this CNL language that, using the context of the entire document, can parse successfully a complete requirements document without the need of an co-operating user.

Using a small seed of annotated data, we developed a statistical parser that successfully parse requirements in CNL with 95% F-Score on a small benchmark of CNL scenarios created by human experts (for more details see chapter 4 table 4.11.). In this, we succeeded in alleviating the need of human disambiguation.

Furthermore, we proved empirically that context matters, i.e., that parsing a requirements document as a whole, instead of each sentence in isolation, improved parsing accuracy. This is because overlapping between sentences helps disambiguation.

Having automated the CNL to LSC mapping, our next goal is to handle unrestricted natural language requirements documents. The first challenge was to find a suitable data. Roth et al., 2014 treat the mapping from Natural Language requirements to formal representations as a semantic parsing task and contribute an annotated data set, which contains a set of requirements documents from CS students homework assignments. Each document describes an atomic simple application such as a messaging system (see more in appendix B).

Using the data of Roth, we developed a rule-based algorithm that integrates statistical tools including Part-of-Speech taggers, a semantic role labeler, phrase and dependency parsers and linguistic ontologies that we created in order to translate each requirements document into an LSC/SM formal representation. Our algorithm exploits the context of the whole document and translates the requirements into an LSC/SM representation. In order to evaluate our results, we developed an evaluation metric for object/data modeling based on tree edit distance.

As reference translations for our evaluation we used human modeling by software engineering experts with diverse backgrounds, such as students, junior and senior software engineers, in order to compare our automatically generated System Models against expert created System Models. We found that the agreement scores of the auto-generated models compared with the models created by human experts was almost the same as the agreement scores between different engineers.

Our experiments raise an interesting and important question for further research in this area, namely: is there a perfect model? and more interestingly, do we need to assume a single perfect model when developing models for the text-to-code translation task? and finally, with no attested gold, what would be a sound evaluation method for text-to-code?

1.4 Outline

In the next sections we describe in details the different parts of this research. Firstly, in chapter 2 we survey related work on text-to-code translation and in chapter 3 we discuss the formal preliminaries that are common to chapters 4 and 5. In chapter 4 we describe our work on automating the CNL translation into LSC representation, and how we modeled, developed and evaluated it. In chapter 5 we describe how we extend our domain to unrestricted Natural Language. We describe our algorithm, its implementation and evaluation. Finally, in chapter 6 we conclude our work and discuss possible future research directions.

Chapter 2

Related Work

2.1 When SE met NLP

One of the oldest question in computer science is whether one could program a computer by speaking to it in natural language (Dijkstra, 1979). Programming in natural language may seem impossible, because it requires complete natural language understanding and dealing with the vagueness of human specification of programs.

Natural language programming intersects two disciplines in computer science, on the one hand we are dealing with *Natural Language Processing* (*NLP*), since our input is a specification document written in *natural language* that describe a desired software system, and on the other hand with *Software Engineering* (*SE*), since our output is an executable code (a software system).

To be more precise, in *SE* we focus on *Requirements Engineering* which is a subfield of SE that deals with the process of defining, documenting and maintaining requirement, while in *NLP* we focus on *Semantic Parsing*, the task of accepting a requirements document as input, and output a formal representation that captures the meaning, in our case the desired system.

2.1.1 Requirements Engineering

Every software system is ultimately measured by the degree to which it is able to meet the purpose for which it was designed. Requirements Engineering (RE) is the process of describing that purpose, by identifying the stakeholders and their needs. The output of *RE* is a specification document, that is written in a rich natural language. After that, the system analyst transforms the captured information into formal and semi-formal artifacts, mostly diagrams (Leite, 1987).

There is a number of inherent difficulties in the *RE process*. The stakeholders might be numerous and distributed, their goals may vary and even conflicting. Also, their goals may not be explicit and are difficult to articulate, which makes this process critical to the success of the final system (Nuseibeh and Easterbrook, 2000). We aim to close this gap by allowing a deeper analysis of stakeholder desired scenarios, which are mostly expressed in natural language. Moreover, by allowing to execute these NL scenarios the analysis might be able to detect conflicts early in the process.

2.1.2 Natural Language Processing

Natural Language Processing (NLP) is a field of research in computer science and a sub-field of artificial intelligence, that mainly deals with automatically analyzing and understanding human language. The algorithms in NLP can be abstracted to a structure prediction function, where the input is a natural language utterance, such as words, sentences and whole documents, and the output is some formal representation that represents the linguistic structure and content. Common NLP tasks are usually divided to analyzing the syntactic and semantic levels. Syntax level algorithms focus on capturing the grammar and structure of the language, for example, determine sentences, phrases, part-of-speech tags, derivation trees and more. Semantic level algorithms focus on the meaning of the given utterance. For example, in question-answering systems, in order to capture the knowledge and make reasoning possible, the knowledge is usually captured by some formal logic that supports the reasoning process. In our case, our output is an executable program that captures the meaning of the system specification in the requirements document.

2.1.3 Natural Language Programming

Early work in natural language programming has been rather ambitious, targeting the generation of complete computer programs that would compile and run. For instance, Ballard and Biermann, 1979 created the “NLC” prototype which aimed to create a natural language interface for processing data stored in arrays and matrices, with the ability of handling low level operations such as the turning natural language statements like add y1 to y2 into the programmatic expression y1 + y2. These first attempts triggered the criticism of the community. Dijkstra, 1979 took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish". However, a couple of decades later, Lieberman and Liu, 2005 describe why new developments in *NLP* might now make programming in natural language feasible. In their work, they proposed NL programming as a testbed for NL processing algorithms. Their main claim is that the tremendous improvement of natural language parsers and semantic extraction techniques make the NL programming dream feasible.

2.2 SE for NL-Programming

The first researchers to face the challenge of *natural language programming* came from the *software engineering* community. SE researchers often use existing off-the shelf *natural language processing* tools (for example — using part-of-speech tagging to determine nouns as possible objects) and apply additional transformations on their output to map them to executable code. In the following sections we will review two main approaches : *NL-Programming using Controlled Natural Language* and *NL-Programming using off-the-shelf tools*.

2.2.1 Controlled Natural Language

Software programming languages are formal languages that have an unambitious translation to machine code. So, one of the first approaches to natural language programming was to try to define a formal language that will look like a natural language. *Controlled natural languages* (CNLs) are subsets of natural language, obtained by restricting the grammar and vocabulary in order to reduce or eliminate ambiguity and complexity. Traditionally, controlled languages fall into two major types: those that improve readability for human readers (e.g. non-native speakers), and those that enable reliable automatic semantic analysis of the language (Kuhn, 2014).

Several CNLs were introduced for Requirements Elicitation. Fuchs and Schwitter, 1995 developed the *Attempto Controlled English* (ACE), a sublanguage of English whose utterances can be unambiguously translated into Prolog clauses, hence become formal and executable. Over the years, ACE has evolved into a mature controlled language, which is used mainly for reasoning about software requirements specification.

Bryant and Lee, 2002, use a CNL called *Two-Level Grammar* (TLG) to first extract the objects and methods and then extract classes, hierarchies and methods. TLG is designed to output UML class diagrams and Java code. The methods are described in natural language as a sequence of intra-object behaviors.

Cabral and Sampaio, 2008, propose a strategy that automatically translates use cases, written in a Controlled Natural Language with a fix grammar, into specification in CSP process algebra. They define templates that represent requirements at different levels of abstraction .Moreover, a refinement notion is defined based on events mapping between abstract and concrete models.

Zapata and Losada, 2012, proposed a model for knowledge representation of the transformation process from a natural language discourse into controlled language specifications (within the context of the requirements elicitation process) by using pre-conceptual schemes.

Gordon and Harel, 2009, created a controlled natural language interface, which, for a useful class of systems, yields the automatic production of executable code from structured requirements.

The main drawbacks of the CNL approach is the limitation on the expressivity (a.k.a what can be expressed by the language) of the formal language and the non-intuitiveness to the *average Joe* compared to unrestricted rich natural language.

2.2.2 Off-the-shelf solutions

In order to deal with the drawbacks of the CNL approach, some researchers aimed to use off-the-shelf natural language processing solutions to parse requirements documents written in unrestricted natural language and extract different levels of software artifacts abstraction.

The first one to employ this approach was Abbott, 1983. In his paper he introduces a technique for extracting data types, objects, variables and operators from informal English texts. His method was based on a simple rule-based algorithm, in which nouns were determined as objects and

verbs as operators between them. Later, Booch, 1986 has extend Abbott's approach to the object oriented paradigm.

The first automatic prototype for constructing object-oriend models from informal requirements was introduced by Saeki, Horai, and Enomoto, 1989. Unlike Abbott and Booch, Seaki's system is based on automatically extracted nouns and verbs. Due to high ambiguities that characterize informal requirements, Saeki realized that in order to achieve diagrams of reasonable quality human intervention was still needed to distinguish between words that are relevant for the model and irrelevant nouns and verbs.

Nanduri and Rugaber, 1995 proposed to further automate the object-oriented analysis of requirement texts by applying a syntactic parser and a set of post-processing rules. In a similar setting, Mich, 1996 employed a full NLP pipeline that contains a semantic analysis module, thus omitting the need for additional post-processing rules. More recent approaches include those by Harmain and Gaizauskas, 2003 and Kof, 2004, who relied on a combination of NLP components and human interaction.

As mention before, Lieberman and Liu, 2005 have conducted a feasibility study and showed how a partial understanding of a text, coupled with a dialog with the user, can help non-expert users make their intentions more precise when designing a computer program. Their study resulted in a system called METAFOR (Liu and Lieberman, 2005), able to translate natural language statements into class descriptions with the associated objects and methods.

Gulwani and Marron, 2014 developed NLyze, a system that synthesizes spreadsheet formulas from NL. Their translation algorithm builds over ideas of keyword programming and semantic parsing.

Most approaches aimed to create class diagrams. Ghosh et al., 2014 proposed a pipeline architecture that converts syntactic parses to logical expressions via a set of heuristic post-processing rules.

The biggest drawback of the off-the-shelf approach is that these NL processing tools were not designed for the specific domain of NL programming, and therefore, those works are not able to go beyond capturing static models and shallow interactions between object. In order to capture the dynamic nature of programming, domain-oriented semantic parsers that aim to capture the dynamic aspects of meaning of the specification documents are needed.

2.3 NL-Processing for NL-Programming

Mihalcea, Liu, and Lieberman, 2006 were the first to tackle the NL Programming problem from a *Natural Language Processing* perspective. In this research, Mihalcea et al. argue that modern Natural Language Processing techniques can make possible the use of natural language to express programming ideas, thus drastically increasing the accessibility of programming to non-expert users. To demonstrate it, they tackle two main programming patterns : steps and loops. Their algorithms consisted of three components:

- *Step finder*, which has the role of identifying in a natural language text the action statements to be converted into programming language statements.

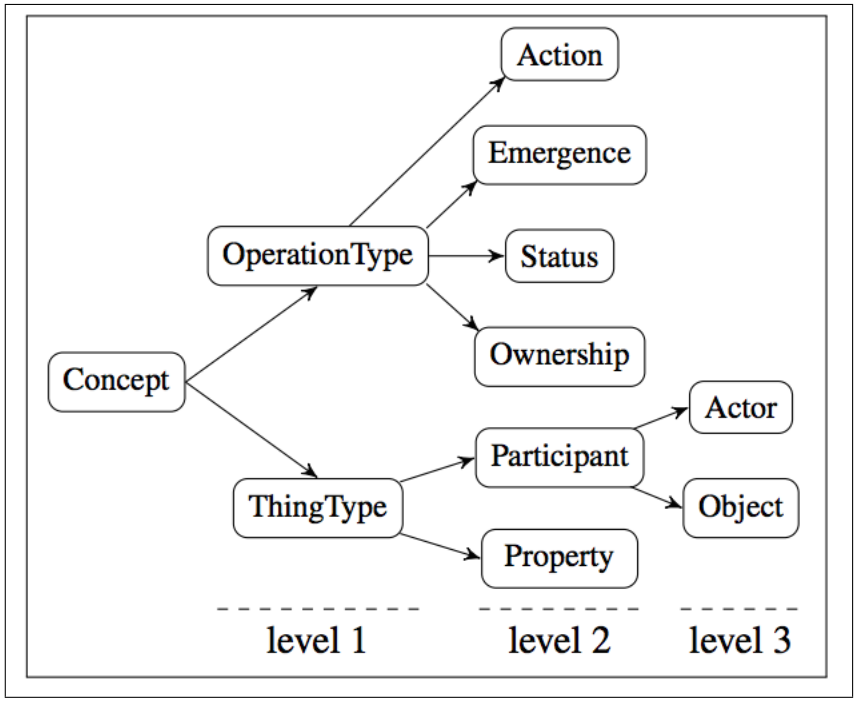


FIGURE 2.1: Class hierarchy of Michael Roth's conceptual ontology for modeling software requirements.

- *Loop finder*, which identifies the natural language structures that indicate repetition.
- *Comment identification* components, which identify the descriptive statements that can be turned into program comments.

The step and loop finder components are based on information extraction methods that use keywords and patterns to extract the action and then linguistic structures to extract the arguments.

Later on, Roth et al., 2014 suggest to treat the mapping from NL requirements documents to formal representation as a semantic role labeling task. This research establishes an ontology for formally representing requirements (as shown in figure 2.1). The ontology is hierarchal and includes three levels. In the first level they distinguish between *things* and *operations*. Where a *thing* can be a *Participant* which is involved in an operation. They further subdivide Participants into *Actors*, which can be users of a system or the system itself, and *Objects*. A *Property* is an attribute of an *Object* or a characteristic of an *operation*. *Operations* are divided as well to few subclasses — for example an *Action* that describes an operation that is performed by an *Actor* on one or several *Object(s)* (see figure 2.1).

Later, Roth and Klein, 2015 adapt techniques from semantic role labeling on these concepts and relations for describing static software functionalities. In our research, we aim to go beyond the static structure and first level interaction (a.k.a. subject-verb-object), and we target capturing the dynamic parts and interactions involving events, quantifiers and other software artifacts such as loops, conditions, and so on, which cannot be captured via the terms defined in this ontology.

2.3.1 Semantic Parsing

Semantic parsing is the process of automatically mapping a natural-language sentence into a formal representation of its meaning. A shallow form of semantic representation is a case-role analysis (a.k.a. a semantic role labeling), which identifies roles such as *agent*, *patient*, *source*, and *destination*. A deeper semantic analysis provides a representation of the sentence in predicate logic or other formal language which supports automated reasoning.

So how do we define such formalization and translate natural language to it? Many different formal representations were defined, some of them capture only the static parts of the meaning, others tried to capture dynamic and temporal aspects. After the formalization is defined, the translation task is usually done using machine learning algorithms.

Moens and Steedman, 1988 use temporal logic to describe the semantics of natural language, they proposed an ontology based on notions such as causation and consequences. Later studies combine temporal logic with distributional statistics (Lewis and Steedman, 2014). These approaches remained theoretical and there currently exists no algorithms to map NL sentences to these dynamic semantic representations.

Clark, Coecke, and Sadrzadeh, 2011 proposed a mathematical framework for unification of the distributional theory of meaning in terms of vector space models. Das et al., 2010 contribute a formalization of frame-semantic parsing as a structure predication problem, their parser finds words that evoke FrameNet frames, select frames for them, and locate their arguments. Liang, Jordan, and Klein, 2011 use logical form as semantic representation to learn how to map questions to answers via latent logical forms, which are induced from question-answer pairs (supervised model). In a later study, Liang and Potts, 2015 present a discriminative learning framework for compositional semantic models and relate them to logical theories.

Poon and Domingos, 2009 are the first research to apply unsupervised modeling to semantic parsing, by using Markov logic. Their USP system transforms dependency trees into quasi-logical forms, recursively induce lambda forms and clusters them to abstract away syntactic variations of the same meaning.

Every semantic system differs in the input, the learning method and the output. All those parameters are derived from the task that the parser aims to solve. In this work, we focus on understanding natural language description, therefore, we use a semantic representation that aims to capture the dynamic, executional meaning of the processes that will be executed by the specified system.

2.3.2 Applications

Recently, different studies on semantic parsing have been applied to the general problem of natural language programming, or text-to-code translation, in various sub-domains of programming. All these studies try to go beyond the creation of formal specification or structure.

Examples include analyzing API documents to infer API library specifications (Zhong et al., 2009), analyzing code comments to detect concurrency bugs (Tan, Zhou, and Padioleau, 2011) and automatically generating parser programs from natural language input format descriptions (Barzilay et al.,

2013). Kushman and Barzilay, 2013 consider the problem of translating natural language text queries into regular expressions.

Other popular applications of semantic parsing to NL programs are database queries. Thompson, Mooney, and Tang, 1997 learned a deterministic shift reduce parser for this task, while Zettlemoyer and Collins, 2012 use a log-linear model to induce a grammar for the problem.

Other works focus on mapping natural language instructions to actions. Artzi and Zettlemoyer, 2013 used a grounded CCG semantic parsing approach to interpret and execute instructions using a weakly supervised model. Branavan, Zettlemoyer, and Barzilay, 2010 also deal with mapping high level instructions to commands in an external environment.

All of the aforementioned studies focus on sentence or paragraph level and a specific task, while we target the analysis of document/discourse level, towards describing a complete software system.

2.4 Summary

The text-to-code translation is a challenge at the intersection of *SE* and *NLP* disciplines. This task is an old dream that was heavily criticized by Dijkstra. At the beginning, SE researchers tackle this challenge with two main approaches — definitions of CNLs and off-the-shelf NLP tools. Recent NLP studies approach the task by casting it as a semantic parsing tasks and translating sentences into some, often static, formal representation.

Chapter 3

Formal Preliminaries

In this chapter we review the formal preliminaries required for the next chapters which will in turn describe our text-to-code translation algorithms.

3.1 The Semantic Representation

Let us restate our desired function: our input is a requirements document (a list of sentences where every sentence is a requirement of a desired system) written in natural language, and our desired output is a software system that meets those requirements. Formally, our input is a requirements document $D \in \mathcal{D}$, which consists of n requirements - $D = d_1, ..., d_n$, and our output is a desired software - SW. So our function is $f : D \rightarrow SW$. The first question that arises is: what formal representation is suitable for implementing our function?

In order to implement this function, we choose an intermediate formal representation called *Live sequence charts* (Damm and Harel, 2001). The first contribution of this thesis is the proposal that the LSC formal language, previously proposed for manual scenario-based programming, could provide a viable semantic representation for automatic (and statistical) text-to-code translation. In addition to LSCs which capture the dynamic behavior, our target formal representation also contains a *System model* (SM), which captures the static structure (i.e., the objects structure) of the desired system.

From the *SE* perspective, the LSC language has been defined for formalizing requirements and in a fashion that accommodates the way humans think. Moreover, its constructs are scenarios that can be usually described with a single sentence in natural language. From the *NLP* perspective, LSC has a very good alignment with linguistic constructs that NLP algorithms aim to recover, such as *entities, events, temporality* and more. Finally, the LSC representation directly fulfills the basic requirement of text-to-code translation, since it has a direct translation into executable java code (Harel and Maoz, 2006, Harel et al., 2010).

Harel et al., 2010, develop an algorithm that maps the LSC/SM representation into an executable code. So, relying on this work, we already have a deterministic function that maps an LSC/SM representation to code:

$$f_1 : LSC/SM \rightarrow SW$$

In order to complete our task we will develop a model for the function that maps a requirements document into its LSC/SM representation:

$$f_2 : D \rightarrow LSC/SM$$

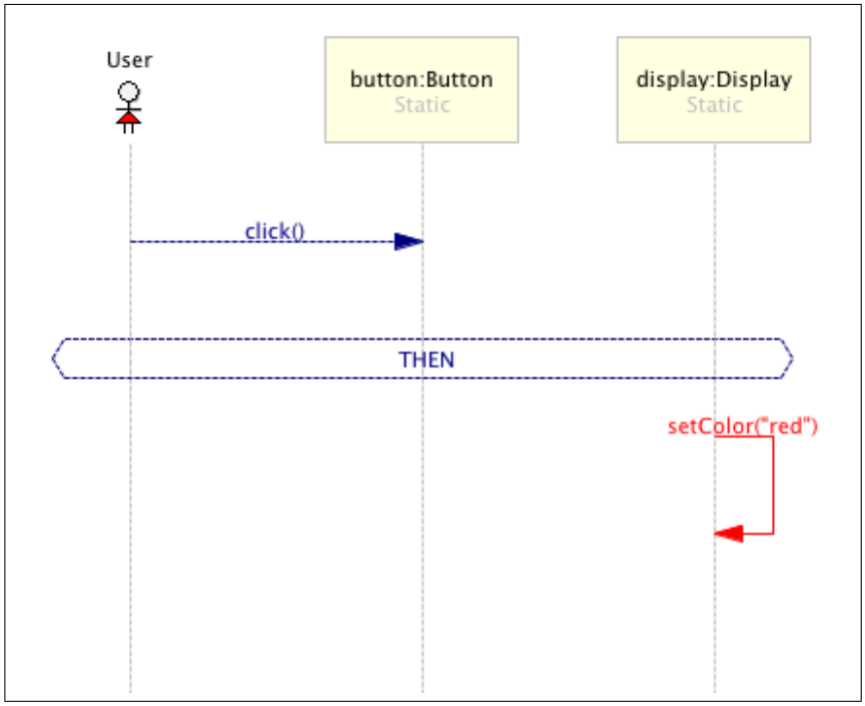


FIGURE 3.1: The LSC graphical depiction of the scenario: "When the user clicks the button, the display color must change to red."

By composing these functions, f_1 and f_2, we will achieve our translation goal.

$$f(D) \quad = \quad f_1(f_2(D)) \quad = \quad f_1(LSC/SM) \quad = \quad SW$$

In this work we make two preliminary assumptions on the task:

1. We assume that our input documents describe reactive systems.
2. Every sentence in the document is a self-contained requirement that can be mapped to a single LSC.

We will 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.

3.1.1 Live Sequence Charts

A **Live Sequence Chart** (LSC) is an extension of *Message Sequence Chart* (MSC). Entities in LSC diagrams are represented as vertical lines called *life-lines*, and interactions between entities are represented by horizontal arrows between lifelines (or a self-pointing arrow) called *messages*. 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, imposing a partial order on the execution of all messages.

A message can be "hot" (obligatory), represented by a red color, or "cold" (optional), colored in blue. Every message has an execution status: solid

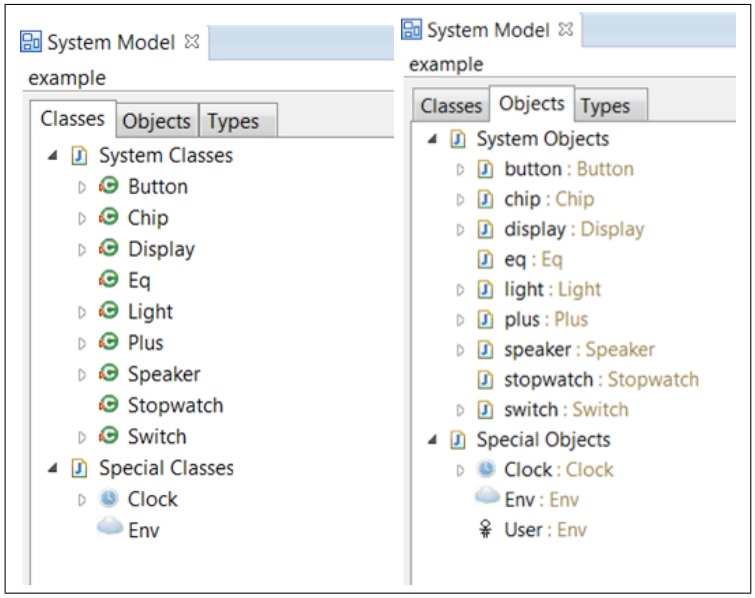


FIGURE 3.2: A System Model representing the System Architecture: *Classes* and *Objects* Views.

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 3.1 shows the LSC for a simple scenario involving two objects from the architecture presented in Figure 3.2.

So what can be expressed effectively by the LSC representation? The predecessors of LSCs, Message sequence charts (MSCs), are widely used in industry to document the interworking of processes or objects – but they are expressively weak, based on the modest semantic notion of a partial ordering of events (Damm and Harel, 2001). LSCs propose an extension of MSCs which deals with specifying "liveness", i.e., things that must occur in the lifespan of objects. LSCs allow distinguishing between possible and necessary behavior, both globally, on the level of an entire chart, and locally, when specifying events and conditions within a chart.

Furthermore, they enable naturally specifying constructs such as subcharts, branching (i.e., if statements - as in figure 3.3) and iteration (i.e., loops - as in figure 3.4), and allow specifying forbidden scenarios (as in figure 3.5). The representation also allows us to distinguish universal binding that applies on all instances of the participating lifelines, and existential binding that applies to only specific instance (figure 3.6).

Formally, it can be shown that the execution semantics of the LSC language is embedded in a fragment of a branching temporal logic called CTL* (Kugler et al., 2005). Gordon and Harel, 2011 showed empirically that LSC is a suitable formalism for specifying dynamic reactive systems, which is in turn our selected domain for formal specification.

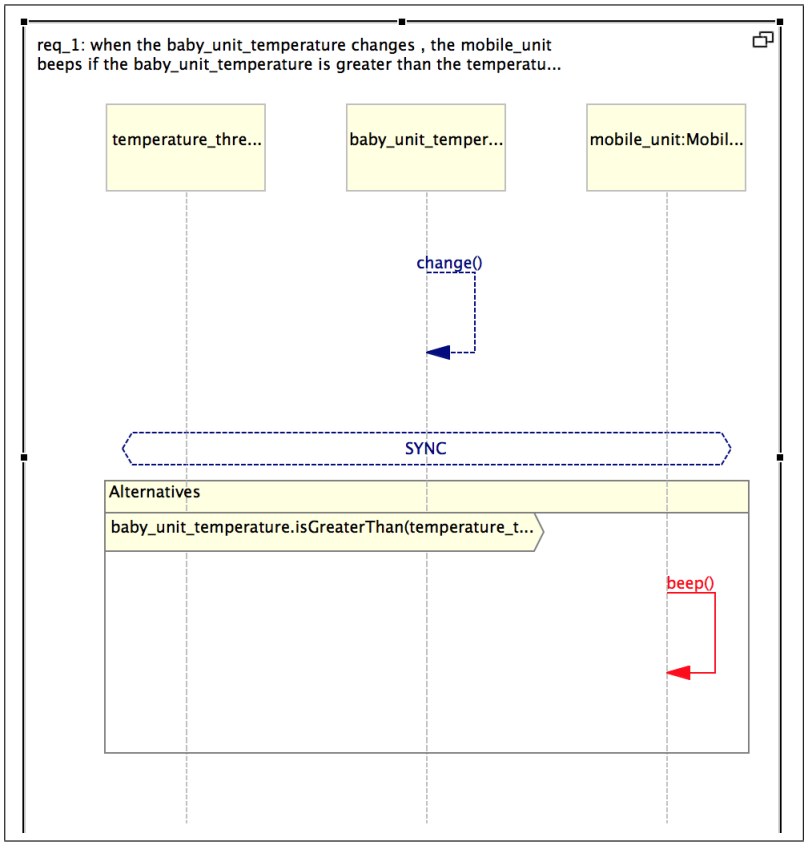


FIGURE 3.3: An LSC for an if statement that represents the requirement : "When the baby unit temperature changes, the mobile unit beeps if the baby unit temperature is greater than temperature threshold" (This requirement is taken from the Baby Monitor episode from the data of Gordon)

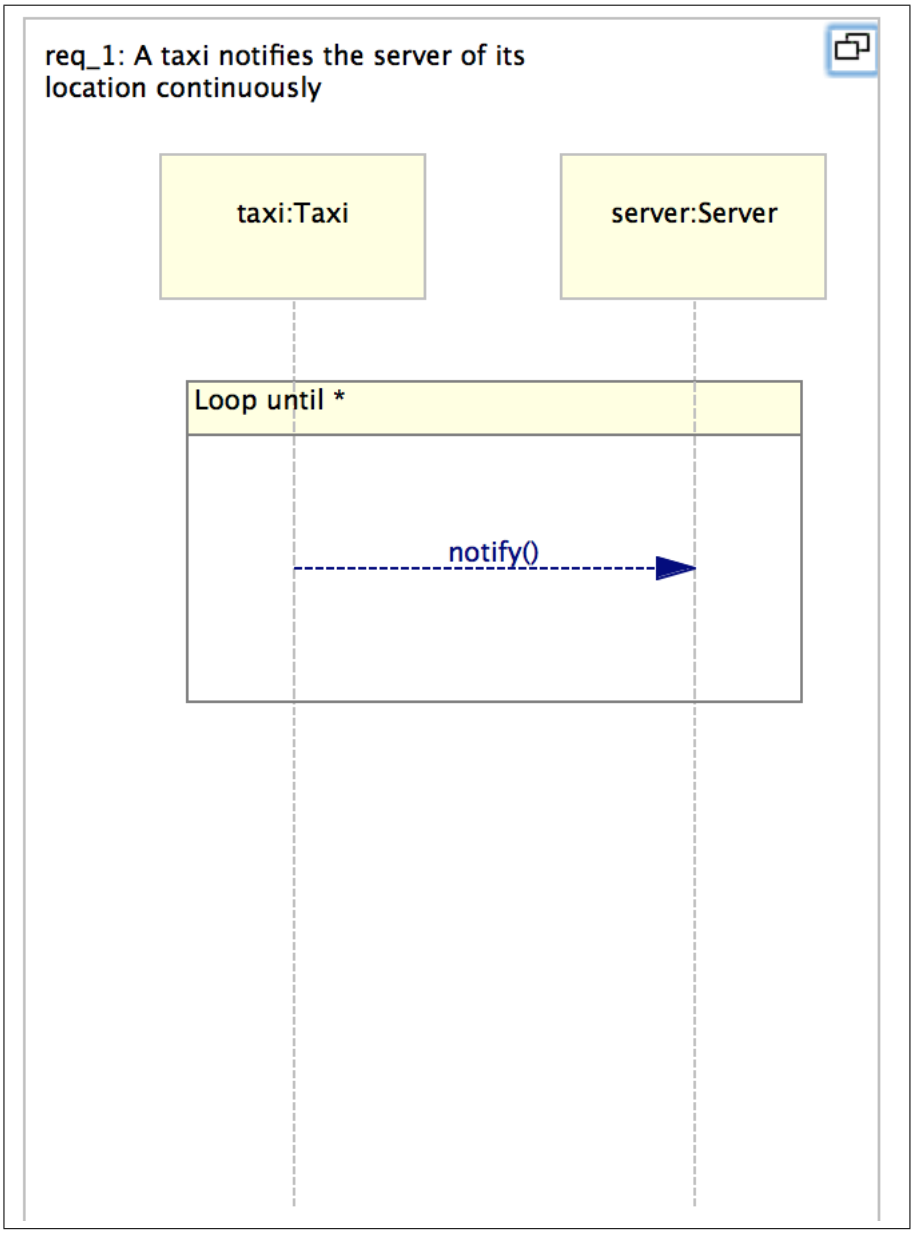


FIGURE 3.4: An LSC for a loop operation that represents the requirement: "A taxi notifies the server of its location continuously." (This requirement is taken from the Taxi episode of Roth et al., 2014).

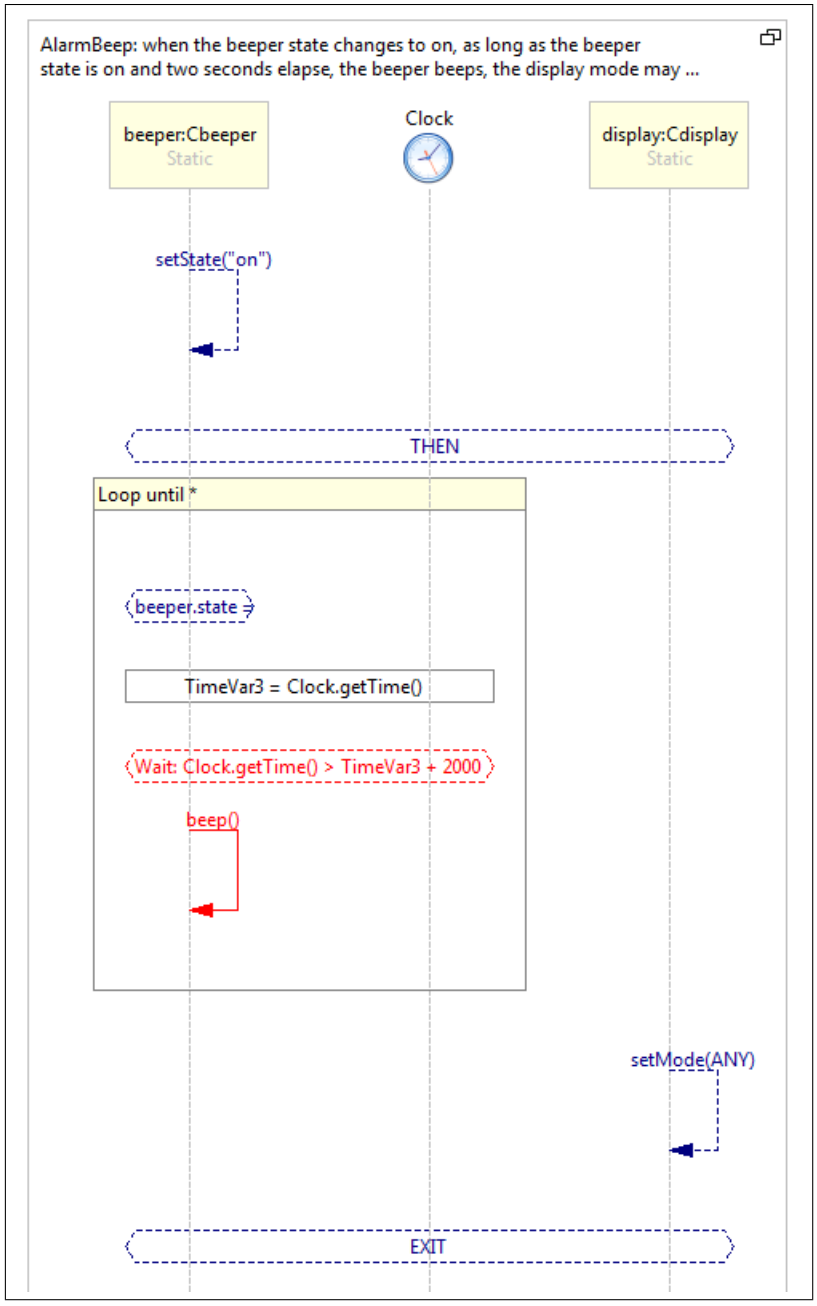


FIGURE 3.5: An LSC for a forbidden state that represents the requirement: "when the beeper state changes to on, as long as the beeper state is on and two seconds elapse, the beeper beeps, the display mode may not change." (This requirement is taken from the "wristwatch" episode at - http://wiki.weizmann.ac.il/playgo/index.php/Wristwatch_Example)

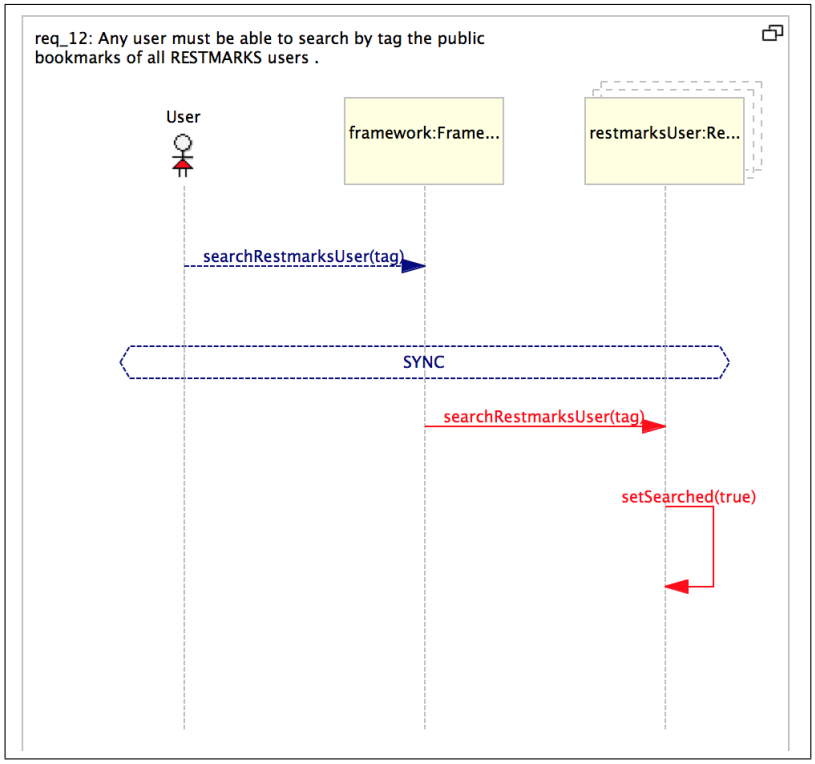


FIGURE 3.6: An LSC for the requirement: "Any user must be able to search by tag the public bookmarks off al RESTMARKS users" (This requirement is taken from the RESTMARK episode of Roth et al., 2014).

3.1.2 The System Model

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 3.2 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 reactive 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. Those three special objects represent the outside world, and allow us to model a system that can be integrated in a real world (interact with users, sample different real-world phenomenas - temperature, time and more). These objects effectively trigger the system events and cannot be expanded.

3.2 Play-In and Play-Out

Play-in/Play-out is a methodology for scenario-based specification of reactive systems developed be Harel and Marelly, 2003. In this methodology the behavior is "played in" directly from the system's GUI or some abstract version thereof, and can then be "played out", i.e., executed according to the specification. Play-in is a user-friendly high-level way of specifying behavior and play-out is a way of working with a fully operational system

directly from its inter-object requirements. This methodology is relevant to many stages of system development, including requirements engineering, specification, testing, analysis and implementation. Let us specify about these two concepts further.

3.2.1 Play-In

The basic unit of an LSC system is a "scenario", which describes a facet of the system's behavior. We assume that every requirement sentence can be mapped directly to a single "scenario". Play-in is the process of creating a scenario, it consists of demonstrating user actions and specifying possible or mandatory system reactions. There are at least two ways of playing-in a scenario, using a GUI application or NL play-in interface.

- GUI: 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* that represent the desired behavior.

- NL Play-in: In this approach, the user specifies scenarios by describing them in (semi-)natural language. Every requirement is parsed separately, ambiguities are resolved by user intervention (using interactive mode) and finally the requirement is mapped to LSC representation and the SM is updated accordingly. Post edit operations are also available using the GUI interface (Gordon and Harel, 2009). Figure 3.7 presents the NL Play-in interface, the user types a single requirement and the parser tries to resolve it. If there is more than one possible parse for the requirement, the user should disambiguate by selecting one of the offered possibilities.

3.2.2 Play-Out

Play-out is a method introduced in Harel and Marelly, 2003 for executing LSC specifications. According to them, play-out is the means for running 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 from the user action, and which terminates either at a stable situation, where the next event is a user action, or when the entire execution terminates. By executing the events in these charts and causing the GUI application and object map to reflect the effect of these events on the system objects, the user is provided with a simulation of an executable application. Figure 3.8 presents the LSC and SM during play-out. The blue traces manifesting the system behavior on the LSC in real time.

3.2.3 The PlayGo Tool

The Play-in/Play-out methodology is implemented by a tool called PlayGo [1]. PlayGo supports three modes of play-in:

- Editor: a drag and drop interface to create LSCs.
- GUI: the user specifies scenarios by playing them in directly from a graphical user interface of the system being developed.
- CNL Play-in: an initial implementation of NL Play-in, the user describes the desired system using a Controlled Natural Language.

As the behavior is played in, the play-go tool automatically generates a formal LSC/SM representation of the system. PlayGo then allows to play it out, causing the application to react according to the specified behavior and can be monitored to check their successful completion.

The execution can be visually depicted as a set of traces, as shown in Figure 3.8.

In this work, we aim to lift two inherent restrictions of the current CNL play-in. The first one is to remove the need of user disambiguation (see Figure 3.7), chapter 4 will focus on this task. The second is to augment the PlayGo tool with an additional mode of *unrestricted NL play-in*, chapter 5 will describe our steps towards this target.

3.3 Summary

The main contribution of this thesis is the proposal to use the LSC representation as a formal semantic representation of specification documents of reactive systems. Subsequently, the algorithms described in this thesis translating texts into their LSC representation are implemented and integrated in the PlayGo Tool (Harel et al., 2010).

[1] PlayGo project page - www.playgo.co

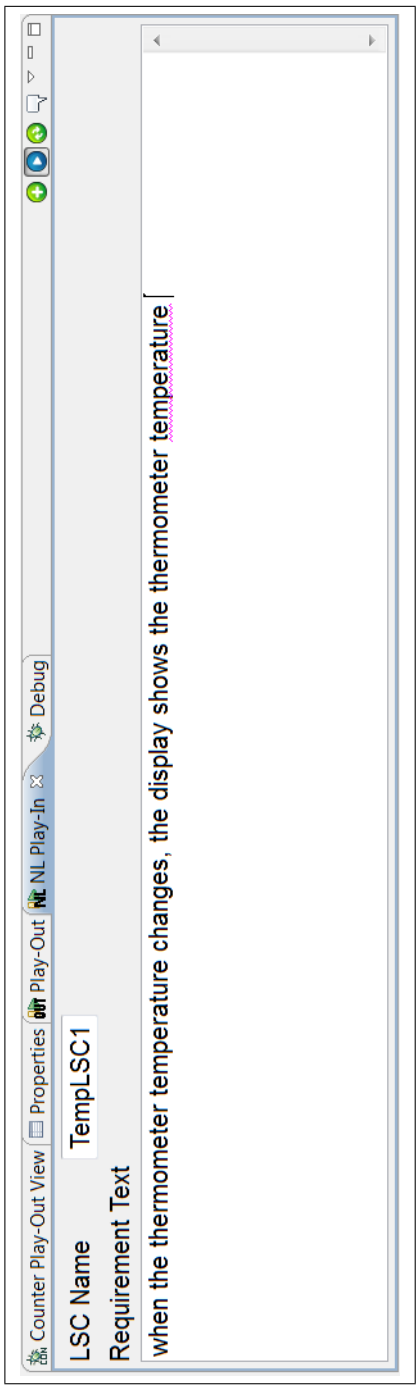


FIGURE 3.7: A screenshot of NL Play-In. Note the red squiggly line that indicates an ambiguity, i.e., that there is more than one possible parse for the sentence. The user should disambiguate by selecting one of the offered possibilities. The selection can have a significant effect on the created LSC.

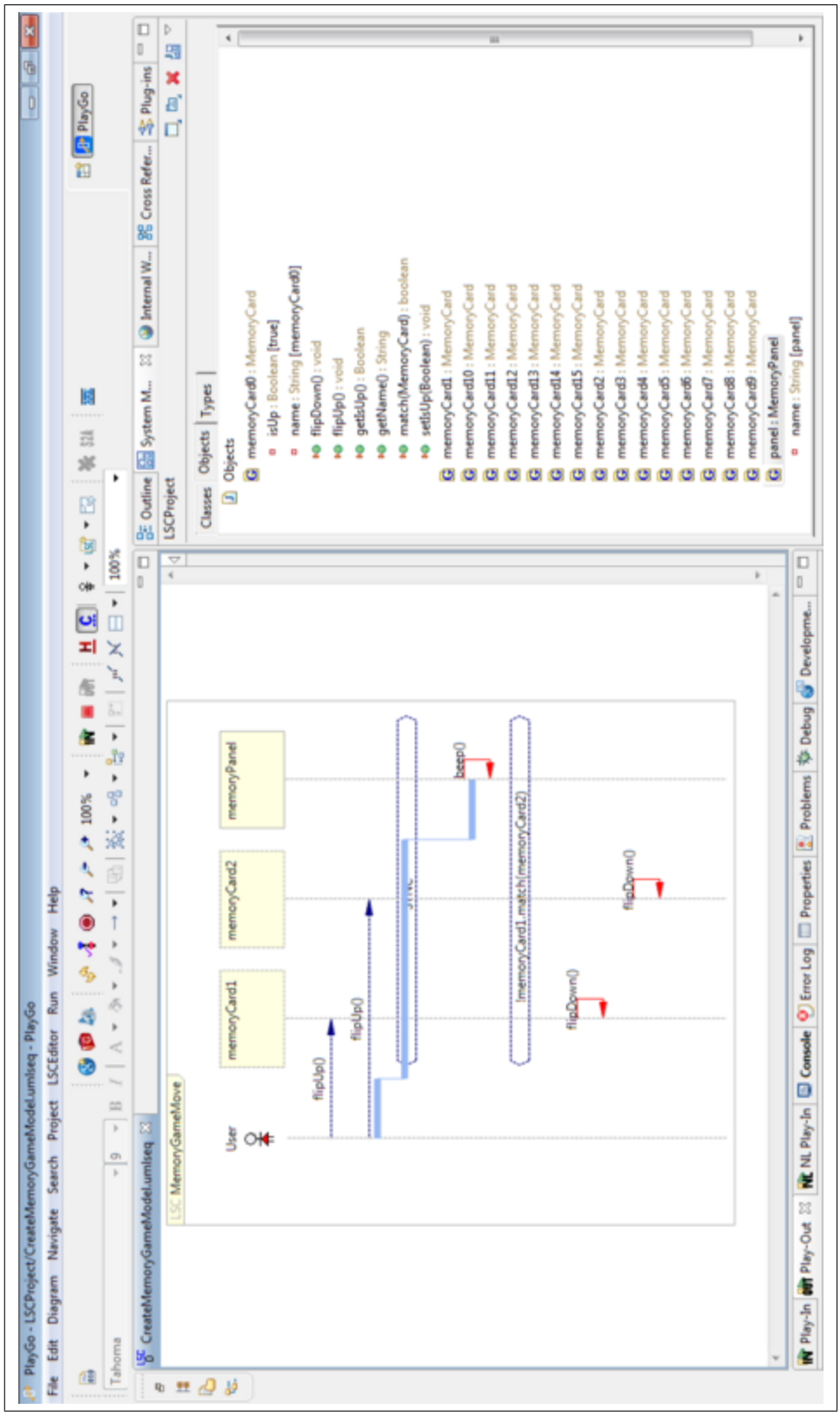


FIGURE 3.8: A Live Sequence Chart (LSC) and a System Model (SM) during play-out. Note the blue traces manifesting the system behavior on the LSC in real time, and the real-time properties of the Objects in the SM.

Chapter 4

Programming in Controlled Natural Language

4.1 Rule-Based NL Play-In

The starting point for this research is the work of Gordon and Harel, 2009, who define a CNL Play-in interface that is implemented in the PlayGo tool. This CNL defines a subset of English that is generated by a context-free grammar that has a direct translation into the LSC/SM representation. Currently the translation is semi-automatic and requires a user intervention when ambiguities arise. Our goal in this chapter is to create a fully automatic statistical parser that removes the need for user intervention by relying on statistical learning as well as exploiting document context in order to solve ambiguities automatically.

4.1.1 Controlled NL

The CNL defined by Gordon and Harel is based on a hand-crafted context-free grammar. A context-free grammar (CFG) is a set of production rules that allow to generate possible strings in the specified formal language. Production rules indicate simple replacements. Most of the programming languages are CFG languages and they are unambiguous in order to make the translation to machine code trivial.

Gordon and Harel, 2009 defined a *Control Natural Language* , a subset of English, that has a direct translation into LSC/SM representation. The CNL is described by *Context-free grammer* that includes 189 rules (for the version that we used in this project). In figure 4.1 we show an example of a requirement that can be derived by the grammar. The full specification of the CNL grammar can be found in appendix A.

4.1.2 Semi-Automatic Parser

Gordon and Harel, 2009 also developed a semi-automatic parser for the described CNL. The parser consists of a context-free grammar bottom-up parser and a dialog system that helps the programmer create both a system model and a set of LSC scenarios.

When a requirement is analyzed, it first tokenizes the sentence, than for each token the parser tries to fit a production rule. The parser uses a WordNet dictionary 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 ambiguity arise, i.e., when more than one rule can induce

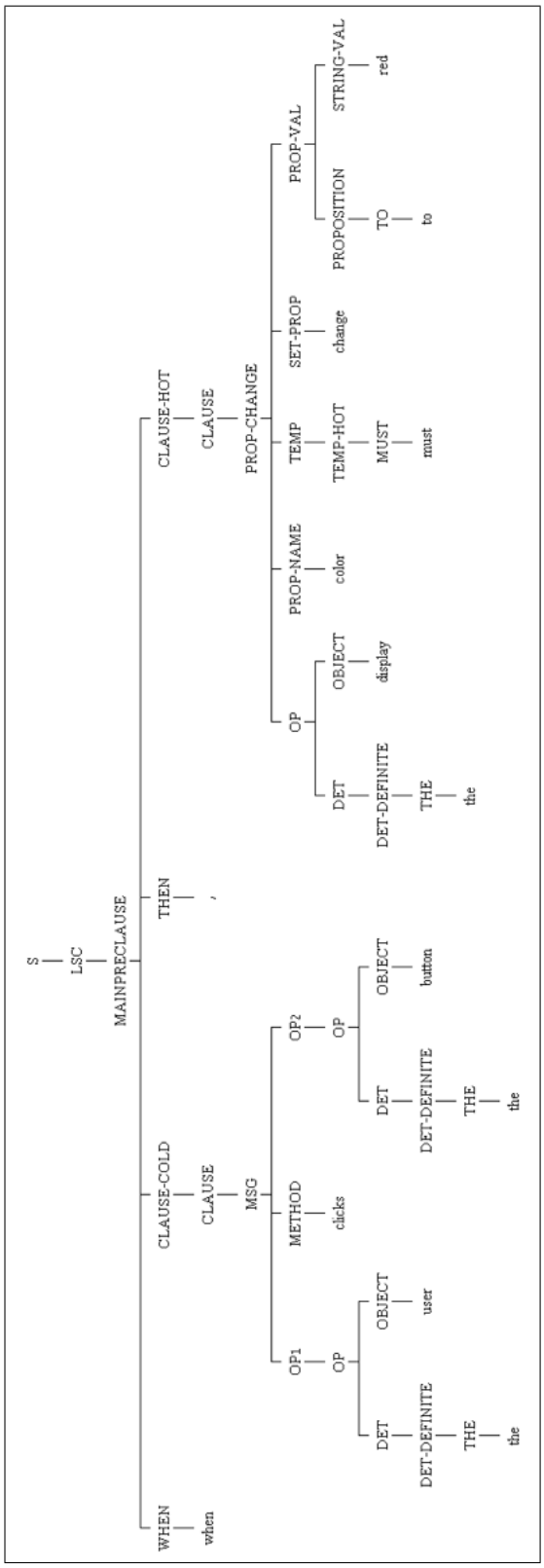


FIGURE 4.1: A derivation tree for the requirement: "When the user clicks that button, the display color must change to red".

the terminal, the system requires an interactive user intervention in order to solve the ambiguity.

As previously mention, our main goal in this chapter is to replace the semi-automatic parser with a fully automatic parser that uses statistical methods and exploits the context to resolve ambiguities and remove the need of the human precious time and effort.

4.2 What happen Next?

In this chapter we aim to create a fully automatic parser for the CNL-to-LSC task, i.e., remove entirely the need of an interactive user for solving ambiguities. In order to do that, we will use a common statistical modeling strategy based on Shannon's *Noisy Channel Model* (more details in subsection 4.2.1). Moreover, in order to solve ambiguities that might arise during parsing, we will exploit the discourse context of the entire document (more on that in subsection 4.2.2). Finally, by parsing complete documents a whole, we add the capacity of entering a complete system description and processing it in one go, without requiring a sentence-by-sentence manual process, as is currently the case in previous solutions.

4.2.1 Noisy Channel Model

Shannon, 1948 introduce one of the most important theories of computer since in the 20th century: A mathematical Theory of Communication. The Noisy-Channel coding theorem establishes that for any given degree of noise contamination of a communication channel, it is possible to communicate discrete data (digital information) nearly error-free up to a computable maximum rate through the channel. This model has been applied to a wide range of NLP problems. Bahl, Jelinek, and Mercer, 1983 formulated the speech recognition problem as a problem of maximum likelihood decoding based on the noisy channel model, Brill and Moore, 2000 used the noisy channel for spelling correction algorithm and Brown et al., 1993 used the noisy-channel model as the basis of a statical model of machine translation.

4.2.2 Discourse Analysis

Discourse analysis is a broad term for the study of the ways in which language is used in texts and contexts. The discourse properties of text have long been recognized as critical to language technology, and over the past 40 years, our understanding of and ability to exploit the discourse properties of text has grown in many ways. Webber and Joshi, 2012 recount these developments, the technology they employ, the applications they support, and the new challenges that each subsequent development has raised. As we are aiming to analyze specification documents and not separated sentences, it is very natural to use discourse methods to improve the results of the automatic analysis.

4.3 Statistical NL Play-In

In this chapter we define the probabilistic models for translating requirements written in CNL into LSC/SM representations. Based on our probabilistic models, we implement a statistical parser that automatically parses requirement documents and converts them into executable code.

4.3.1 Formal Settings

In the NL Programming task (i.e., text-to-code translation task), we aim to implement a prediction function $f : \mathcal{D} \rightarrow \mathcal{M}$,such that $D \in \mathcal{D}$ is a specification document consisting of an ordered set of requirements $D = d_1, d_2, ..., d_n$ and $f(D) = M \in \mathcal{M}$ is a code-base hierarchy that grounds the semantic interpretation of D (that is, the code that fits all the requirements). We formally denote the fact that M grounds the semantics of D as $M \triangleright \mathbf{sem}(D)$.

Furthermore, we assume that each requirement $d_i \in D$ represents a certain aspect of the model M. We assume a sequence of snapshots of M that correspond to the timestamps $1...n$, that is: $m_1, m_2, ..., m_n \in M$ where each SM snapshot is the section of the code-based that grounds the implementation of the specific LSC that represents the meaning of the requirements. Formally, we term this $\forall i : m_i \triangleright \mathbf{sem}(d_i)$.

Let Σ be a finite lexicon and let $\mathcal{L}_{req} \subseteq \Sigma^*$ be a language for specifying requirements. At the beginning, we limit our domain to controlled natural language, so we assume that requirements in $\mathcal{L}_{req}$ have been generated by a context-free grammar G. In this entire chapter we assume the d_i requirement sentences are all from $\mathcal{L}_{req}$.

We define $\mathcal{T}_{req}$ to be the set of trees strongly generated by G, and utilize an auxiliary yield function $yield : \mathcal{T}_{req} \rightarrow \mathcal{L}_{req}$ returning the leaves of the given tree $t \in \mathcal{T}_{req}$. Different parse-trees can generate the same utterance, so the task of analyzing the structure of every requirement $d \in \mathcal{L}_{req}$ is modeled via a function $syn : \mathcal{T}_{req} \rightarrow \mathcal{L}_{req}$ that returns the correct, human-perceived, parse $t \in \mathcal{T}_{req}$ of $d \in \mathcal{L}_{req}$.

4.4 Probabilistic Modeling

Our formal representation consists of two parts, LSCs and a SM. LSCs are responsible for capturing the *content* of every requirement, while the SM is a global object that captures the *context*. According to these phenomena, we defined two probabilistic models. The first one is a content model, a sentence-level model which is based on a probabilistic grammar augmented with compositional semantic rules. The second model is a context model, discourse-level sequence model that takes into account the content of every sentence as well as the relation between System model snapshots (a local SM) at different time stamps.

In both models, given a requirements document D that consists of n requirements, $D = d_1, ..., d_n$, our focus is to map every d_i to a correct derivation tree of the CNL grammar. Then, we rely on the existence of a mapper algorithm $M : \mathcal{T}_{req} \rightarrow (LSC, SM)$ (implemented in Gordon and Harel, 2009), and we return an LSC/SM representation for each requirement.

In order to obtain a SM as a global object that represents the structure of the whole system, we combine all SM snapshots, mapped from each of the requirements, to a single global SM.

4.4.1 Sentence-Based Modeling

The theory

Our task is to learn a mapping function from each sentence to its correct LSC and SM snapshot. Formally, given a requirements document with n requirements $D = d_1, ..., d_n$, we map every d_i to the tuple (LSC_i, SM_i) where SM_i is a partial snapshot of the global SM. In this model, given a requirements document, we parse and analyze each requirement separately. Than, in order to complete our task we combine all partial SM_i together to create a global SM ($SM = \bigcup_{i=1}^{n} SM_i$).

We address this task via a probabilistic context-free grammar augmented with a semantic interpretation function. More formally, given a requirements document with n requirements $D = d_1, ..., d_n$, we assume that each d_i has been generated by a *probabilistic context-free grammar* (PCFG) G.

The syntactic analysis of d_i might be ambiguous and return more than one derivation tree. So, in order to overcome it we implement a syntactic analysis function $syn : \mathcal{L}_{req} \rightarrow \mathcal{T}_{req}$ using a probabilistic model that selects the most likely syntax tree. Formally, $syn(d) = argmax_{t \in \mathcal{T}_{req}} P(t|d)$. This proposal relies on Shannon's (1948) noisy channel model (more details in 4.2.1) and starts with the idea that pure semantic content (an SM snapshot) existed at every timestamp, and through the noisy channel, the SM snapshot gets "corrupted" into a requirement description written in NL. The task then is to remove the "noise" and restore the original SM snapshot.

Formally, given a requirement d, we can simplify $syn(d)$ with respect to the maximization as follows:

$$
\begin{aligned}
syn(d) &= argmax_{t \in \mathcal{T}_{req}} P(t|d) \\
&= argmax_{t \in \mathcal{T}_{req}} \frac{P(t,d)}{P(d)} && \text{conditional probability rule} \\
&= argmax_{t \in \mathcal{T}_{req}} P(t,d) && P(d) \text{ is constant for all trees} \\
&= argmax_{t\{t | t \in \mathcal{T}_{req} \wedge yield(t) = d\}} P(t) && \text{projection to trees that their yield is } r
\end{aligned}
$$

Since we assume context-freeness, it holds that $P(d) = \prod_{d \in der(t)} P(r)$, where $der(t)$ return the rules that derive t. Finally, our model will take the following form:

$$
syn(d) = argmax_{t\{t | t \in \mathcal{T}_{req} \wedge yield(t) = d\}} \prod_{r \in der(t)} P(r)
$$

The implementation

The sentence-based model relies on two main components - the PCFG model and the CKY parsing algorithm (Younger, 1967). The PCFG learning phase is an offline process that is learned once and used multiple times, while the CKY decoding is an online process, applied fresh for every new requirement.

The probability of $P(t)$ is estimated using a treebank PCFG (Charniak, 1996), based on all pairs $(d_{ij}, t_{ij}) \in D \times \mathcal{T}_{req}$ in the parallel corpus that consists of requirements $d_1, ..., d_n$ with their corresponded syntax trees $t_1, ..., t_n$. We have estimated the rule probabilities using maximum-likelihood estimations and smoothed unknown words based on rare words distributions. This means that words that haven't passed a minimal cutoff number of occurrences, were considered as special token ##UNKNOWN##, and this token probability is assigned to future unseen words.

In order to simplify our model, we perform a binarization on our trees that reserve the tree probability, such that every rule will be unary or binary, formally, $X \to Y$ or $X \to YZ$. The maximum likelihood will be as follow :

$$\hat{P}(r) \quad = \quad P(x \to \alpha) = \quad \begin{cases} \frac{count(x \to y \quad z)}{count(x)}, & \text{for binary rules} \\ \frac{count(x \to y)}{count(x)}, & \text{for unary rules} \end{cases}$$

with respect to the smoothing cutoff.

After we have learned a PCFG model, we iterate through the given requirements document, and for every sentence we apply the CKY algorithm to find the most likely syntax tree. In this greedy approach , we takes the best syntax tree for every sentence without taking into consideration the context.

Runtime complexity

The overall complexity of decoding a document with n requirements, where the number of tokens of the longest requirement is l, using a grammar G of size $|G|$, is given by: $O(n \times l^3 \times |G|)$. CKY worse case complexity is $O(l^3 \times |G|)$. The space complexity is the size of the table $O(l^2)$ (Younger, 1967).

4.4.2 Discourse-Based Modeling

The theory

In contrast to the sentence-base modeling, in this model, the selection of a derivation tree for every $d \in D$ will not rely only on a local maximization, but will take into account also the context. As before, our input is a requirements document $D \in \mathcal{D}$ and we aim to find the most probable system model $M \in \mathcal{M}$ that fits the requirements.

We assume that M reflects a single domain that the stakeholders have in mind, and we are provided with an ambiguous natural language evidence, an elicited discourse D, in which they convey it.

We instantiate this view as a noisy channel model (Shannon, 1948), which provides the foundation for many NLP applications, such as speech recognition Bahl, Jelinek, and Mercer, 1983 and machine translation Brown et al., 1993. According to the noisy channel model, when a signal is received it does not uniquely identify the message being sent. A probabilistic model is then applied to decode the original message.

In our case, the signal is the discourse and the message is the overall system model. In formal terms, we want to find a system model M that maximizes the following:

$$f(D) = argmax_{M \in \mathcal{M}} P(M|D)$$

We can simplify further :

$$\begin{aligned} f(D) &= argmax_{M \in \mathcal{M}} P(M|D) \\ &= argmax_{M \in \mathcal{M}} \frac{P(D|M)P(M)}{P(D)} && \text{Bayes law} \\ &= argmax_{M \in \mathcal{M}} P(D|M)P(M) && P(D) \text{ is constant} \end{aligned}$$

Therefore, we would like to estimate two types of probability distributions, $P(M)$ over the source and $P(D|M)$ over the channel. Both M and D are structured objects with complex internal structure. In order to assign probabilities to objects involving such complex structures it is customary to break them down into simpler, more basic, events. We know that $D = d_1, ..., d_n$ is composed of n individual requirement sentences, each representing a certain aspect of the model M. We assume a sequence of snapshots of M that correspond to the timestamps $1...n$, that is: $m_1, m_2, ..., m_n \in M$ where $\forall i : m_i \triangleright sem(di)$.

The complete SM is given by the union of the different snapshots reflected in different requirements, $M = \bigcup_i m_i$. We then rephrase:

$$\begin{aligned} P(M) &= P(m_1, ..., m_n) \\ P(D|M) &= P(d_1, ..., d_n | m_1, ..., m_n) \end{aligned}$$

These events may be seen as points in a high dimensional space. In actuality, they are too complex and would be too hard to estimate directly. We then define two independence assumptions. First, we assume that a system model snapshot at time i depends only on k previous snapshots (a stationary distribution). Secondly, we assume that each sentence i depends only on the SM snapshot at time i. We now get:

$$\begin{aligned} P(m_1, ..., m_n) &\approx \prod_i P(m_i | m_{i-k}, ..., m_{i-1}) \\ P(d_1, ..., d_n | m_1, ..., m_n) &\approx \prod_i P(d_i | m_i) \end{aligned}$$

Therefore, assuming bi-gram transitions, our objective function is now represented as follows:

$$f(D) = argmax_{M \in \mathcal{M}} \prod_{i=1}^{n} P(m_i | m_{i-1}) P(d_i | m_i)$$

Note that m_0 may be empty if the system is implemented from scratch, and non-empty if the requirements assume an existing code-base, which makes $P(m_1|m_0)$ a non-trivial transition.

The implementation

The discourse-based model is in essence a Hidden Markov models, where the states capture SM snapshots, state-transitions probabilities model that transition between SM snapshots and emission probabilities model the verbal description of each state.

In order to implement this, we defined a training algorithm that automatically learns the values of model parameters $P(m_i|m_{i-1})$ and $P(d_i|m_i)$ given annotated data, and a decoding algorithm that, given the learned parameters, generates SM snapshot candidates for each requirement and searches through possible sequences of SM snapshots to find the one that maximizes our objective function.

So, the learning algorithm learns to assign numerical estimates to the $P(m_i|m_{i-1})$ and $P(d_i|m_i)$ parameters, and the decoding algorithm implements the search for $argmax$. And our objective function looks as follows:

$$f(D) \quad = \quad \underbrace{argmax_{M \in \mathcal{M}}}_{decoding} \prod_{i=1}^{n} \underbrace{P(m_i|m_{i-1})P(d_i|m_i)}_{training}$$

Training : Estimation of model parameters

During the training phase, our task is to estimate our parameter values - the emission and transition probabilities. We assume a supervised training set which consists of examples that were annotated by human experts. As will be discussed at length in subsection 4.5.1, our seed is fairly small and in order to generalize from it, we have also created a larger set of synthetic examples that were generated by the rules of the CNL grammar.

Training Emission Parameters The parameter $P(d_i|m_i)$ represents the probability of a verbal description of a requirement given an SM snapshot which grounds the semantics of that description. A single SM may result from many different syntactic derivations. We estimate the probability by using all $t \in \mathcal{T}_{req}$ that their yield is a requirement d and their grounded semantics is m, divided by all possible trees that their grounded semantics is m.

$$P(d|m) \quad = \quad \frac{P(d,m)}{P(m)} \quad = \quad \frac{\sum_{t \in \{t|yield(t)=d \wedge m \triangleright sem(t)\}} P(t)}{\sum_{t \in \{t|t \in \mathcal{T}_{req} \wedge m \triangleright sem(t)\}} P(t)}$$

$P(t)$ is calculated by the CKY algorithm with the PCFG model that we described in the sentence-based model.

Estimating Transition Parameters The parameter $P(m_i|m_{i-1})$ represents the probability of seeing a SM snapshot in time i given the SM snapshot in time $i-1$. We look at the current and the previous system model, and aim to estimate how likely the current SM is given the previous one. There are different assumptions that may underly this probability distribution, reflecting different principles of human communication in general and software design in particular. We first define a generic estimator as follow :

$$\hat{P}(m_i|m_j) \quad = \quad \frac{gap(m_i|m_j)}{\sum_{m_k} gap(m_i|m_k)}$$

Transition:	$gap(m_i, m_j)$
max-overlap	$\frac{\lvert set(m_i) \cap set(m_j) \rvert}{\lvert set(m_i) \rvert}$
max-expansion	$1 - \frac{\lvert set(m_i) \cap set(m_j) \rvert}{\lvert set(m_i) \cup set(m_j) \rvert}$
min-distance	$1 - \frac{ted(m_i, m_j)}{\lvert set(m_i) \rvert + \lvert set(m_j) \rvert}$

TABLE 4.1: Quantifying the gap between SM snapshots, $set(m_i)$ is a set of nodes marked by path to root in the tree that represent the m_i, $ted(m_i, m_j)$ is a tree edit distance metric.

where $gap(m_i|m_j)$ captures the information sharing between SM snapshots. By definition, and with a constraint that the $gap(.)$ function is non-negative, it is easily shown that $\hat{P}$ is a conditional probability distribution where, $\hat{P} : \mathcal{M} \times \mathcal{M} \rightarrow [0, 1] \wedge \forall m_i : \sum_{m_j} \hat{P}(m_i|m_j) = 1$.
We define several $gap(.)$ implementations, reflecting different assumptions about how requirements document are written. Our first assumption here relies on the idea that different SM snapshots share the same context, that is, refer to the same conceptual world, so there should be a large overlap between them. We call this the **max-overlap** function. A second assumption relies on collaborative communication, a new requirement will only be stated if it provide a new, significant information about the system, akin to Grice, 1975. That is, every new requirement will try to expand the system, this is the **max-expansion** function. A third assumption prefers "easy" transitions over "hard" ones, meaning that the new information expands locally. We call this the **min-distance** function, and based it on *tree edit distance* metrics. The different gap calculations are listed in Table 4.1.

Decoding : Applying the model to translate requirements documents into code

The input to our decoding algorithm is a specification document D that contains n requirements , formally - $D = \{d_1, ..., d_n\}$. For every requirement d_i our algorithm derives and considers N-best syntactic trees , which are retrieved via a CKY chart parser. Each syntactic tree t_i implies a semantic SM snapshot, and some trees may have the same SM snapshots, so we combine all the tree for same SM snapshot and summing their probabilities. So, at each step $i = 1, ...n$ we assume at most N SM snapshots states which represent the semantics of the N best syntax trees.

Thus, setting $N = 1$ is equivalent to our sentence-based model, a greedy algorithm in which for each requirement we simply select the most likely tree according to a probabilistic grammar, and construct a semantic representation for it.

For each document with n requirements, we assume that the entire universe of the system models $\mathcal{M}$ is composed of $N \times n$ SM snapshots, reflecting the N most-likely analyses of n requirements, as derived by the probabilistic syntactic model. As shall be seen shortly, even with the restricted[1] universe approximating $\mathcal{M}$, our discourse-based model achieves substantial improvements over the sentence-based model.

[1]This restriction is akin to pseudo maximum-likelihood estimation, as in "Pseudolikelihood Estimation: Some Examples". In the pseudo likelihood-estimation, instead normalizing over the entire set of elements, one uses a subset that reflects only the possible outcomes.

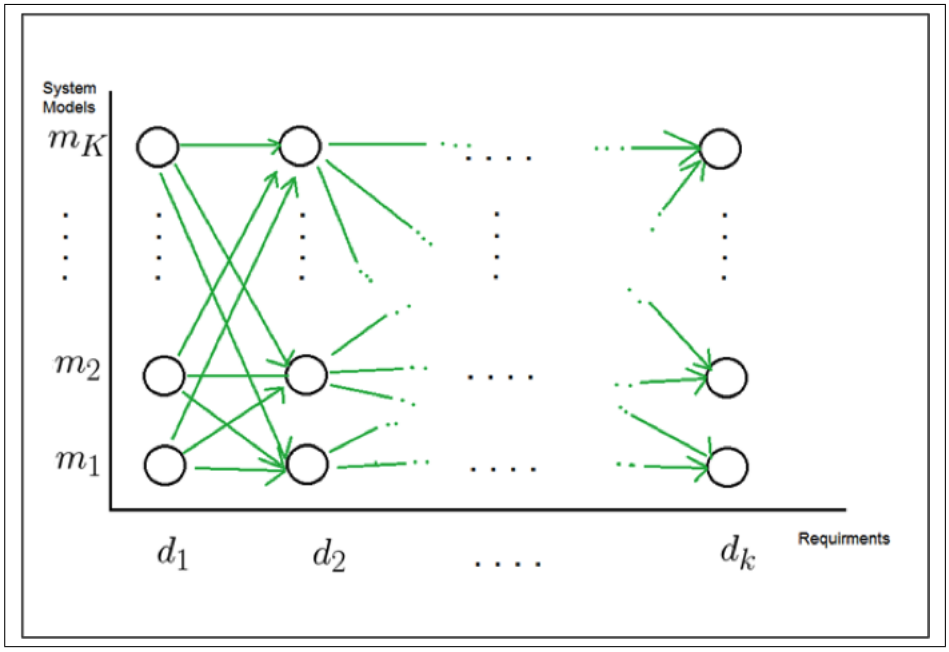


FIGURE 4.2: An illustration of the algorithm, on X axis we have the requirements and on Y axis we have the candidate system models. The circles represent the emission probabilities and the arrows represent the transition probabilities.

Our discourse-based model is a Hidden Markov Model (HMM) where every requirement is an observed signal, and each $i = 1, ..., N$ is a state representing the SM that grounds the i^{th} best syntactic tree. Because of the Markov independence assumption our step satisfies the *optimal subproblem* and *overlapping problem* properties, and we can use efficient *Viterbi* decoding (Viterbi, 1967) to exhaustively search through the different state sequences, and find the most probable sequence of SM snapshots that has generated the sequence of requirements according to our discourse-based probabilistic model. In figure 4.2 we see an illustration of the sequence selection phase. The X axis holds the requirements as ordered in the document and on the Y axis we have the candidate SM snapshots. The circles represent the emission probabilities, while the green arrows represent the transition probabilities and what we look for is the most likely sequence of green arrows leading connecting d_1 to d_n.

Time complexity

The overall complexity of decoding a document with n requirements, where the number of tokens of the longest requirement is l, using a grammar G of size $|G|$ and a fixed N (number of considered syntax trees per requirement), is given by:

$$O(n \times l^3 \times |G| + l^2 \times N^2 \times n + n^3 \times N^2)$$

We can break this expression into 3 parts as follows:

In our case, instead of summing SM probabilities over all possible sentences in the language, we sum up the SM analyses of the sentences observed in the document only.

1. $O(n \times l^3 \times |G|)$ - we generate N best trees for each one of the n requirements using CKY chart (Younger, 1967). (CKY worse case complexity is $O(l^3 \times |G|)$)

2. $O(l^2 \times N^2 \times n)$ - we create the universe $\mathcal{M}$ based on the N best trees of the n requirements, and calculate $N \times N$ transitions in every adjacent requirements d_i, d_{i+1} . The transition calculation complexity is $O(l^2)$.

3. $O(n^3 \times N^2) = O((N \times n)^2 \times n)$ - we decode the $n \times N$ grid using Viterbi decoding. (Viterbi complexity is $O(|S|^2 \times T)$ where S represents the possible states and T is the number of the observations — Viterbi, 1967.)

4.5 Empirical Evaluation

4.5.1 The Data

Every statistical supervised model requires annotated data that represent the gold (correct) analyses as assigned by human experts. Another important property of the data is that it is representative in order to be able to generalize from it and prevent over-fitting.

Our gold annotated data is a parallel corpus that at one side contains a CNL sentence and on the other side a correct parse tree based on the CNL rules.

We fortunately could benefit from a small seed of correctly annotated requirements-specification case studies that describe simple reactive systems in the LSC language. These case studies were created by software engineering graduate students and experts at the software engineering team at the Weizmann Institute.

Each document contains a sequence of requirements, each of which is annotated with the correct LSC diagram. The entire program is grounded in a java implementation. As training data, we use the case studies provided by Gordon and Harel (2009). Table 4.2 lists the case studies and basic statistics concerning these data (We list here only episodes with at least ten requirements).

As our annotated seed is quite small, it is hard to generalize from it to unseen examples. In particular, we are not guaranteed to have observed all possible structures that are theoretically permitted by the assumed grammar. To cope with this, we generated a synthetic set of examples by sampling the grammar of Gordon and Harel, 2009 in generation mode, and randomly generate trees $t \in \mathcal{T}_{req}$ using the derivation rules.

The grammar we use to generate the synthetic examples clearly overgenerates. That is to say, it creates many trees that do not have a sound interpretation. In fact, only 3000 our of 10000 generated examples turn out to have a sound semantic interpretation grounded in an SM. Nonetheless, these data allow us to smooth the syntactic distributions that are observed in the seed, and increase the coverage of the grammar learned from it.

This data set, the real-world seed along with the generated data , will be used to learn a probabilistic context-free grammar (PCFG) model in order to estimate probabilities of derivation trees for unseen CNL requirement sentences.

System	# of requirements	avg requirement length
Phone	21	23.33
Wrist Watch	15	29.8
Chess	18	15.83
Baby Monitor	14	20
Total	68	22.395

TABLE 4.2: Small seed of gold-annotated requirements documents

4.5.2 Experimental Setup

Goal: Our goal is to evaluate the accuracy of the semantic parser for requirements documents, and compare the two modes of analysis, sentence-based and discourse-based. Our evaluation methodology is as standardly assumed in supervised machine learning and NLP: given a set of annotated examples, in our case, given a set of requirements documents, we have for each requirement a gold representation of the LSC and a complete SM for the whole document. Formally we have a parallel corpus with the following pairs:

$$(D, \langle LSC, SM \rangle) = (\{d_1, ..., d_n\}, \langle \{lsc_1, ..., lsc_n\}, SM \rangle)$$

We partition this set into two disjoint sets - *training set* and *test set*. We train our statistical model on the examples in the training set and apply the learned model to the test set, and then automatically evaluate its performance. We compare the predicted semantic analyses of the test set with gold analyses, and empirically quantify the prediction accuracy.

Metrics: Our semantic LSC object has the form of a tree (reflecting the sequence of nested events in our scenarios). Therefore, we can use standard tree evaluation metrics, such as ParseEval (Black, Lafferty, and Roukos, 1992), to evaluate the accuracy of our prediction. Overall, we have defined three metrics to evaluate the accuracy of the LSC trees:

- **POS**: this metric capture the percentage of part-of-speech tags predicated correctly. (POS of the defined CNL Grammar).
- **LSC-F1**: F1, is a F-Measure metric with $\beta = 1$, which is a harmonic means of precision and recall of the predicted tree.
- **LSC-EM**: EM abbreviation stands for *exact match*, this metric is 1 if the predicted tree is an exact match of the gold tree, and 0 otherwise.

In the case of SM trees, as opposed to the LSC trees, we cannot assume identity of the yield between the gold and predicted trees for a same requirement sentence, so we cannot benefit from ParseEval. Therefore, we implemented a distance-based metrics in the spirit of Tsarfaty et al. (2012). Then, to evaluate the accuracy of the SM, we use two kinds of scores:

- **SM-TED**: Tree-edit distance (TED) is the normalized edit distance between the predicated and the gold SM trees.

- **SM-EM**: As in **LSC-EM**, this metric is 1 if the predicted SM is an exact match with the gold SM, and 0 otherwise.

Parameters: Our models consist of several hyper parameters to play with.

First, we would like to test the impact of the P-CFG models on the final results, for that we will create 3 different models - one based only on the synthetic data, the second based only on the real-world data, and the last one using real-world data together with the synthetic data.

Secondly, we will test how does **N**, the number of considered candidate syntactic trees, generated by CKY algorithm, impact the final results. An interesting observation is that when $N = 1$ the discourse-model is equivalent to the sentence-based model.

The last hyper parameter will be the **Transitions** we describe previously, we define 3 estimators — *max-overlap*, *max-expansion* and *min-distance*. We will test each of them separately and also interpolation between them.

4.5.3 Results and Analysis

In all our experiments we used the *Phone* episode as our development set in order to optimize our hyper-parameters. When then verified our models on the other episodes using the cross-fold validation methodology.

Real-world data improves substantially the results.
The goal of our first experiment was to check the impact of using a small seed of real-world data. Table 4.3 presents the results for parsing our four episodes using a leave-one-out method. The parameter that we change here is the training data for our PCFG model. We can see that despite the small size of the real-world seed, adding it to the synthetic examples substantially improves all our evaluation metrics relative to the model that has been trained only on the synthetic examples.

Another setup we examined is using as training data our annotated seed only, and parsed each one of the episodes. Interestingly, this setup outperformed the seed+synthetic examples setup. This result surprised at first. Investigating further, we have been able to show that this result is due to overfitting. Tables 4.4, 4.5 and 4.6 show that when we learn the grammar from a single episode and try to parse another single episode, the result improved when adding the synthetic rules. This is reasonable as part of the derivation structure cannot be observed and learn from a small and specific data set.

In our next experiment, we aim to determine empirical upper-bounds and lower-bounds for the discourse-based model. Table 4.7 presents the results of the discourse-based model for $N \geq 1$ on the $Phone$ episode. $Gen-Only$ presents the results of the discoursed-based model with a PCFG learned from synthetic trees only, incorporating transitions obeying the **max-overlap** assumption. Already here, we can see a mild improvement for $N > 1$ relative to the sentence-based model where $N = 1$, which indicates that even a weak signal such as the overlap between neighboring requirement sentences already improves sentence disambiguation in context. These results provide a lower bound on parser performance for each N.

We next present the results of the $Orcale$ experiment (Table 4.8), where every requirement is assigned the highest scoring tree in terms of LSC-F1

Scenarios	POS	LSC-F1	LSC-EM	SM-TED	SM-EM
Phone					
Gen-Only	89.04	86.38	9.52	85.12	9.52
Seed-Only	93.54	93.73	61.90	98.19	90.48
Gen+Seed	91.78	87.71	14.29	85.48	14.29
Baby Monitor					
Gen-Only	94.63	91.87	7.14	87.88	28.57
Seed-Only	85.57	90.84	42.86	93.51	71.43
Gen+Seed	94.63	91.26	7.14	87.88	28.57
Wrist Watch					
Gen-Only	44.19	58.54	10.00	70.19	10.00
Seed-Only	55.97	59.26	35.00	82.25	60.00
Gen+Seed	44.84	57.60	20.00	70.39	20.00
Chess Game					
Gen-Only	91.23	93.08	0.00	96.42	55.56
Seed-Only	100.0	99.31	94.44	100.0	100.0
Gen+Seed	92.63	95.79	5.56	95.35	61.11

TABLE 4.3: Sentence-based modeling: Cross-fold validation - comparing the gen+seed grammar vs only generated grammar.

Learning / Testing	Phone	Baby Monitor	Wrist Watch	Chess Game
Seed-Only				
Phone	/	48.66	36.99	100.0
Baby Monitor	69.47	/	43.37	61.05
Wrist Watch	50.10	61.74	/	76.49
Chess Game	49.12	44.63	21.44	/
Gen+Seed				
Phone	/	93.96	43.37	88.77
Baby Monitor	86.69	/	43.04	87.02
Wrist Watch	90.61	94.63	/	91.23
Chess Game	90.02	94.63	44.19	/

TABLE 4.4: Sentence-based modeling: in this setup, we learn the PCFG only on single episode and test against each of the other episodes separately. These results show that the seed-only method is limited and that additional generated data improves the results in most cases. (POS metric comparison)

Learning / Testing	Phone	Baby Monitor	Wrist Watch	Chess Game
Seed-Only				
Phone	/	68.68	54.36	98.62
Baby Monitor	78.39	/	58.54	76.88
Wrist Watch	66.84	75.94	/	86.27
Chess Game	68.03	65.19	38.70	/
Gen+Seed				
Phone	/	91.12	56.50	93.73
Baby Monitor	87.11	/	57.69	91.56
Wrist Watch	88.80	91.26	/	93.32
Chess Game	88.30	91.26	56.41	/

TABLE 4.5: Sentence-based modeling: in this setup, we learn the PCFG only on single episode and test against each of the other episodes separately. These results show that the seed-only method is limited and that additional generated data improves the results in most cases. (LSC-F1 metric comparison)

Learning / Testing	Phone	Baby Monitor	Wrist Watch	Chess Game
Seed-Only				
Phone	/	85.91	76.89	100.00
Baby Monitor	90.37	/	74.24	94.38
Wrist Watch	83.49	88.55	/	89.33
Chess Game	77.89	82.41	70.63	/
Gen+Seed				
Phone	/	87.39	71,09	95.25
Baby Monitor	84.71	/	70.79	96.42
Wrist Watch	85.26	87.88	/	96.42
Chess Game	85.25	87.88	70.19	/

TABLE 4.6: Sentence-based modeling: in this setup, we learn the PCFG only on single episode and test against each of the other episodes separately. These results show that the seed-only method is limited and that additional generated data improves the results in most cases. (SM-TED metric comparison)

	N=1	N=2	4	8	16	32	64	128
Gen-Only								
POS	89.04	89.24	89.04	90.22	91.00	91.19	**91.39**	91.39
LSC-F1	86.38	86.38	87.33	90.12	91.29	91.82	91.62	**91.90**
LSC-EM	9.52	9.52	19.05	**33.33**	33.33	33.33	33.33	33.33
SM-TED	85.12	85.19	86.98	88.78	91.58	92.86	92.29	**93.14**
SM-EM	9.52	9.52	19.05	33.33	33.33	38.10	**42.86**	38.10
Seed-Only								
POS	**93.54**	93.15	93.35	93.15	93.35	92.95	93.15	93.15
LSC-F1	**93.73**	93.59	93.66	93.59	93.72	93.16	92.73	92.86
LSC-EM	**61.90**	61.90	61.90	61.90	61.90	61.90	61.90	61.90
SM-TED	**98.19**	97.31	97.75	97.31	97.31	96.06	93.52	95.13
SM-EM	**90.48**	85.71	85.71	85.71	85.71	71.43	66.67	66.67
Gen+Seed								
POS	91.78	91.78	92.76	93.74	94.13	94.32	**94.91**	94.32
LSC-F1	87.71	87.69	89.15	90.92	91.47	91.87	**92.55**	92.35
LSC-EM	14.29	19.05	38.10	47.62	47.62	**52.38**	52.38	52.38
SM-TED	85.48	85.38	90.93	92.05	93.82	94.45	**95.73**	93.17
SM-EM	14.29	19.05	42.86	57.14	57.14	**61.90**	61.90	57.14

TABLE 4.7: Discourse-based modeling: Evaluation results on the $Phone$ episode, our development set. Gen-only selects the most probable tree, relying on synthetic examples only, providing the lower bound.

metrics form the N-best derived candidates by the CKY with respect to the gold tree, while keeping the same transitions estimator. Again, we can see that the results improve with higher N, indicating that the syntactic model alone does not provide optimal disambiguation. These results provide an upper bound on the parser performance for each N.

$Gen + Seed$ presents the results of the discourse-based model where the PCFG interpolates the seed and the synthetic train set together, with **max-overlap** transitions. In this setup, we can see larger improvements over the $Gen - Only$ PCFG. That is, modeling grammaticality of individual requirement sentences improves the interpretation of the document.

Here we can see another evidence that using only *Seed* data is not enough, and even though the result of the *Seed-only* models are much higher, the increase of N and usage of the context does not improves that model and even reduce the accuracy, this further supports our conjecture that when using only seed data severe overfitting takes place.

Transition functions comparison

The next experiment aims to test the impact and performance of the different implementations of the $gap(m_i, m_j)$ functions, that reflect the transition probabilities in our model. We have estimated probability distributions that reflect each of the assumptions previously discussed, and also added an additional method called **hybrid**, in which we interpolate the **max-overlap** and **max-expansion** methods. Formally, the hybrid estimator is :

Gen+Seed								
POS	91.78	91.78	92.76	93.74	94.13	94.32	**94.91**	94.32
LSC-F1	87.71	87.69	89.15	90.92	91.47	91.87	**92.55**	92.35
LSC-EM	14.29	19.05	38.10	47.62	47.62	**52.38**	52.38	52.38
SM-TED	85.48	85.38	90.93	92.05	93.82	94.45	**95.73**	93.17
SM-EM	14.29	19.05	42.86	57.14	57.14	**61.90**	61.90	57.14
Oracle								
POS	N/A	91.98	93.54	94.91	95.30	96.09	96.67	**96.87**
LSC-F1	N/A	88.73	91.33	93.19	94.39	95.11	95.91	**96.70**
LSC-EM	N/A	23.81	42.86	61.90	61.90	66.67	76.19	**76.19**
SM-TED	N/A	86.54	91.28	94.28	94.88	96.24	97.51	**98.80**
SM-EM	N/A	23.81	42.86	66.67	71.43	**76.19**	76.19	76.19

TABLE 4.8: Discourse-based modeling: Evaluation results on the *Phone* episode, our development set. The Oracle selects the highest scoring LSC tree among the N-best candidates using gen+seed PCFG model, providing an upper bound.

$$\begin{aligned} hybrid(m_i, m_j) =& w_1 * maxOverlap(m_i, m_j) + w_2 * maxExpansion(m_i, m_j) \\ =& w_1 * \frac{|set(m_i) \cap set(m_j)|}{|set(m_i)|} + w_2 * (1 - \frac{|set(m_i) \cap set(m_j)|}{|set(m_i) \cup set(m_j)|}) \end{aligned}$$

We tested different approaches for w_1 and w_2 and the one that presents the best results was $w_1 = w_2 = 0.5$. In table 4.9, we can see that the trend from the previous experiment persists, meaning, as we increase N our results improve. Notably, the **hybrid** model provides a larger error reduction than each of its components (i.e., max-overlap and max-expansion) used separately. This phenomenon indicates that in order to capture the discourse context we may need to balance various, possibly conflicting, factors.

Another interpolated gap function that we tried is called **greedy**. **Greedy** method is also an interpolation of the **max-overlap** and **max-expansion** methods. But compared to the **hybrid** method, **greedy** selects the maximum score between **max-overlap** and **max-expansion**. Formally :

$$\begin{aligned} greedy(m_i, m_j) =& \max(maxOverlap(m_i, m_j), maxExpansion(m_i, m_j)) \\ =& \max(\frac{|set(m_i) \cap set(m_j)|}{|set(m_i)|}, (1 - \frac{|set(m_i) \cap set(m_j)|}{|set(m_i) \cup set(m_j)|})) \end{aligned}$$

This method outperformed all previous methods, including **hybrid**. These results also support that we may need to balance possibly conflicting factors in order to capture the discourse context.

We further added a control setup, called **no emissions**, in this case, we rely solely on the probability of the state transitions , and again increasing N leads to improvement (see results it Table 4.10). This result confirms that *context* is indispensable for sentence interpretation even when probabilities for sentence's semantics (a.k.a. *content*) are entirely absent. Another

control setup, we called **no transitions**, in this setup, we eliminate the transition probabilities, and consider the emission probabilities only. Our experiments showed that the results improved when we increase N, as we aggregate the syntactic trees that implies the same semantic together, we see that the correct semantic is rising up.

Next, we preform a cross-fold experiment, in which we leave one real-world episode (document) out as a test set, and use the rest as our seed to generate a P-CFG model (i.e., we learn for every episode e a $Gen + Seed_{-e}$ grammar). The results of this experiment are provided in Table 4.11. The discourse-based model outperforms the sentence-based model $N = 1$ in all cases ($N > 1$). Moreover, the drop in $N = 128$ for $Phone$ seems incidental to this set and the other cases level off beforehand.

Despite our small seed, the persistent improvement on all metrics is consistent with our hypothesis that modeling the interpretation process within the discourse has substantial benefits for automatic understanding of the text, than modeling sentence by sentence interpretation.

The significant of sentences order

Finally, we tested the impact of the requirements order. Formally, given a requirements document $D = d_1, d_2, ..., d_n$, does the requirements order affect the parsing results? To test this, we shuffled the individual requirements in each of our episodes, and repeated the parsing experiments.

Surprisingly, regardless of the requirements order, we observe the similar results and the same trends, i.e., we observe significant improvements as the number of candidates increase (see results in table 4.12). It seems that this result is related to CNL episodes and the way they were created, namely, each requirement is independent. We do believe that in real-world requirements documents, the order will be important as humans intend to rely on previous information.

So, if the order of the sentences does not seem to matter, why do we observe improvements when allowing more candidates and when using a global inference procedure? This is because the requirements document as a whole describes a complete system, and refers to the same elements over and over again. An algorithm that relies on more information (i.e., context) provides a better disambiguation capacity when it optimizes a global function (document level) and not a local function (sentence level) that takes into account overlaps in the document and easy transition between requirements.

4.6 Conclusion

The requirements understanding task presents an exciting challenge for CL/NLP. We ought to automatically discover the entities in the discourse, the actions they take, conditions, temporal constraints, and execution modalities. Furthermore, it requires us to extract a single code-base ontology that satisfies all individual requirements.

The contributions of this part are three-fold: we formalize the text-to-code prediction task where the input is a requirements document and the output is an executable code. We propose a semantic representation with well-defined grounding by means of LSC/SM formalism. And we show

Transitions	N=2	4	8	16	32	64	128
Min Dist							
POS	90.61	91.98	92.76	92.95	93.54	94.13	**94.52**
LSC-F1	88.80	90.65	91.50	92.45	92.36	93.37	**93.72**
LSC-EM	14.29	42.86	**52.38**	47.62	52.38	52.38	52.38
SM-TED	85.26	90.50	90.76	93.72	92.37	94.83	**95.59**
SM-EM	14.29	42.86	52.38	47.62	52.38	**61.90**	61.90
Max Overlap							
POS	90.61	92.37	92.56	92.95	93.35	93.93	**94.32**
LSC-F1	88.80	91.11	91.43	92.45	92.29	93.30	**93.65**
LSC-EM	14.29	**47.62**	47.62	42.86	47.62	47.62	47.62
SM-TED	85.26	90.69	90.08	93.51	91.72	94.21	**94.99**
SM-EM	14.29	47.62	47.62	42.86	47.62	**57.14**	57.14
Max Expansion							
POS	90.61	91.59	91.98	92.17	92.76	93.35	**93.74**
LSC-F1	88.80	90.18	90.65	91.60	91.50	92.52	**92.88**
LSC-EM	14.29	38.10	**42.86**	38.10	42.86	42.86	42.86
SM-TED	85.26	90.31	90.50	93.46	92.11	94.58	**95.34**
SM-EM	14.29	38.10	42.86	38.10	42.86	**52.38**	52.38
Hybrid							
POS	90.61	91.59	92.17	92.37	92.95	93.54	**93.74**
LSC-F1	88.80	90.30	91.04	91.99	91.90	92.91	**93.00**
LSC-EM	14.29	38.10	**42.86**	38.10	42.86	42.86	42.86
SM-TED	85.26	89.85	90.01	93.02	91.65	94.15	**94.94**
SM-EM	14.29	38.10	42.86	38.10	42.86	**52.38**	52.38
Greedy							
POS	90.61	92.37	93.15	93.35	93.93	94.52	**94.91**
LSC-F1	88.80	91.11	91.97	92.91	92.82	93.83	**94.18**
LSC-EM	14.29	47.62	**57.14**	52.38	57.14	57.14	57.14
SM-TED	85.26	90.69	90.95	93.91	92.56	95.02	**95.77**
SM-EM	14.29	47.62	57.14	52.38	57.14	**66.67**	66.67
No transition							
POS	90.61	92.37	92.76	92.95	93.54	93.74	**93.93**
LSC-F1	88.80	91.11	91.57	92.51	92.42	93.04	**93.20**
LSC-EM	14.29	47.62	**52.38**	47.62	52.38	47.62	42.86
SM-TED	85.26	90.69	90.88	93.50	92.15	94.56	**95.13**
SM-EM	14.29	47.62	52.38	47.62	52.38	**57.14**	52.38
No Emissions							
POS	91.78	91.98	92.37	92.37	92.17	92.76	**93.15**
LSC-F1	88.11	88.79	89.12	89.12	89.39	89.67	**89.89**
LSC-EM	19.05	19.05	**23.81**	23.81	23.81	23.81	23.81
SM-TED	85.49	85.74	85.82	85.82	85.87	86.85	**86.92**
SM-EM	19.05	19.05	**23.81**	23.81	23.81	23.81	23.81

TABLE 4.9: Discourse-based modeling: Experiments on the *Phone* development set. Comparison of all metrics for different transition estimators. All experiments use the Gen+Seed P-CFG for emission probabilities

Transitions	N=2	4	8	16	32	64	128
Min Dist							
POS	90.61	91.78	92.17	92.76	93.54	93.35	**93.74**
LSC-F1	88.80	90.37	90.77	91.59	92.57	92.29	**92.78**
LSC-EM	14.29	38.10	42.86	47.62	**52.38**	52.38	52.38
SM-TED	85.26	89.53	89.60	92.47	92.63	91.33	**93.49**
SM-EM	14.29	38.10	42.86	47.62	**52.38**	52.38	52.38
Max Overlap							
POS	90.61	91.39	91.39	91.59	91.59	**91.78**	91.78
LSC-F1	88.80	89.74	90.00	90.41	90.45	**90.73**	90.73
LSC-EM	14.29	**23.81**	19.05	19.05	19.05	19.05	19.05
SM-TED	85.26	**85.64**	84.64	84.33	83.68	83.64	83.64
SM-EM	14.29	**23.81**	19.05	19.05	19.05	19.05	19.05
Max Expansion							
POS	90.61	91.59	91.39	91.78	91.59	92.56	**92.76**
LSC-F1	88.80	90.18	90.11	90.66	90.85	**91.26**	91.18
LSC-EM	14.29	**38.10**	38.10	33.33	33.33	38.10	33.33
SM-TED	85.26	90.31	89.89	**92.44**	91.55	91.98	92.26
SM-EM	14.29	38.10	38.10	33.33	33.33	**42.86**	33.33
Hybrid							
POS	90.66	90.87	91.29	91.29	91.19	91.39	**91.59**
LSC-F1	89.01	89.43	90.16	**90.52**	90.26	90.14	90.36
LSC-EM	15.00	**20.00**	20.00	20.00	19.05	19.05	19.05
SM-TED	85.26	85.81	85.63	**86.78**	86.16	85.06	86.03
SM-EM	14.29	**19.05**	19.05	19.05	19.05	19.05	19.05
Greedy							
POS	90.61	91.78	91.98	92.56	92.56	92.37	**92.76**
LSC-F1	88.80	90.29	90.67	91.82	92.03	92.03	**92.30**
LSC-EM	14.29	33.33	38.10	38.10	**42.86**	38.10	42.86
SM-TED	85.26	87.72	87.12	90.16	**90.70**	89.18	89.78
SM-EM	14.29	33.33	38.10	38.10	**42.86**	38.10	42.86

TABLE 4.10: Discourse-based modeling: Experiments on the *Phone* development set. Comparison of all metrics for different transition estimators. All didn't take into account emission probabilities - based only on transitions

Data Set	N=2	4	8	16	32	64	128
Baby Monitor							
POS	94.63	95.64	96.31	96.64	**96.98**	96.98	96.98
LSC-F1	91.26	93.27	93.98	94.56	**95.14**	95.14	95.14
LSC-EM	7.14	14.29	**21.43**	21.43	21.43	21.43	21.43
SM-TED	87.88	91.79	92.59	92.11	**93.53**	93.53	93.53
SM-EM	28.57	50.00	57.14	57.14	**64.29**	64.29	64.29
Chess							
POS	91.23	91.23	92.63	**93.33**	93.33	93.33	93.33
LSC-F1	93.32	93.54	95.22	**95.46**	95.46	95.46	95.46
LSC-EM	0.00	0.00	**5.56**	5.56	5.56	5.56	5.56
SM-TED	96.42	95.77	95.53	**96.61**	96.61	96.61	96.61
SM-EM	55.56	55.56	**61.11**	61.11	61.11	61.11	61.11
Phone							
POS	91.78	92.76	93.74	94.13	94.32	**94.91**	94.32
LSC-F1	87.69	89.15	90.92	91.47	91.87	**92.55**	92.35
LSC-EM	19.05	38.10	47.62	47.62	**52.38**	52.38	52.38
SM-TED	85.38	90.93	92.05	93.82	94.45	**95.73**	93.17
SM-EM	19.05	42.86	57.14	57.14	**61.90**	61.90	57.14
Wrist Watch							
POS	43.86	44.35	44.84	44.84	**45.34**	45.34	45.34
LSC-F1	57.73	58.88	59.62	59.89	60.75	**61.02**	61.02
LSC-EM	5.00	20.00	**30.00**	30.00	30.00	30.00	30.00
SM-TED	71.39	74.14	74.88	74.88	77.33	**78.28**	78.28
SM-EM	10.00	20.00	30.00	30.00	35.00	**40.00**	40.00

TABLE 4.11: Discourse-based modeling: Impact of requirements order in the document - N=1,..,128. Gen+Seed for emissions, Greedy estimator for transitions

Data Set	N=1	2	4	8	16	32	64	128
Phone - Ordered								
POS	91.78	91.78	92.76	93.74	94.13	94.32	**94.91**	94.32
LSC-F1	87.71	87.69	89.15	90.92	91.47	91.87	**92.55**	92.35
LSC-EM	14.29	19.05	38.10	47.62	47.62	**52.38**	52.38	52.38
SM-TED	85.48	85.38	90.93	92.05	93.82	94.45	**95.73**	93.17
SM-EM	14.29	19.05	42.86	57.14	57.14	**61.90**	61.90	57.14
Phone - random 1								
POS	91.78	91.78	92.76	93.74	94.13	94.32	**94.91**	94.32
LSC-F1	87.71	87.69	89.15	90.92	91.47	91.87	**92.55**	92.35
LSC-EM	14.29	19.05	38.10	47.62	47.62	**52.38**	52.38	52.38
SM-TED	85.48	85.38	90.93	92.05	93.82	94.45	**95.73**	93.17
SM-EM	14.29	19.05	42.86	57.14	57.14	**61.90**	61.90	57.14
Phone - random 2								
POS	91.78	91.98	92.95	93.93	94.13	94.52	**94.91**	94.52
LSC-F1	87.71	87.97	89.43	91.20	91.47	92.15	92.55	**92.62**
LSC-EM	14.29	23.81	42.86	52.38	47.62	**57.14**	52.38	57.14
SM-TED	85.48	86.42	91.88	92.99	93.82	95.34	**95.73**	94.07
SM-EM	14.29	23.81	47.62	61.90	57.14	**66.67**	61.90	61.90

TABLE 4.12: Discourse-based modeling: Cross-Fold Validation for N=1,..,128. Gen+Seed for emissions, Greedy estimator for transitions

consistent improvement of the discourse-based over sentence-based models, in all case studies.

Chapter 5

Programming in Natural Language

Having lifted the *manual, sentence-by-sentence* restriction on Play-In, we are ready to tackle the greater question: can we lift the restriction of using a *controlled* fragment of NL? That is, can we automatically interpret requirements documents written in reasonably rich and natural English? Here, we propose an implementation that can accept a requirements document written in English as input and provide its LSC/SM representation as output. We evaluate our output using evaluation method that we have developed called *semnaticTED* that compares that output SM with an expert-annotated SM, and we show that the agreement between the models almost reaches the agreement when comparing two human written SMs.

5.1 Data

In every supervised machine learning model, the most critical step is obtaining a collection of annotated data, that will be used for learning and evaluation phases. Our departure point is the data set of Roth et al., 2014, which contains 324 syntactically parsed and semantic role-labeled requirements.

As one of our main insights from the previous chapter was that context indeed matters (i.e., discourse-based models outperform sentence-based models) , we divided this data into disjoint systems and created 25 distinct documents with an average of 12 requirements per system. A requirement has an average of 12.53 tokens and the median of 12 tokens.

Every requirement contains at least one verb and at least one explicit entity (that is, we assume at least a subject-predicate pair). In average each requirement contains 1.32 actions (verbs), and 1.31 entities.

We devise an algorithm that aims to pair this corpus of NL requirements with aligned LSC/SM representations. Our target is to make our outputs as close as possible to the correct LSC/SM representations, meaning, minimal manual post-editing.

This corpus is intended to facilitate the development of statistical models for translating unrestricted NL requirements directly into LSC/SM representations (and hence to executable code).

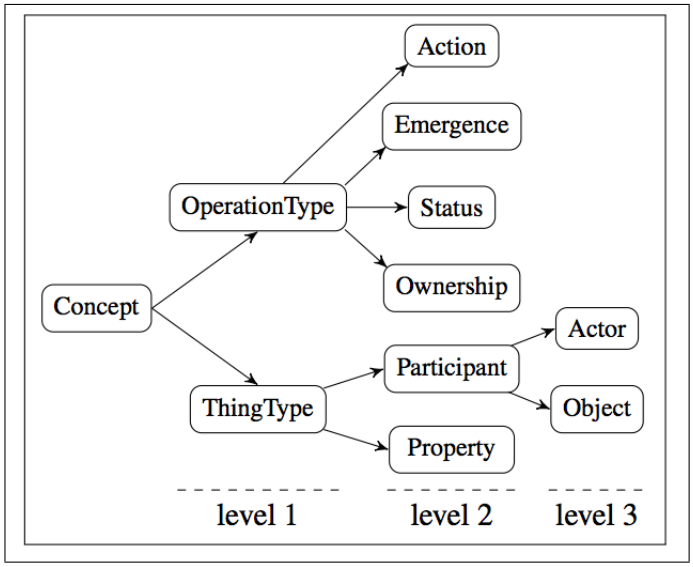


FIGURE 5.1: The class hierarchy of Michael Roth's conceptual ontology for modeling software requirements. (A duplication of figure 2.1).

5.2 The Parsing Algorithm

Our implementation makes use of the joint dependency parser and semantic-roles labeler (SRL) of Roth and Klein, 2015. The representation Roth's parser delivers is an extension of the CoNLL 2009 dependency format, which also identifies SRLs — the actors, objects, actions and properties[1] in each parse. Figure 5.1 shows the class hierarchy of the SRL. [2]

Our algorithm is built around three conceptual steps:

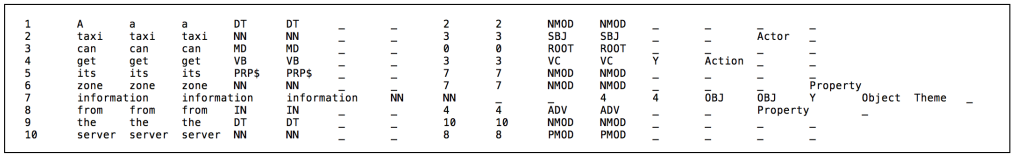
- Lifeline candidates extraction - focus on high recall, that is, not missing any class candidates.
- LSCs and SM creation - focus on precision, while using the candidates from previous step as a global context that aids local disambiguation .
- Cleanup phase - false (or un-needed) candidates are discarded.

Let us review these different steps in turn:

Step 0: Preprocessing We start our data processing by applying a batch of syntactic and semantic algorithms in order to enrich our NL requirements with linguistic constructs. We hereby list the tools and representations that we use:

- OpenNLP POS tagger - gives part-of-speech tags. Figure 5.3 presents an example of the POS tagger output.
- Semantic-Role-Labeler parser - the output provides semantic role labels that adheres to Roth et al., 2014. In addition, the output of the

[1] where 'properties' is a cover term for various sorts of verbal and nominal modification.
[2] The ontology of semantic roles that is provided by Roth is described in details in Roth et al., 2014.



1	A	a	a	DT	DT	_	_	2	2	NMOD	NMOD	_	_	_	_
2	taxi	taxi	taxi	NN	NN	_	_	3	3	SBJ	SBJ	_	_	Actor	_
3	can	can	can	MD	MD	_	_	0	0	ROOT	ROOT	_	_	_	_
4	get	get	get	VB	VB	_	_	3	3	VC	VC	Y	Action	_	_
5	its	its	its	PRP$	PRP$	_	_	7	7	NMOD	NMOD	_	_	_	_
6	zone	zone	zone	NN	NN	_	_	7	7	NMOD	NMOD	_	_	_	Property
7	information	information	information	NN	NN	_	_	4	4	OBJ	OBJ	Y	Object	Theme	_
8	from	from	from	IN	IN	_	_	4	4	ADV	ADV	_	_	Property	_
9	the	the	the	DT	DT	_	_	10	10	NMOD	NMOD	_	_	_	_
10	server	server	server	NN	NN	_	_	8	8	PMOD	PMOD	_	_	_	_

FIGURE 5.2: A SRL parser output for : "A taxi notifies the server of its location continuously.".

.

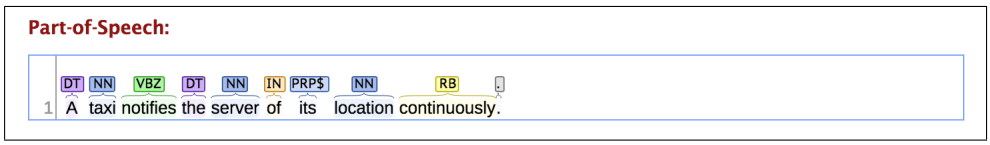


FIGURE 5.3: A POS tagger output for : "A taxi notifies the server of its location continuously.". Every token in the sentence is tagged with its part-of-speech tag.

parser provides internal features such as part-of-speech tags and joint dependency parser output. Figure 5.2 presents an example of the SRL parser output.

- Coreference Resolution by Stanford's CoreNLP algorithms (Raghunathan et al., 2010. Lee et al., 2011; Lee et al., 2013; Recasens, Marneffe, and Potts., 2013). This algorithm is responsible for finding all expressions that refer to the same entity in a sentence. Figure 5.4 presents an example of the coreference resolution output.

Step 1: Lifeline-candidates extraction In this step, based on the enrichment from the preprocessing step, we extract all possible class candidates (henceforth, *lifelines*) of the specified system.

In order to extract all possible classes, we iterate over the requirements. For each requirement, first, we use the following labels of the SRL parser — *Actor*, *Object* or *Theme* labels as potential candidates for lifelines.

Secondly, since the output of the semantic role labeler is noisy, we have added two fallback strategies for lifeline/class identification: We use dependency relations (i.e., subject-predicate-object relations) marked in the dependency tree and add all subjects and the objects as possible lifelines. We also use POS tags and mark NN or NN-sequences as optional lifelines.

The output for this step is a set of candidate lifelines for each requirement and an additional set of all possible unique lifelines, which would

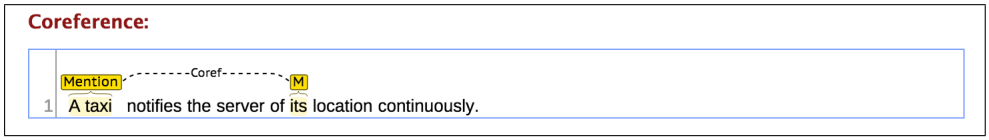


FIGURE 5.4: A co-reference analyzer output for : "A taxi notifies the server of its location continuously.". The algorithm connects phrases that point to the same entity, in this example "a taxi" and "its" refer to the same entity.

then become the classes in the system model (SM), and provide structural context for the entire analysis of the document.

In order to identify co-mentions of the same entities within this set, we use the co-references tags for grouping co-mentions of entities within every requirement.

Step 2: Analysis and LSC creation In this phase, we iterate over the requirements again. For every requirement we create a feature structure that captures the semantic information for creating the LSC representation for the requirement and incremental creation of the global SM.

- **Step 2.1: Feature Structure Generation** In this phase we map every requirement to a feature structure that captures the lifelines (objects), actions (methods) and themes (method arguments) which are required for executing the scenario. In figure 5.5 we illustrate the feature structure for the requirement: "A user must be able to create a user account by providing a username and a password.", where we have two possible lifelines, two actions and two themes, in every such artifact we hold the token, the semantic-role label and additional linguistic information (i.e., binding, modifier, modal). For each of these feature-structures we extract attribute:value pairs that add different dimensions of semantic interpretation, as we will specify in the next paragraph. The attributes are the fields in the feature structures. 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 information that is discovered to expand the global system model for the entire document.

  The attribute:value pairs we extract include the following information: semantic value (i.e., the reference of each attribute), the type of binding of each reference (i.e., should we use an existing instance or create a new one?), the role of each identified argument (i.e., should the argument be a modifying property or a theme?) , the modality of actions (i.e, 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 5.6). We created a linguistic ontology that implies ordering based on the modalities.

  We further use the co-reference component to determine the 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 feature structure and the corresponding LSC for the first requirement in Roth et al., 2014 are provided in Figures 5.5-5.6.

- **Step 2.2: Feature Structure to LSC**

  Having extracted the feature structures we are ready to create an LSC for each requirement.

  Lifelines creation
  For each requirement we initially add all possible lifelines based the

candidates set that we have initialized in step 1. For every possible lifeline, we retrieve from the feature structure and co-references component the properties for those classes , to expend the system model.[3]

Actions creation
Having created all possible lifelines for the scenario, we add to the LSC the interaction between them , i.e., the actions.

As time proceeds from top to bottom in LSC, this imposes a partial order on the execution of the actions. Therefore, we have to sort the action verbs before we insert them as methods. The sorting is done by understanding the dependencies between the actions, based linguistic (lexically-specified) rules on dependency parse relations. For example, in the requirement "A user must be able to create a user account by providing a username and a password.", we have two actions, *create* and *providing*. The *by* preposition helps us to determine that *providing* happens before *creating*.

Now, for each action in the ordered sequence we detect the sender, the receiver and the arguments based on feature structures and attribute:value pairs. We also detect the state (HOT/COLD, Monitored/Executed) based on modifiers and action orders, and add SYNC elements to impose a linear order between actions.

Triggers creation
As our data captures reactive systems, each requirement should have some trigger that causes the execution. The trigger might be some state of the system, action that happened, (e.g., a user clicks a button), periodically or initialization execution (e.g., at the beginning the color of the display is red), etc. Based on the content of the dependency relations and adverbial modifiers we determine if we already have an explicit trigger in a given requirement or need to create one (i.e., need to add clock trigger).

Step 3: Cleanup As a final step, for each created LSC, we go over all of its candidate lifelines and remove those which do not participate in any interaction i.e., no in-coming or out-coming message for this lifeline. By doing so, we separate the wheat from the chaff, keeping the important information of the document explicit in the semantic, and executable, representation of the system.

See a complete pseudo-code of the algorithm in appendix C.

5.3 Empirical Evaluation

The previous section described a baseline, rule-based algorithm for translating a requirements document written in natural language into the LSC/SM representation, that makes use of statistical NLP components. Each of these

[3]There is a subtlety here concerning an implicit"Controller" lifeline in the system. In "The user must be able to login" we have an explicit lifeline a *User* but also an implicit one that we call a *Controller* through which the login is executed. We infer the name for this "Controller" based on the existence of an explicit *system* object in other requirements in the document.

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

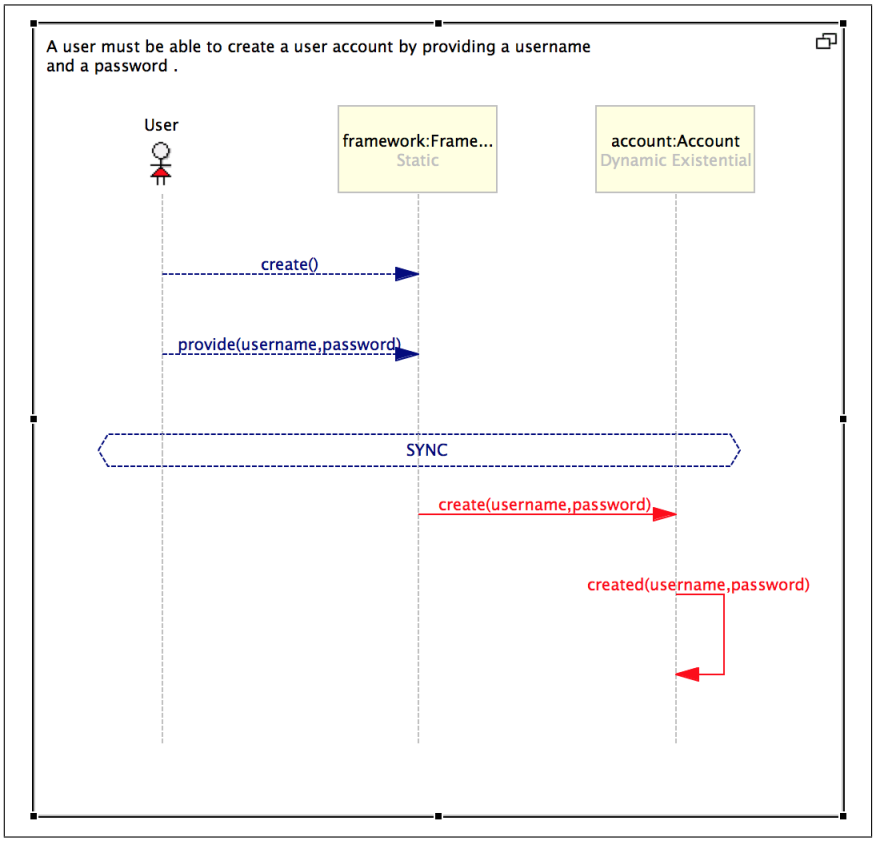


FIGURE 5.6: An LSC scenario for the requirement: "A user must be able to create a user account by providing a username and a password."

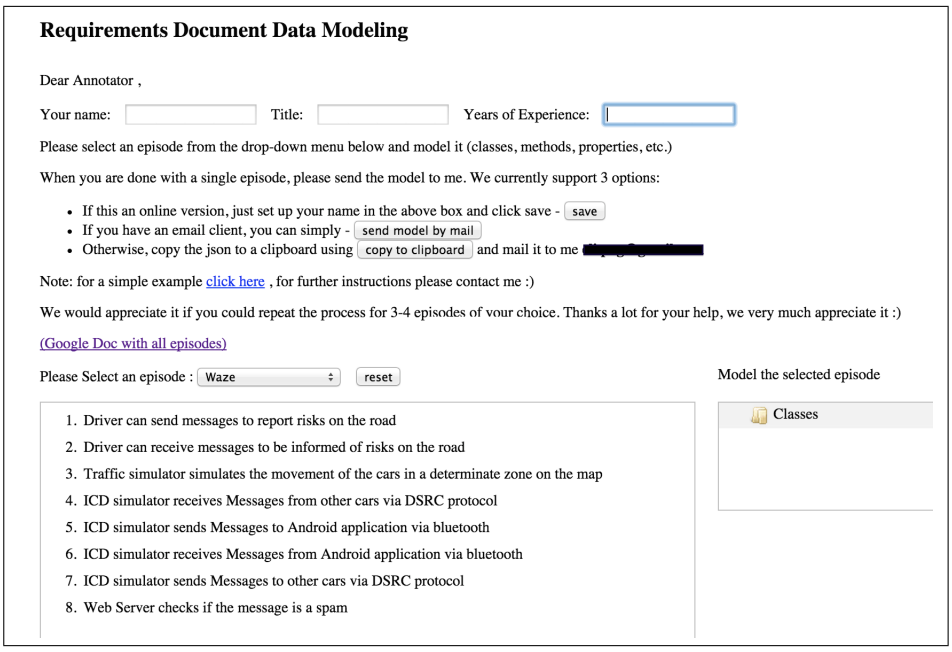


FIGURE 5.7: The online annotation platform for modeling requirements

artifacts require a different evaluation method. In this section we will focus on evaluating the accuracy of the global SM predicted for the system.

In the CNL-to-LSC/SM (Chapter 4) we used a simple *tree edit distance* (TED) metric to evaluate the SM (every SM can be represented as tree; see figure 4.1). Simple TED requires two conditions - order of leafs and strict comparison of leads labels. In chapter 4 we parsed requirements based on a controlled language, so the simple TED metric satisfied our needs. Here, we aim to parse unrestricted natural language, and simple TED is no longer an option. Our predicted systems are compared to systems from human experts, and we cannot force the order or usage of the same token during their modeling process. So, we adjust the TED metric to our NL programming task.

Note that the SM tree (Figure 4.1) contains 2 subtrees: the static object models (Classes) and the instantiation of those classes (Objects). The nodes in the trees are of two different types. The first type contains *frozen* structural/functional labels — *Classes, Methods ,Properties* — let's call them functional nodes. The second type holds the information content - classes names , methods return type , methods names, methods arguments, classes properties, etc — let's call them content nodes.

We define TED-based evaluation that removes the order constraint. Our objective function is to maximize the similarity of the compared SM trees, by minimized TED in every sub-tree comparison. Our edit operations compare apples to apples, using the functional nodes as anchors. This means that our TED operations compare classes to classes, properties to properties, and so on.

Our metric is specified as follows, where *SemanticTED(Tree1,Tree2)* contains the Tree edit distance between the compared SMs. We used functional nodes to compare sub-trees, meaning, given two functional nodes of the same type, we compared all children sub trees (a quadratic number of comparisons), and match them with respect to our objective function - maximize the similarity (minimized TED). The distance is normalized to provide a score between 0 and 1 by dividing on the size of the SM trees.

$$Score = 1 - \frac{SemanticTED(Tree1, Tree2)}{|Tree_1| + |Tree_2|}$$

5.3.1 Experiments

We aim to evaluate the accuracy of the predicted SM of the system, described in the input requirements document. We set out to provide the baseline results that would encourage more researchers to join the text-to-code effort.

To achieve this, we selected 9 requirements documents (aka *episodes*), with average length of 10 requirements each, to be modeled by experts. We intendedly selected episodes that contain varied logical expressions (if statements, loops, existence and universal bindings and more). We developed an online annotation platform (as depicted in figure 5.7) which we sent out to human expert annotators (in our case, graduate students who work as software engineers). Each annotator can pick a requirements document from the pre-selected set and model it by determining the classes, methods, properties and so on.

As our algorithm depends on the joint dependency parser and semantic-roles labeler (SRL) of Roth and Klein, 2015, and our data is also taken from same work, we re-trained the SRL models with "leave-one-out" approach. That is, when we parse episode X, we trained a new SRL model with all data except for episode X, by doing this we reassure that we do not overfit our training data.

5.3.2 Results and Analysis

For every episode we got at least two outside human annotators, and a control human annotation done by us. In table 5.3 we see the scores of comparison between our algorithm and the manual annotators models according to our *semanticTED* metric. As expected, our controlled annotation achieves the highest scores — this is expected as the algorithm has been designed by us, so it reflects the way we tend to conceptualize model requirements.

To obtain an upper bound, we applied our algorithm to gold SRL annotation provided by Roth et al. (Table 5.2). As expected, those scores are higher than the ones using the statistical SRL parser, as this setup enjoys error-free SRL annotations. Another insight that comes from the table is that our scores decrease as the number of requirements per document increase. This is reasonable, as every additional information (requirement) adds new structure and increases the error space.

We applied our evaluation method to compare between the expert models. In table 5.1 we compared 4 annotators (including our own human annotation) on the Taxi episode. The agreement scores are a bit higher compared to the previous scores, as expected – we get higher agreement between humans, than between the human and the machine. Means, the agreement between the auto-generated models and the expert models is almost the same as the agreement between two experts. Interestingly though, the results are far from perfect. When we analyzed the results qualitatively we observe that “gold” models for a given requirements set vary widely between any two experienced developers.

We conducted a qualitative error analysis to cluster the type of differences between the algorithm and the various human annotators. The most common error types are when comparing auto-generated models and human expert models:

- *Name Conventions:* there are conventions and best practices guiding the modeler in how to call a class/method/member. Our method does not capture small difference in names, for example. “change" vs “changeStatus", “Zone" vs “ZoneInformation".

- *Synonyms & Prior:* developers are using some world knowledge that does not only depend on the data itself. It is sometimes reflected as synonyms of composites - for example “location” vs “coordinates”.

- *Design Patterns:* developers often apply known design patterns and the algorithm doesn’t. For example, our annotators applied Observable Pattern, Producer-consumer pattern for *Restmarks* episode.

- *Object Handling:* Our algorithm handle every extracted Class as a new object that extends java.object while human annotator use known Object types - Strings, Maps, Lists and more.

Scores Comparison				
Annotator / Episode	1	2	3	Control
Taxi (6)	64.31	66.31	57.48	**68.13**
Game (6)	60.13	**75.84**	62.77	63.03
Restmarks (13)	41.44	/	39.67	48.8
Social (17)	/	62.98	45.66	52.91

TABLE 5.1: In this table we compare 3 external annotators and our control annotation against the algorithm. The number in the brackets in the first column indicates the amount of requirements of each episode, the number inside each cell indicate the *semanticTED* score.

- *Logical Entities vs Physical Entities:* The notation of user in the LSC representation is a physical entity, while sometimes a logical entity that represents the user is also needed.
- *Implicit controller:* who owns a given method? For example, in "The taxi notifies its location continuously to the server", does "notify" belong to the Taxi or to the Server? We notice in our data that in most cases the document assumes an implicit controller that "coordinates" the activities, so our algorithm creates one if such controller does not exist explicitly in the requirements document.

Beyond these observed differences, our experimental results raise a more foundational question, namely, is there a correct model for a given desired system? Based on the variance of the created models by different experts, it seems that the answer is no. We have seen that even experienced software engineers think differently about modeling and each one of them has his or her own style of software design.

Based on these differences, another question has been raised: does our algorithm have to target the perfect model? As industry nowadays shifts towards working in small development cycles and focus on fast deliveries. This paradigm is known as CI/CD (stands for continuous integration / continuous deployment). This leads us to batch of open research questions: Is the output of our algorithm a better starting point than starting with nothing at all? Can our auto-generated models be utilized for SW bootstrapping? How much time could be saved in practice using such computational SW design methodology?

5.4 Conclusion

We present an end-to-end system for NL programming based on statistical syntactic and semantic parsing components. We showcased the NL programming capacity for reasonably rich, uncontrolled, NL requirements data. We developed an evaluation method that extends the TED method, and observed that software *gold* modeling is, for the most part, subjective. In the future we intend to develop advanced models for statistical natural language programming, as well as more sophisticated evaluation methods

Gold SRL Scores Comparison				
Annotator / Episode	1	2	3	Control
Taxi (6)	58.06	74.15	69.11	65.31
Game (6)	48.55	68.18	44.58	56.21
Restmarks (13)	41.44	/	39.67	48.8
Social (17)	/	62.98	45.66	52.91

TABLE 5.2: Annotators scores across episodes: gold semantic roles

Scores Comparison between Annotators - Taxi				
Annotators	1	2	3	Control
1	/	66.46	70.0	64.1
2	66.46	/	77.47	75.49
3	70.0	77.47	/	77.96
Control	64.1	75.49	77.96	/

TABLE 5.3: Score comparison between manual annotators for a single episode

for the task, taking into account multiple reference annotations. We conjecture that the representation and tool we provide herein will prove valuable for future exploration of text-to-code translation.

Chapter 6

Conclusion and Future Research

6.1 Conclusion

The requirements understanding task presents an exciting challenge for NLP, a task that might make programming accessible to non-experts. In our proposed models, we see out to automatically discover the entities in the discourse, the actions they take, conditions, temporal constraints, and execution modalities. Furthermore, it requires us to extract a single ontology that satisfies all individual requirements in a document, forcing us to take into account discourse phenomena.

In the first part of this thesis (Chapter 4), we developed an automatic statistical parser for requirements document written in CNL, and manage to successfully parse whole documents, we have also integrated our solution in the *PlayGO* tool. The main contributions of the first part are (i) formalization the text-to-code predication task, (ii) proposal of a semantic representation with well-defined grounding, and (iii) empirical evaluation of our models for this prediction showing consistent improvement of discourse-based over sentence-based models, in all case studies.

In the second part (Chapter 5), we proposed a model for interpreting requirements in unrestricted English. We present an end-to-end system for NL programming based on statistical syntactic and semantic parsing components. We showcased the NL programming capacity for reasonably rich, uncontrolled, NL requirements data. We developed an evaluation method that extends the TED method, and observed that software *gold* modeling is, for the most part, subjective. The main contributions of this part are (i) an initial step for creating a parallel corpus for the text-to-code translation task and (ii) an evaluation method and platform for system modeling comparison.

In the future we intend to develop advanced models for statistical natural language programming, as well as more sophisticated evaluation methods for the task, taking into account multiple reference annotations. We conjecture that the representation and tool we provide herein will prove valuable for future exploration of the topic and development of interactive programming methodologies.

6.2 Future Extensions and Discussion

In this project, we first automated the translation of requirement documents written in CNL into LSC/SM representation. Than we introduced a first attempt to translate requirements documents written in unrestricted Natural Language into LSC/SM representation. We have also introduced an evaluation method to evaluate our intermediate representation. In the following paragraphs we will suggest several future directions in each of our endeavors.

As mentioned before, modeling is subjective, and even two experts might model the same requirements document completely differently. During our error analysis, we have noticed that some differences might be due to usage of synonyms or "close" terms to describe that same thing - for example, methods : change vs changeStatus, remove vs removeRestaurant, click vs press classes: ZoneInfo vs ZoneInformation). We suggest to add prior information concerning synonyms using WordNet and VerbNet knowledge bases. We also suggest to improve the node's content comparator to allow more "fuzzy" matching than restricted String.equals, that will match terms such as "change" vs "changeStatus".

We have looked at different approaches to evaluating our system. One of those was an existing tool for translating between models in software engineering called *Dozer* [1]. We noticed that these tools do not exploit textual information apart from trivial one (i.e., same name fields or code styling such as camel casing vs underscores - $firstName$ vs $first\_name$), and mostly focus on data types (i.e., primitives and basic object such as string). We are convinced that NLP approaches might improve such tools dramatically along the lines proposed here.

Another challenge that remains unsolved is the evaluation of the behavioral part of our models - The LSCs. We suggest to explore the use of crowd-sourcing techniques to ask which "image" captures more information of the requirement. But how to properly apply this evaluation process to our task is still an open question.

As mentioned before, this is a first attempt at translating unrestricted natural language requirements into SM/LSC representation. As our algorithm is integrated in *PlayGo* and the framework allows for post-edit operations, bootstrapping a parallel corpus is not only necessary but also possible.

In future text-to-code models, we suggest to use discriminative models and estimate directly the conditional probability $P(M|D)$. In order to learn new parameters, we suggest to use *Conditional Random Fields* (CRF) and Max Entropy methods. The decoding architecture can remain as in our CNL model (chapter 4), with small adaptations to support discriminative modeling. Use of discriminative models will allow us to integrate more sophisticated linguistic features and by this exploiting better local and global (context) phenomena.

Now, what about the answer to the question: can we lift the restriction of using a *controlled* fragment of NL? The question is still open, as of yet.

[1] Dozer supports simple property mapping, complex type mapping, bi-directional mapping, implicit-explicit mapping, as well as recursive mapping. This includes mapping collection attributes that also need mapping at the element level. - http://dozer.sourceforge.net/

However, in this research we did take a first step towards this goal. We introduced a first system that translates requirements documents written in real natural language into LSC/SM representation. We have also introduced an evaluation method for this task.

During the evaluation phase, our experimental results raise a more foundational question, namely, is there a correct model for a given desired system? Based on the variance of the created models by different experts, it seems that the answer is no. We have seen that even experienced software engineers think differently about modeling and each one of them has his or her own style of software design.

Based on these differences, another question has been raised: does our algorithm have to target the perfect model? As industry nowadays shifts towards working in small development cycles and focus on fast deliveries. This paradigm is known as CI/CD (stands for continuous integration / continuous deployment). This leads us to batch of open research questions: Is the output of our algorithm a better starting point than starting with nothing at all? Can our auto-generated models be utilized for SW bootstrapping? How much time could be saved in practice using such computational SW design methodology? We aim to investigate these questions, among others, in future cycles of our text-to-code research project.[2]

We are hopeful that this interesting challenge will be perused further within both the SE and NLP communities.

[2]This Text-to-Code project is kindly funded by a generous ERC Starting Grant awarded to the thesis supervisor, Dr. Reut Tsarfaty.

Appendix A

Full Specification of CNL CFG

We present here a full specification of CNL-Play context-free grammar defined by Gordon and Harel, 2009. In chapter 4 we implemented a statistical parser for this grammar.

The -> arrow indicates substitution in the course of derivation, and the {...} brackets defined the semantic interpretation function of the production rule.

```
S -> LSC {LSC.sem}
S -> MODEL {MODEL.sem}
LSC -> MAINPRECLAUSE {fCreateLsc(MAINPRECLAUSE.sem)}
LSC -> FORBID THEN CLAUSE {fCreateForbiddenLsc(CLAUSE.sem)}
LSC -> FORBID THEN MAINPRECLAUSE
       {fCreateForbiddenLsc(MAINPRECLAUSE.sem)}
MAINPRECLAUSE -> WHEN CLAUSE-COLD THEN CLAUSE-HOT
             {fAddPreMain(CLAUSE-COLD.sem, CLAUSE-HOT.sem)}
CLAUSE-COLD -> CLAUSE  {CLAUSE.sem}
CLAUSE-HOT -> CLAUSE   {CLAUSE.sem}
CLAUSE -> CLAUSE CONNECT CLAUSE
         {fAddCharts(0.CLAUSE.sem, CONNECT.sem, 2.CLAUSE.sem)}
CLAUSE -> COND-CLAUSE  {COND-CLAUSE.sem}
CLAUSE -> LOOP-CLAUSE  {LOOP-CLAUSE.sem}
CLAUSE -> MSG {MSG.sem}
CLAUSE -> PROP-CHANGE  {PROP-CHANGE.sem}
CLAUSE -> TIME-CHANGE {TIME-CHANGE.sem}
CLAUSE -> PROPBABILITY-CLAUSE {PROPBABILITY-CLAUSE.sem}
CLAUSE -> SAVE-VARIABLE {fAddToLsc(SAVE-VARIABLE.sem)}
CLAUSE -> LOAD-VARIABLE {fAddToLsc(LOAD-VARIABLE.sem)}
CLAUSE -> MAINPRECLAUSE {fAddToLsc(MAINPRECLAUSE.sem)}
CLAUSE -> ELSE-IF-CLAUSE {fAddToLsc(ELSE-IF-CLAUSE.sem)}
CLAUSE -> ELSE-CLAUSE {fAddToLsc(ELSE-CLAUSE.sem)}
CLAUSE -> EXPRESSION {fAddToLsc(EXPRESSION.sem)}
LOOP-CLAUSE -> WHILE EXPRESSION THEN CLAUSE
          {fCreateLoopWithCondition(EXPRESSION.sem, CLAUSE.sem)}
COND-CLAUSE -> IF-CLAUSE {IF-CLAUSE.sem}
COND-CLAUSE -> IF-CLAUSE ELSE-IF-CLAUSE
          {fAddCharts(IF-CLAUSE.sem, NULL.sem, ELSE-IF-CLAUSE.sem)}
COND-CLAUSE -> IF-CLAUSE ELSE-CLAUSE
          {fAddCharts(IF-CLAUSE.sem, NULL.sem, ELSE-CLAUSE.sem)}
COND-CLAUSE -> IF-CLAUSE ELSE-IF-CLAUSE ELSE-CLAUSE
          {fAdd3Charts
            (IF-CLAUSE.sem, ELSE-IF-CLAUSE.sem, ELSE-CLAUSE.sem)}
```

```
IF-CLAUSE -> IF EXPRESSION THEN CLAUSE
             {fCreateCondAddChart(EXPRESSION.sem, CLAUSE.sem)}
ELSE-IF-CLAUSE -> ELSE-IF EXPRESSION THEN CLAUSE
          {fCreateElseCondAddChart(EXPRESSION.sem, CLAUSE.sem)}
ELSE-IF-CLAUSE -> ELSE-IF-CLAUSE ELSE-IF-CLAUSE
      {fAddCharts
         (0.ELSE-IF-CLAUSE.sem, NULL.sem, 1.ELSE-IF-CLAUSE.sem)}
ELSE-CLAUSE -> [THEN] ELSE [THEN] CLAUSE
               {fCreateElseAddChart(CLAUSE.sem)}
MSG -> OP1 [TEMP] METHOD OP2 [PROP-VAL]
      {fCreateMessage
          (OP1.sem, OP2.sem, TEMP.sem, METHOD.sem, PROP-VAL.sem)}
MSG -> OP [TEMP] METHOD [PROP-VAL]
      {fCreateMessage
           (OP.sem, OP.sem, TEMP.sem, METHOD.sem, PROP-VAL.sem)}
MSG -> OP1 [TEMP] METHOD PROP-VAL PROPOSITION OP2
       {fCreateMessage
         (OP1.sem, OP2.sem, TEMP.sem, METHOD.sem, PROP-VAL.sem)}
MSG -> OP1 [TEMP] METHOD OP2 PROP-NAME
       {fCreateMessageWithPropArg
          (OP1.sem, OP2.sem, TEMP.sem, METHOD.sem, PROP-NAME.sem)}
MSG -> OP1 [TEMP] METHOD DET PROP-NAME OF OP2
       {fCreateMessageWithPropArg
          (OP1.sem, OP2.sem, TEMP.sem, METHOD.sem, PROP-NAME.sem)}
PROP-CHANGE -> OP1 [TEMP] SET-PROP OP2 PROP-NAME [PROP-VAL]
       {fCreatePropChange
         (OP1.sem, OP2.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem)}
PROP-CHANGE -> OP PROP-NAME [TEMP] SET-PROP [PROP-VAL]
       {fCreatePropChange
           (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem)}
PROP-CHANGE -> OP [TEMP] SET-PROP [PROP-VAL] PROP-NAME
       {fCreatePropChange
           (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem)}
PROP-CHANGE -> OP [TEMP] SET-PROP ITS PROP-NAME [PROP-VAL]
       {fCreatePropChange
           (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem)}
PROP-CHANGE -> OP1 PROP-NAME1 [TEMP] SET-PROP [PROPOSITION]
         [DET] OBJECT PROP-NAME2
         {fCreatePropChangeEx
          (OP1.sem, OP1.sem, TEMP.sem, PROP-NAME1.sem,
          OBJECT.sem, PROP-NAME2.sem)}
PROP-CHANGE -> OP1 PROP-NAME1 [TEMP] SET-PROP [PROPOSITION]
      [DET] OBJECT PROP-NAME2
      {fCreatePropChangeEx
       (OP1.sem, OP1.sem, TEMP.sem, PROP-NAME1.sem,
        OBJECT.sem, PROP-NAME2.sem)}
PROP-CHANGE -> OP1 [TEMP] SET-PROP-BY-VAL OP2 PROP-NAME PROP-VAL
     {fCreatePropChangeByVal
         (OP1.sem, OP2.sem, TEMP.sem, PROP-NAME.sem,
           PROP-VAL.sem, SET-PROP-BY-VAL.sem)}
```

```
PROP-CHANGE -> OP PROP-NAME [TEMP] SET-PROP-BY-VAL PROP-VAL
        {fCreatePropChangeByVal
            (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem,
             PROP-VAL.sem, SET-PROP-BY-VAL.sem)}

PROP-CHANGE -> OP [TEMP] SET-PROP-BY-VAL PROP-VAL PROP-NAME
    {fCreatePropChangeByVal
          (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem,
           SET-PROP-BY-VAL.sem)}

PROP-CHANGE -> OP [TEMP] SET-PROP-BY-VAL ITS PROP-NAME by PROP-VAL
    {fCreatePropChangeByVal
        (OP.sem, OP.sem, TEMP.sem, PROP-NAME.sem, PROP-VAL.sem,
         SET-PROP-BY-VAL.sem)}

EXPRESSION -> EXPRESSION AND EXPRESSION
   {fConcatExpressions(0.EXPRESSION.sem, 2.EXPRESSION.sem)}

EXPRESSION -> OP PROP-NAME COMPARE PROP-VAL
  {fCreateExpression
     (OP.sem, PROP-NAME.sem, NULL.sem, COMPARE.sem, NULL.sem, NULL.sem,

EXPRESSION -> OP PROP-NAME TEMP COMPARE PROP-VAL
  {fCreateExpression
     (OP.sem, PROP-NAME.sem, TEMP.sem, COMPARE.sem, NULL.sem, NULL.sem,

EXPRESSION -> OP PROP-NAME COMPARE OP PROP-NAME
 {fCreateExpression
    (0.OP.sem, 1.PROP-NAME.sem, NULL.sem, COMPARE.sem, 3.OP.sem,
     4.PROP-NAME.sem, NULL.sem)}

EXPRESSION -> OP PROP-NAME TEMP COMPARE OP PROP-NAME
 {fCreateExpression
   (0.OP.sem, 1.PROP-NAME.sem, TEMP.sem, COMPARE.sem, 4.OP.sem,
    5.PROP-NAME.sem, NULL.sem)}

EXPRESSION -> OP PROP-NAME COMPARE OBJECT
 {fCreateExpressionForObject
      (0.OP.sem, 1.PROP-NAME.sem, NULL.sem, COMPARE.sem, OBJECT.sem)}

EXPRESSION -> OP PROP-NAME TEMP COMPARE OBJECT
 {fCreateExpressionForObject
      (0.OP.sem, 1.PROP-NAME.sem, TEMP.sem, COMPARE.sem, OBJECT.sem)}

EXPRESSION -> TIME-CHANGE {TIME-CHANGE.sem}

PROPBABILITY-CLAUSE -> PROPBABILITY-COND CLAUSE
          {fCreateProbabilityCond(PROPBABILITY-COND.sem, CLAUSE.sem)}
PROPBABILITY-COND -> SOMETIMES {-1}
PROPBABILITY-COND -> NUMBER PERCENT OF THE TIME {NUMBER.sem}
FORBID -> THE FOLLOWING CAN NEVER HAPPEN
```

```
FORBID -> THE FOLLOWING IS FORBIDEN
NULL -> null
OP -> DET OBJECT {fCreateObject(DET.sem, OBJECT.sem)}
OP -> DET OBJECT WITH PROP-NAME PROP-VAL
 {fCreateObjectWithCond(DET.sem, OBJECT.sem, PROP-NAME.sem, PROP-VAL.sem)}
OP -> DET PROP-VAL PROP-NAME OBJECT
 {fCreateObjectWithCond(DET.sem, OBJECT.sem, PROP-NAME.sem, PROP-VAL.sem)}
OP -> DET-INDEFINITE CLASS-NAME
      {fCreateObject(DET-INDEFINITE.sem, CLASS.sem)}
OP1 -> OP {OP.sem}
OP2 -> OP {OP.sem}
OP-NO-DET -> OBJECT {fCreateObject(eInstance, OBJECT.sem)}
OP -> OP-NO-DET {OP-NO-DET.sem}
PROP-NAME1 -> PROP-NAME {PROP-NAME.sem}
PROP-NAME2 -> PROP-NAME {PROP-NAME.sem}
CONNECT -> THEN {eControlExit}
CONNECT -> AND AFTER THAT {eControlSync}
CONNECT -> AND ONLY THEN  {eControlSync}
CONNECT -> AND {eControlContinue}
CONNECT -> THEN AND {eControlContinue}
CONNECT -> UNLESS {eControlColdForbid}
CONNECT -> UNTIL {eControlColdForbid}
DET -> DET-INDEFINITE {DET-INDEFINITE.sem}
DET -> DET-DEFINITE  {DET-DEFINITE.sem}
DET-INDEFINITE -> a | an | any | all | some | other | another {eSymbolic}
DET-DEFINITE -> THE {eInstance}
TEMP -> TEMP-HOT {eTempHot}
TEMP -> TEMP-COLD {eTempCold}
TEMP -> TEMP-HOT-NOT {eTempHotNot}
TEMP -> TEMP-COLD-NOT {eTempColdNot}
TEMP-HOT -> MUST [EVENTUALLY] | EVENTUALLY | [EVENTUALLY] MUST
MUST -> must | shall | should | will
TEMP-COLD -> may | could | can | does
TEMP-COLD-NOT -> MUST NOT | CANNOT | TEMP-COLD NOT
TEMP-HOT-NOT -> TEMP-HOT NEVER | TEMP-COLD NEVER
SET-PROP -> turn | change | set | turns | changes | IS SET | sets
            {ePropSet}
SET-PROP-BY-VAL -> INCREASE [by] {ePropIncrease}
SET-PROP-BY-VAL -> DECREASE [by] {ePropDecrease}
PROPOSITION -> TO | from | by  | IN | of
WHEN -> when | whenever
THEN -> then | , | do {eControlExit}
ELSE -> else | [THEN] OTHERWISE {else}
ELSE-IF -> ELSE IF {elseif}
EVENTUALLY -> eventually
EQUAL -> equal
EQUALS -> equals
COMPARE -> IS {eEqual}
COMPARE -> IS NOT {eNotEqual}
COMPARE -> BE {eEqual}
CANNOT -> cannot
```

```
COMPARE -> CANNOT BE {eNotEqual}
COMPARE -> EQUALS [TO] {eEqual}
COMPARE -> EQUAL [TO] {eEqual}
COMPARE -> IS NOT EQUAL [TO] {eNotEqual}
COMPARE -> IS EQUAL [TO] {eEqual}
COMPARE -> IS GREATER THAN {eGreaterThan}
COMPARE -> IS LESS THAN {eLessThan}
COMPARE -> IS GREATER [THAN] OR EQUAL [TO] {eGreaterEqual}
COMPARE -> IS LESS [THAN] OR EQUAL [TO] {eGreaterEqual}
COMPARE -> IS NOT EQUAL TO {eNotEqual}
COMPARE -> DOES NOT EQUAL [TO] {eNotEqual}
SOMETIMES -> sometimes | OTHER TIMES
NUMBER -> CARDINAL-NUMBER {CARDINAL-NUMBER.sem}
NUMBER -> CARDINAL-NUMBER NUMBER
                    {fCreateNumber(CARDINAL-NUMBER.sem, NUMBER.sem)}
WHILE -> while | AS LONG AS
TIME-PASSED -> [have] PASSED | [have] ELAPSED | elapses | passes |
               elapse | pass
TIME-UNIT -> minutes | minute | seconds | second | hour | hours |
             day | days
TIME-INTERVAL -> NUMBER TIME-UNIT
                    {fCreateTimeInterval(NUMBER.sem, TIME-UNIT.sem)}
TIME-CHANGE -> TIME-INTERVAL [TEMP] TIME-PASSED
               {fCreateTimeChange(TIME-INTERVAL.sem, TEMP.sem)}
STORED -> stored | saved
SAVE-VARIABLE -> SAVE OP PROP-NAME IN UNKNOWN-NAME
   {fCreateSetVariable(OP.sem, PROP-NAME.sem, UNKNOWN-NAME.sem, eSave)}
SAVE-VARIABLE -> SAVE [THE] CURRENT OP-NO-DET PROP-NAME
    {fCreateSetVariable(OP-NO-DET.sem, PROP-NAME.sem, NULL.sem, eSave)}
LOAD-VARIABLE -> SET OP PROP-NAME TO [THE] STORED UNKNOWN-NAME
   {fCreateSetVariable(OP.sem, PROP-NAME.sem, UNKNOWN-NAME.sem, eLoad)}
LOAD-VARIABLE -> SET OP PROP-NAME1 TO [THE] LAST OBJECT PROP-NAME2
          {fCreateSetVariable(OP.sem, PROP-NAME1.sem, NULL.sem, eLoad)}
LOAD-VARIABLE -> LOAD [THE] STORED UNKNOWN-NAME TO OP PROP-NAME
   {fCreateSetVariable(OP.sem, PROP-NAME.sem, UNKNOWN-NAME.sem, eLoad)}
LOAD-VARIABLE -> LOAD [THE] LAST OP-NO-DET PROP-NAME1 TO OP PROP-NAME2
   {fCreateSetVariable(OP-NO-DET.sem, PROP-NAME1.sem, NULL.sem, eLoad)}
UNKNOWN-NAME -> STRING-VAL {STRING-VAL.sem}
MODEL -> OP IS A TYPE OF [A] OP
         {fAddObjectToModel(OP.sem, OP.sem)}
MODEL -> OP IS A OP-NO-DET
         {fAddObjectToModel(OP.sem, OP-NO-DET.sem)}
LAST -> last
CURRENT -> current
SAVE -> save | store | stores
LOAD -> load
PASSED -> passed
ELAPSED -> elapsed
OF -> of
TIME -> time
PERCENT -> percent
```

```
OTHER -> other
TIMES -> times
DOT -> .
OTHERWISE -> otherwise
IS -> is | be
IF -> if
NOT -> not
IN -> in
TO -> to
SET -> set
THE -> the
WITH -> with
WHEN -> when
THAT -> that
ITS -> its
FOLLOWING -> following
CAN -> can
NEVER -> never
HAPPEN -> happen
GREATER -> greater
LESS -> less
THAN -> than
EQUAL -> equal
OR -> or
ONLY -> only
AFTER -> after
AND -> and
AS -> as
LONG -> long
DOES -> does
UNTIL -> until
UNLESS -> unless
INCREASE -> increase | increases | increased
DECREASE -> decrease | decreases | decreased
METHOD -> increase | increases | increased {increase}
METHOD -> decrease | decreases | decreased {decrease}
A -> a | an
TYPE -> type
CLASS -> Category
ARE -> are
FORBIDEN -> forbiden
METHOD -> METHOD PROPOSITION {METHOD.sem}
PROP-VAL -> [PROPOSITION] STRING-VAL
            {fCreateStringValue(STRING-VAL.sem, null)}
PROP-VAL -> [PROPOSITION] TIME-INTERVAL {TIME-INTERVAL.sem}
PROP-VAL -> [PROPOSITION] [DET] STRING-VAL
            {fCreateStringValue(STRING-VAL.sem, DET.sem)}
STRING-VAL -> coins
STRING-VAL -> coin
```

Appendix B

Michael Roth Data

We present here a few example episodes from Michael Roth's data (Roth et al., 2014). Each episode describes a simple system via a set of roughly a dozen individual requirements or less. The original data of Roth can be downloaded at URL http://www.coli.uni-saarland.de/~mroth/data/requirements.tar.gz.

Restmarks

1. A user must be able to create a user account by providing a username and a password.
2. A user must be able to login to his account by providing his username and password.
3. A user that is logged in to his account must be able to update his password.
4. A logged in user must be able to add a new bookmark to his account.
5. A logged in user must be able to retrieve any bookmark from his account.
6. A logged in user must be able to delete any bookmark from his account.
7. A logged in user must be able to update any bookmark from his account.
8. A logged in user must be able to mark his bookmarks as public or private.
9. A logged in user must be able to add tags to his bookmarks.
10. Any user must be able to retrieve the public bookmarks of any RESTMARKS's community user.
11. Any user must be able to search by tag the public bookmarks of a specific RESTMARKS's user.
12. Any user must be able to search by tag the public bookmarks of all RESTMARKS users.
13. A logged in user , must be able to search by tag his private bookmarks as well.

Taxi

1. A taxi notifies the server of its location continuously.
2. A taxi can get its zone information from the server.
3. A taxi can change its status.
4. A customer can place an order for a taxi.
5. A taxi can receive a customer order.
6. After the taxi is selected , customer gets taxi info.

Teacher and Students

1. Teachers could browse as well as create topic or course with the possibility to add and manipulate material or resources as per need.
2. Teachers could easily create exercise on a selected topic or course.
3. Teachers could add and manipulate file resources with respect to specific course or a topic.
4. Teachers can create flash cards.
5. The students could browse and search a topic in any domain.
6. Students have the ability to write notes and could also take printouts of them.
7. It is also possible for students that they can create cards and share them among others.
8. Students could do some exercises and the systems should show results at the end highlighting score.

Appendix C

Rule-Based Algorithm Pseudo Code

1. Role labeler annotation <- parse each requirement according to Michael Roth role labeler.
2. Co-references <- for each requirement get it's co-references.
3. Possible lifelines detection (add context) <- iterate over all requirements and create list of all possible lifelines (actors, theme and (NN, SBJ))
4. For each requirement :
   (a) create lifelines based on annotation + possible lifelines
   (b) add clock if init step is needed (or periodically ticking is needed)
   (c) add properties to lifelines by usage of co-references + annotations + rules - .
   (d) add Framework (System) lifeline if needed.
   (e) sort action by linguistic hints (based on modifiers ontology)
   (f) for each action:
      i. detect sender, receiver and parameters according to SRL and dependency tree
      ii. detect state (HOT/COLD, Monitored/ Executed) according to the feature structure
      iii. add sync
   (g) remove lifelines that do not participate in the current LSC (do not have any income/outcome message/ part of loop/if statement/ assignment and so..)

Bibliography

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

Arnold, Barry C. and David Strauss. “Pseudolikelihood Estimation: Some Examples”. In:

Artzi, Y. and L. S. Zettlemoyer (2013). “Weakly Supervised Learning of Semantic Parsers for Mapping Instructions to Actions.” In: *Trans. of ACL* 1, pp. 49–62. URL: http://dblp.uni-trier.de/db/journals/tacl/tacl1.html#ArtziZ13.

Bahl, R., F. Jelinek, and R. Mercer (1983). “A maximum likelihood approach to continuous speech recognition”. In: *IEEE Trans. Pattern Anal. Mach. Intell., 5(2):179–190*.

Ballard, B. W. and A. W. Biermann (1979). “Programming in Natural Language: “NLC" As a Prototype”. In: *Proceedings of the 1979 ACM Annual Conference*. New York, NY, USA, pp. 228–237. ISBN: 0-89791-008-7. DOI: 10.1145/800177.810072. URL: http://doi.acm.org/10.1145/800177.810072.

Barzilay, R. et al. (2013). “From Natural Language Specifications to Program Input Parsers”. In: *Proceedings of ACL*, pp. 1294–1303.

Black, E., J. D. Lafferty, and S. Roukos (1992). “Development and Evaluation of a Broad-Coverage Probabilistic Grammar of English-Language Computer Manuals”. In: *Proceedings of ACL*, pp. 185–192.

Booch, G. (1986). “Object Oriented Development”. In: *IEEE Transactions on Software Engineering, (2):211–221.*

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

Brill, E. and R. C. Moore (2000). “An Improved Error Model for Noisy Channel Spelling Correction”. In: *Proceedings of ACL*.

Brown, P. F. et al. (1993). “The Mathematics of Statistical Machine Translation: Parameter Estimation”. In: *Comp. Ling.* 19.2, pp. 263–311. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=972470.972474.

Bryant, B. and B. S. Lee (2002). “Two-Level Grammar As an Object-Oriented Requirements Specification Language”. In: *Proceedings of the 35th Hawaii Annual International Conference on System Sciences*. Washington, DC, USA., pp. 3627–3636. ISBN: 0-7695-1435-9. URL: http://dl.acm.org/citation.cfm?id=820747.821314.

Cabral, G. and A. Sampaio (2008). “Formal Specification Generation from Requirement Documents”. In: *Electronic Notes in Theoretical Computer Science Volume 195, 18 January 2008, Pages171-188.*

Charniak, Eugene (1996). “Tree-bank Grammars”. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1031–1036.

Clark, S., B. Coecke, and M. Sadrzadeh (2011). "Mathematical Foundations for a Compositional Distributed Model of Meaning". In: *Linguistic Analysis* 36.1-4. Ed. by van Bentham and Moortgat, pp. 345–384.

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

Das, D. et al. (2010). "Probabilistic Frame-semantic Parsing". In: *Proceedings of NAACL-HLT*. HLT '10. Los Angeles, California, pp. 948–956. ISBN: 1-932432-65-5. URL: http://dl.acm.org/citation.cfm?id=1857999.1858136.

Dijkstra, E. W. (1979). *On the Foolishness of Natural Language Programming*. EWD 667.

Fuchs, N. E. and R. Schwitter (1995). "Attempto: Controlled Natural Language for Requirements Specifications." In: *LPE*. Ed. by M. P. J. Fromherz, M. Kirschenbaum, and A. J. Kusalik. URL: http://dblp.uni-trier.de/db/conf/lpe/lpe95.html#FuchsS95.

Ghosh, S. et al. (2014). "Automatic Requirements Specification Extraction from Natural Language". In: *arXiv preprint arXiv:1403.3142.*

Gordon, M. and D. Harel (2009). "Generating Executable Scenarios from Natural Language". In: *Proceedings of CICLing*. Mexico City, Mexico: Springer-Verlag, pp. 456–467. ISBN: 978-3-642-00381-3. DOI: 10.1007/978-3-642-00382-0_37. URL: http://dx.doi.org/10.1007/978-3-642-00382-0_37.

Gordon, Michal and David Harel (2011). "Show-and-Tell Play-In: combining natural language with user interaction for specifying behavior". In: *Proc. IADIS Interfaces and Human Computer Interaction*. Vol. 360-364.

Grice, H. P. (1975). "Logic and Conversation". In: *Syntax and Semantics: Vol. 3: Speech Acts*. Ed. by P. Cole and J. L. Morgan. San Diego, CA: Academic Press, pp. 41–58.

Gulwani, Sumit and Mark Marron (2014). "NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation". In: *SIGMOD '14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data Pages 803-814*.

Harel, D. and S. Maoz (2006). "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams". In: *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*. SCESM '06. Shanghai, China: ACM, pp. 13–20. ISBN: 1-59593-394-8. DOI: 10.1145/1138953.1138958. URL: http://doi.acm.org/10.1145/1138953.1138958.

Harel, D. and R. Marelly (2003). *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Secaucus, NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540007873.

Harel, David. et al. (2010). "PlayGo: Towards a Comprehensive Tool for Scenario Based Programming". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. Antwerp, Belgium: ACM, pp. 359–360. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859075. URL: http://doi.acm.org/10.1145/1858996.1859075.

Harmain, H. M. and R. Gaizauskas (2003). "CM-Builder: A Natural Language-based CASE Tool". In: *Journal of Automated Software Engineering* 10, pp. 157–181.

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

Kugler, H. et al. (2005). "Temporal Logic for Scenario-Based Specifications". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 3440. Springer Berlin Heidelberg, pp. 445–460.

Kuhn, T. (2014). "A Survey and Classification of Controlled Natural Languages". In: *Comp. Ling.* 40.1, pp. 121–170.

Kushman, N. and R. Barzilay (2013). "Using Semantic Unification to Generate Regular Expressions from Natural Language". In: *Proceedings of NAACL-HLT*, pp. 826–836.

Lee, H. et al. (2011). "Stanford's Multi-Pass Sieve Coreference Resolution System at the CoNLL-2011 Shared Task." In: *CoNLL*.

Lee, H. et al. (2013). "Deterministic coreference resolution based on entity-centric, precision-ranked rules." In: *Computational Linguistics*.

Leite, Julio Cesar (1987). "A survey on requirements analysis." In: *Department of Information and Computer Science, University of California, Irvine, Advanced Software Engineering Project Technical Report RTP071*.

Lewis, M. and M. Steedman (2014). "Combining Formal and Distributional Models of Temporal and Intensional Semantics". In: *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pp. 28–32. URL: http://homepages.inf.ed.ac.uk/s1049478/publications/sp2014.pdf.

Liang, P., M. I. Jordan, and D. Klein (2011). "Learning Dependency-Based Compositional Semantics". In: *Proceedings of ACL*, pp. 590–599.

Liang, P. and C. Potts (2015). "Bringing machine learning and compositional semantics together". In: *Ann. Rev. of Ling.* 1.1, pp. 355–376.

Lieberman, H. and H. Liu (2005). "Feasibility Studies for Programming in Natural Language". In: *End-User Development*. Ed. by H. Lieberman, F. Paterno, and V. Wulf. Springer: Springer. URL: http://web.media.mit.edu/~lieber/Publications/Feasibility-Nat-Lang-Prog.pdf.

Liu, Hugo and Henry Lieberman (2005). "Metafor: visualizing stories as code". In: *IUI '05 Proceedings of the 10th international conference on Intelligent user interfaces Pages 305-307*.

Mich, L (1996). "NL-OOPS: From natural language to object oriented requirements using the natural lan- guage processing system LOLITA". In: *Natural Language Engineering , 2(2):161–187*.

Mihalcea, R., H. Liu, and H. Lieberman (2006). "NLP (Natural Language Processing) for NLP (Natural Language Programming)". In: *Proceedings of CICLING*. Springer-Verlag, pp. 319–330.

Moens, M. and M. Steedman (1988). "Temporal Ontology and Temporal Reference". In: *Comp. Ling.* 14.2, pp. 15–28. ISSN: 0891-2017. URL: http://dl.acm.org/citation.cfm?id=55056.55058.

Nanduri, S. and S. Rugaber (1995). "Requirements validation via automated natural language parsing". In: *Journal of Management Information Systems - Special section: Information technology and its organizational impact archive Volume 12 Issue 3, December 1995 Pages 9 - 19*.

Nuseibeh, B. and S. Easterbrook (2000). "Requirements Engineering: A Roadmap". In: *Proceedings of ICSE*.

Poon, H. and P. Domingos (2009). "Unsupervised Semantic Parsing". In: *Proceedings of the Conference on Empirical Methods in Natural Language*

Processing. Singapore, pp. 1–10. ISBN: 978-1-932432-59-6. URL: http://dl.acm.org/citation.cfm?id=1699510.1699512.

Raghunathan, K. et al. (2010.). "A Multi-Pass Sieve for Coreference Resolution". In: *EMNLP*.

Recasens, Marta, Marie Catherine de Marneffe, and Christopher Potts. (2013). "The Life and Death of Discourse Entities: Identifying Singleton Mentions." In: *NAACL*.

Roth, M. 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.

Roth, M. et al. (2014). "Software Requirements: A New Domain For Semantic Parsers". In: *Proceedings of the ACL 2014 Workshop on Semantic Parsing (SP14)*.

Saeki, M., H. Horai, and H Enomoto (1989). "Software Development Process from Natural Language Specification". In: *Proceeding ICSE '89 Pages 64-73*.

Shannon, C. (1948). "A Mathematical Theory of Communication". In: *Bell System Technical Journal* 27, pp. 379–423. URL: http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

Tan, Lin, Yuanyuan Zhou, and Yoann Padioleau (2011). "aComment: mining annotations from comments and code to detect interrupt related concurrency bugs". In: *Proceeding ICSE '11 Pages 11-20*.

Thompson, C. A., R. J. Mooney, and L. R. Tang (1997). "Learning to Parse NL Database Queries into Logical Form". In: *The ML-97 Workshop on Automata Induction, Grammatical Inference, and Language Acquisition*. Nashville, TN. URL: http://www.cs.utexas.edu/users/ai-lab/?thompson:ml97ws.

Viterbi, A. (1967). "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm". In: *IEEE Trans. Inf. Theor.* ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1054010. URL: http://dx.doi.org/10.1109/TIT.1967.1054010.

Webber, B. and A. Joshi (2012). "Discourse Structure and Computation: Past, Present and Future". In: *Proceedings of the ACL-2012 Special Workshop on Rediscovering 50 Years of Discoveries*, pp. 42–54.

Younger, D. H. (1967). "Recognition and parsing of context-free languages in time n3". In: *Information and Control* 10.2, pp. 189–208.

Zapata, C. and B. Losada (2012). "Transforming Natural Language into Controlled Language for Requirements Elicitation: A Knowledge Representation Approach". In: *Ramrez Gutirrez, ISBN 978-953-51-0597-8, Published: May 9, 2012 under CC BY 3.0 license.*

Zettlemoyer, L. S. and M. Collins (2012). "Learning to Map Sentences to Logical Forms: Structured Classification with Probabilistic Categorial Grammars". In: *CoRR* abs/1207.1420.

Zhong, H. et al. (2009). "Inferring resource specifications from natural language api documentation." In: *Proceedings of ASE*, pp. 307–318.

גורדון והראל [Gordon 2009], הגדירו שפה חסרת הקשר (כלומר, סט של חוקי גזירה המצורף בנספחים) ואלגוריתם חצי-אוטומטי שעל ידי עזרה מן המשתמש ממפה משפטים השייכים לשפה לייצוגם הפורמלי. בעבודתנו הראשונה [Tsarfaty et al. 2014] פיתחנו מנוע לימוד סטטיסטי, שלמד מדוגמאות מנותחות (annotated data) כיצד למפות באופן אוטומטי (ללא צורך בהתערבות של משתמש אנושי) מסמכי דרישות שנכתבו בCNL לייצוג הפורמלי.

ניסויים אמפיריים שערכנו הראו איכות תרגום של 95% במדד F-score על סט קטן של דוגמאות למסמכי דרישות בשפה זו. כמו כן הראנו ששימוש במסמך כולו (בהקשר הדרישה) משפר את הדיוק בניתוח דרישות לעומת ניתוח ברמת המשפט הבודד.

לאחר שהורדנו את התלות במשתמש, המשכנו לאתגר הבא שלנו, ניתוח מסמכי דרישות שנכתבו בשפה טבעית ללא מגבלות. מסמכי הדרישות הכתובים בשפה טבעית לקוחים ממחקר אחר [Roth 2014]. מסמכי דרישות אלה הינם אפיונים של מערכות שניתנו לסטודנטים במדעי המחשב במסגרת שיעורי הבית. כל מסמך דרישות מתאר מערכת יחידה ופשוטה. פיתחנו אלגוריתם המשלב כלים סטטיסטים כגון מנתח חלקי דיבר, מנתח תחבירי, מנתח תלויות ומנתח סמנטי, ועבורם הגדרנו סט חוקים (rule based) המשתמש בתוכן המסמך כולו (context) על מנת לתרגם את המסמך לייצוג הפורמלי הרצוי. כמו כן, פיתחנו מדדי בקרת איכות לבדיקת איכות הפלטים שלנו, המבוססת על מרחק עריכת העצים (edit distance) של המבנה הסטטי.

על מנת להבין את איכות המערכת שלנו, ביקשנו ממספר מהנדסי תוכנה לייצר עבורנו את המודל הסטטי (object model) עבור מספר מסמכי דרישות. על-ידי שימוש במדדי בקרת האיכות שלנו, ראינו שממד ההסכמה בין שני מהנדסי תוכנה שונים כמעט זהה למדד ההסכמה בין המודל שנוצר על-ידי האלגוריתם למודל שנוצר על-ידי מהנדסי תוכנה עבור אותן מסמכי דרישות. תוצאות אלו העלו לנו מספר שאלות למחקרים הבאים. האם יש דבר כזה מודל נכון? האם יש צורך להתחיל ממודל נכון? האם נקודת התחלה של מודל אוטומטי עדיף מהתחלה ריקה?

תקציר

כל תהליך של הנדסת תוכנה מתחיל מרעיון, רוב הרעיונות הנ״ל מתוארים על ידי מסמכי אפיון הנכתבים על ידי בני אדם (בעלי עניין, מהנדסי מערכת, מנתחי דרישות) בשפה טבעית. לאחר מכן, מסמכי דרישות אלו מתורגמים על ידי מהנדסי תוכנה לתוכנית מחשב בשפת קוד העונה לדרישות.

כבר בעבר התמודדו חוקרים עם המשימה של תרגום אוטומטי של מסמכי דרישות בשפה טבעית לתכניות מחשב בשפת קוד. המחקר הראשון [Abbot 1983] התמקד בחילוץ העצמים והמאפיינים מתוך הדרישות, כלומר - עצמים (objects), מאפייני עצמים (properties), ושיטות (methods). מחקרים מאוחרים יותר, [Mich 1996] הגדירו שפות חלקיות מבוססות אנגלית לכתיבת מסמכי דרישות. שפות אלו (לרוב שפות חסרות הקשר עם חוקי גזירה מוגדרים, בדומה לשפות תכנות) נהנו מתרגום ישיר לייצוג פורמלי, ומתרגום חד משמעי לשפת קוד. אך אליה וקוץ בה --- שפות אלו הנן מוגבלות ביכולת הביטוי שלהן, ואינן טבעיות לביטוי והבנה עבור דוברי שפה אנושית.

מטרתנו במחקר זה היא לתרגם בצורה אוטומטית מסמכי דרישות שנכתבו בשפה טבעית לתוכניות בשפת קוד הניתנת להרצה, על ידי שימוש בכלים ושיטות הלקוחים מתחום עיבוד השפה הטבעית ומכונות לומדות. המטרה העיקרית שלנו, להרחיב מעבר לחילוץ מבנה התוכנה (object model) , על ידי הבנת הדרך בה נוצרים מופעים (instances) של עצמים ומידול התקשורת ביניהם. עם זאת, אין ברצוננו להניח שום הנחה מקלה על הקלט - מטרתנו היא להתמודד עם תרגום של שפה טבעית אמיתית.

שיטתנו לתרגום מסמכי הדרישות הכתובים בשפה טבעית לשפת קוד היא דרך תרגומם לייצוג ביניים פורמלי. המורכב מ2 ישויות מרכזיות : Live Sequence Charts שנסמן כ LSC ו System Model שנסמן כ SM. כל מסמך דרישות המורכב מאוסף של דרישות ממופה לאוסף של LSC (אחד לכל דרישה) וSM גלובלי ברמת המסמך כולו. LSC הינו הרחבה מעל UML המתאר את ההתנהגות הדינמית של העצמים המתוארים בדרישה זו, ואילו SM הינו מידול הסטטי של המערכת כולה. בסיום תרגום זה,אנו מסתמכים על תרגום ישיר הקיים בין הייצוג הפורמלי לקוד JAVA.

עבודה זו בוצעה בהדרכתם של דר' רעות צרפתי מהאוניברסטיה הפתוחה ופרופסור שמעון שוקן מבי"ס אפי ארזי למדעי המחשב, המרכז הבינתחומי, הרצליה.

המרכז הבינתחומי בהרצליה
בית ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.) מסלול מחקרי

מדרישות מערכת בשפה טבעית לקוד הניתן להרצה

מאת
איליה פוגרבצקי

עבודת תזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.
במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

ינואר 2017