Text Understanding for Programming in Natural Language: Control Structures

Mathias Landhäußer
Karlsruhe Institute of Technology
Email: landhaeusser@kit.edu

Ronny Hug
Karlsruhe Institute of Technology
Email: ronny.hug@student.kit.edu

***Abstract*—We investigate how natural languages such as English can be used as programming languages. Often, in natural language (as well as in programming) different actions happen at the same time or are repeated. In natural language we just say what is going to happen – in programming we use control structures. Story lines without equivalent language/phrase structures are unnatural and programs without control structures are hard to read. An empirical study showed how users express control structures in natural language.**

We propose a new and automatic text analysis. It leverages Stanford's typed dependencies to detect sentence structures that imply strictly sequential control flows, repetition, and parallelism.

The technique is analyzed in the context of Alice, a 3D programming environment, and AliceNLP, a system for programming Alice in ordinary English. We evaluate our approach with 52 texts with 795 control-flow-affected elements in total and show that 82% of these elements can be detected successfully. We performed a second evaluation with manually corrected input and find that our approach successfully detects the control structures 97% of the time in the absence of parser errors.

I. Introduction

We aim at programming in natural language – up to now, computers have been instructed in specialized synthetic languages: programming languages. The obvious alternative, natural language, is not considered because being too complex, ambiguous, or (too) unconstrained. Programming languages on the other hand are "simple", "strict", and "constrained". Even though Sammet already proposed natural language as an alternative for programming in 1966 [1], there has not been much progress. However, significant progress has been made when it comes to language modeling, text mining, and language understanding. This progress lead to impressive, usable tools like Watson [2], Apple's Siri [3], and Google Translator.

We develop a system that translates plain English into executable code using available NLP technologies and tool chains. Our project AliceNLP builds a prototype that can be seen as a black box that takes English text as input and produces source code as output. Inside the black box, AliceNLP analyzes the input text with various NLP tools and asks the user for clarification if information is missing. Figure 1 shows the overall architecture of AliceNLP: It uses the text as information store and every analysis module can read the results from its predecessors; its own results are readily available for its successors. Some modules are proven NLP tools such as Stanford's CoreNLP [4]. Others are specialized modules that analyze the input text to identify programming-specific information. The domain knowledge (i. e. the knowledge about the target API) is provided to the analysis modules in an ontology. It contains information about the available classes, their properties and their methods.

The application domain of AliceNLP is programming 3D animations in Alice. Alice is a sophisticated 3D animation tool developed at Carnegie Melon University and is used in introductory programming classes [5]. Alice provides a rich set of everyday objects (e.g. vehicles and tools), people and imaginary characters (fairies etc.). Users can program Alice animations using these objects. Alice provides a set of basic functions (for movement etc.) only but users can extend the objects' capabilities by programming customized methods. AliceNLP takes natural language scripts as input and produces Alice animations.

To evaluate our prototype, we collected scripts from various authors that describe manually programmed animations. Following this procedure we can compare the results of AliceNLP with the original animations and the programs that were used to create them. The scripts form a growing corpus of English scripts; we have gold solutions (and the manually created programs) for all scripts in the corpus and use them for automatic evaluation.

The first animation that we considered was simple: The code to produce this animation does not contain loops and is strictly sequential. Then we allowed the authors to rearrange the actions in their scripts (e.g. "do a, but before, do b") and determined the correct sequential order [6]. As the project matured, we constructed more complex animations with more actions, parallel actions, loops, and so on. The natural language scripts for these animations are more complicated and use many different ways to describe the same flow of actions.

This paper explains how we map the described flow of actions to classical control structures. We designed an extensible approach that leverages Stanford's typed dependencies [7] and implemented a tool called control structure recognizer (CSR).

Section II reviews related work, Section III presents our approach in detail. In Section IV we evaluate our approach with 52 texts for six different animations. Section V concludes the paper.

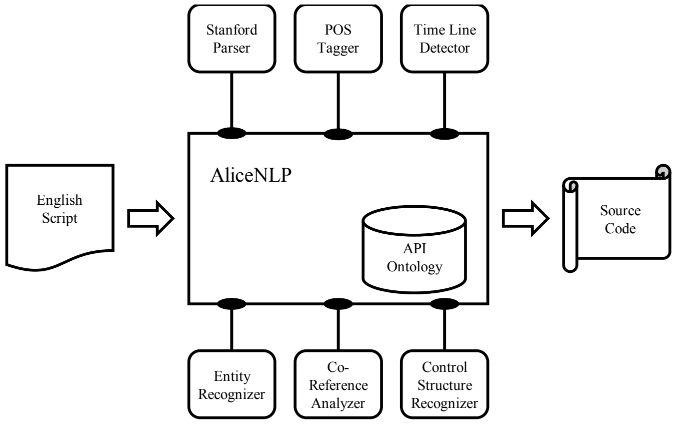

Fig. 1. The AliceNLP Architecture: Natural language analyses are independent but can share information; analysis results are annotated in the text.

II. Related Work

Programming in natural language has a long history. Already in 1979, Ballard and Bierman presented NLC, the natural language computer [8]. It interprets direct commands such as “add x1 to y2”, performs matrix calculations, and gives immediate feedback about the results. After entering a series of commands, users can let NLC repeat them either a given number of times (e.g. “repeat the last command 5 times”) or for a given number of entries in the matrix (e.g. “repeat for all entries in row 3”) [9].

In 2000, Price et al. presented NaturalJava, a prototypical user interface based on natural language for creating, modifying, and examining Java programs [10]. Users are expected to dictate Java and one can use all control structures that are available in Java. But as users of AliceNLP neither write nor dictate code, NaturalJava is not directly comparable to AliceNLP.

Pane and Myers investigated how non-programmers describe programming solutions to make future programming languages more user-centered [11]. They report that users tend not to dictate loops directly: they describe actions set-wise instead (e.g. “A, B, and C do X”) and rely on the listener's interpretative power. The scripts in our corpus support this finding.

Metafor, an approach described by Liu and Lieberman in references [12] and [13], creates python classes from English stories. As only stubs are being generated, Metafor does not deal with control structures. In reference [14] Mihalcea, Liu, and Lieberman describe how programming steps (“actions” in our terms), loops, and comments can be identified in natural language and how they can be mapped to programming constructs. They use signal words to identify loops. Words in plural also trigger the creation of loops.

Knöll and Mezini research an approach to natural language programming called Pegasus [15]. Pegasus translates natural language input in an intermediate “idea language”, which is then compiled into code. Control structures can be expressed in the idea language but it remains unclear how the transition from natural language to the idea language is performed.

Dependencies	conj, prep
POS Tags	VBG, VBZ
Ignored Words	times
Signal Phrases	at the same time

TABLE I
An Example for a Trace Profile for *do together*.

There is a body of research on temporal analysis of texts. It mainly deals with the identification of events in a text and with putting them on a time line (e.g. references [16] and [17]). Some systems tackle temporal notions such as “noon” and “midnight” and translate them to numeric points (or ranges) in time [18]. Other systems use statistics to infer relations between events: E.g. Mani et al. detect simultaneous events (among other relations) using a maximum entropy learner [19]; Chambers et al. classify the relation between two events using part-of-speech (POS) tags, lemmas, WordNet synsets and modal words [20]; Kolya et al. use conditional random fields to infer the temporal order of a sequence of events [21]. Berglund et al. use decision trees to order events for a car crash simulation based on witness statements [22]. Lapata and Lascarides propose a statistical approach for inferring sentence-internal temporal relations, i.e. relations between events in main clauses and its sub clauses [23]. Their method exploits the presence of signal words such as “after” and “before”.

Nemec proposes a rule-based system of automatic analysis of temporal relations within a Czech discourse [24]. His algorithm predicts relative ordering relations between finite verbs in a sentence based on the information provided only by the grammar. It looks at finite verbs, because they bear one of the three basic tenses – past, present and future.

Also, there is a body of research on temporal reasoning [25]. For example, Russell et al. describe an event calculus [26]. The emphasis in this work is on automated reasoning using the calculus, not the extraction of event order from texts. In summary, temporal reasoning is related to the detection of control structures (e.g. “while A, do B”) but the work focuses on detecting events and anchoring them on a time line.

III. Control Structures from Text

To extract control structures from natural language, we analyze the text sentence-by-sentence. The first step is to analyze the text with Stanford's CoreNLP tool chain which produces the needed linguistic information [4], namely part-of-speech tags (POS tags) and typed dependencies. Then CSR takes over. CSR processes every control structure in isolation as the linguistic properties differ, but the analyses work the same. It traverses the dependency graph to identify the actions and their (temporal) relationships. It visits different edges in the graph from different starting points depending on the control structure to be identified.

The configuration for a control structure is called a trace profile. A trace profile contains a list of dependency types, a list of POS tags, a list of signal words (such as “meanwhile”), and a list of words that should not be treated as actions. The latter is a list of entities (objects from the domain or: class names) and the list is identical in all configurations. CSR uses POS tags to identify the words that refer to actions (usually it recognizes verbs as actions). We use the POS tag list to mitigate parser errors (sometimes the parser labels nouns as verbs and vice versa); if the parser gives perfect results, the list can and should be empty. Also, trace profiles are tunable: By including more/less dependencies and including more/less signal words, we can counter some (but not all) parser errors. Table I shows an excerpt from a trace profile.

After all sentences are processed, a post-processing step ensures that the nesting of the detected control structures is correctly recorded. Also, unnecessary control structures are deleted (e.g. sequential blocks with only one statement). Then the results are annotated in the text.

We evaluate two different configurations: The default configuration that follows linguistic considerations and a tuned configuration that additionally makes provisions for parser errors. The latter performs better when the dependency graphs contain errors, the former performs better without them.

The following subsection explains the dependency graph traversal. The second subsection illustrates how CSR works with an example. The last subsection discusses the control structures that can be detected with CSR.

A. Control Structure Independent Dependency Graph Traversal

CSR traverses the dependency graph to identify the actions that belong to a specific control structure. A dependency graph is a directed graph with the words of a phrase as nodes. The typed edges express the dependencies between two words: Figure 2 shows a dependency graph for the phrase "Mathias and Ronny analyze dependencies." The node without incoming edges is called the root of the phrase; as in Figure 2 most often the main verb of the phrase is the root node, here "analyze". *subj* denotes the subject of the verb: "Mathias". The dependencies *conj* from "Mathias" to "Ronny" and *cc* to "and" show that they are connected with the conjunction "and". And finally, *dobj* denotes that "dependencies" is the direct object of "analyze". There are ca. 50 different dependencies in total (c.f. reference [7] for a thorough explanation).

Graph traversal works essentially the same for all control structures: CSR starts the traversal at a signal word (or phrase) and follows the edges in the graph to collect the actions; CSR ignores the direction of the edges. Actions are identified by their POS tags; the default configuration lists all POS tags for verbs so that verbs are being considered actions regardless of their tense. Edges are only visited once and if their type is allowed by the active trace profile. If CSR encounters an action, the action is recorded and traversal starts from this node again. The types of the dependency edges that are traversed are specific for each control structure. Therefore CSR's configuration has several parts: one for each control structure. Similar control structures such as *do together* and *do in order* share a configuration; the exact type (i.e. *together* vs. *in order*) is determined after the first processing.

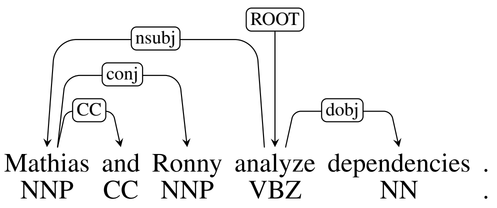

Fig. 2. A Stanford Typed Dependency Graph for the Phrase "Mathias and Ronny analyze dependencies". POS Tags are Shown Below the Words.

B. Dependency Tracing by Example

In this section we illustrate CSR's approach with an example. It shows how CSR detects *do together* and how it identifies the actions that belong together. The trace profile in Table I is used in this example. Figure 3 shows the dependency graph.

Dependency tracing starts at the signal phrase "at the same time". From there all edges are considered: CSR checks whether an edge is contained in the trace profile. The only edge in the example is labeled with *prep* and the trace profile contains this dependency type and so the edge is traversed. CSR reaches "nods" which has the POS tag VBZ; the trace profile defines VBZ as action and thus "nods" is recorded. Afterwards, the traversal starts again from "nods". *prep* has already been visited and *nsubj* is ignored because it is not in the profile. *conj* is in the trace profile and so CSR traverses this edge. "hops" has the POS tag VBZ as well and is recorded. As there are no more edges coming to/from "nods", the traversal starts again from the node "hops". There the traversal stops because *conj* has already been visited and neither *nsubj* nor *cc* are in the trace profile. There are no further edges.

The result set contains the actions "nods" and "hops" and therefore CSR annotates in the document that they happen simultaneously.

C. Covered Control Structures

Alice – as every programming language – supports various control structures. In addition to the usual it provides simple structures for parallelism: *do together* performs the contained (action) block in parallel and *for all together* performs the loop body with a set of objects in parallel. All control structures have a body that encompasses a single action or multiple actions; of course, control structures can be nested. The following paragraphs describe the available control structures and give an idea on how CSR identifies the control structure and its contents.

a) do together and do in order: do control structures describe the flow of actions in a single sentence. There is a *do together* and a *do in order*; the former expresses that the actions happen simultaneously and the latter enforces a strictly sequential order. To detect actions that are within the same do structure, one has to identify all actions within a sentence that are linked by words like "simultaneously" and phrases like "first ... then".

b) while: while – as well as *do* – expresses that two actions are connected: The actions within the body of *while*

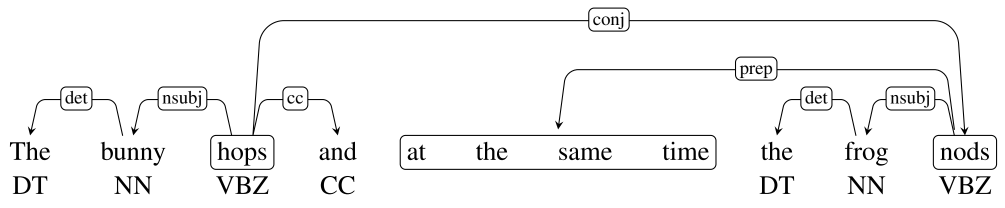

Fig. 3. A Stanford Typed Dependency Graph for the Phrase "The bunny hops and at the same time the frog nods". POS Tags are Shown Below the Words.

happen in a loop until the condition of the *while* loop evaluates to false. CSR identifies the actions that happen within the *while* body and within the *while* condition.

c) for all in order and for all together: for all executes all actions within its body on (or with) a list of entities. The list of entities must be inferred by CSR. At the moment, CSR uses AliceNLP's ontology to determine which entities of the current script belong to a group (e.g. "all animals"). Also it handles exceptions (e.g. "all animals but not the rabbit") and enumerations ("the rabbit, the penguin, and the dog do A"). The actions in the body can either be invoked *in order* (i.e. the body of the loop is executed for every entity in the list in full before the actions start with the second entity) or *together* (i.e. all entities start immediately with the execution of the loop).

The analysis for *for all* control structures uses information that has been added to the script by a different analyzer of AliceNLP: The entity analyzer annotates in the text which token refers to which entity (or instance). Therefore CSR can check the ontology for further information about the groups. In short: If there is an action being performed by a group, CSR creates a *for all* structure and dereferences the group. If there is an explicit exception, the excepted actor is removed from the dereferenced group. If there is an enumeration of actors, CSR creates a *for all* structure with the enumerated entities. If CSR detects a group but fails to dereference it, it asks the user which objects of the current program should be included in the group.

d) loop: The last – and maybe the simplest – control structure covered by CSR is the *loop*. A *loop* executes an action a predefined number of times. CSR searches a sentence for cardinal numbers connected to "times" (such as "two times"); it also considers adverbs such as "twice" and "thrice". Then it identifies the action that is connected to the repetitive phrase in the dependency graph; groups of actions are formed by enumerations and signal words such as "simultaneously".

e) if/else: if/else is not supported by CSR at the moment; in our application there is no conditional branching so the detection of *if/else* is not needed. But the parse tree contains the needed information: The conditions are in sub phrases that start with signal words such as "if", "when" and so on. Furthermore, the conditionally executed action is connected with the verb of the condition with an *advcl* dependency (adverbial clause). The condition verb is connected with the signal word itself with a *mark* dependency. CSR can easily be extended to include conditionals using either the parse tree or the typed dependencies.

Evaluation	$\sum$	C	W	M	WP
A-E-D	43	36	0	7	0
B-E-D	221	151	26	66	4
C-E-D	26	11	4	15	0
F-E-D	308	287	6	20	1
M-E-D	47	27	19	19	1
R-E-D	150	137	22	13	0
Sum	795	649	77	140	6
		82%	10%	18%	1%
A-E-T	43	34	4	9	0
B-E-T	221	195	10	17	9
C-E-T	26	16	4	10	0
F-E-T	308	295	9	12	1
M-E-T	47	27	24	19	1
R-E-T	150	138	27	12	0
Sum	795	705	78	79	11
		89%	10%	10%	1%
B-F-D	221	215	2	6	0
F-F-D	308	298	3	10	0
Sum	529	513	5	16	0
		97%	1%	3%	0%

TABLE III
THE RESULTS OF THE THREE EVALUATION RUNS. THE FIRST LETTER REPRESENTS THE ANIMATION; THE SECOND LETTER INDICATES WHETHER THE INPUT CONTAINED NLP ERRORS (E) OR IF THE ERRORS WERE FIXED (F); THE THIRD LETTER INDICATES THE CONFIGURATION USED (D=DEFAULT, T=TUNED)

IV. EVALUATION

We use 52 English scripts for six animations and evaluated whether CSR can correctly identify the control structures and the actions therein. Therefore we created short animations and let subjects describe the animations in their own words. Table II shows the details of the animations and their programs respectively: They are short and should be easy to describe and use a relatively low number of different methods. Beach and Farm make heavy use of control structures. The first two columns show the name of the animations and their durations. LOC indicates how many lines of code are needed to produce the animation. The following two columns show how many objects were used and how many methods were called in the code. The sixth column shows how many control structures have been used for the animation (the numbers in parentheses give the number of different control structures used). The last column shows the number of natural language scripts describing the respective animation.

The subjects are mostly computer science students or staff from our chair but ten of the 30+ subjects have no programming background whatsoever. We asked the subjects to describe the animation with their own words and we also explained AliceNLP's intention. We first showed an animation to

Animation	Duration	LOC	#Objects	#Methods	#Control Structures	#Scripts
Alice	32 sec	31	3	23	7 (3)	4
Beach	20 sec	28	9	11	10 (5)	14
Cheerleader	19 sec	12	2	11	1 (1)	3
Farm	40 sec	39	9	15	10 (7)	14
Moon	38 sec	36	3	25	10 (2)	3
Rabbit	23 sec	21	3	15	0 (0)	14

TABLE II
STATISTICS FOR THE ANIMATIONS USED IN THE EVALUATION.

the subjects and gave them a description written by ourselves as an example. Then we gave the two animations to the subjects including the names of the objects used and a complete list of the available methods and control structures. Given this information they had unlimited time to write the scripts and were allowed to pause and replay the animation as often as they wished. Every subject described both animations. Before the evaluation, we corrected typos and spelling mistakes; no further modifications were made. For every script we manually created a gold solution that identifies all control structures that are contained in the respective script. The gold solution can be compared to the computed results automatically. CSR, all texts, gold solutions, and the results produced by CSR can be downloaded from our website[1].

We first ran the evaluation with the default configuration for CSR that is based on linguistic considerations. During the manual inspection of the input texts and the output we learned that many errors stem from parser errors such as incorrectly recognizing verbs as nouns. Therefore, we tried to improve CSR's performance by configuring the dependency tracer more loosely. We wanted to know whether CSR's configuration can be tweaked to remedy some of the parser's flaws. For example, we added nouns and noun phrases to the POS tag lists. CSR then treats nouns as actions and records more input tokens, which should drastically reduce the miss rate. On the other hand, CSR then treats nouns (correctly) tagged as nouns as actions as well, which should increase the error rate. As we wanted to push CSR to its limits, we ignored the design rationales behind the configuration options and re-ran the evaluation and analyzed the results several times to identify a good configuration. The second block of Table III shows the best results that we achieved.

Now that we knew the average performance of our approach we wanted to know how good it possibly can become. Therefore we manually corrected the input (i.e. we fixed POS tags and other parser results) in the texts that describe the Beach and Farm animations. We omitted the other texts because correcting the parser's and tagger's results is very tedious and we wanted to limit the work necessary. The Beach and Farm animations contain most of the control structures and therefore we narrowed our focus to these texts. The evaluation in this ideal world uses the default configuration for CSR.

Table III summarizes the results of the evaluations. The first row indicates the evaluation run. $\sum$ gives the number of expected annotations; this number differs from text to text as the texts do not reproduce the respective animation perfectly. C is the number of correct annotations, W is the number of incorrect annotations, M is the number of missing annotations and WP is the number of annotations that were incorrectly placed. Incorrectly placed annotations are correct in essence, i.e. they describe the correct control structure but would lead to a wrong nesting of control structures in the final program.

The numbers in the table summarize all texts per animation. Usually every element of a control structure needs one annotation: E.g. every action in a loop needs an annotation that indicates which loop the action belongs to. If a text calls for a loop with three actions and CSR detects the loop but misses one action the numbers are as follows: In total one needs three annotations (one for each action); two count as correct, one as missing. The number of times the loop should be executed is encoded as an attribute of the loop annotation and therefore needs no separate annotation.

For a realistic evaluation one has to include the parsing errors as they would remain in the input during normal operation. As one can see in the first block of Table III CSR makes few mistakes if parser errors remain in the input but 18% of the annotations are missing. The percentage of correct annotations is 82% and we were able to improve this number up to 89% with a tuned configuration. Against our assumptions, tuning increased the number of wrong annotations only by one. The number of missing annotations can be cut down to 10%.

The performance in the absence of parsing errors is almost perfect: Only 3% of the annotations are missing and only five incorrect annotations have been made. As one can see, the results for the Beach animation improved significantly: In the respective texts, many parser errors surfaced; after the manual correction, 97% of the annotations are correct. Only two incorrect annotations are added to the text and only six of 221 annotations are missing. In the Beach texts the parser tagged many verbs as nouns which lead to the lion's share of errors. The uncorrected Farm texts contain much less parser errors; this is why the results can hardly be improved. This underlines that our approach performs very well under ideal circumstances but is sensitive when it comes to tagging errors in the input.

V. CONCLUSION

Control structures are essential for programming and therefore have to be available when programming in natural language. We introduce an analysis that uses key phrases, POS tags, and Stanford's typed dependencies to identify control structures in English texts. Our experience supports

[1]https://svn.ipd.kit.edu/trac/AliceNLP/wiki/Papers/LH2015

the findings of Pane and Myers. They report that users tend to describe actions set-wise when programmers would use a loop. Therefore our analysis recognizes terms that represent a group and uses an ontology to dereference the group; enumerations are also supported. Our analyses are implemented in a tool called Control Structure Analyzer (CSR) and are almost fully automatic; only if CSR cannot dereference a group (or "set" to use the terminology of Pane and Myers), it asks the user which objects of the program should be included in the group.

We evaluated our approach in the domain of Alice, a programming environment where (usually non-expert) users can program 3D animations. The evaluation considers 52 texts written by over 30 subjects. CSR produced correct information 82% of the time. We also created a configuration that is more robust to parser errors; it improved the performance to 89%. If the input contains no parer errors, the performance with the default configuration is as high as 97%.

There is still room for improvement: CSR does not yet treat *if/else* constructions; the information needed to derive *if/else* from a text is present but an automatic analysis waits to be implemented in CSR. Also the evaluation leaves room for improvement: We have a collection of 95 texts that could be used for the evaluation but as of today only 52 are used.

References

[1] J. E. Sammet, "The use of english as a programming language," *Commun. ACM*, vol. 9, no. 3, pp. 228–230, Mar. 1966. [Online]. Available: http://doi.acm.org/10.1145/365230.365274

[2] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty, "Building watson: An overview of the DeepQA project," *AI Magazine*, vol. 31, no. 3, pp. 59–79, 2010. [Online]. Available: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2303

[3] J. R. Bellegarda, "Spoken language understanding for natural interaction: The siri experience," in *Natural Interaction with Robots, Knowbots and Smartphones*, J. Mariani, S. Rosset, M. Garnier-Rizet, and L. Devillers, Eds. Springer New York, 2014, pp. 3–14.

[4] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 55–60. [Online]. Available: http://www.aclweb.org/anthology/P/P14/P14-5010

[5] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen, "Alice: Lessons learned from building a 3d system for novices," in *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*. The Hague, The Netherlands: ACM Press, 2000, pp. 486–493.

[6] M. Landhäußer, T. Hey, and W. F. Tichy, "Deriving timelines from texts," in *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, Jun. 2014, pp. 45–51.

[7] M.-C. de Marneffe and C. D. Manning, "Stanford typed dependencies manual," Tech. Rep., Dec. 2013.

[8] B. W. Ballard and A. W. Biermann, "Programming in natural language: NLC as a prototype," in *Proceedings of the 1979 annual conference*, ser. ACM '79. New York, NY, USA: ACM, 1979, pp. 228–237.

[9] A. W. Biermann and B. W. Ballard, "Toward natural language computation," *Comput. Linguist.*, vol. 6, no. 2, pp. 71–86, Apr. 1980. [Online]. Available: http://dl.acm.org/citation.cfm?id=972439.972440

[10] D. Price, E. Rilofff, J. Zachary, and B. Harvey, "NaturalJava: A natural language interface for programming in java," in *Proceedings of the 5th international conference on Intelligent user interfaces*, ser. IUI '00. New Orleans, Louisiana, USA: ACM, 2000, pp. 207–211. [Online]. Available: http://doi.acm.org/10.1145/325737.325845

[11] J. F. Pane, C. Ratanamahatana, and B. A. Myers, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237–264, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6WGR-458NDXY-M/2/e785863fea0552236a5587e1fbe7a57f

[12] H. Liu and H. Lieberman, "Programmatic semantics for natural language interfaces," in *CHI '05 extended abstracts on Human factors in computing systems*, ser. CHI '05. Portland, OR, USA: ACM, 2005, pp. 1597–1600.

[13] ——, "Metafor: Visualizing stories as code," in *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces*. San Diego, California, USA: ACM, 2005, pp. 305–307. [Online]. Available: http://portal.acm.org/citation.cfm?id=1040830.1040908

[14] R. Mihalcea, H. Liu, and H. Lieberman, "NLP (natural language processing) for NLP (natural language programming)," in *Proceedings of the 7th International Conference, CICLing 2006, Mexico City, Mexico, February 19-25, 2006*, ser. Lecture Notes in Computer Science, A. Gelbukh, Ed., vol. 3878. Berlin, Heidelberg: Springer, 2006, pp. 319–330.

[15] R. Knöll and M. Mezini, "Pegasus: First steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '06. ACM, 2006, pp. 542–559.

[16] J. Pustejovsky, R. Knippen, J. Littman, and R. Saurí, "Temporal and event information in natural language text," *Language Resources and Evaluation*, vol. 39, no. 2-3, pp. 123–164, May 2005. [Online]. Available: http://link.springer.com/article/10.1007/s10579-005-7882-7

[17] F. Schilder, "Event extraction and temporal reasoning in legal documents," in *Annotating, Extracting and Reasoning about Time and Events*, ser. Lecture Notes in Computer Science, F. Schilder, G. Katz, and J. Pustejovsky, Eds. Springer, Jan. 2007, no. 4795, pp. 59–71.

[18] H. J. Ohlbach, "Computational treatment of temporal notions: The CTTN–system," in *Annotating, Extracting and Reasoning about Time and Events*, ser. Lecture Notes in Computer Science, F. Schilder, G. Katz, and J. Pustejovsky, Eds. Springer Berlin Heidelberg, Jan. 2007, no. 4795, pp. 72–87.

[19] I. Mani, M. Verhagen, B. Wellner, C. M. Lee, and J. Pustejovsky, "Machine learning of temporal relations," in *ACL-44: Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Sydney, Australia: Association for Computational Linguistics, 2006, pp. 753–760. [Online]. Available: http://portal.acm.org/citation.cfm?id=1220175.1220270

[20] N. Chambers, S. Wang, and D. Jurafsky, "Classifying temporal relations between events," in *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ser. ACL '07. Prague, Czech Republic: Association for Computational Linguistics, 2007, pp. 173–176. [Online]. Available: http://dl.acm.org/citation.cfm?id=1557769.1557820

[21] A. K. Kolya, A. Ekbal, and S. Bandyopadhyay, "A first step towards evaluating events, time expressions and temporal relations," in *Proceedings of the 5th International Workshop on Semantic Evaluation*, ser. SemEval '10. Los Angeles, California: Association for Computational Linguistics, 2010, pp. 345–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1859664.1859741

[22] A. Berglund, R. Johansson, and P. Nugues, "A machine learning approach to extract temporal information from texts in swedish and generate animated 3d scenes," in *Proceedings of EACL-2006, 11th Conference of the European Chapter of the Association for Computational Linguistics*. Trento, Italy: Association for Computational Linguistics, 2006, pp. 385–392.

[23] M. Lapata and A. Lascarides, "Inferring sentence-internal temporal relations," in *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004*, Jan. 2005, pp. 153–160. [Online]. Available: http://aclweb.org/anthology/N04-1020

[24] P. Nemec, "Automatic analysis of temporal relations within a discourse," in *14th International Symposium on Temporal Representation and Reasoning*, Jun. 2007, pp. 117–128.

[25] S. K. Sanampudi and G. V. Kumari, "Temporal reasoning in natural language processing: A survey," *International Journal of Computer Applications*, vol. 1, no. 4, pp. 68–72, Feb. 2010.

[26] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, Dec. 2009.