

# Uhura: A Tool for Translating Controlled Natural Language into Answer-Set Programs

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Software & Information Engineering**

by

**Tobias Kain**

Registration Number 1329088

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Mag.rer.nat. Dr.techn. Hans Tompits

Vienna, 18<sup>th</sup> August, 2017

---

Tobias Kain

---

Hans Tompits



# Erklärung zur Verfassung der Arbeit

Tobias Kain  
Gstetten 12, 3281 Oberndorf/Melk

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. August 2017

---

Tobias Kain



# Abstract

In the past couple of years, answer-set programming (ASP) has evolved to an important logic programming language. ASP has been used as a host language for many different projects. ASP was even applied to build a decision support system for the Space Shuttle.

A controlled natural language (CNL) is language that is based on a natural language. However, compared to its base language a CNL is more restricted concerning vocabulary, semantic and syntax. Due to these constraints, some controlled natural languages can be translated and interpreted efficiently by computers. Nevertheless, they are still easy to understand and use by humans since they are based on a natural language.

In this theses, we present a tool, called Uhura, which allows its users to express a problem description in controlled natural language. The system translates the composed sentences into an ASP program, which can then be solved by an ASP solver.

The goal we want to achieve with our system is, to allow people who are inexperienced in ASP using ASP anyway to solve problems. Furthermore, we designed our tool in such a way that the users can learn how to implement ASP programs by observing the ASP rules generated by Uhura.



# Contents

<b>Abstract</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>3</b>
2.1 Answer-Set Programming . . . . .	3
2.2 Controlled Natural Language . . . . .	6
2.3 Controlled Natural Languages for Knowledge Representation . . . . .	8
<b>3 The System Uhura</b>	<b>13</b>
3.1 The Jobs Puzzle . . . . .	13
3.2 CNL Problem Description . . . . .	14
3.3 ASP Translation . . . . .	18
3.4 Solve the Described Problem . . . . .	20
<b>4 Implementation</b>	<b>23</b>
4.1 System Architecture . . . . .	23
4.2 User Interface . . . . .	24
4.3 Sentence Type Detector . . . . .	26
4.4 CNL to ASP Translator . . . . .	26
4.5 DLV . . . . .	27
4.6 Features . . . . .	27
<b>5 Related Work</b>	<b>31</b>
5.1 PENG <sup>ASP</sup> System . . . . .	31
5.2 LOGICIA . . . . .	32
<b>6 Conclusion and Future Work</b>	<b>33</b>
<b>A User Manual</b>	<b>35</b>
A.1 Overview . . . . .	35
A.2 Implement Program . . . . .	35

A.3	Solve ASP Program . . . . .	37
A.4	Automatic Translation Mode . . . . .	38
A.5	Dictionary . . . . .	39
A.6	Self-Defined Sentence Pattern . . . . .	40
A.7	Manual Translations . . . . .	41
<b>B</b>	<b>Sentence Patterns</b>	<b>43</b>
	<b>Bibliography</b>	<b>47</b>



# Introduction

In the past couple of years, answer-set programming (ASP) has evolved to an important logic programming language. ASP has already served as a host language for several different application in the field of artificial intelligence and knowledge representation as well as other scientific areas.

However, ASP is rarely used by developers who do not work in an academic environment. This is attributable, among other things, to a lack of support tools designed for ASP novices. Especially those, who have never before used any logical programming languages have a hard time become acquainted with ASP.

In this thesis, we present a tool which addresses the previously mentioned issue. Our tool Uhura, named after the Star Trek character Nyota Uhura, supports ASP novices as well as professional ASP developers in developing ASP programs. Uhura provides the opportunity to express a problem definition in controlled natural language (CNL) close to natural English. This problem definition is then translated into an ASP program, which can be solved by the powerful ASP solver DLV. Compared to some other CNL to ASP translation tools, our system translates CNL sentences into ASP rules in the same way a professional ASP developer would do. For this reason, the resulting ASP program looks natural to developers experienced in ASP.

A developer, who has no experience in ASP so far, can use this tool to specify a problem definition expressed in the supported controlled natural language and observed how the single sentences are translated into ASP. Through this observation, the user can learn how to convert logical statements into ASP.

Furthermore, this tool can improve the communication between those who provide the required knowledge to solve a certain problem (domain experts) and those who put the knowledge into code (knowledge engineers), since the domain expert can read and understand the CNL sentences the knowledge engineer has composed.

One of our design goals was to develop an easy to use tool for which the user does not need any previous knowledge. Therefore, domain experts can easily perform changes to the code implemented by the knowledge engineer. Some of them might even be capable of implementing an ASP program without the help of a knowledge engineer.

This thesis is organized as follows: In Chapter 2, we provide background information about the two concepts our tool is based on, namely answer-set programming and controlled natural languages. Chapter 3 gives an overview of the system Uhura. In this section, we will describe how a typical workflow looks like when using Uhura. In Chapter 4, we will explain in detail the technical implementation of Uhura. In Chapter 5 we will compare our work to systems that address a similar issue. Finally, in Chapter 6 we will conclude this thesis and name some ideas of possible future developments of Uhura.

# Background

## 2.1 Answer-Set Programming

Answer-set programming (ASP) is a declarative programming paradigm used to solve complex search problems [Eiter et al., 2009, Brewka et al., 2011, Baral, 2003, Lifschitz, 2008]. ASP was found by Michal Gelfond and Vladimir Lifschitz at the beginning of the 1990s and has its roots in nonmonotonic reasoning and knowledge representation. ASP, therefore, is ideally suited for knowledge-intensive applications. The name *answer-set programming* leads back to the solutions of ASP programs, which are called answer sets (stable models) [Gelfond and Lifschitz, 1988].

In contrast to imperative programming languages, like for example C or Java, the idea of ASP is not to define an algorithm, which solves a particular problem, but rather describe the problem. This means that developers do not have to care about how to solve the specified problem. So-called answer-set solvers undertake the task of solving answer-set programs by generating the solutions (answer sets).

The past couple of years have shown, that ASP finds use in various scientific fields. The following listing is just a brief extract of all the areas ASP has been used in:

- Planing [Lifschitz, 2002]
- Learning [Sakama, 2001]
- Robotics [Erdem et al., 2012]
- Natural language processing [Baral and Son, 2008]
- Semantic-web reasoning [Eiter et al., 2008]
- Software testing [Brain et al., 2012]

### 2.1.1 Syntax of Answer-Set Programs

An answer-set program is composed of a finite set of rules. The building blocks of such rules are *atoms* and *literals*.

An atom is a factual statement that is either true or false. It is an expression of the form

$$p(t_1, \dots, t_n),$$

where  $p$  is a predicate symbol and  $t_1 \dots t_n$  are terms, whereby a term is either a variable (denoted by a capital letter) or a constant (denoted by a lower-case character).

Literals refer to atoms  $a$  and strongly negated atoms  $\neg a$ .

The rules, a program is composed of are of form

$$a_1 \vee \dots \vee a_m : -b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n \quad (2.1)$$

where  $a_1 \dots a_m$  and  $b_1 \dots b_n$  are literals. The first part of the rule  $a_1 \vee \dots \vee a_m$  is referred to as the *head* of the rule. The rear part of the rule  $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n$  is referred to as the *body* of the rule. The body is subdivided into a positive and a negative body. The keyword *not* that is placed in front of those literals that are part of the negative body is called *negation as failure* or *default negation*.

Both head and body of a rule might be empty. In the case that the body does not contain any literals, the resulting rule is referred to as *fact*. On the other hand, rules that do not have any literals in the head are called *constraints*.

A rule that does not comprise any variables is characterized as a *ground rule*, otherwise as a *non-ground rule*. Similar, ASP programs that only contain ground rules are classified as grounded and otherwise as non-grounded programs.

### 2.1.2 Semantic of Answer-Set Programs

The intuitive meaning of rule 2.1 is that if  $b_1, \dots, b_k$  are derivable and  $b_{k+1}, \dots, b_n$  are not derivable then  $a_1 \vee \dots \vee a_m$  is true. It is noted that ASP rules are similar to default rules used in default logic [Reiter, 1980]. The equivalent default rule of the rule shown in 2.1 is expressed as follows:

$$\frac{b_1 \wedge \dots \wedge b_k : \neg b_{k+1}, \dots, \neg b_n}{a_1 \vee \dots \vee a_m} \quad (2.2)$$

A Solution of an ASP program is intuitively described as a set of literals that can be derived from the ASP program. These Solutions correspond to the extensions of the default theory  $T = (W, D)$ , where the set of the certain knowledge  $W$  is empty and  $D$ , the set of default rules, contains all rules of the ASP program transformed into default rules as shown in 2.2.

Since these intuitive definitions are vague and difficult to determine we will take a look at the precise definition of the ASP semantic. At first we will only consider grounded ASP programs.

An interpretation  $I$  of an answer-set program is a set of ground literals that does not contain any atom  $a$  such that  $\{a, \neg a\} \subseteq I$  ( $I$  is consistent). A literal  $l$  is true if  $l \in I$ , otherwise  $l$  is false. If a literal is default negated then *not*  $l$  is true if  $l \notin I$  and otherwise false.

We further define an interpretation  $I$  as a model  $M$  of a rule, constructed like the one shown in 2.1, if it holds that whenever the body of the rule is true, meaning that  $\{b_1, \dots, b_k\} \subseteq M$  and  $\{b_{k+1}, \dots, b_n\} \not\subseteq M$ , the head is true, and therefore  $\{a_1, \dots, a_m\} \cap M \neq \emptyset$  holds. If an interpretation  $I$  is a model  $M$  of all rules of a program  $P$  then  $M$  is called a (classical) model of  $P$ . However, not all models of program  $P$  are considered as solutions (answer sets) of  $P$ .

To determine, which interpretations are answer sets of a program, we first have to define the Gelfond-Lifschitz reduct of a program.

### Determine Answer Sets of a Grounded Program

The Gelfond-Lifschitz reduct of a program  $P$  and a interpretation  $M$  is defined as follows:

$$\begin{aligned} P^M = \{ & a_1 \vee \dots \vee a_m : -b_1, \dots, b_k | \\ & a_1 \vee \dots \vee a_m : -b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_n \in P, \\ & \{b_{k+1}, \dots, b_n\} \not\subseteq M \} \end{aligned} \quad (2.3)$$

An interpretation  $M$  for which holds, that  $M$  is a minimal model of the Gelfond-Lifschitz reduct  $P^M$  is considered as an answer set of  $P$ .  $M$  is a minimal model of  $P^M$  if no other interpretation  $N \subset M$  is also a model of  $P^M$ . For instance, consider the following program  $P$ :

*tweety.*  
*bird* : - *tweety*, *not penguin*.  
*penguin* : - *tweety*, *not bird*.

To check whether the interpretation  $M = \{\textit{tweety}, \textit{bird}\}$  is an answer set of  $P$ , we first have to determine the reduct  $P^M$ . If we compute the reduct of program  $P$  and interpretation  $M$  as stated in 2.3 we get the following result:

$$P^M = \{ \textit{tweety}., \\ \textit{bird} : - \textit{tweety}. \}$$

$M$  is a model of  $P^M$  since  $M$  is a model of both the rules contained in  $P^M$ . As  $M$  is a minimal model of the reduct, meaning that no subset of  $M$  is also a model of  $P^M$ , the model  $M$  is an answer set of program  $P$ . Besides  $M$ , the interpretation  $\{\textit{tweety}, \textit{penguin}\}$  is also an answer set of  $P$ .

We now remove the first rule of  $P$ . The resulting program  $P'$  is given as follows:

*bird* : - *tweety*, *not penguin*.  
*penguin* : - *tweety*, *not bird*.

Again, we check if the interpretation  $M = \{tweety, bird\}$  is an answer set of the reduct  $P'$ , which is given as follows:

$$P'^M = \{bird : - tweety.\}$$

However, since  $\emptyset$  is a model of  $P'^M$  and  $\emptyset \subseteq M$  the interpretation  $M$  is not an answer set of  $P'$ .

### Determine Answer Sets of a Non-Grounded Program

In the case of a non-grounded program  $P$ , like the following:

$$\begin{aligned} &bird(tweety). \\ &cat(sylvester). \\ &fly(X) : - bird(X), not penguin(X). \end{aligned}$$

grounding is required in order to determine its answer sets.

At the process of grounding, all variables of a rule are replaced by the constants appearing in the program. The grounded program  $grnd(P)$  is given as follows:

$$\begin{aligned} &bird(tweety). \\ &cat(sylvester). \\ &fly(tweety) : - bird(tweety), not penguin(tweety). \\ &fly(sylvester) : - bird(sylvester), not penguin(sylvester). \end{aligned}$$

Since this naive grounding approach can cause grounded programs that are enormous in length, grounding tools such as Gringo [Gebser et al., 2011] or DLV's grounder [Faber et al., 2012] optimize the grounding task.

Once a program is grounded, we can feed it into an ASP solver, which generates the answer sets of our problem description. Among the most efficient ASP solvers are DLV [Leone et al., 2006] and clasp [Gebser et al., 2012].

## 2.2 Controlled Natural Language

Controlled natural language (CNL) or simply controlled language is a fuzzy term, and therefore many different definitions exist.

Tobias Kuhn [Kuhn, 2014] tried to differentiate controlled natural languages from other languages by the following detailed definition:

**Definition 2.2.1** *A language is characterized as controlled natural language if it fulfills the following four points:*

- *A controlled natural language is only based on one natural language (e.g. English, German, Spanish).*
- *Compared to its base language the controlled natural language is more restricted concerning vocabulary, syntax and/or semantics.*
- *The controlled natural language should be similar to the natural language, meaning that a person capable of reading and writing the base language should also be able to read and write the controlled natural language without much effort.*
- *A controlled natural language is a constructed language, meaning that in contrast to its base language, the controlled natural language is explicitly and consciously defined.*

Controlled natural languages can be divided into two the following two categories [Huijsen, 1998]:

- Human-oriented CNLs
- Computer-oriented CNLs

In the following sections, we will take a closer look at these two groups and discuss their attributes.

Besides these two classes of controlled natural languages, many other properties of CNLs can be used to categorize them. [Kuhn, 2014] characterizes 100 controlled natural languages by introducing a general classification scheme.

### 2.2.1 Human-Oriented Controlled Natural Languages

The main purpose of human-oriented controlled natural languages is to provide a language that is easier to read and understand, especially for those people who are non-native speakers of the base language. Often these kinds of controlled natural languages are used to write technical documentations that have to be read by a wide variety of people.

A few popular examples of human-oriented CNLs are Caterpillar Technical English [Kamprath et al., 1998], IBM’s EasyEnglish [Bernth, 1997], and ASD Simplified Technical English [ASD, 2005], which is a language for the aerospace industry.

### 2.2.2 Computer-Oriented Controlled Natural Languages

In contrast to human-oriented CNLs, computer-oriented controlled natural languages are languages that can be translated and interpreted efficiently by computers.

Natural languages are the most expressive types of languages. Furthermore, they are easy to use and understand. However, statements expressed in a natural language

are often ambiguous and require a certain background knowledge to understand them [Schwitter, 2010]. On the other hand, formal languages, like for example first-order logic, are unambiguous and have a well-defined syntax, and are therefore for example suited for knowledge representation. The downside of formal languages is that formal sentences are more difficult to formulate and understand than sentences expressed in a natural language.

Computer-oriented CNLs build the bridge between natural languages and formal languages. They combine the advantages of both these types of languages, meaning that they are easy to use and read as well as qualified to be used as a knowledge representation language. Besides knowledge representation, computer-oriented CNLs find for example application in Semantic Web applications [Kaljurand, 2007] and machine translation tools [Nyberg and Mitamura, 2000].

### 2.3 Controlled Natural Languages for Knowledge Representation

In the following sections, we will discuss some computer-oriented CNLs suited for knowledge representation.

#### 2.3.1 Attempto Controlled English (ACE)

Attempto Controlled English (ACE) [Fuchs and Schwitter, 1996] is a controlled natural language that was found in 1996. The motivation to develop this language was to provide a language that can be used for writing complete, unambiguous and consistent software specification. However, the last couple of years have shown, that the main area of application of ACE is the Semantic Web [Kaljurand, 2007].

A text written in ACE can be translated unambiguously into first-order logic through its discourse representation structure (DRS) [Kamp and Reyle, 2013]. The Attempto Project<sup>1</sup> provide some tools to for handling ACE texts.

The following short excerpt of a specification, written in ACE, explores various kinds of ACE features, like anaphoric references and abbreviations:

The user enters a card and a numeric code.  
If it is not valid then SM rejects the card.

These two sentences have been parsed by the ACE Parser provided by the Attempto Project's website. The parser provides the feature to show a paraphrase of the entered text:

---

<sup>1</sup>Attempto Project: <http://attempto.ifi.uzh.ch>



There is a user X1.

The user X1 enters a numeric code X2 and a card X3.

If it is false that the numeric code X2 is valid then SM rejects the card X3.

Furthermore, this tool also allows to display the discourse representation structure of the entered specification:

```
[A, B, C, D, E]
object (E, user, countable, na, eq, 1) -1/2
object (A, card, countable, na, eq, 1) -1/5
has\_part (D, A) -1/
predicate (B, enter, E, D) -1/3
object (C, code, countable, na, eq, 1) -1/9
property (C, numeric, pos) -1/8
has\_part (D, C) -1/
object (D, na, countable, na, eq, 2) -1/
  []
    NOT
    [F, G]
    property (F, valid, pos) -2/5
    predicate (G, be, C, F) -2/3
=>
[H]
predicate (H, reject, named (SimpleMat), A) -2/8}
```

### 2.3.2 PENG Light

PENG Light [Schwitter, 2008] is the successor of PENG (Processable English) [Schwitter, 2002], a computer-oriented controlled natural language similar to Attempto Controlled English. In contrast to its predecessor, PENG Light can be processed by a bidirectional grammar, meaning that first-order translations of sentences expressed in PENG Light can be used to create answers to these sentences.

#### Syntax of PENG Light

The syntax of PENG Light specifies two types of sentences: simple and complex sentences. The following two sections will give a short overview of these types of sentences. For a more detailed specification of the PENG Light Syntax see [Schwitter, 2008].

**Simple sentences** are sentences of the following structure:

$$\textit{subject} + \textit{predicator} + [\textit{complements}] + \{\textit{adjuncts}\}$$

A subject consists of an optional determiner (e.g. the, a, her, ...), followed by an optional adjective (a so called pre-nominal modifier), a noun, and finally a post-nominal modifier,

like for example a variable or an of-construct. The subject is followed by a predicator, which is a non-negated verb. It is important to note that verbs are only allowed in present tense, active voice, indicative mood, and third person singular/plural. Depending on the verb, the predicator is or is not followed by a compliment. A compliment, which is either a noun phrase, an adjective phrase or a prepositional phrase, completes a predicate. The last part of a simple sentence is any number of adjuncts, which are either realized as adverbs or prepositional phrases.

2.4 shows the shortest possible PENG Light sentence. It only consists of a subject (*Roberta*) and a predictor (*plays*).

2.5 illustrates a more complex simple sentence. It is composed of an determiner (*a*), which forms together with an adjective (*tall*) and a noun (*person*) the subject. It is followed by a predicator (*plays*) and a complement (*the piano*). The sentence is then completed by two adjuncts (*in the opera house*, *on Monday*).

Compared to 2.5, 2.6 refers to a specific person *X*. We can use this variable in the same or in other sentences to refer to exactly that person.

Roberta plays. (2.4)

A tall person plays the piano in the opera house on Monday. (2.5)

A tall person X plays the piano in the opera house on Monday. (2.6)

**Complex sentences** are sentences that are modifications of simple sentences or constructed of several simple sentences. PENG Light supports six different types of complex sentences: Quantification, Negation, Conditionals, Subordination, Coordination, Definitions.

**Quantification** PENG Light allows three different kinds of quantification: existential quantification, universal quantification, and cardinality restrictions.

2.7 and 2.8 illustrate two examples of quantified PENG Light sentences together with their first-order translation.

Sentence 2.7 comprises an existential as well as an universal quantification. As can be seen from the first-order expression, the scope of the existential quantification covers the entire sentence, where on the other hand the universal quantification only encompasses a part of the sentence. The scope of a quantification in a PENG Light sentence always begins at the position the quantification is mentioned and ends at the end of the sentence.

Sentence 2.8 contains a cardinality restrictions (exactly one). Besides the restriction used in 2.8, PENG Light also supports the cardinality restrictions *at least* and *at most*.

A person plays the piano in every opera house. (2.7)  
 $\exists x(person(x) \wedge (\forall y(operaHouse(y) \supset playsThePianoIn(x, y))))$

$$\begin{aligned}
&\text{Roberta plays the piano in exactly one opera house.} \\
&\exists x(\text{operaHouse}(x) \wedge \text{playsThePianoIn}(\text{roberta}, x) \wedge \\
&\quad (\forall y(\text{playsThePianoIn}(\text{roberta}, y) \supset x = y)))
\end{aligned} \tag{2.8}$$

**Negation** PENG Light supports three different kinds of negation. We can negate an entire sentence by starting it with the phrase *it is false that* (2.9). Further, we can negate a noun by adding *no* in front of it (2.9). In the same fashion, we can negate a verb with *do/does not* and *is not* (2.10).

$$\begin{aligned}
&\text{It is false that no person plays the piano.} \\
&\neg(\neg \exists x \text{ play}(x, \text{piano})) \iff \exists x \text{ play}(x, \text{piano})
\end{aligned} \tag{2.9}$$

$$\begin{aligned}
&\text{Roberta does not play the piano.} \\
&\neg \text{play}(\text{roberta}, \text{piano})
\end{aligned} \tag{2.10}$$

**Conditionals** With the phrase *If ... then ...* we can construct conditional sentences. As 2.11 shows, conditionals consume two simple sentences. The first one (*Roberta plays the piano.*) specifies the antecedent and the second simple sentence (*Roberta is in the opera house.*) defines the consequence.

$$\begin{aligned}
&\text{If Roberta plays the piano then Roberta is in the opera house.} \\
&\text{play}(\text{roberta}, \text{piano}) \supset \text{in}(\text{roberta}, \text{operaHouse})
\end{aligned} \tag{2.11}$$

We will not discuss the other three types of complex sentences (Subordination, Coordination, and Definitions) since they are not of relevance in the course of this thesis.

### 2.3.3 PENG<sup>ASP</sup>

PENG<sup>ASP</sup> [Schwitter, 2013] is controlled natural language similar to PENG Light and Attempto Controlled English. The primary purpose of designing PENG<sup>ASP</sup> was to provide a CNL that can be unambiguously translated into ASP.

Like PENG Light, PENG<sup>ASP</sup> also differentiates between simple and complex sentences. Table 2.1 shows some examples of simple sentences. Depending on the application, more sentence patterns can be added. The two sample patterns in table 2.1 are factual statements, meaning that they can be translated into an ASP rule. Other simple sentences (e.g. A *CNoun* is *Adjective*.) can not be transformed directly into ASP. Those sentences are used in complex sentences.

---

<sup>2</sup>*PNoun* = Proper Noun (e.g. Vienna, Roberta, TU-Wien, ...)

<sup>3</sup>*CNoun* = Common Noun (e.g. person, animal, university, ...)

Pattern	Example	ASP-Translation
$PNoun^2$ is a $CNoun^3$ .	Roberta is a person.	person(Roberta) .
$PNoun$ is <i>Adjective</i> .	Roberta is lovely.	lovely(Roberta) .

Table 2.1: Simple Sentence Examples

2.12 and 2.13 are the both complex sentences that are supported by PENG<sup>ASP</sup>.

If *SimpleSentence* and *SimpleSentence* then *SimpleSentence*. (2.12)

Exclude that *SimpleSentence* and that *SimpleSentence*. (2.13)

A sentence that fulfills pattern 2.12 corresponds to an ASP rule which has a non-empty head and a non-empty body. On the other hand, a sentence that matches pattern 2.13 corresponds to an ASP constraint.

2.14 and 2.15 show two examples of complex PENG<sup>ASP</sup> sentences as well as their translation into ASP.

If Roberta is female then Roberta is a woman.  
 $woman(roberta) : \neg female(roberta).$  (2.14)

Exclude that Roberta is female and that Roberta is a man.  
 $: \neg female(roberta), man(roberta).$  (2.15)

# The System Uhura

Uhura is a tool for translating a problem definition expressed in a controlled natural language into an answer-set program. We named our system after the Star Trek character Lieutenant Uhura, who serves as translation and communication officer of the USS Enterprise.

The development of Uhura was driven by our motivation to provide a system which supports students doing their first steps in answer-set programming.

By specifying a problem in a language similar to natural language, but influenced by the syntax of ASP, students learn how to express domain knowledge. Furthermore, they learn how to translate domain knowledge into ASP rules, by observing the generated ASP rules. Due to this reason, our tool converts CNL sentences in such a way, that the user recognizes which sentence is translated into which ASP rule(s). We also aim to keep the ASP rules as natural as possible, meaning that the resulting ASP program should be easy to understand by developers experienced in ASP.

Uhura, as well as its source code, is available for download at <https://github.com/TobiasKain/Uhura>. For a detailed description of how to use Uhura see Appendix A.

In the following sections, we will describe a typical workflow when using Uhura.

## 3.1 The Jobs Puzzle

To demonstrate how a typical workflow looks like and how the system works we will refer in the upcoming sections to the following puzzle:

1. There are four people: Roberta, Thelma, Steve, and Pete.
2. Among them, they hold eight different jobs.

3. Each holds exactly two jobs.
4. The jobs are chef, guard, nurse, clerk, police officer (gender not implied), teacher, actor, and boxer.
5. The job of nurse is held by a male.
6. The husband of the chef is the clerk.
7. Roberta is not a boxer.
8. Pete has no education past the ninth grade.
9. Roberta, the chef, and the police officer went golfing together.

This puzzle is the so-called *jobs puzzle*, which was first introduced in 1984 along with other puzzles designed for automated reasoning [Wos et al., 1984].

The goal of this puzzle is to find out which person holds which jobs. Clearly, the solution to this puzzle can not be found only by analyzing the explicit information stated in the puzzle. Instead, we have to extract the implicit information of the puzzle and use this knowledge together with the explicit information to solve the puzzle.

## 3.2 CNL Problem Description

In order to solve the job puzzle using Uhura, the user has to phrase the implicit and explicit knowledge of the puzzle in a controlled natural language similar to PENG<sup>ASP</sup>.

Compared to PENG<sup>ASP</sup>, the controlled natural language we are working with comprises a lot more and slightly different sentence patterns. Furthermore, we also allow our users to define their own sentence patterns. Appendices B provides a listing of all sentence patterns supported by Uhura.

The first sentence of the job puzzle states that our problem domain holds four different people. We express this factual statement by applying the following pattern:

*PNoun* is a *CNoun*.

As PNouns (proper nouns) we insert the names of the four people (*Roberta*, *Thelma*, *Steve*, *Pete*) and as CNoun (common noun) we use *person*.

Roberta is a person.  
Thelma is a person.  
Steve is a person.  
Pete is a person.

(3.1)

From the characters' names, we can imply that two of them are male and the other two are female.

Roberta is female.  
 Thelma is female.  
 Steve is male.  
 Pete is male.

(3.2)

Furthermore, we have to express that a person can not be male and female at the same time. This means we want to drop all solutions which claim that some person has two genders. We do this by using the following patterns:

Exclude that *SimpleSentence* {and that *SimpleSentence*}.

Since we want that this constraint is applied to every person in our problem domain, we use a variable instead of proper nouns.

Exclude that person X is male and that person X is female. (3.3)

Next, we guess all the possible solution candidates. In order to do that, we use the following sentence pattern:

If *SimpleSentence* {and *SimpleSentence*} that *SimpleSentence*.

To guess whether a person holds a specific job or not, we use a disjunctive sentence.

If there is a person X and there is a job Y then person X holds job Y or  
 person X does not hold job Y. (3.4)

The second sentence of the job puzzle tells us, that our problem domain includes eight different jobs and that every job is held by exactly one person. To formulate this statement, we use the following pattern as part of a constraint:

*CNoun Variable Verb* (more/less) than *Number CNoun Variable*.

Since we want to express that a job is held by **exactly** one person, we have to use this pattern twice.

Exclude that there is a job Y and that person X holds more than one job Y.  
 Exclude that there is a job Y and that person X holds less than one job Y.

(3.5)

The third sentence of the job puzzle states that each person holds exactly two jobs. As before, we use the cardinality pattern twice to formulate this statement.

Exclude that there is a person X and that person X holds more than two jobs Y.  
Exclude that there is a person X and that person X holds less than two jobs Y. (3.6)

The next sentence of the job puzzle enumerates all jobs our problem domain holds. We formulate these facts using the same pattern we applied to phrase the first sentence.

Chef is a job.  
Guard is a job.  
Nurse is a job.  
Clerk is a job.  
Police officer is a job. (3.7)  
Teacher is a job.  
Actor is a job.  
Boxer is a job.

Chef, guard, nurse, clerk, police officer (gender not implied), teacher, and boxer are gender neutral jobs, meaning that they can be held by both genders. However, the job of an actor can only be done by a male, since a female actor is called an actress.

Furthermore, due to the fifth sentence of the job puzzle, we know that the job of a nurse is held by a male.

If a person X holds a job as actor then person X is male.  
If a person X holds a job as nurse then person X is male. (3.8)

According to the sixth sentence of the job puzzle, the husband of the chef is the clerk.

If a person X holds a job as chef and a person Y holds a job as clerk then a person Y is a husband of a person X. (3.9)

The previous sentence implies, that the clerk is male since he is the husband of another person and that the chef is female since she has a husband.

If a person X is a husband of a person Y then person X is male.  
If a person X is a husband of a person Y then person Y is female. (3.10)

Since the seventh sentence of the job puzzle states, that Roberta is not a boxer, we exclude all solution which claims that Roberta is a boxer.



Exclude that Roberta holds a job as boxer. (3.11)

The eighth sentence of the job puzzle specifies that Pete is not educated. Therefore we drop all solutions that assert that Pete is educated.

Exclude that Pete is educated. (3.12)

Furthermore, we can assume that a person has to be educated to hold a job as a nurse, police officer, or teacher.

If a person X holds a job as nurse then person X is educated.  
If a person X holds a job as police officer then person X is educated. (3.13)  
If a person X holds a job as teacher then person X is educated.

The last sentence of the job puzzle contains the implicit information that Roberta is neither a chef nor a police officer. Furthermore, this sentence also implies, that the person who works as a chef and the person who works as a police officer are two different individuals.

Exclude that Roberta holds a job as chef.  
Exclude that Roberta holds a job as police officer.  
Exclude that a person X holds a job as chef and that person X holds a job as police officer. (3.14)

### 3.2.1 Composing the CNL Problem Description using Uhura

As the previous section illustrated, the user has to work precisely to extract all the implicit information contained in the puzzle. Especially for inexperienced users, the task of extracting implicit information is probably one of the most complicated jobs.

To support the user as good as possible during the process of composing the problem description, our tool provides a separate text field where the user writes down the domain knowledge (see Figure 3.1). Alternatively, our tool also allows importing text files composed in a different text editor.

Since remembering all the different sentence patterns is unrewarding, the user can look them up by switching in the lower tab pane to the tab called *Sentence Patterns*.

In case the user enters a sentence that does not match any pattern, our system responds with an error message that explains the user where exactly in his sentence an error occurred.

### 3. THE SYSTEM UHURA

---

For example, if the user types in the following sentence

Roberta is not a lovely.

then the system responds with the following error message, which tells the user that the which sentence pattern was detected and which word caused the error:

```
Error in sentence "Roberta is not a lovely .":  
  "lovely" is not a common noun.  
  (detected sentence-pattern: 'PNoun is [not] a CNoun.')
```

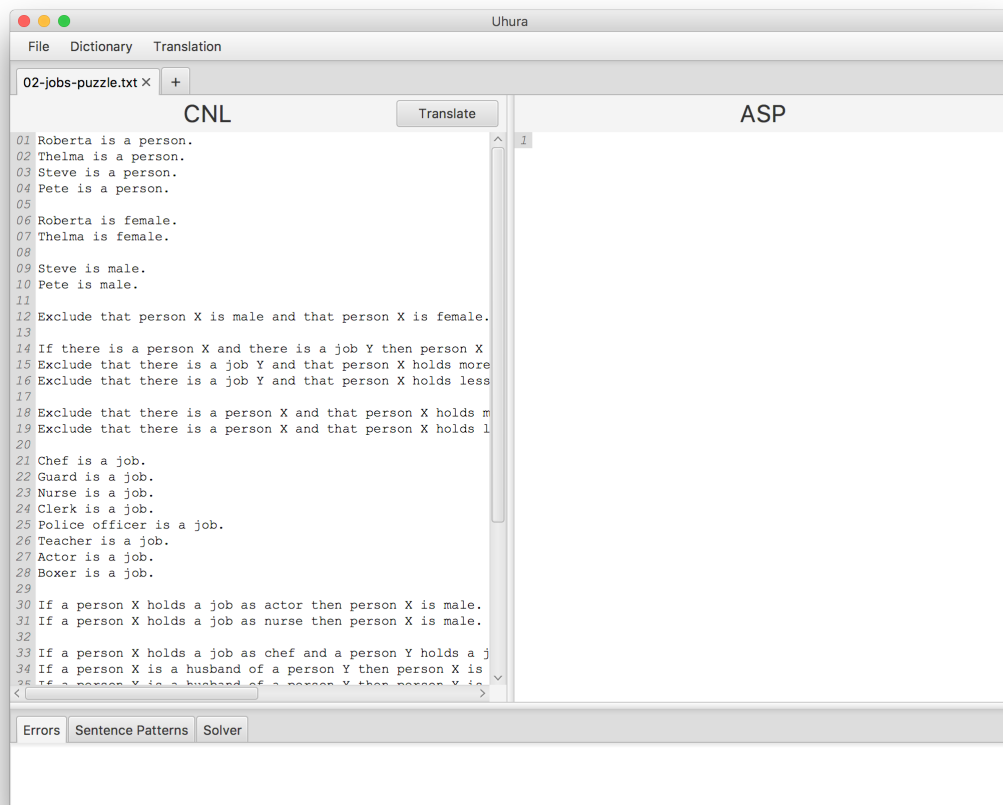


Figure 3.1: The user interface with the entered CNL problem description.

### 3.3 ASP Translation

Once the user clicks the *Translation* button, the system starts translating the entered CNL problem description into ASP rules.

The following ASP program is the result of translating the CNL problem description shown in the previous section. To demonstrate which sentence corresponds to which ASP rule, we added the sentence numbers next to the ASP rules.

```

person(roberta).    % 3.1.a
person(thelma).    % 3.1.b
person(steve).     % 3.1.c
person(pete).      % 3.1.d

female(roberta).   % 3.2.a
female(thelma).    % 3.2.b
male(steve).       % 3.2.c
male(pete).        % 3.2.d

:- male(X), person(X), female(X), person(X). % 3.3

hold(X,Y) v -hold(X,Y) :- person(X), job(Y). % 3.4

:- job(Y), #count{X : hold(X,Y)} > 1. % 3.5.a
:- job(Y), #count{X : hold(X,Y)} < 1. % 3.5.b

:- person(X), #count{Y : hold(X,Y)} > 2. % 3.6.a
:- person(X), #count{Y : hold(X,Y)} < 2. % 3.6.b

job(chef).          % 3.6.a
job(guard).         % 3.6.b
job(nurse).         % 3.6.c
job(clerk).         % 3.6.d
job(policeofficer). % 3.6.e
job(teacher).       % 3.6.f
job(actor).         % 3.6.g
job(boxer).         % 3.6.h

male(X) :- hold(X,actor), person(X), job(actor). % 3.7.a
male(X) :- hold(X,nurse), person(X), job(nurse). % 3.7.b

husband(Y,X) :- hold(X,chef), person(X), job(chef),
                hold(Y,clerk), person(Y), job(clerk). % 3.8

male(X) :- husband(X,Y), person(X), person(Y). % 3.9.a
female(Y) :- husband(X,Y), person(X), person(Y). % 3.9.b

:- hold(roberta,boxer), job(boxer). % 3.10

```

```

:- educated(pete).      % 3.11

educated(X) :- hold(X,nurse), person(X), job(nurse). % 3.12.a
educated(X) :- hold(X,policeofficer), person(X),
               job(policeofficer). % 3.12.b
educated(X) :- hold(X,teacher), person(X),
               job(teacher). % 3.12.c

:- hold(roberta,chef), job(chef). % 3.13.a
:- hold(roberta,policeofficer),
   job(policeofficer). % 3.13.b
:- hold(X,chef), person(X), job(chef),
   hold(X,policeofficer), person(X),
   job(policeofficer). % 3.13.c

```

As can be seen in Figure 3.2, the result of the translation process is displayed in the text editor next to the CNL problem description.

## 3.4 Solve the Described Problem

Now that the problem description is translated into an ASP program, the user can solve the specified problem. To do so, the user has to switch to the *Solver* tab. Since the goal of the job puzzle is to find out which person holds which job, we are only interested in the predicate *hold*. Therefore, we filter the resulting models by the predicate *hold*.

Figure 3.3 shows that the job puzzle has exactly one solution, namely:

- Roberta holds a job as guard and teacher.
- Thelma holds a job as chef and boxer.
- Steve holds a job as nurse and police officer.
- Pete holds a job as clerk and actor.

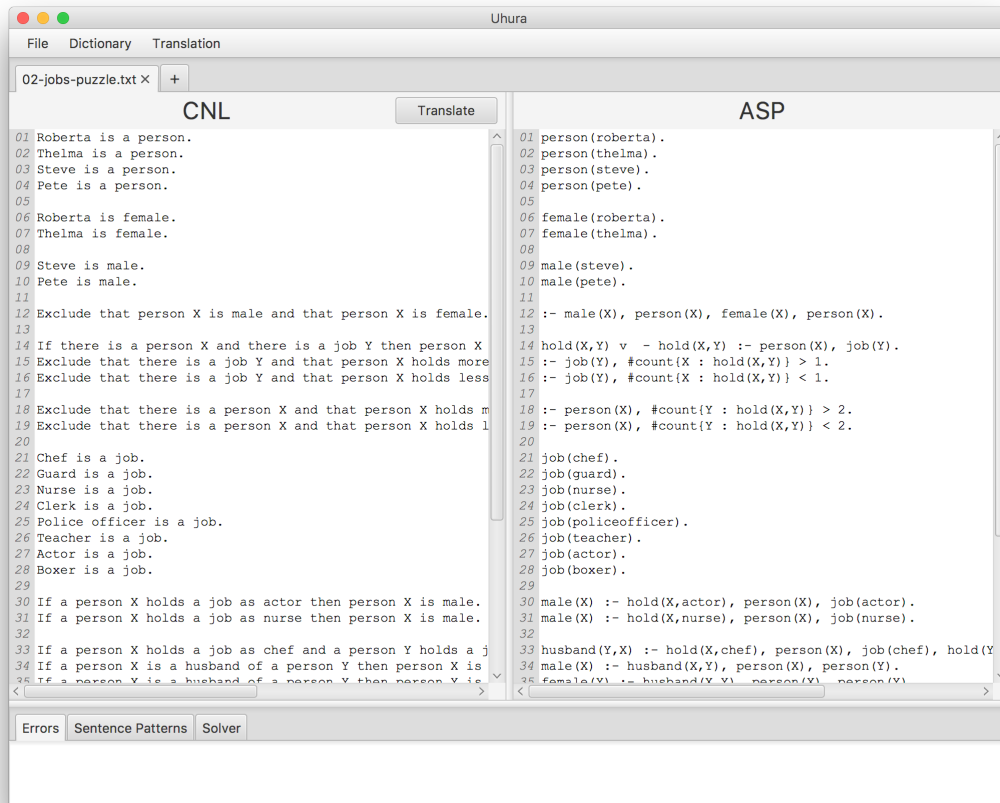


Figure 3.2: The user interface showing the translated CNL program.

### 3. THE SYSTEM UHURA

---

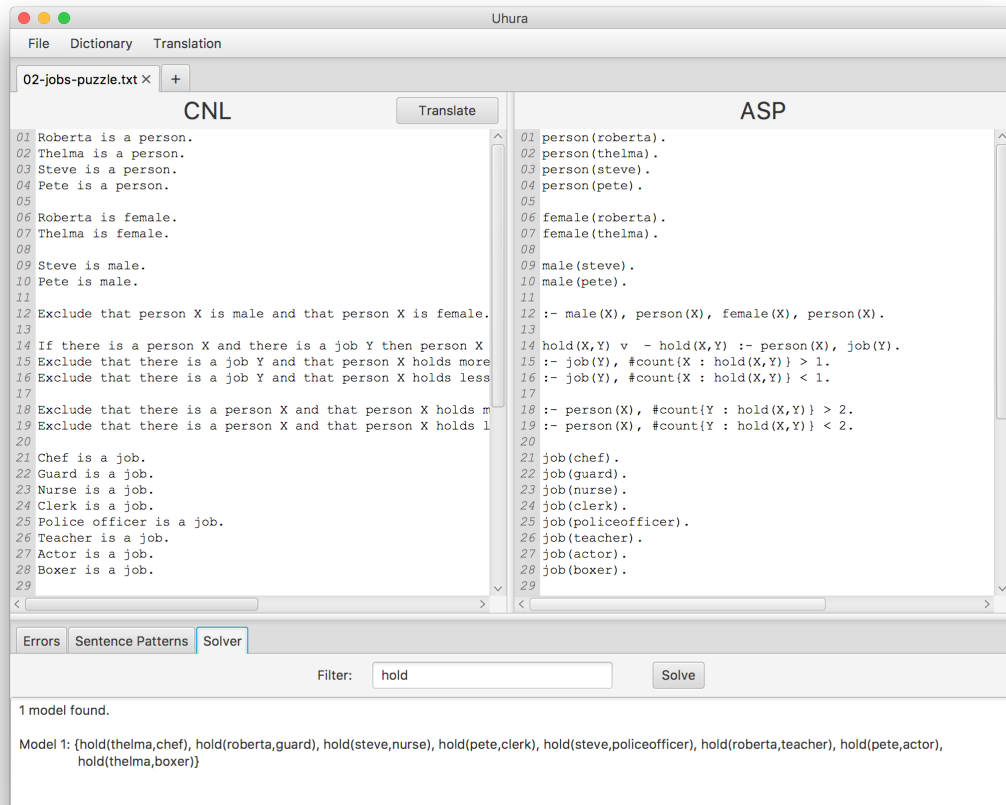


Figure 3.3: The user interface showing the solution of the specified problem.

# Implementation

To implement Uhura we used the programming language Java. One of the main reasons why we used Java is that DLVSYSTEM<sup>1</sup> provides a product called DLVWrapper [Ricca, 2003], which is a Java library that provides all the functionalities of DLV. Another reason, why we decided to implement our tool using Java is that we want to keep the option to integrate our tool into SeaLion<sup>2</sup>, which is an integrated development environment for ASP.

## 4.1 System Architecture

Basically, Uhura consists of four different components:

- User interface
- Sentence type detector
- CNL to ASP translator
- DLV (ASP solver)

Figure 4.1 shows how these components are connected to each other.

In the following sections, we will describe the components our system is composed of.

---

<sup>1</sup>DLVSYSTEM: <http://www.dlvsystem.com>

<sup>2</sup>SeaLion: <http://www.sealion.at>

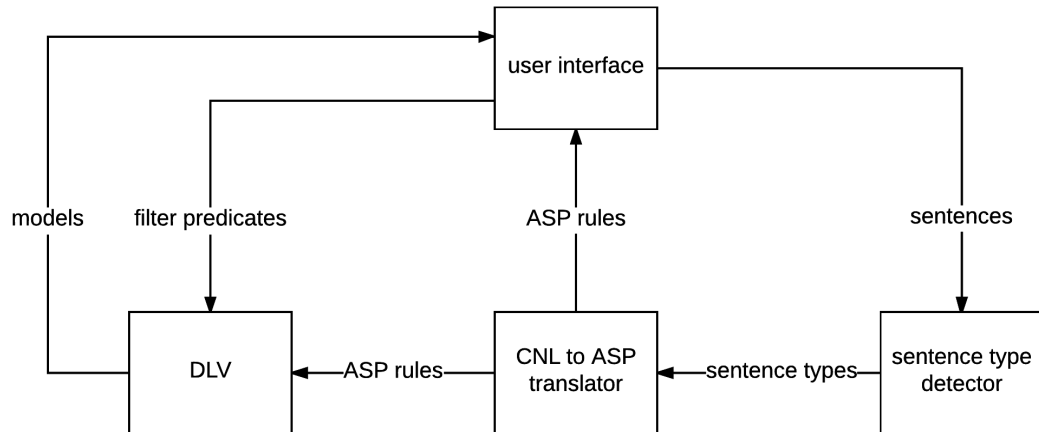


Figure 4.1: The system architecture of Uhura.

## 4.2 User Interface

To design and create the user interface (UI) of our tool we used JavaFX. Since one of our goals was to provide an intuitively to use graphical user interface, we decided to design the UI of Uhura similar to the user interfaces of popular IDEs, like for example IntelliJ, Eclipse, or Visual Studio.

### 4.2.1 User Interface Segments

As can be seen in Figure 4.2, the user interface of Uhura can be segmented into three parts.

#### Menu

The red marked area contains the menu, which is composed of three submenus. The *File* submenu provides basic features like for example open, save a file. The *Dictionary* submenu allows modifying the dictionary (see Section 4.6.1). The *Translation* submenu allows the user to switch between manual translation mode and automatic translation mode, to manage translation patterns (see Section 4.6.2), and to modify manual translations (see Section 4.6.3).

#### Coding Area

The region highlighted in blue is the main area of user interface. This is the area where the user composes the source code. It consists of two text editors, one for writing down



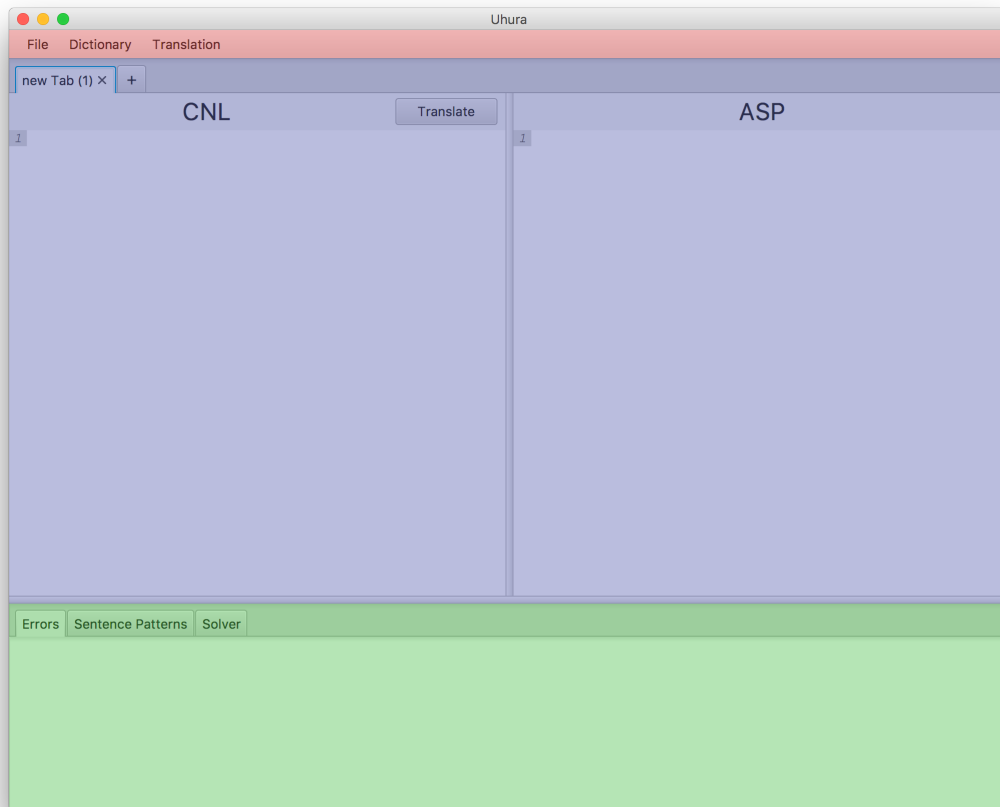


Figure 4.2: The user interface of Uhura.

the CNL problem description, and one for the translated ASP program. The editor we use is a powerful and memory efficient JavaFX text area called RichTextFX<sup>3</sup>.

### Bottom Bar

The area highlighted in green shows the bottom bar, which consists of three tabs. The *Error* tab tells the user, which sentences cannot be translated correctly. The *Sentence Pattern* tab holds a list of all the sentence patterns the user can make use of. The *Solver* tab allows the user to solve the ASP program and inspect the models found by the ASP solver.

<sup>3</sup>RichTextFX: <https://github.com/TomasMikula/RichTextFX>

### 4.3 Sentence Type Detector

Once the user has typed in a sentence and clicked the *Translate* button, the sentence type detector tries to find out which sentence pattern the entered sentence fulfills. Therefore, the sentence type detector holds for each sentence pattern a regular expression (regex) pattern. For example, the regex pattern of the sentence pattern

*PNoun* is a *CNoun*.

is defined as follows:

`. * is(n't | n't | not | ) (a|an) . * \\. $`

Note that also the negated sentence pattern (*PNoun* is not a *CNoun*.) matches this regex pattern.

To find out of which type the entered sentence is, the sentence type detector sequentially checks if the sentence matches one of the regex patterns. In case a match was found, the sentence type detector initiates the CNL to ASP translation of the sentence entered by the user.

### 4.4 CNL to ASP Translator

The CNL to ASP translator is the heart of Uhura. It is responsible for translating the CNL sentences entered by the user into ASP rules.

The following code snippet shows the method that translates sentences that match the regex pattern shown in the previous section:

```
AspRule pNounIsACNoun(ArrayList<TaggedWord> taggedWords)
    throws SentenceValidationException {

    String pNoun = wordDetector.getPoun(taggedWords);
    wordDetector.removeWord(taggedWords, "is");
    boolean negated = wordDetector.isNegation(taggedWords);
    wordDetector.removeWord(taggedWords, " (a|an) ");
    String cNoun = wordDetector.getCNoun(taggedWords);
    wordDetector.removeWord(taggedWords, ".");

    Literal literal = new Literal(cNoun, negated);
    literal.getTerms().add(pNoun);

    AspRule aspRule = new AspRule();
    aspRule.getHead().add(literal);

    return aspRule;
}
```

The basic idea of the CNL to ASP translator is to filter out the key words of a sentence and remove those words that are not used to put together the ASP rule. As can be seen in the example above, the keywords are the proper noun, the negation, and the common noun. Those words are used to assemble the resulting ASP rule.

The CNL to ASP translator also checks if the words are of the right category (e.g. PNoun, CNoun, adjective, verb, ...). To do so, the translator makes use of the Stanford Parser<sup>4</sup>, a natural language parser that uses statistical data to parse sentences. In case a word is of a different type than expected, an exception is thrown, which tells the sentence type detector, to try if the sentence matches one of the remaining sentence patterns.

## 4.5 DLV

As already mentioned at the beginning of this chapter, we use DLV to solve the generated ASP programs. We chose DLV because it is a powerful state of the art answer-set solver.

Besides the generated ASP program, another input parameter is the list of predicates that the user entered in the intended text field.

The resulting models are returned to the graphical user interface, where the user can inspect them.

## 4.6 Features

In the following sections, we will describe some features of Uhura.

### 4.6.1 Dictionary

Sometimes the Stanford Parser tags some words wrong, and therefore, the problem description cannot be translated correctly.

Let's consider the following example:

Exclude that yellow is a color.

If we translate this sentence using Uhura, the system responds with the following error message:

```
Error in sentence "Exclude that yellow is a color .":
  Error in sentence "yellow is a color .":
    "yellow" is not a proper noun.
    (detected sentence-pattern: 'PNoun is [not] a CNoun.')
    (detected sentence-pattern: 'Exclude that SimpleSentence
                                {and that SimpleSentence}.')
```

---

<sup>4</sup>Stanford Parser: <https://nlp.stanford.edu/software/lex-parser.html>

In this particular case, the word *yellow* is used as a noun. However, the Stanford Parser tags the word *yellow* as an adjective.

To solve errors of such type, we provide a dictionary, which holds pairs of words and their corresponding types. In our case we add the word *yellow* and specify it as a *PNoun* (see Figure 4.3). Once we have added this pair, Uhura will translate our sentence correctly.

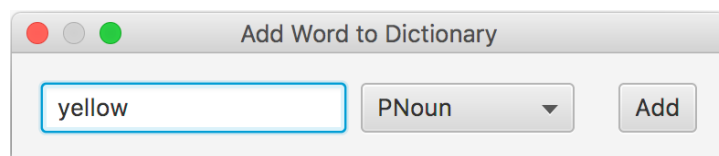


Figure 4.3: Add the word *yellow* to the dictionary.

#### 4.6.2 Self-Defined Sentence Patterns

Our tool only provides a limited number of sentence patterns. Therefore, we allow the user to add new sentence patterns that are based on those provided by Uhura. For instance, let's consider, the user wants to add the following pattern:

*PNoun* and *PNoun* are *Adjective*.

As can be seen in Figure 4.4, in order to add the discussed pattern, the user has to define how the pattern looks like and how it should be translated to the patterns known by the system. When specifying the pattern, the user has to insert capital letters (reference

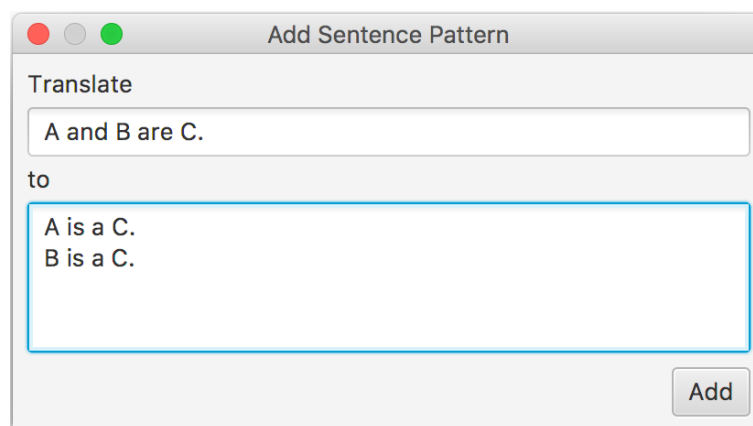


Figure 4.4: Add a new sentence pattern.

variables) at those position where key words are expected. Those reference variables can then be used defining the translation.

Once the user clicks the button *Add*, the system generates a regex pattern for the newly created pattern. In the case of our example, Uhura generates the following regex pattern:

`. * and . * are . * \ . $`

As can be seen, Uhura simply replaces the reference variables by `.*`, a random sequence of characters. This regex pattern is used in the same fashion as the other regex patterns. The sentence type detector checks if a sentence matches any of those regex patterns. Let's consider the user enters the following sentence:

Roberta and Thelma are female.

The sentence type detector will detect, that this sentence matches the sentence pattern, we just defined. Thus, it looks up the translation pattern the user defined and replaces the reference variables with the appropriate words. Our sentence gets translated into the following two sentences:

Roberta is female.

Thelma is female.

These two sentences are again used as an input for the sentence type detector.

### 4.6.3 Manual Translation

Sometimes the user does not agree with the translation of a specific CNL sentence. For instance, the user enters the following sentence:

If Roberta is not female then Roberta is male.

Uhura translates this sentence into the following ASP rule:

*male(roberta) : - not female(roberta).*

Nevertheless, the user expected that the literal *female(roberta)* would be strong negated. One option to solve this problem is that the user edits the generated ASP code and replaces the weak negation by a strong negation. The problem with this option is that the modification made by the user will be lost when translating the problem description again.

Therefore, we recommend that the user creates its own translation of that particular sentence. This can be done by selecting the sentence, then opening the context menu and clicking the appropriate context menu item. Then, the user has to specify the ASP translation for that sentence (see Figure 4.5)

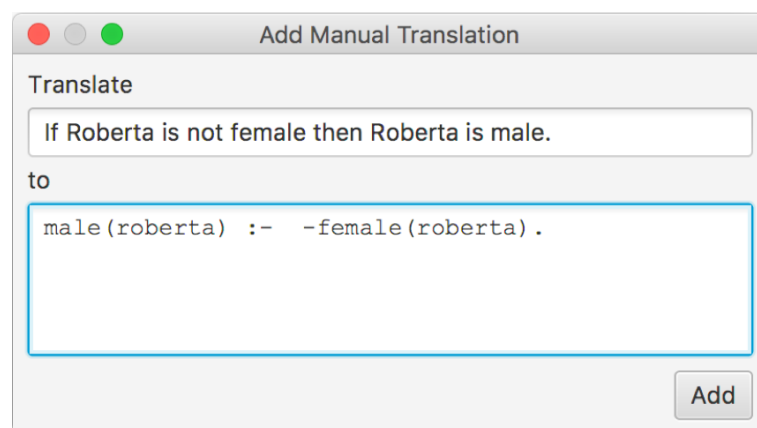


Figure 4.5: Manually translate a sentence.

## Related Work

The idea of building a tool for translating a problem description expressed in a (controlled) natural language into an ASP program is not a new one.

However, what makes our approach unique is that the aim of Uhura is not only to translate CNL sentences into ASP rules but to perform the translation in such a way that the user can learn how to write ASP programs. Most of the other tool available focus on solving the described problem.

In the following sections, we will discuss two related works.

### 5.1 PENG<sup>ASP</sup> System

Schwitter and Guy presented a web-based predictive editor for PENG<sup>ASP</sup> [Guy and Schwitter, 2016] [Guy and Schwitter, 2014]. The editor suggests for every word that is typed in a selection of words, from which the user can choose. Thus, sentences entered by the user are always valid PENG<sup>ASP</sup> sentences. Another advantage of this approach is that the user does not have to remember the different sentence patterns.

The unit that creates these suggestions is called controlled natural language processor, and it is located on the server. The editor, which acts as a client, communicates with the natural language processor asynchronously by receiving and transmitting JSON objects via the established connection. The natural language processor is also responsible for translating the problem description expressed in PENG<sup>ASP</sup> into an ASP program. The ASP program is then transferred to the clingo ASP tool, which solves the program.

Compared to Uhura, this system can handle more complicated sentences. For example, to express the information provided by the third Sentence of the jobs puzzle (see 3.1) the user types in the following sentence:

Every person holds exactly two jobs.

In order to formulate the same sentence using Uhura, the following two sentences must be used:

Exclude that there is a person X and that person X holds more than two jobs Y.

Exclude that there is a person X and that person X holds less than two jobs Y.

### 5.2 LOGICIA

Baral and Mitra developed a system, called LOGICIA, that can solve logic grid puzzles [Mitra and Baral, 2015]. To solve those puzzles, this tool translates the puzzles, described in natural language, into an answer-set program, which is then solved by an answer-set solver. The system automatically learns how to translate the information given by the puzzle description.

Obviously, the aim of LOGICIA differs from the goal of Uhura quite strongly. While our tool focuses on generating answer-set programs that can be easily understood even by developers inexperienced in ASP, LOGICA concentrates on the translation of sentences expressed in natural language. To do so, this tool represents answer-set rules in a more complicated and unintuitive way. For example, the clue

Hannah paid more than Teri's client.

is translated into the following ASP rules:

```
clue1 :-
    greaterThan(hannah, 1, X, 1, 2),
    sameTuple(X, 1, teri, 3).
:- not clue1.
```



## Conclusion and Future Work

In this thesis we presented a tool, named Uhura, that is capable of translating sentences expressed in a controlled natural language into an answer-set program. We showed that our system translates the sentences composed by the user in such a way that the resulting ASP program gives the impression that it was written by a developer experienced in ASP. Therefore, users inexperienced in ASP can learn how to express domain knowledge in ASP by observing the ASP program generated by Uhura.

Furthermore, in the course of this thesis, we demonstrated that our system consists of several units (user interface, sentence type detector, CNL to ASP translator, DLV). Due to this segmentation, we can replace a single unit without changing the others. As a result, our tool can be relatively easily adjusted to meet new goals.

One possible enhancement of Uhura would be to integrate this system into Sealion, which is an integrated development environment for answer-set programming. In this case, the only unit that has to be changed is the user interface. Furthermore, this integration would also allow providing a second ASP solver to solve the ASP programs generated by Uhura since Sealion supports two answer-set solvers: DLV and clingo.

Another promising future work would be to use Uhura for specifying test cases for software projects. [Irlinger, 2017] describes how answer-set programming can be used to define sequence-covering arrays. The idea would be using Uhura to allow the user specifying the sequence-covering array in the controlled natural language offered by our system and then translate this definition into ASP.



# User Manual

## A.1 Overview

As can be seen in Figure A.1, the user interface of Uhura can be segmented into three parts:

1. Menu (highlighted in red)
2. Coding area (highlighted in blue)
3. Bottom bar (highlighted in green)

## A.2 Implement Program

The coding area consists of two text editors, one for writing down the CNL problem description, and one for the translated ASP program.

The first task the user has to accomplish is to express the CNL problem description. To help the user formulating valid CNL sentences, our tool provides in the *Sentence Patterns* tab, which is located in the bottom bar, a list of all sentence patterns our tool supports.

To translate the CNL sentences into an ASP program the user simply has to click the *Translate* button (red framed in Figure A.2) or press *CTRL+T* respectively *CMD+T*<sup>1</sup>. Once the system has translated all the sentences entered by the user, the resulting ASP program is displayed in the editor next to the CNL editor. If the system is not able to translate a sentence entered by the user an error message regarding that sentence is displayed in the *Error* tab, which is located in the bottom bar (see Figure A.2).

---

<sup>1</sup>macOS

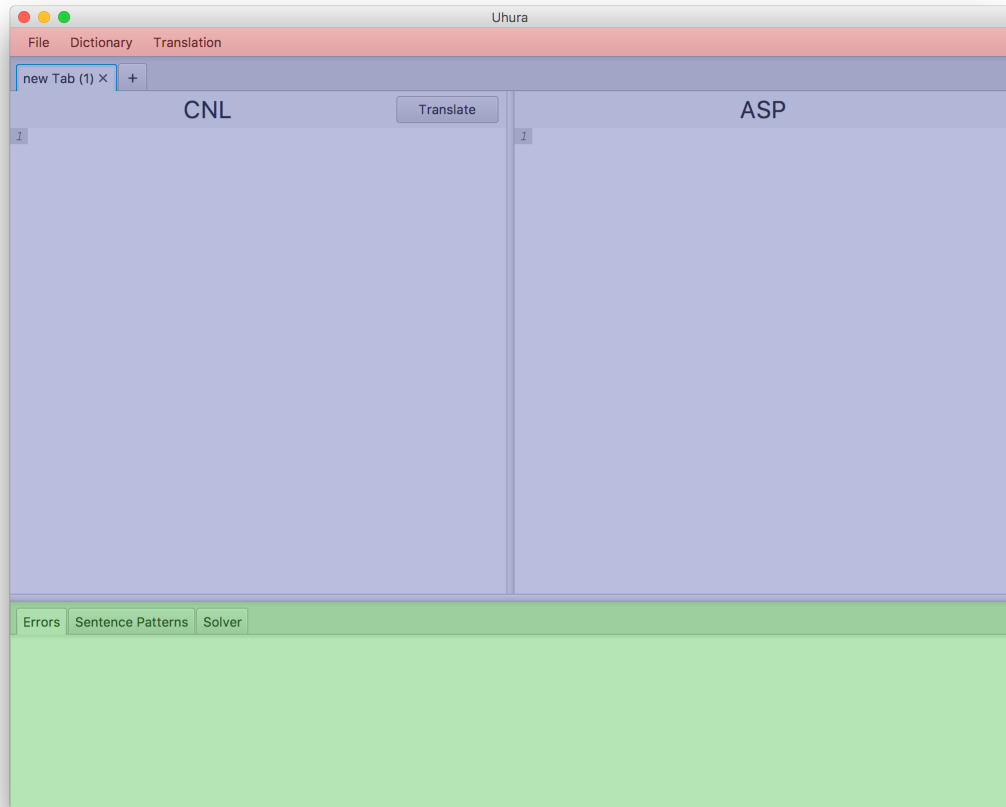


Figure A.1: The user interface of Uhura.

### A.2.1 File Operations

Besides formulating the CNL problem description in editor provided by our tool, the user can also use another text editor. To open a CNL problem description composed in another text editor the user has to open the *File* menu, as shown in Figure A.3, and select the menu item *Open*. Note that only files with the file extension *.txt* can be opened.

The *File* menu also allows the user to save the CNL problem description and to export the ASP generated by our tool.

### A.2.2 Comments

Uhura supports two kinds of comments:

- CNL comments
- ASP comments

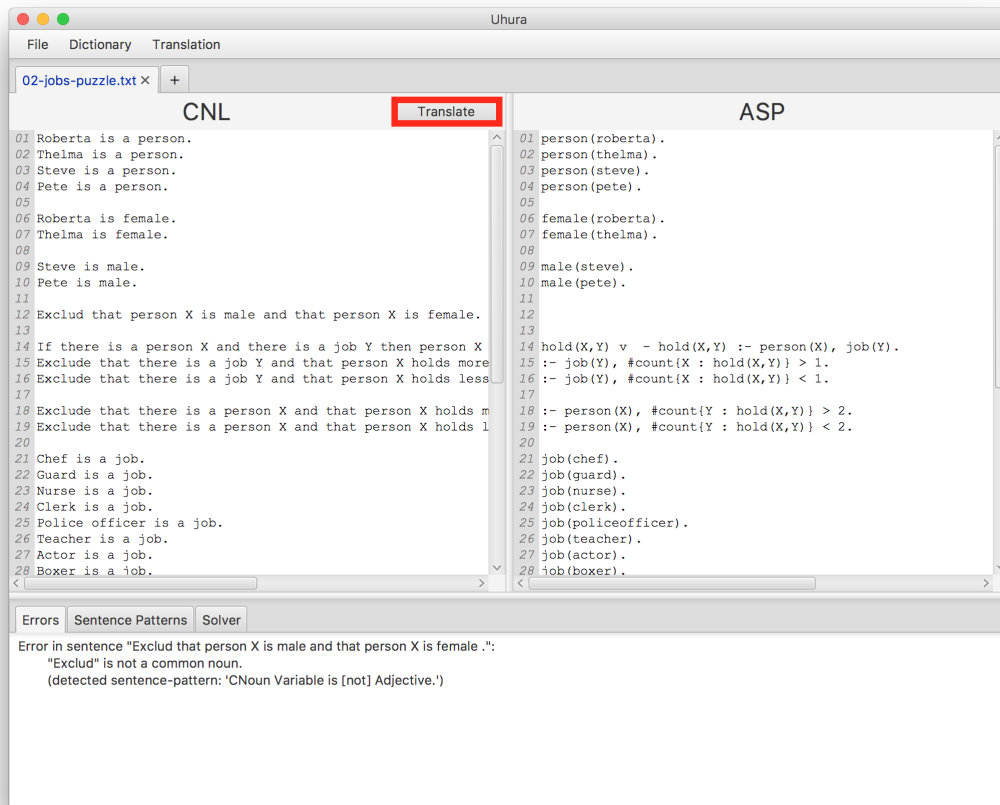


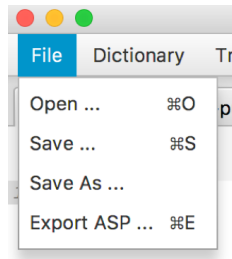
Figure A.2: The user interface showing the translated CNL program.

A CNL comment is a sequence of characters that starts with two slashes (//). As the name already implies, CNL comments can only be used in the CNL problem description.

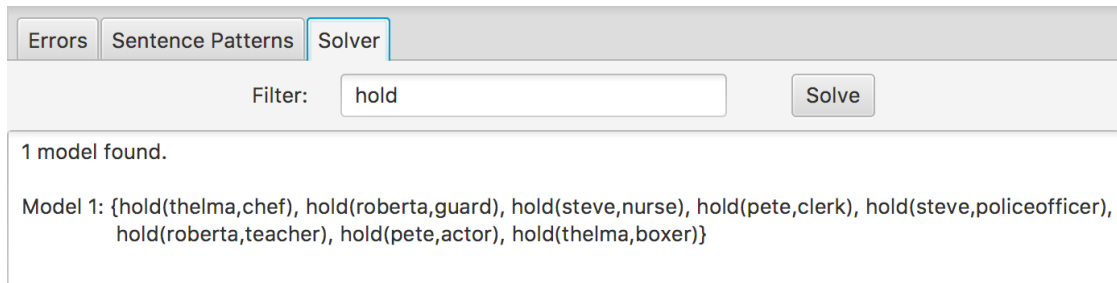
On the other hand, ASP comments can be used when writing the CNL problem description as well as in the ASP program. If the CNL problem description contains an ASP comment, Uhura copies that comment to the ASP program when the user clicks the *Translate* button. To indicate that a sequence of characters is an ASP comment, the user has to put a percent sign (%) in front of that sequence.

### A.3 Solve ASP Program

To solve the ASP program generated by Uhura, the user has to switch to the *Solver* tab, which is located in the bottom bar. Then the user has to click

Figure A.3: The *File* menu.

the *Solve* button and wait until the resulting models are displayed. As can be seen in Figure A.4, the models can be filtered by predicates defined by the user.

Figure A.4: The *Solver* tab.

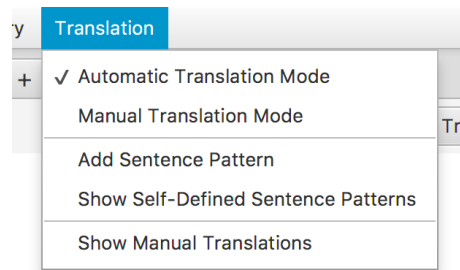
## A.4 Automatic Translation Mode

Uhura offers two translation modes:

- Manual translation mode
- Automatic translation mode

In manual translation mode (default mode) the user himself has to initiate the translation of the CNL problem description into an ASP program. This means that the user has to click the *Translate* button each time he wants to translate the composed CNL sentences.

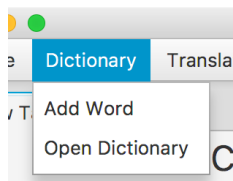
When the user activates the automatic translation mode Uhura translates the CNL problem description each time the user edits, deletes or adds a sentence. The automatic translation mode can be activated by clicking on the menu item called *Automatic Translation Mode* in the *Translation* menu (see Figure A.5).

Figure A.5: The *Translation* menu.

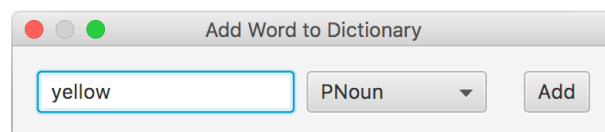
## A.5 Dictionary

Sometimes our tool tags some words wrong, and therefore, the problem description cannot be translated correctly. To solve errors of such type, we provide a dictionary, which holds pairs of words and their corresponding types.

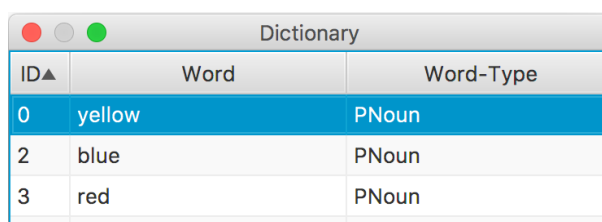
In order to add a word to the dictionary, the user has to open the *Dictionary* menu (see Figure A.6) and click the first menu item.

Figure A.6: The *Dictionary* menu.

Next, a dialog for adding a word to the dictionary appears. In this dialog, the user has to type in the word and select the corresponding type.

Figure A.7: Add the word *yellow* to the dictionary.

It is also possible to inspect and edit the entries stored in the dictionary. Therefore, the user has to open the dictionary by click the second menu item of the *Dictionary* menu (see Figure A.8). To edit or delete a word stored in the dictionary the user has to right-click the entry and select the appropriate item from the appearing context menu.



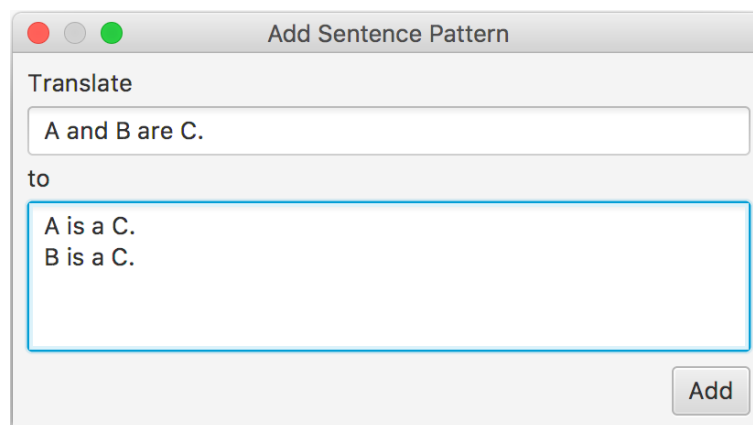
ID▲	Word	Word-Type
0	yellow	PNoun
2	blue	PNoun
3	red	PNoun

Figure A.8: The dictionary.

## A.6 Self-Defined Sentence Pattern

Our tool only provides a limited number of sentence patterns. Therefore, we allow the user to add new sentence patterns that are based on those provided by Uhura.

To add a new sentence pattern, the user has to click the menu item called *Add Sentence Pattern* of the *Translation* menu. As can be seen in Figure A.9, in order to add a pattern, the user has to define how the pattern looks like and how it should be translated to the patterns known by the system. When specifying the pattern, the user has to insert capital letters (reference variables) at those positions where keywords are expected. These reference variables can then be used defining the translation.



Translate

A and B are C.

to

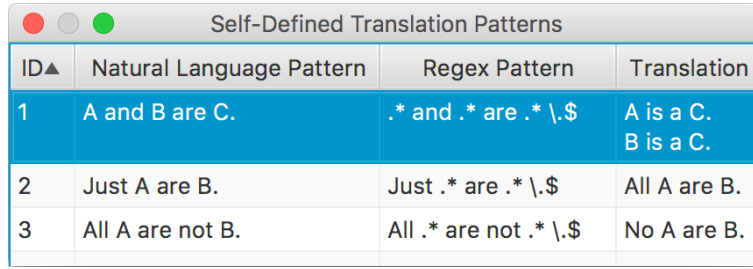
A is a C.  
B is a C.

Add

Figure A.9: Add a new sentence pattern.

To display and edit the self-defined sentence patterns, the user has to open the *Translation* menu and click *Show Self-Defined Sentence Patterns* (see Figure A.10).



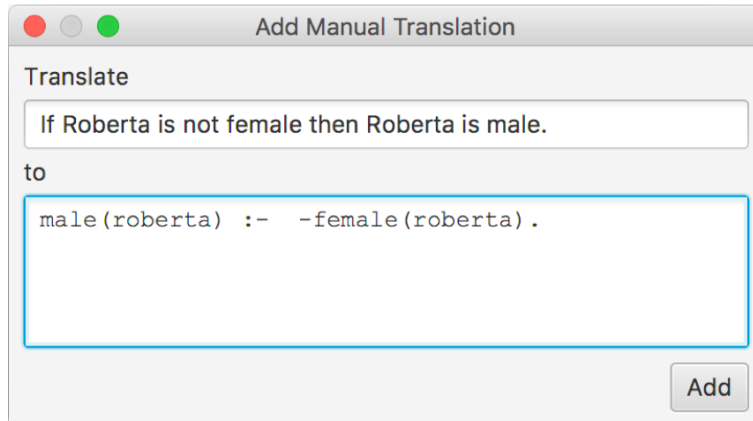


ID▲	Natural Language Pattern	Regex Pattern	Translation
1	A and B are C.	. * and . * are . * \. \$	A is a C. B is a C.
2	Just A are B.	Just . * are . * \. \$	All A are B.
3	All A are not B.	All . * are not . * \. \$	No A are B.

Figure A.10: List of self-defined sentence patterns.

## A.7 Manual Translations

Sometimes the user does not agree with the translation of a specific CNL sentence. Therefore, Uhura allows the user to create its own translation of that particular sentence. He can do this by selecting the sentence he wants to translate, then opening the context menu and clicking the appropriate context menu item. In the appearing dialog the user has to specify the translation of that sentence (see Figure A.11)



**Add Manual Translation**

Translate

If Roberta is not female then Roberta is male.

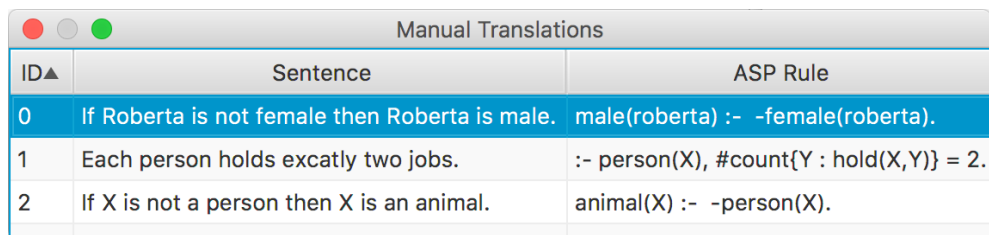
to

male(roberta) :- -female(roberta).

Add

Figure A.11: Manually translate a sentence.

The user can also view and edit the manually translated sentences. To do so, he has to open the *Translation* menu and click the last menu item. The appearing window (see Figure A.12) allows to edit and delete the manual translations by right-clicking a row and selecting the proper item from the context menu.



Manual Translations		
ID▲	Sentence	ASP Rule
0	If Roberta is not female then Roberta is male.	male(roberta) :- ¬female(roberta).
1	Each person holds exactly two jobs.	:- person(X), #count{Y : hold(X,Y)} = 2.
2	If X is not a person then X is an animal.	animal(X) :- ¬person(X).

Figure A.12: List of manual translations.

# Sentence Patterns

The following tables show the sentence patterns that our tool is aware of:

Nr.	Structure	Example
1	If <i>SimpleSentence</i> {and <i>SimpleSentence</i> } then <i>SimpleSentence</i> .	If X is a human then X is mortal.
2	Exclude that <i>SimpleSentence</i> {and that <i>SimpleSentence</i> }.	Exclude that Robert is a dog.

Table B.1: Complex Sentences

Table B.2: Simple Sentences

Nr.	Structure	Example
3	<i>PNoun</i> is a <i>CNoun</i> .	Roberta is a person.
4	<i>PNoun</i> is not a <i>CNoun</i> .	Roberta is not a dog.
5	<i>Variable</i> is a <i>CNoun</i> .	X is a person.
6	<i>Variable</i> is not a <i>CNoun</i> .	X is not a dog.
7	<i>CNoun Variable</i> is a <i>CNoun</i> .	Person X is a teacher.
8	<i>CNoun Variable</i> is not a <i>CNoun</i> .	Person X is not a headmaster.
9	<i>PNoun Verb</i> .	Roberta reads.
10	<i>PNoun</i> not <i>Verb</i> .	Roberta does not read.
11	<i>CNoun Variable Verb</i> .	Person X reads.
12	<i>CNoun Variable</i> not <i>Verb</i> .	Person X does not read.
13	<i>PNoun</i> is <i>Verb</i> .	Roberta is reading.
14	<i>PNoun</i> is not <i>Verb</i> .	Roberta is not reading.
15	<i>CNoun Variable</i> is <i>Verb</i> .	Person X is reading.

## B. SENTENCE PATTERNS

16	<i>CNoun Variable is not Verb.</i>	Person X is not reading.
17	<i>PNoun is Adjective.</i>	Roberta is educated.
18	<i>PNoun is not Adjective.</i>	Roberta is not tall.
19	<i>Variable is Adjective.</i>	X is tall.
20	<i>Variable is not Adjective.</i>	X is not small.
21	<i>CNoun Variable is Adjective.</i>	Person X is educated.
22	<i>CNoun Variable is not Adjective.</i>	Person X is not educated.
23	<i>PNoun is Preposition PNoun.</i>	Roberta is at work.
24	<i>PNoun is not Preposition PNoun.</i>	Roberta is not at work.
25	<i>PNoun is Adjective Preposition PNoun.</i>	Roberta is married to Bob.
26	<i>PNoun is not Adjective Preposition PNoun.</i>	Roberta is not married to Bob.
27	<i>CNoun Variable is Preposition PNoun.</i>	Person X is at work.
28	<i>CNoun Variable is not Preposition PNoun.</i>	Person X is not at work.
29	<i>CNoun Variable is Adjective Preposition PNoun.</i>	Person X is married to bob.
30	<i>CNoun Variable is not Adjective Preposition PNoun.</i>	Person X is not married to bob.
31	<i>PNoun is Preposition CNoun Variable.</i>	Roberta is in room X.
32	<i>PNoun is not Preposition CNoun Variable.</i>	Roberta is not in room X.
33	<i>PNoun is Adjective Preposition CNoun Variable.</i>	Roberta is married to person X.
34	<i>PNoun is not Adjective Preposition CNoun Variable.</i>	Roberta is not married to person X.
35	<i>CNoun Variable is Preposition CNoun Variable.</i>	Person X is in room Y.
36	<i>CNoun Variable is not Preposition CNoun Variable.</i>	Person X is not in room Y.
37	<i>CNoun Variable is Adjective Preposition CNoun Variable.</i>	Person X is married to person Y.
38	<i>CNoun Variable is not Adjective Preposition CNoun Variable.</i>	Person X is not married to person Y.
39	<i>[A] PNoun Verb a CNoun as [a] PNoun.</i>	Roberta holds a job as a nurse.
40	<i>[A] PNoun not Verb a CNoun as [a] PNoun.</i>	Roberta does not hold a job as a nurse.
41	<i>[A] CNoun Variable Verb a CNoun as [a] PNoun.</i>	Person X holds a job as nurse.
42	<i>[A] CNoun Variable not Verb a CNoun as [a] PNoun.</i>	Person X does not hold a job as a nurse.

---

43	[A] <i>CNoun Variable Verb</i> [a] <i>CNoun Variable</i> .	Person X holds job Y.
44	[A] <i>CNoun Variable not Verb</i> [a] <i>CNoun Variable</i> .	Person X does not hold job Y.
45	[A] <i>CNoun Variable Verb Preposition</i> [a] <i>CNoun Variable</i> .	Person X works for person Y.
46	[A] <i>CNoun Variable not Verb Preposition</i> [a] <i>CNoun Variable</i> .	Person X does not work for person Y.
47	<i>CNoun Variable Verb PNoun</i> .	Person X studies computer science.
48	<i>CNoun Variable not Verb PNoun</i> .	Person X does not studies computer science.
49	<i>CNoun Variable Verb Preposition PNoun</i> .	Person X studies at TU Wien.
50	<i>CNoun Variable not Verb Preposition PNoun</i> .	Person X does not study at TU Wien.
51	<i>PNoun Verb CNoun Variable</i> .	Roberta holds job X.
52	<i>PNoun not Verb CNoun Variable</i> .	Roberta doesn't hold job X.
53	<i>PNoun Verb Preposition CNoun Variable</i> .	Roberta works for person X.
54	<i>PNoun not Verb Preposition CNoun Variable</i> .	Roberta does not work for person X.
55	<i>PNoun Verb PNoun</i> .	Roberta loves Bob.
56	<i>PNoun not Verb PNoun</i> .	Roberta doesn't love Bob.
57	<i>PNoun Verb Preposition PNoun</i> .	Roberta works for Bob.
58	<i>PNoun not Verb Preposition PNoun</i> .	Roberta does not work for Bob.
59	There is a <i>CNoun Variable</i> .	There is a person X.
60	There is not a <i>CNoun Variable</i> .	There is not a person X.
61	<i>CNoun Variable Verb</i> more than <i>Number CNoun Variable</i> .	Person X holds more than two jobs Y.
62	<i>CNoun Variable Verb</i> less than <i>Number CNoun Variable</i> .	Person X holds less than two jobs Y.
63	A <i>CNoun Variable</i> is a <i>CNoun</i> of a <i>CNoun Variable</i> .	A person X is a husband of a person Y.
64	A <i>CNoun Variable</i> is not a <i>CNoun</i> of a <i>CNoun Variable</i> .	A person X is not a husband of a person Y.
65	<i>PNoun</i> is [the] <i>CNoun</i> of [a] <i>CNoun Variable</i> .	Bob is the father of person X.
66	<i>PNoun</i> is not [the] <i>CNoun</i> of [a] <i>CNoun Variable</i> .	Bob is not the father of person X.
67	[A] <i>CNoun Variable</i> is [the] <i>CNoun</i> of [a] <i>PNoun</i> .	Person X is the father of Bob.

## B. SENTENCE PATTERNS

---

68	[A] <i>CNoun Variable</i> is not [the] <i>CNoun</i> of [a] <i>PNoun</i> .	Person X is not the father of Bob.
69	<i>PNoun</i> is [the] <i>CNoun</i> of [a] <i>PNoun</i> .	Bob is the father of Roberta.
70	<i>PNoun</i> is not [the] <i>CNoun</i> of [a] <i>PNoun</i> .	Bob is not the father of Roberta.
71	<i>Fact</i> {or <i>Fact</i> }.	Roberta is a person or Roberta is a dog.

Nr.	Structure	Example
72	<i>CNoun</i> normally <i>Verb</i> .	Birds normally fly.
73	<i>CNoun</i> normally not <i>Verb</i> .	Animals normally do not fly.
74	<i>CNoun</i> normally are <i>Adjective</i> .	Birds normally are beautiful.
75	<i>CNoun</i> normally are not <i>Adjective</i> .	Birds normally are not ugly.
76	<i>CNoun</i> normally are <i>Adjective Preposition CNoun</i> .	Students normally are afraid of math.
77	<i>CNoun</i> normally are not <i>Adjective Preposition CNoun</i> .	Students normally are not afraid of architecture.

Table B.3: Default Sentences

Nr.	Structure	Example
78	All <i>CNoun</i> are <i>CNoun</i> .	All politicians are humans.
79	All <i>CNoun</i> are <i>Adjective</i> .	All politicians are corrupt.
80	No <i>CNoun</i> are <i>CNoun</i> .	No politicians are humans.
81	No <i>CNoun</i> are <i>Adjective</i> .	No politicians are corrupt.
82	Some <i>CNoun</i> are <i>CNoun</i> .	Some politicians are humans.
83	Some <i>CNoun</i> are not <i>CNoun</i> .	Some politicians are not humans.
84	Some <i>CNoun</i> are <i>Adjective</i> .	Some politicians are corrupt.
85	Some <i>CNoun</i> are not <i>Adjective</i> .	Some politicians are not corrupt.

Table B.4: Categorical Proposition Sentences

# Bibliography

- [ASD, 2005] ASD (2005). Asd simplified technical english: Specification asd-ste100. *AeroSpace and Defence Industries Association of Europe*.
- [Baral, 2003] Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press.
- [Baral and Son, 2008] Baral, C. and Son, T. C. (2008). Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions.
- [Bernth, 1997] Bernth, A. (1997). Easyenglish: a tool for improving document quality. In *Proceedings of the fifth conference on Applied natural language processing*, pages 159–165. Association for Computational Linguistics.
- [Brain et al., 2012] Brain, M., Erdem, E., Inoue, K., Oetsch, J., Pührer, J., Tompits, H., and Yilmaz, C. (2012). Event-sequence testing using answer-set programming. *International Journal on Advances in Software Volume 5, Number 3 and 4, 2012*.
- [Brewka et al., 2011] Brewka, G., Eiter, T., and Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103.
- [Eiter et al., 2009] Eiter, T., Ianni, G., and Krennwallner, T. (2009). Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer.
- [Eiter et al., 2008] Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., and Tompits, H. (2008). Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539.
- [Erdem et al., 2012] Erdem, E., Aker, E., and Patoglu, V. (2012). Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, 5(4):275–291.
- [Faber et al., 2012] Faber, W., Leone, N., and Perri, S. (2012). The intelligent grounder of dl<sub>v</sub>. In *Correct Reasoning*, pages 247–264. Springer.

- [Fuchs and Schwitter, 1996] Fuchs, N. E. and Schwitter, R. (1996). Attempto controlled english (ace). In *The First International Workshop On Controlled Language Applications*. Katholieke Universiteit Leuven.
- [Gebser et al., 2011] Gebser, M., Kaminski, R., König, A., and Schaub, T. (2011). Advances in gringo series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 345–351. Springer.
- [Gebser et al., 2012] Gebser, M., Kaufmann, B., and Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89.
- [Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080.
- [Guy and Schwitter, 2014] Guy, S. and Schwitter, R. (2014). Architecture of a web-based predictive editor for controlled natural language processing. In *International Workshop on Controlled Natural Language*, pages 167–178. Springer.
- [Guy and Schwitter, 2016] Guy, S. C. and Schwitter, R. (2016). The pengasp system: architecture, language and authoring tool. *Language Resources and Evaluation*, pages 1–26.
- [Huijsen, 1998] Huijsen, W.-O. (1998). Controlled language—an introduction. In *Proceedings of CLAW*, volume 98, pages 1–15.
- [Irlinger, 2017] Irlinger, M. (2017). Combinatorial testing using answer-set programming.
- [Kaljurand, 2007] Kaljurand, K. (2007). *Attempto controlled English as a Semantic Web language*. Tartu University Press.
- [Kamp and Reyle, 2013] Kamp, H. and Reyle, U. (2013). *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*, volume 42. Springer Science & Business Media.
- [Kamprath et al., 1998] Kamprath, C., Adolphson, E., Mitamura, T., and Nyberg, E. (1998). Controlled language for multilingual document production: Experience with caterpillar technical english. In *Proceedings of the Second International Workshop on Controlled Language Applications*, volume 146.
- [Kuhn, 2014] Kuhn, T. (2014). A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170.
- [Leone et al., 2006] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The dlv system for knowledge



- representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562.
- [Lifschitz, 2002] Lifschitz, V. (2002). Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54.
- [Lifschitz, 2008] Lifschitz, V. (2008). What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597.
- [Mitra and Baral, 2015] Mitra, A. and Baral, C. (2015). Learning to automatically solve logic grid puzzles. In *EMNLP*, pages 1023–1033.
- [Nyberg and Mitamura, 2000] Nyberg, E. and Mitamura, T. (2000). The kantoo machine translation environment. *Envisioning Machine Translation in the Information Future*, pages 21–22.
- [Reiter, 1980] Reiter, R. (1980). A logic for default reasoning. *Artificial intelligence*, 13(1-2):81–132.
- [Ricca, 2003] Ricca, F. (2003). The dlv java wrapper. In *APPIA-GULP-PRODE*, pages 263–274.
- [Sakama, 2001] Sakama, C. (2001). Learning by answer sets. In *Answer Set Programming*.
- [Schwitter, 2002] Schwitter, R. (2002). English as a formal specification language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 228–232. IEEE.
- [Schwitter, 2008] Schwitter, R. (2008). Working for two: A bidirectional grammar for a controlled natural language. In *Australasian Joint Conference on Artificial Intelligence*, pages 168–179. Springer.
- [Schwitter, 2010] Schwitter, R. (2010). Controlled natural languages for knowledge representation. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 1113–1121. Association for Computational Linguistics.
- [Schwitter, 2013] Schwitter, R. (2013). The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming*, 13(4-5):487–501.
- [Wos et al., 1984] Wos, L., Overbeck, R., Lusk, E., and Boyle, J. (1984). Automated reasoning: introduction and applications.