lips: An IDE for Model Driven Engineering Based on Natural Language Processing

Oliver Keszocze[1] Mathias Soeken[1,2] Eugen Kuksa[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{keszocze,msoeken,eugenk,drechsle}@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/agra/eng/lips.php

***Abstract*—Combining both, state-of-the art natural language processing (NLP) algorithms and semantic information offered by a variety of ontologies and databases, efficient methods have been proposed that assist system designers in automatically translating text-based specifications into formal models. But due to ambiguities in natural language, these approaches usually require user interaction. Following these achievements, we consider natural language as a further input language that is used in the design flow for systems and software. Consequently, concepts from integrated development environments (IDE) as they can be found for programming languages such as Java need to be made available for natural language specifications as well.**

In this paper, we propose lips, an integrated development environment that is seamlessly implemented on top of Eclipse. It contains recent NLP algorithms that extract formal models suited for the Eclipse Modeling Framework and therefore provide a starting point for an ongoing implementation. Whenever user interaction is required, lips makes use of well-known IDE concepts such as markers and quick fixes thereby enabling a holistic user experience.

I. Introduction

In the recent past, several approaches have been presented that assist the designer in extracting an implementation from a specification by making use of natural language processing (NLP) techniques (see e.g. [1]). It turned out that these approaches are in particular successful when an interactive dialog with the designer is employed in contrast to fully automatic approaches. Latter ones are often only applicable after several restrictions have been applied to the input document, e.g. by means of a domain specific dictionary or a restricted set of sentence forms that are allowed (see e.g. [2]).

Interactive approaches [3] do not have to restrict the input document because they can overcome obstacles such as ambiguities by systematically asking questions to the designer in order to resolve them. However, it is important that this additional overhead caused by this "conversation" is not getting in the way of creating an implementation and depreciates the efficiency. A clever implementation of the dialog system is therefore of paramount importance.

We envision a design flow for systems and software in which natural language is treated as a further input language for the specification level just as modeling languages are used for formal modeling and programming languages are used for the actual implementation. This is also illustrated by means of Fig. 1, which shows the first abstraction levels in the *Model Driven Engineering* (MDE, [4]) based design flow along with typical languages used to describe the respective artifacts. However, the way of working with these languages is fundamentally different. Whereas sophisticated *Integrated Development Environments* (IDE) exist for modeling and developing software programs, natural language has often been entered using a word processor.

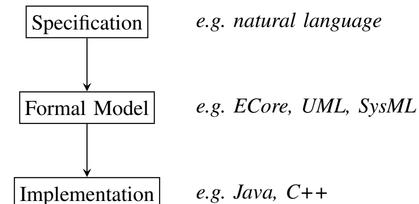


Fig. 1: MDE-based design flow

Various platforms and development environments tailored for natural language processing have been proposed over the years. These systems often focus only on the linguistic aspects and put sophisticated natural language processing techniques in the focus. However, dedicated integrated systems that extend the conventional MDE-based design flow have not been proposed thus far to the best of our knowledge.

In this paper, we propose lips, an integrated development environment particularly suited for natural language processing in the context of MDE-based systems and software development. Instead of using a word processor, well known concepts that are used in program development such as markers, syntax highlighting, and outlines are mapped and adjusted for the use with natural language documents.

For this purpose, we have implemented recently proposed interactive NLP algorithms that extract both structural models and formal expressions from natural language specifications as an Eclipse[1] plugin. The user interaction with the algorithms is enabled by means of common IDE concepts leading to a holistic user experience. The above NLP algorithms are adjusted such that they produce models that are compatible with the *Eclipse Modeling Framework* (EMF, [5]). Hence, the extracted model elements serve as a starting point for the ongoing implementation.

[1]www.eclipse.org

978-1-4673-6271-9/13 © 2013 IEEE

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

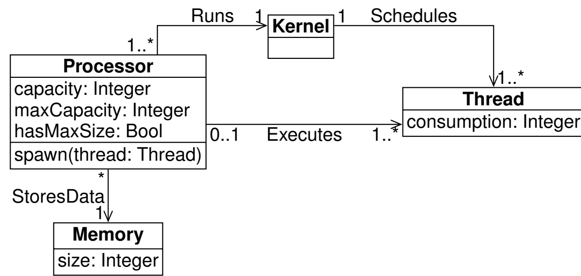


Fig. 2: ECore model depicted as ECore diagram

```
1. <noun.person> waiter, server
-- (a person whose occupation is to serve at table (as
in a restaurant))
2. <noun.person> server1
-- ((court games) the player who serves to start a point)
3. <noun.artifact> server1, host
-- ((computer science) a computer that provides client
stations with access to files and printers as shared
resources to a computer network)
4. <noun.artifact> server -- (utensil used in serving food
or drink)
```

Fig. 3: WordNet output of the query for “server”

The remainder of this paper is structured as follows. Section II provides the necessary background in order to comprehend the paper, whereas NLP algorithms that have been implemented in lips are reviewed in Section III. The features of the IDE are presented in Section IV before the paper finishes with a related work discussion and conclusions in Sections V and VI, respectively.

II. Preliminaries

A. Model Driven Engineering

The main purpose of lips is to transform natural language descriptions into models as they appear in *Model Driven Engineering* based design flows. In MDE, models embody the central elements during the design process. Instead of working exclusively with source code, the most important aspects and properties of the software or system to be implemented are represented by means of so-called domain or meta models. They abstract from precise implementation details, but are already expressive enough to perform many interesting tasks on this meta level such as model transformation [6], formal verification [7], or automatic test case generation [8].

The MDE paradigm has been standardized by means of the *Meta Object Facility* (MOF, [9]), a meta modeling architecture which serves as base for many modeling languages and frameworks such as the *Unified Modeling Language* (UML, [10]), the *Object Constraint Language* (OCL, [11]), or *ECore* which is part of the *Eclipse Modeling Framework*.

The algorithms and methods presented in this paper are implemented on top of the EMF, as it is currently one of the most popular meta modeling frameworks [12]. The central element in an ECore model are classes, which can contain attributes that represent the data elements of the class. Besides attributes, operations can be defined which can access and modify the attributes. Classes can be related to each other by means of associations. For precise modeling, the formal declarative language OCL can be used which allows for extending ECore models by means of invariants, pre- and postconditions. Invariants constrain the attributes and associations of classes globally, whereas pre- and postconditions are associated to operations, such that the preconditions and postconditions are valid before and after the operation call, respectively.

Example 1: Fig. 2 illustrates the specification of a simple computer architecture in ECore using the ECore diagram notation that is following the style of UML class diagrams. The structure of the system is defined by means of four classes, namely a *Processor*, a *Kernel*, a *Thread*, and a *Memory*. Attributes such as *maxCapacity* provide further details on the respective components (e.g. the maximal capacity of the processor).

B. Natural Language Processing

The lips IDE combines algorithms that use different natural language processing techniques which are briefly reviewed in this section.

1) Word Sense Disambiguation: When the correct sense of a word must be determined, *Word Sense Disambiguation* (WSD, [13]) comes into play. Given e.g. the sentence “The server delivers the website,” it is easy for a human to identify “server” as a technical device. For a computer, in contrast, it is an impossible task to determine the correct sense with no additional information. Thinking in terms of an MDE context, it is not clear whether “server” describes a class which is part of the model or an actor that interacts with the model.

In case of ambiguity, lips uses WordNet [14] for dictionary-based word sense disambiguation [15]. WordNet is a lexical dictionary of English, consisting of more than 90,000 word senses and 166,000 pairs connecting senses with a semantic meaning. For many senses, WordNet also provides example sentences. It is designed to be used by external programs, such as automated scripts or IDEs such as lips. Fig. 3 displays the results of a WordNet query for the word “server”.

2) Constituency Grammars: A constituency grammar [16] is used to decompose a sentence into its constituent parts, usually depicted as a phrase structure tree (PST, see Fig. 4). A PST is a tree whose root is the most general phrase structure, in case of Fig. 4 it is *S*, the whole sentence. The leaves of the tree are the words of the sentence. Moving along the branches from the root to the leaves, the vertices become more specialized phrase structures. Following the leftmost branch from the example shown in Fig. 4 we get the following structures: *S* (sentence) $\rightarrow$ *NP* (noun phrase) $\rightarrow$ *DT* (determiner) $\rightarrow$ “The” (word within the sentence). The parent of a leaf corresponds to the part-of-speech (POS, or tag) of the leaf, i.e. “The” and “the” are determiners, “server” and “website” are nouns and “delivers” is a verb. For details on how a PST is extracted from a sentence, we refer to [15] and [17].

3) Dependency Grammars: In order to represent dependencies between individual words, natural language processing techniques make use of dependency parses [18], i.e. binary

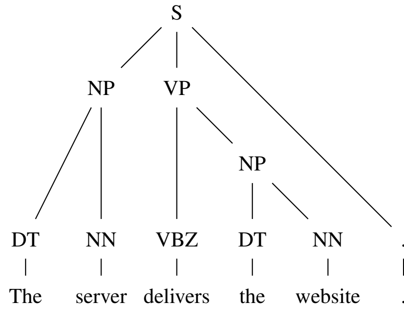
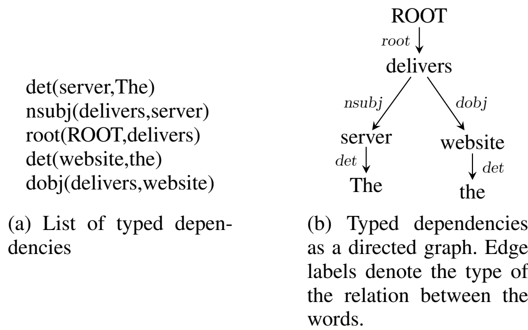


Fig. 4: Phrase structure tree (PST) of the sentence "The server delivers the website."



(a) List of typed dependencies

(b) Typed dependencies as a directed graph. Edge labels denote the type of the relation between the words.

Fig. 5: Typed dependencies of the sentence "The server delivers the website.".

dependency relations are extracted from the sentences. As an example the relation *nsubj* binds a verb to its subject. The usual notation for this relation is *relation(governor,dependent)*, e.g. *nsubj(delivers,server)*. To avoid ambiguities, the position of the word within the sentence can be appended to the words of the relation (i.e. *nsubj(delivers-3,server-2)*, see Fig. 5(a)).

The typed dependencies of a sentence *s* can be understood as an edge-labeled graph whose vertices represent words and labels the type of the dependency. There is an edge $v \xrightarrow{r} w$ if and only if $r(v, w)$ is a dependency of s. For a visualization see Fig. 5(b).

III. Interactive NLP Extraction

We aim for a design flow in which natural language is the starting point not only for specifications but also for (semi-)automatic algorithms. For this purpose, in the first stage we address approaches that extract a formal model from natural language descriptions in an interactive fashion. Two existing approaches that target the extraction of structural models and OCL expressions are briefly reviewed in this section. For lips, they have been adjusted in order to generate models that align with the Eclipse Modeling Framework.

A. Extracting Structure

Among other techniques, the algorithm described in [19] is aiming for extracting UML class diagrams and UML sequence diagrams from acceptance tests which can be found

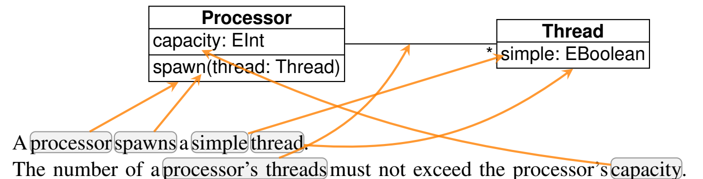


Fig. 6: Extracting models

in *Behavior-Driven Development* (BDD, [20]) based design flows. The considered UML class diagrams are so elementary that they can be mapped to equivalent ECore models. In the scope of the present paper, we only focus on the structure extraction by means of ECore models.

From simple sentences much information can already be determined automatically. As an example, consider the following specification excerpt describing processors and threads:

> *A processor spawns a simple thread.*
> *The number of a processor's threads must not exceed the processor's capacity.*

Fig. 6 illustrates that already from these two sentences a significant amount of structural information can be extracted: Since "thread" is an object noun, it can be concluded that it represents a component of the considered system (to be represented by a class). Recent progress in the development of NLP technologies enables to extract these information in a (semi-)automatic manner. More precisely, constituency grammars and corresponding phrase structure trees provide first insights for the classification of words with respect to the model to be created. However, sometimes the syntactical and grammatical information alone is not sufficient. For example in the first sentence from Fig. 6, two nouns are identified in the PST, i.e. "processor" and "thread," but only the latter one is determined to be a class. Due to ambiguities in language, a "processor" can also be a person. This information cannot be derived using constituency grammars. Hence, we are making additionally use of word sense disambiguation using WordNet, which reveals that "processor" can be both a person or an artifact.

In order to determine the correct sense, lips uses the *Simplified Lesk* algorithm [13]. The algorithm chooses the sense whose gloss and example sentences provided by a dictionary have the most words in common with the sentence containing the word in question.

If the determined sense does not allow to automatically classify the noun as either class or actor, the designer is interactively asked to clarify this ambiguity.

Similarly, other information can be extracted from the sentences. Adjective nouns such as "capacity" shall be added as attributes to the corresponding class. Verbs correlate to operations which can be invoked by components or actors.

Overall, exploiting these NLP technologies, ECore models formally representing the structure of the considered system can automatically be determined in many cases. However, since the textual description can always contain ambiguities, manual interactions with the design engineer cannot entirely be

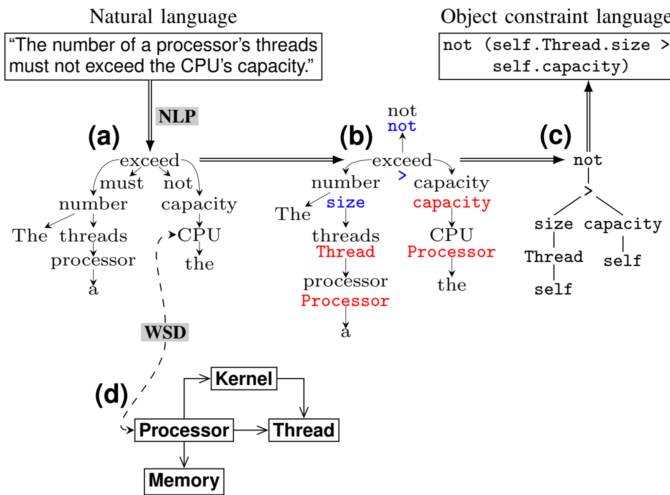


Fig. 7: Extracting OCL constraints

excluded leading to a (semi-)automatic and assisted approach as evaluated in [19].

B. Extracting OCL Constraints

When a model is available, e.g. determined using methods described in the previous section, formal behavioral specifications can be extracted from English sentences using the approach presented in this section. More precisely, the goal is to support the designer in creating a formal specification in OCL from given informal natural language requirements. During the generation process, it is exploited, that despite the undoubtedly existing differences, the given input (i.e. the sentence in natural language) and the desired output (i.e. the formal requirement in OCL) are indeed quite similar. While this is often not evident in a direct comparison, structural analyses unveil the correlation between the input and the output. This is illustrated in the following example.

Consider the informal requirement "The number of a processor's threads must not exceed the CPU's capacity" and its formal counterpart

```
not (self.Thread.size > self.capacity).
```

A direct mapping of these two descriptions (see the boxes at the top of Fig. 7) is not straightforward. However, after a prior application of semantical and grammatical analyses followed by a normalization, a promising representation can be obtained as shown in Fig. 7 **(a)** through **(c)**. In fact, the resulting normalized dependency graph of the sentence (see Fig. 7(**b**)) is almost identical to the resulting abstract syntax tree (AST) of the OCL constraint (see Fig. 7(**c**)).

However, the example in Fig. 7 also shows that, due to the wide scope of natural language, a direct mapping of *all* parts of the informal requirement to the appropriate identifier or OCL operations is not guaranteed. Often different grammatical forms of words (e.g. due to declension or conjugation) or the use of synonymous descriptions represent obstacles to a one-to-one mapping from the dependency graph to the AST. Dictionary-based word sense disambiguation can be applied to address these problems. Using this technique, normal forms and synonymous identifiers are determined.

With respect to the example in Fig. 7, while e.g. *not* and *capacity* can easily be mapped from the dependency graph to the corresponding OCL expression (highlighted in blue color) or model element (highlighted in red color), respectively, a correct mapping of *CPU* is not obvious at a first glance. However, the application of WSD unveils that among others the word "CPU" is a synonym for "processor" (for a visualization of the WSD process see Fig. 7(**d**)). Since *Processor* is a class in the ECore model, it can be assumed that "CPU" is just an alternative description of "processor" in the informal description. Hence, substituting both words does not affect the meaning of the requirement, but enables a correct mapping from the informal requirement to the formal requirement.

IV. Integrated Development Environment

We have implemented lips as a plug-in on top of the Eclipse IDE. The *Eclipse plug-in development environment* (PDE) provides a systematic way of adding new languages and features by offering interfaces to various IDE concepts.

This section lists the program features that are implemented in lips and partly illustrated in Fig. 8.

Code Generation: Code generation is a concept that enables to automatically produce output files without explicit compiling. An example for this is the programming language Xtend[2]: Java code is automatically generated during the editing of source files. We are making use of code generation in order to implement the natural language processing techniques that have been presented in [19] and [21] in order to extract ECore models and OCL expressions, respectively.

In particular, while writing natural language specifications, EMF-compatible models are automatically generated in designated code generation folders. From these models code skeletons can be generated for any object oriented programming language. The resulting models can be seen in Fig. 8 **(2)**. Eclipse also provides means to create a diagram from ECore models, allowing to visually inspect the extracted models.

Furthermore, OCL is generated from both natural language sentences and the underlying ECore model that has been generated in the previous step. When a part of a sentence cannot be resolved automatically, the user needs to pick a candidate from a list of choices.

Project Management: In IDEs source files are organized in terms of projects that store additional information on top of the program code such as build parameters and user-defined settings of the editor. When dealing with natural language documents and in particular their algorithms it is important to store auxiliary data, e.g. user responses to questions in the dialog system.

Outline: An outline is used to display the structure of a document. In lips we display the list of all sentences in the specification and their syntactical structure (i.e. the phrase

[2]www.eclipse.org/xtend

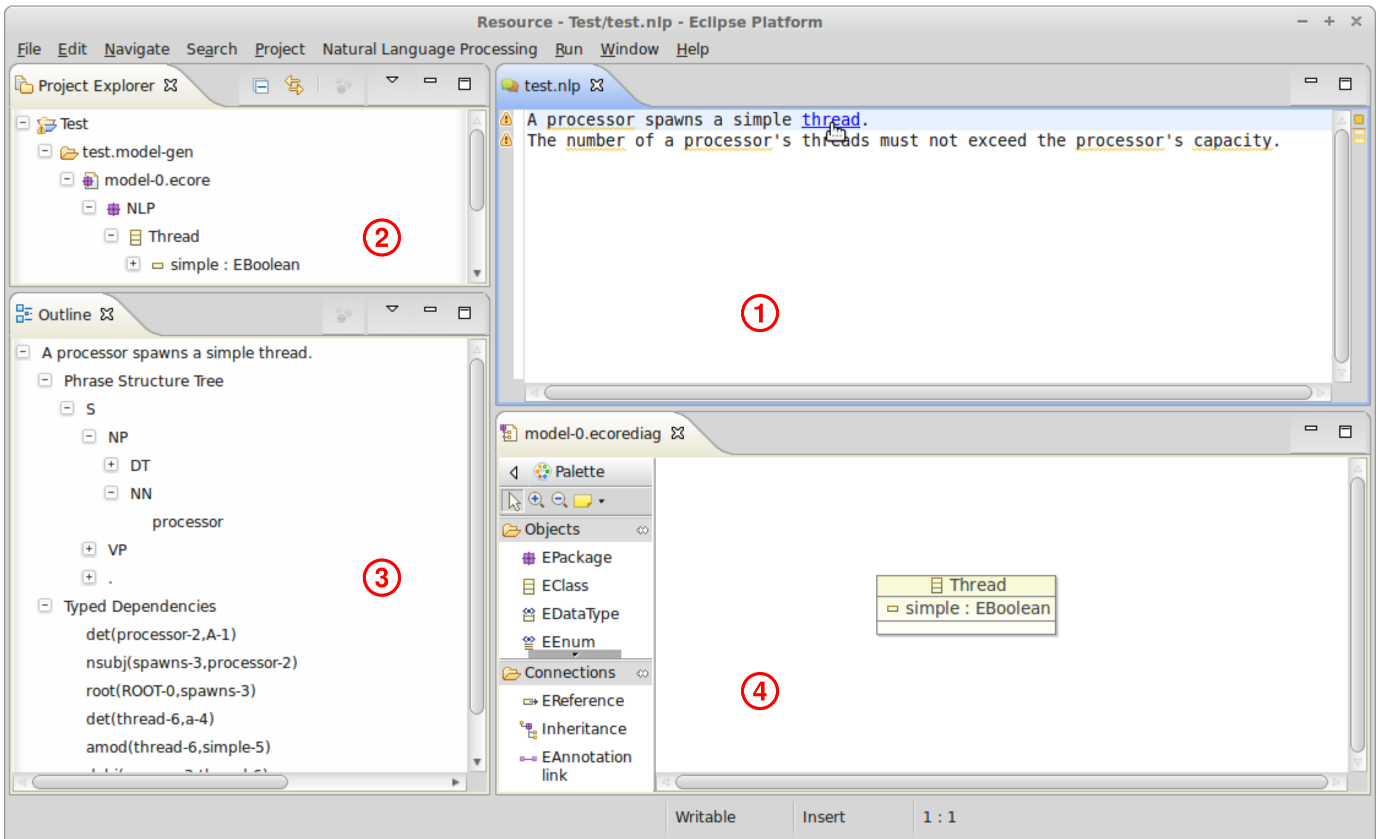


Fig. 8: Screenshot of lips with **(1)** *editor*, **(2)** *project explorer*, **(3)** *outline*, and **(4)** *ECore diagram*

structure tree [17] and the typed dependencies [15]). The outline for an example sentence is depicted in Fig. 8 **(3)**.

Views: A view is a window within the Eclipse IDE that provides additional information to the designer, possibly allowing to modify its content. In lips we use views to

- visualize the phrase structure tree and the typed dependencies as graphs, allowing the user to get a better understanding of the sentence and the underlying syntactical structure. The PST as well as the dependency graph illustrate how a sentence has been understood by the algorithm. This is in particular helpful for debugging purposes in case of ambiguous sentences or unexpected results. Examples of these views are illustrated in Fig. 10;
- display the generated ECore model (see. Fig. 8 **(4)**) and OCL expressions;
- display and modify the context information either automatically generated by lips or directly entered by the user (e.g. that the word “server” refers to the technical device and not a waiter at a restaurant). This view is depicted in Fig. 9 **(3)**;
- display the WordNet output for a given query as depicted in Fig. 9 **(4)**.

The user of lips is in complete control of the views: they can be arbitrarily moved and resized to fit the actual needs of the user.

Markers: Markers are used to underline code fragments that correspond to errors and warnings e.g. given by the compiler. Markers can be additionally equipped with so-called quick fixes, which allow for an automatic resolution of the respective issue. Errors could e.g. point to a missing include or import statement, which can automatically be inserted by a quick fix. We foresee to use markers and quick fixes as major elements to implement an unobtrusive dialog system in the interactive NLP algorithms. Ambiguities are presented to the designer as markers at the respective word with an additional information message. The designer can respond to this problem in terms of a quick fix, which automatically causes an action resolving the problem. In Fig. 8 **(1)**, the word “processor” cannot clearly be assigned to the model, hence it is annotated with a warning. Every marker also appears in the Eclipse “Problems” view (see Fig. 9 **(1)**). This gives an overview of all problems at once. This is especially helpful when working with specifications that span multiple files. When asking lips for a quick fix, a new window with the list of possible solutions is opened (see Fig. 9 **(2)**). WordNet senses for the word in question are used to generate suitable proposals for the word classification.

Hyperlinks: Once models and OCL expressions have been extracted from the natural language specification using code generation, they can be linked to the original document by means of hyperlinks. As an example, if some noun in a sentence corresponds to a class in a model, a click on the

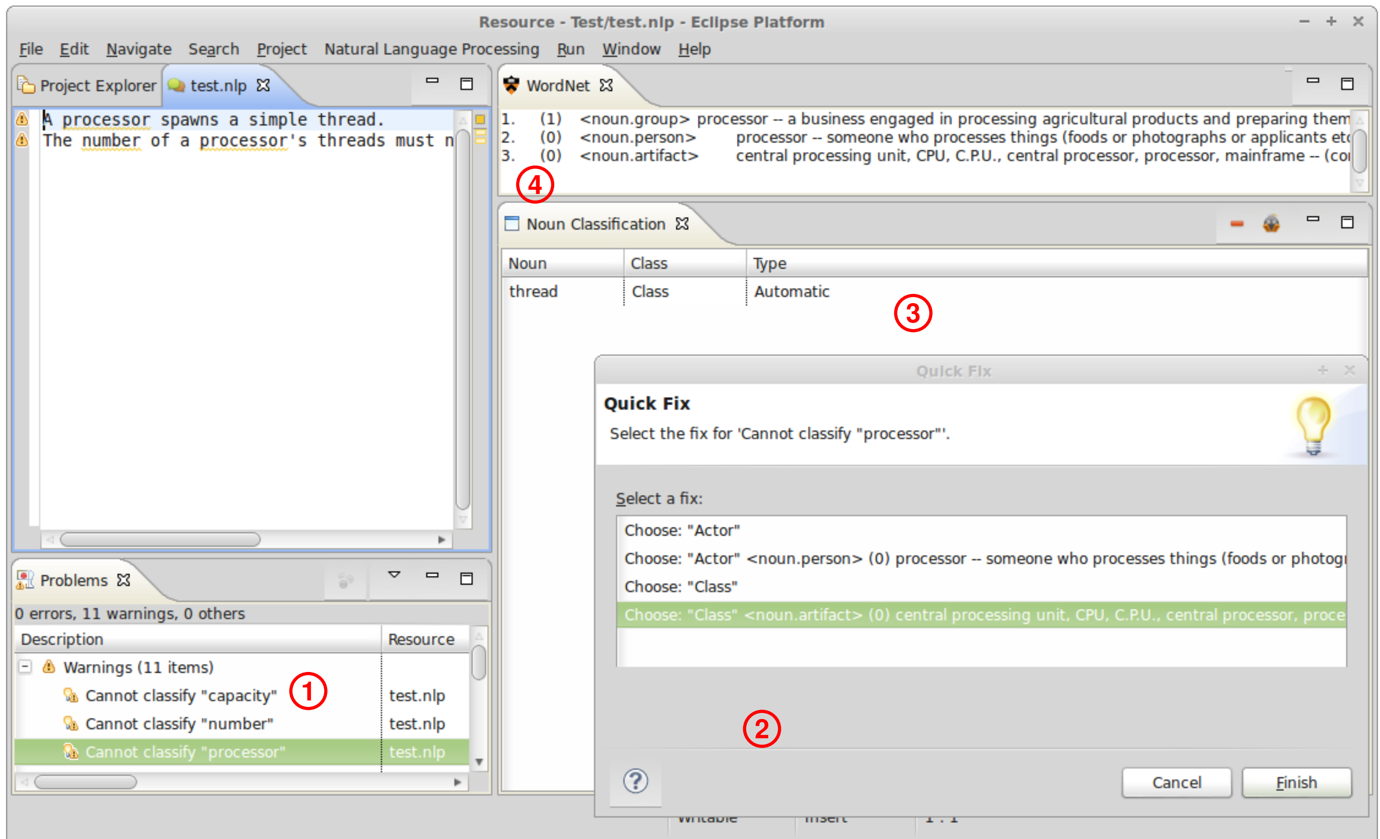


Fig. 9: Screenshot of lips with **(1)** Problems view, **(2)** Quick Fix Window, **(3)** a view showing all classified words, and **(4)** WordNet view

noun opens the model and focuses the class (this is shown in Fig. 8 **(1)**).

V. Related Work

The automatized extraction of information for software from natural language specifications has been intensively studied in the past. Saeki et al. [22] presented an approach similar to [19] that associates words in English sentences to software concepts found in the object oriented programming paradigm. The extraction of OCL constraints from English sentences has been considered in [23] and [21]. Most of the considered approaches are aiming for fully automatic methods and restrict the input language to achieve this goal. Some approaches make use of so-called boilerplates [24] or determine recurring patterns in property specifications [25]. A more general approach is followed by the Attempto controlled English which is a restricted subset of English that can automatically be transformed into formal descriptions such as Prolog [2].

Integrated development environments for natural language processing have been considered for different purposes in the past. Some IDEs are settled at a lower level and focus on the internals of natural language and are targeting linguists as audience rather than software and system developers. Among them, the INTEX platform [26] is tailored for linguist to describe natural language in general, whereas LexGram [27] focuses on categorial grammars. The PAGE system[3] offers a collection of various linguistic resources.

The IDEs that are most related to lips are GATE [28], VisualText[4], and NL-OOPS [29]. However, GATE focuses on a common system for NLP related tasks and VisualText is specialized in the development of user-defined text analysis applications. In contrast, lips does not put the NLP techniques into the foreground but uses them in the background thereby aiming at integrating natural language as a major language for the design flow of software and systems. NL-OOPS is the approach closest to lips, however, it does not emphasize on the integration with other modeling and programming languages in a common IDE.

VI. Conclusions

In this paper, we presented lips, an IDE that integrates natural language as a major language for specifications in the manner modeling languages and programming languages are used to describe a formal model and an implementation, respectively. In particular this works very well when targeting interactive NLP algorithms which do not restrict the input language but occasionally require the user to manually intervene in order to resolve ambiguities. We are mapping widely known IDE concepts such as outlines, markers, views, and

[3] www.dfki.de/lt/systems/page [4] www.textanalysis.com

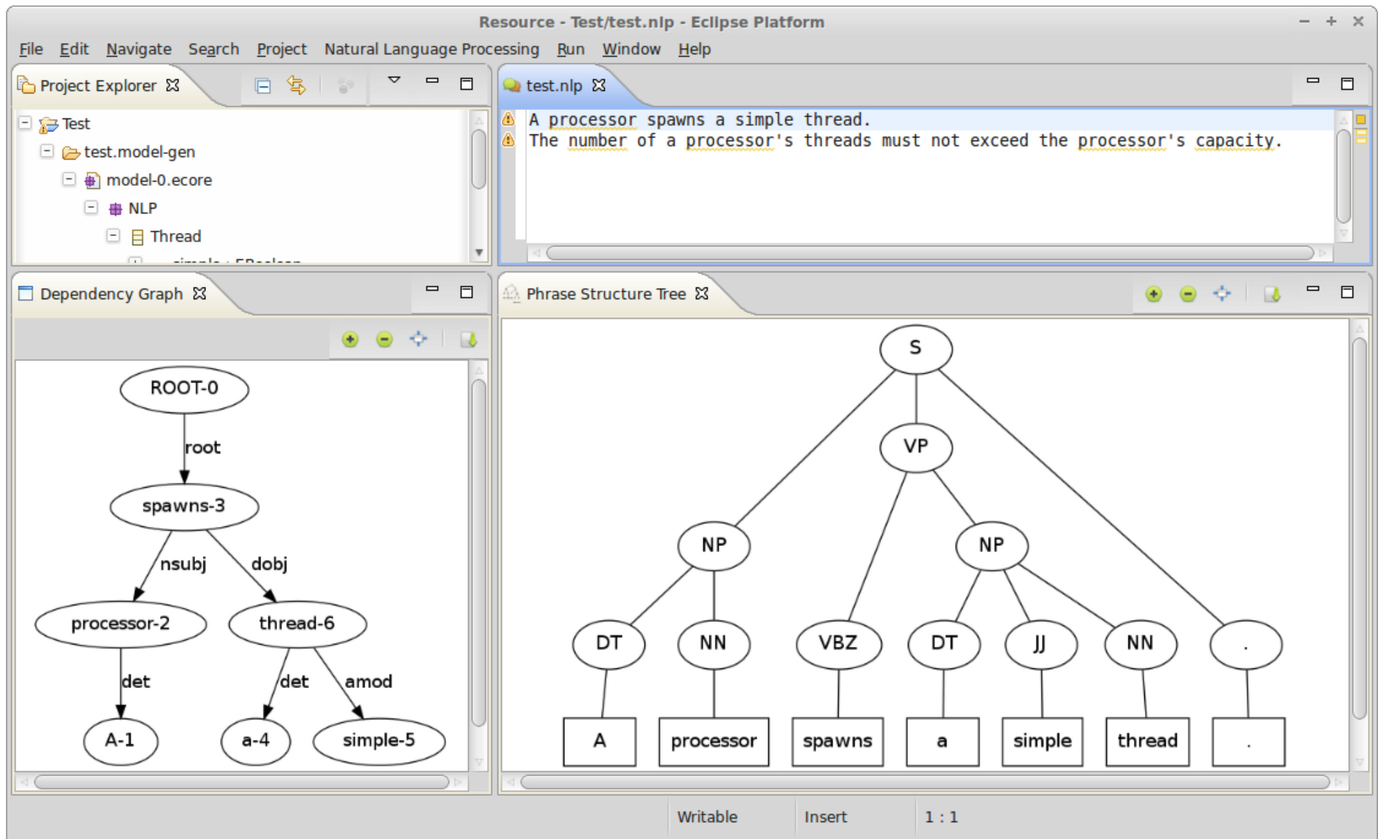


Fig. 10: Screenshot of lips's view showing the phrase structure tree and the dependency graph

quick fixes to implement a dialog system that realizes the user interaction. As a result, a seamless user experience results in which systems and software can be developed from the initial specification down to the implementation in a single the environment.

Furthermore, lips also serves as a development platform on which MDE-related NLP algorithms can be implemented and evaluated. In particular, the quick access to useful information such as WordNet, phrase structure trees, and dependency graphs turns out to be very useful.

However, some open research challenges remain which we would like to tackle in future works. As an example, the incorporation of additional databases, in particular those from the internet, is not straightforward. Furthermore, a clever management of the user acquired data is important in order to make the information available not only for the current project but also for future projects of similar kind.

Acknowledgments

This work was supported by the German Research Foundation (DFG) (DR 287/23-1).

References

[1] R. Drechsler, M. Soeken, and R. Wille, "Formal Specification Level: Towards Verification-driven Design Based on Natural Language Processing," in *Forum on Specification & Design Languages*, 2012, pp. 53–58.

[2] N. E. Fuchs and R. Schwitter, "Attempto Controlled English (ACE)," in *Int'l Workshop on Controlled Language Applications*, 1996.

[3] R. Drechsler, M. Soeken, and R. Wille, "Towards Dialog Systems for Assisted Natural Language Processing in the Design of Embedded Systems," in *Int'l Design and Test Workshop*, 2012.

[4] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, 2008.

[6] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003.

[7] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[8] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.

[9] *OMG Meta Object Facility (MOF) Core Specification*, Object Management Group, 2011, version 2.4.1.

[10] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*. Addison-Wesley Longman, 1999.

[11] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise modeling with UML*. Boston, MA, USA: Addison-Wesley Longman, 1999.

[12] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "EMF Profiles: A Lightweight Extension Approach for EMF Models," *Journal of Object Technology*, vol. 11, no. 1, pp. 8:1–29, 2012.

[13] A. Kilgarriff and J. Rosenzweig, "Framework and results for english senseval," *Computers and the Humanities*, vol. 34, pp. 15–48, 2000. [Online]. Available: http://dx.doi.org/10.1023/A%3A1002693207386

[14] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[15] D. Jurafsky and J. H. Martin, *Speech and Language Processing*. Pearson Prentice Hall, 2008.

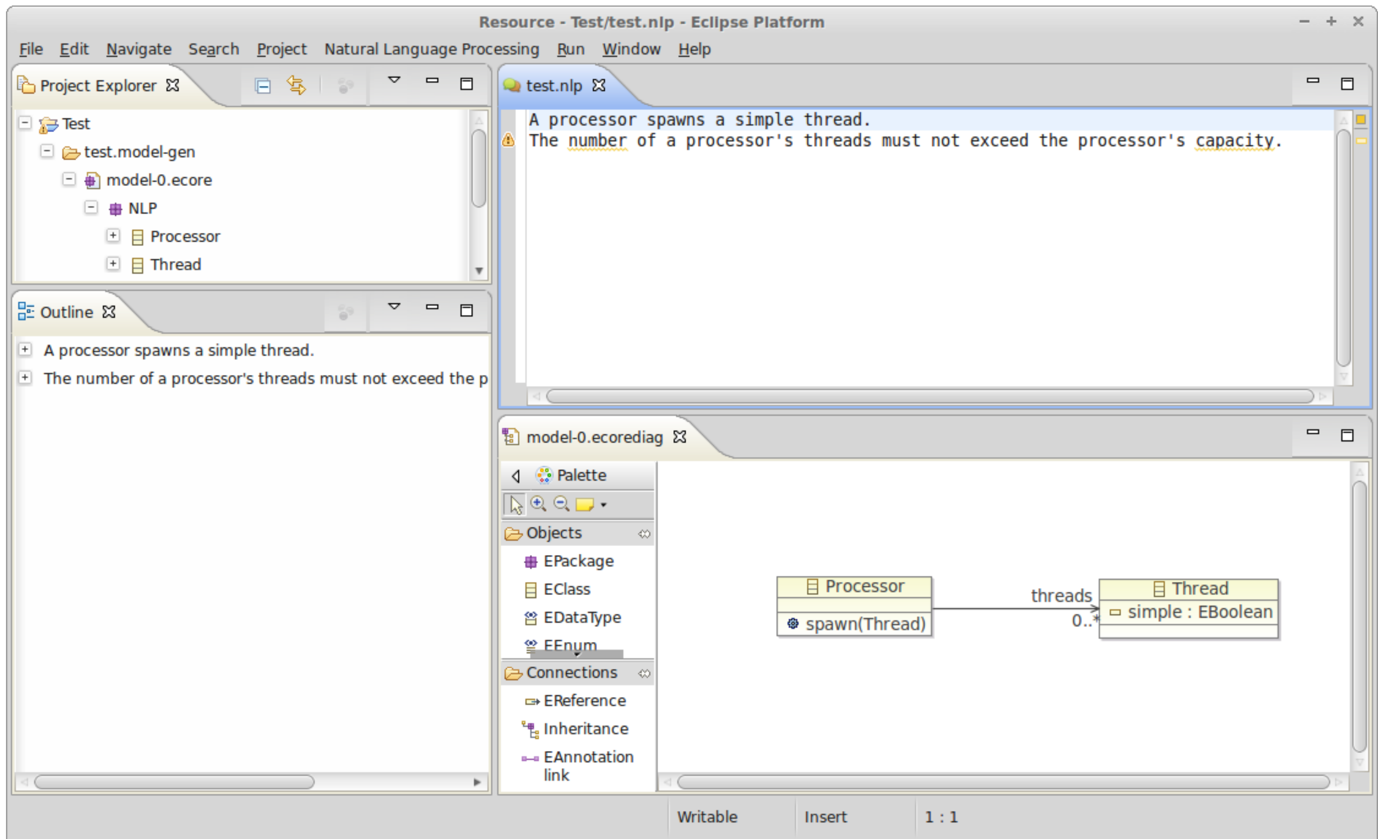


Fig. 11: Screenshot of lips showing the resulting ECore model after the problem resolution for the first sentence

[16] N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.
[17] A. Carnie, *Syntax: a generative introduction*. Blackwell, 2007.
[18] M.-C. de Marneffe, B. MacCartney, and C. D. Manning, "Generating Typed Dependency Parses from Phrase Structure Parses," in *Conf. on Language Resources and Evaluation*, 2006, pp. 449–454.
[19] M. Soeken, R. Wille, and R. Drechsler, "Assisted Behavior Driven Development Using Natural Language Processing," in *Int'l. Conf. on Objects, Models, Components, Patterns*, 2012, pp. 269–287.
[20] D. North, "Behavior Modification: The evolution of behavior-driven development," *Better Software*, vol. 8, no. 3, 2006.
[21] M. Soeken, R. Wille, E. Kuksa, and R. Drechsler, "Supporting the Formalization of Requirements Using Techniques from Natural Language Processing," 2013, in preperation.
[22] M. Saeki, H. Horai, and H. Enomoto, "Software development process from natural language specification," in *Proceedings of the 11th international conference on Software engineering*, ser. ICSE '89. New York, NY, USA: ACM, 1989, pp. 64–73. [Online]. Available: http://doi.acm.org/10.1145/74587.74594
[23] I. S. Bajwa, B. Bordbar, and M. G. Lee, "OCL Constraints Generation from Natural Language Specification," in *Int'l Conf. on Enterprise Distributed Object Computing*, 2010, pp. 204–213.
[24] M. E. C. Hull, K. Jackson, and J. Dick, *Requirements Engineering, Second Edition*. Springer, 2005.
[25] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," 1999, pp. 411–420.
[26] M. Silberztein, "INTEX: A Corpus Processing System," in *Int'l Conf. on Computational Linguistics*, 1994, pp. 579–583.
[27] E. König, "LexGram - a practical categorial grammar formalism," *CoRR*, vol. cmp-lg/9504014, 1995.
[28] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters, *Text Processing with GATE*, 2011.
[29] L. Mich, "Nl-oops: from natural language to object oriented requirements using the natural language processing system lolita," *Natural Language Engineering*, vol. 2, pp. 161–187, 1996.