Development of an Intelligent Tutor for Description Logics

Christel Kemke
Shamima Mithun
Department of Computer Science
University of Manitoba, Winnipeg, Canada
{ckemke,shamima}@cs.umanitoba.ca

Abstract

Description Logic languages have gained a lot of attention and more and more relevance over the past 10 or 20 years. Although it can be foreseen, that this development will have a serious impact on education and training of IT professionals, in academics and industry, the expertise in this area is rare, and training is so far mostly constrained to occasional tutorials at conferences, or written resources like course notes and manuals.

In this paper, we introduce the DL Tutor, an integrated, interactive Tutoring System for Description Logics. We discuss the principle components and architecture of the DL Tutor, which includes an error diagnosis and feedback module, as well as a verbalization component for transforming DL expressions into "normal" natural language, and describe its current implementation. The DL Tutor can be used as an additional interface on top of a DL knowledge representation system. Currently, the DL Tutor works with the Loom/PowerLoom language and knowledge representation system.

1 Introduction

Description Logics (DL) are a family of knowledge representation languages that have been extensively studied in the Artificial Intelligence community over the past 20 years [3, 4, 8, 14, 15, 12]. The development of Description Logics as formal knowledge representation language in Artificial Intelligence is going back to a publication by Brachman and Schmolze in 1985 describing 'KL-ONE' ('Knowledge Language One') [5]. The proposal of KL-ONE was undoubtedly a ground-breaking milestone in the development of Knowledge Representation

Languages, which initiated a trend towards better formal foundations of KR and AI in general. In this context, Description Logics as KR paradigm is becoming more and more influential. DL also gained considerable attention in the past few years, in particular as a core language for the development of the Semantic Web [2, 9, 10, 13]. Therefore, it is to be expected that Description Logics languages will become more widely used in the near future. This will affect the computer science community in academics as well as in industry, and leads to the question of how to introduce the concepts of DL to students and IT professionals, and get them acquainted with the use of DL as KR language and its underlying formal foundations.

Tutorials for Description Logics exist so far only as human presentations, or electronically as written text, with additional exercises, but without integration into an implemented knowledge representation system, like Loom/PowerLoom or Classic. Since description logic languages are declarative languages but in some ways similar to programming languages, they offer an ideal basis for the development of an integrated tutoring system, which allows learning-while-doing as emphasized in [11].

The explicitly well-defined semantics of DL languages provides in addition a good basis for detecting conceptual misunderstandings and errors, and for providing advice on a deeper level, which has not yet been achieved for automated programming languages tutors so far (cf. the Smalltalk Tutor by Chee et al. [6, 7], the interactive Lisp textbook ELM-ART II by Weber and Specht [18], the Java Tutor by Sykes and Franek [17] and - with a different view - the Tutoring System SIAL for Computational Logic [16] and the MinLog system, a Proof System for Logic with an integrated Tutor [1]).

This paper outlines the development of an interactive, integrated tutoring system for Description Logics, the so-called **DL Tutor**. The current version of the DL Tutor is integrated with PowerLoom, a fully functional, implemented knowledge representation system, whose core language is a variant of Description Logics [12].[1]

2 Overview of the DL Tutor

The DL Tutor is aimed at providing user assistance on several levels, starting with a syntactic analysis of the user input. The DL tutor diagnoses syntax errors made by the users and classifies these errors according to the constructs or concepts of the DL language the novice DL user did not know, gives some explanation on the errors made and the correct syntax, and thus provides advice and help on writing DL expressions correctly. In addition, the DL Tutor

[1]This project has been funded by the University of Manitoba Research Grants Program, URGP

comprises a natural language verbalization component, which transfers DL expressions into natural language statements. The natural language formulation of the DL construct should be easier to read and more understandable than the original DL expression, and thus help the user to become clear about what s/he has communicated to the knowledge representation system.

The following sections outline the general architecture of the DL Tutor and its modules, and describe the current status of implementation and design.

3 Architecture of the DL Tutor

The DL Tutor is intended to act as an interface between the user and the PowerLoom system (or another DL system). The user enters DL expressions as usual. The DL Tutor analyzes every entered expression on the syntactic level, checks it for correctness, and if necessary produces an error message and error feedback. The tutor also performs a verbalization of the DL expression, i.e. it generates a natural language sentence corresponding to the meaning of the DL expression. The current DL tutor comprises the following main modules (see fig. 1):

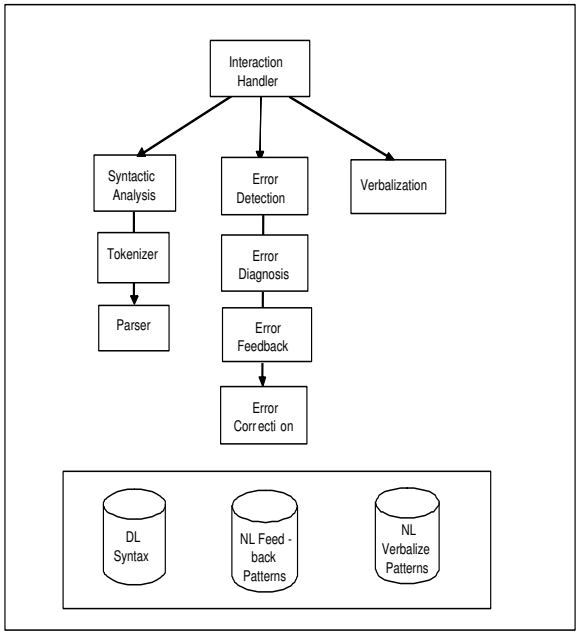

Figure 1: *Overall Architecture of the DL Tutor*

- the Syntactic Analysis module,
- the Error Diagnosis and Feedback module, and

- the Verbalization module.

The *Syntactic Analysis* is a kind of pre-processing module, which checks whether the user input is adhering to the syntax of the given DL language, i.e. in this implementation the PowerLoom language. If the user's input is syntactically correct, it will be directly passed to the *Verbalization*, which generates a natural language form of the DL construct. Both of these modules, as well as the Error Diagnosis, access as common resource a set of grammar rules, which describe the syntax of the underlying DL language. The *Error Diagnosis* provides a classification and specification of the error, and thus enables a focused, adjusted feedback through the *Error Feedback* module. The generation of error feedbacks involves natural language templates, i.e. patterns whose variables are replaced with respective verbal tokens from the analyzed input. The instantiated pattern is then modified to form a proper natural language sentence. The Feedback module has in addition an Error Correction component, which constructs correct DL expressions from the entered ill-formed structures, based on the diagnosed error and the DL (PowerLoom) lexicon and grammar as stored in the DL Syntax source (see fig.2).

3.1 Processing in the DL Tutor

The processing is performed in the DL Tutor according to the following steps:

1. Prompt the user to input a DL expression.
2. Read the user input.
3. Tokenize the input. In the tokenization, the system separates each token by identifying spaces or parentheses.
4. The tokens are then tagged using the pre-defined DL grammar (s. fig.2).
5. Next, the tokenized input is parsed using a top-down parsing method.
6. If a parsing error occurs, the input statement is first checked for a possible mistake in the use of DL constructor keywords (e.g. defconcept, defrelation etc.).
7. If a keyword error is found, the user receives a feedback message with an indication of the error and a suggested replacement for the wrong keyword.
8. If any other error was found during parsing, the respective error is reported to the user. The tutor system generates an error message based on the grammar rule(s) which failed during parsing. The generation of the error message is using natural language templates.
9. The tutor then suggests a correction of the input. If the user accepts the correction, the modified expression is assumed as new DL statement, and the system continues with an analysis in step 4.

10. If no error occurred during parsing, a template based verbalization of the input expression takes place. For various kinds of DL constructs, there are pre-defined templates, which are instantiated with the tagged tokens derived in steps 3 and 4. In a recursive manner, each sub-expression of the parsed DL statement - corresponding to phrases in natural language parsing - is transformed into a natural language expression. Then the transformed sub-expressions are combined and further verbalized to form a complete natural language sentence. This sentence is displayed as output to the user.

3.2 Syntactic Analysis

The Syntactic Analysis module first performs the tokenization in which it separates all the opening brackets, closing brackets, keywords, operators, variables and other words occurring in the DL formula. Then it assigns syntactic markers, so-called 'tags' to every token and parses the input expression, using predefined grammar rules for the specific DL language. We use, for example, the following predefined grammar rules to parse 'assert'-statements in PowerLoom:

assertS $\rightarrow$ *(assert cond)/(assert fa)*
fa $\rightarrow$ (*forall*(*V*)(*opr cond cond*))
V $\rightarrow$ *var* / *var V* / (*var con*) *V*
opr $\rightarrow \Rightarrow$ / $\Leftrightarrow$
var $\rightarrow$ *is-alpha*
cond $\rightarrow$ (*term*) / (*bbool cond cond*) (*ubool cond*) (*term2*)
bbool $\rightarrow$ *and* / *or*
ubool $\rightarrow$ *not*
term2 $\rightarrow$ *con value* / *rel con value*
value $\rightarrow$ *is-alpha*
term $\rightarrow$ *rel V* / *rel value V* / *con var* / *rel value con var*

where rel is a user-defined relation and con is a user-defined concept.[2] If the tutor receives, for example, the input

(*assert* (*company MegaSoft*))

it will first separate 'assert', 'company', 'MegaSoft' and all the brackets as tokens. Next, the system will use the grammar rules to tag the tokens and parse the input. Firstly, it tags the complete formula as an assert statement because it finds a match with the rule asserS $\rightarrow$ (assert cond). The parsing process then recognizes that the sub-expression '(company MegaSoft)' can be marked

[2]Initially, the system has no user-defined concepts or relations. If the user enters during runtime a correct statement defining a concept/relation, this concept/relation is added to the concept-list/relation-list.

as cond, 'company' as con and 'MegaSoft' as var.

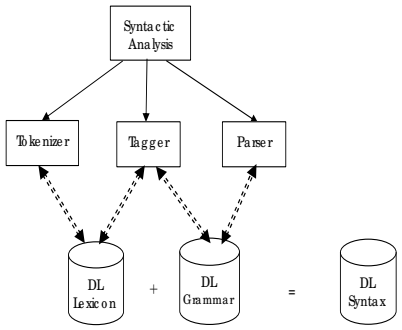

Figure 2: *Syntactic Analysis*

3.3 Error Diagnosis and Feedback

The DL Tutor detects an error based on problems during the syntactic analysis. Currently, diagnosed errors are either on the lexical token level, or on the grammatical phrase/sentence level. Typical errors on the token level include a wrong spelling of DL keywords, which occurs when the user is not yet accustomed with the respective DL syntax and forgot the exact form of the keyword, or in particular when s/he used another DL dialect before. Another typical error on the token level is a wrong reference to a non-existing concept or relation in an assert-statement. The tutor is able to check whether the respective concept or relation referred to in the assert statement has already been defined and exists in the knowledge base; if this is not the case, the statement will be marked as faulty using a respective error identification. Other simple errors on this level are missing parentheses, and wrong use of variable identifiers.

Errors on the grammatical level are detected during parsing, based on violations of the given grammatical rules, which result in a parser failure. Since the parser has from the tokenization process knowledge about the complete input structure, the tutor can diagnose the location of the error as well as indicate a possible correction, by comparing the generated, faulty parse with possibly applicable grammatical rules at the respective parse position.

The Error Feedback takes the error information as determined by the Error Identification, as well as the generated parse output and the general syntactic knowledge to produce an adequate response for the user. Firstly, the identified error is printed out to the user, for example that an unidentified token should be a keyword. The location of the error in the input structure is indicated as

well. Next, a correction statement is produced by referring to a similar proper token or grammatical rule, for example suggesting a proper keyword which can replace the wrong token. In addition, last, the whole statement is corrected and suggested to the user.

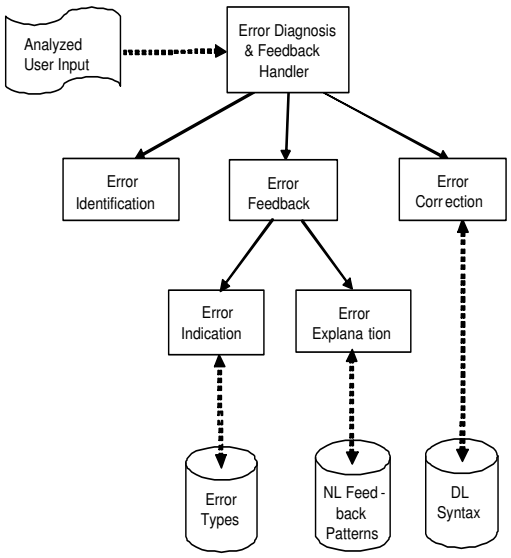

Figure 3: *Components of the Error Diagnosis and Feedback Module*

This whole procedure should achieve that

- the user becomes aware of certain mistakes s/he makes or misconceptions and misunderstandings s/he might have;
- the user receives information about how to correct this mistake;
- the user may receive further information in the form of explanations of grammatical rules (not yet completely implemented)
- the user does not have to retype the DL expression if the tutor's diagnosis and the suggested modified expression is correct and acceptable.

The following are samples of dialogues between the DL Tutor and a user. [3]

Example 1

DL input: *(defrelati has-floor ((?b building) (?f floor)))*
Syntax is not correct.
Incorrect start 'defrelati' of DL statement.

[3]For better readability, the communication printed here has been slightly modified from the original system display.

Is *defrelati* meant to be *defrelation*? (Y/N) : **y**
Verbalization: 'has-floor' is a relation between 'building' and 'floor'.

Comment: The first line contains the tutor's system prompt and the DL input from the user. The tutor detects a syntax error and communicates this to the user (line 2). In line 3, the error is identified as a wrong start of a DL statement, and the respective wrong token is mentioned ('defrelati' should have been a keyword). The tutor can find a suitable keyword and suggests to substitute the wrong token with it. The last line contains the natural language form generated by the Verbalization module.

Example 2

DL input: *(assert (position left)*
Syntax is not correct.
The closing parenthesis) is missing after *left)*.

Comment: In this example, the tutor diagnoses a missing closing bracket, and produces an error feedback which includes an error identification and location, in this case the last matching closing bracket plus the word before.

Example 3

DL input: *(assert (foll (?x ?y)(⇒(has-color ?x ?y)(and (furniture ?x)(color ?y)))))*
Syntax is not correct.
foll must be a defined concept, or substitute *foll* with *forall*.

Comment: Both diagnosed error types, i.e. the wrong token 'foll' referring either to an unknown concept, or meaning the 'forall' constructor, are possible errors.

It is possible to include further mechanisms, like pattern matching and completion, or a further look-ahead and better prediction during the parsing process to resolve ambiguities as in example 3 above. This would improve the quality (specificity and correctness) of the error diagnosis and identification.

3.4 Verbalization

The Verbalization component currently works with a subset of the PowerLoom language. The syntactic constructs which are mostly used and most common to all DL languages, are modelled using a syntax adapted to PowerLoom(see fig.2). Basic constructs included in the current version are *defconcept*, *defrelation*, *assert*, and *forall*.

The Verbalization module transforms DL formulas into natural language expressions using

- the tokenized, parsed input produced by the Syntax Analysis,

- natural language templates for sub-structures,
- construct-specific transformations.

Once the tagging and parsing are complete, the system uses the produced information to transform the DL formula into a natural language sentence. The generation process is progressing recursively from the innermost terms in the DL statement, which is interpreted first, outwards to the outmost expressions, until the entire statement is transformed into a natural language expression.

In the following example, a relation definition 'has-floor' is verbalized. The transformation of the expression:

(defrelation has-floor ((?b building) (?f floor)))

yields the verbalization:

'has-floor' is a relation between 'building' and 'floor'.

In the following example, a relation constraint *forall* is asserted to the knowledge base, involving variables standing for instances of the related concepts. The literal transformation of the expression

(assert (forall (?x ?y) (⇒ (has-color ?x ?y) (and (furniture ?x) (color ?y)))))

produces the output

If the relation 'has-colour' holds between any instances ?x and ?y, then ?x must be a 'furniture' and ?y must be a 'colour'.

Slightly different forms of outputs have been generated so far in various tests of the system interaction. It is intended to make the choice of the output form dependent on either the purpose of the application, i.e. whether it is supposed to act more like a human tutor and communicating agent with the user (meta-natural or meta-technical mode), or whether it is supposed to act as a passive, reactive, feedback system which just reflects the user's input or the user's action (natural or technical mode).

DL input: *(assert (position left))*
natural: *left* is a *position.*
meta/natural: You asserted *left* as a *position.*
technical: *left* is an instance of the concept *position.*
meta/technical: You asserted that *left* is an instance of the concept *position.*

DL input: *(defconcept yes (?x flat-surface))*
meta-natural: You defined *yes* as a kind of *flat-surface.*
meta-technical: You defined a new concept *yes* as sub-concept of the concept *flat-surface.*

Last, the hypothetical example below shows different forms of response by the

DL Tutor. These can be produced relatively easy based on the processing mechanisms already implemented in the current version of the DL Tutor:

DL input: (ssert (position left))
error detection: Syntax is not correct.
error diagnosis: Incorrect start 'ssert' of the DL statement.
exact explanation: 'ssert' should be a DL constructor.
general explanation: The first token after the first opening parenthesis must be a DL constructor.
verbose: 'ssert' should be a DL constructor, since it is the first token after the first opening parenthesis.
correction: Is 'ssert' meant to be assert? (Y/N): **y**
verbalization: 'left' is a 'position'.
confirmation: You asserted 'left' as a 'position'.

4 Conclusion

The DL Tutor has been designed and implemented in a prototypical form, currently with an integrated connection to a paradigmatic, functional DL language, the PowerLoom knowledge representation and reasoning system. The DL tutor provides a syntactic analysis, with an error detection and diagnosis facility for the syntactic language level; an error feedback which provides comments related to occurred errors at different levels of detail and technicality; and a correction of ill-formed DL user inputs. Another branch of the DL Tutor deals with the verbalization of DL expressions, i.e. the generation of natural language sentences corresponding to the DL construct specified by the user.

The DL Tutor still needs improvement, for example in the verbalization and natural language generation part, which should produce more coherent and eloquent expressions. The verbalization should include at a later stage also a 'describe' feature, which is supposed to collect from the knowledge base all descriptive information related to a specific concept (or instance or relation), and provide a coherent textual description of this concept.

Future developments of the DL Tutor include the integration of a User Model. This will provide a better adaptation to the specific user, e.g. by storing preferred interaction and verbalization modes, and - more importantly - to keep track of the user's knowledge and mistakes, by recording the DL constructs which have been properly used ('positive knowledge') and those which have been incorrectly used by the specific user ('negative knowledge').

An adaptation to other DL languages is also in the scope of future work. The current parser, which handles only right-recursion, will be substituted with an Early-Parser to allow arbitrary context-free grammar rules. The verbalization module does require some modified procedures to build a complete verbalization

recursively, based on the pre-defined verbal templates, which are currently used for tranforming sub-structures of a DL-expression into natural language phrases.

In the longer term, the DL Tutor should also include and use semantic knowledge. For example, a more semantically oriented error diagnosis and feedback can address issues of consistency and coherence related to adding new definitions to an existing knowledge base. A mini-version of this feature, which checks on the proper use of existing concepts and relations in input constructs, is already integrated in the current system.

Tests of an improved version of the DL Tutor are planned for the following months, with higher-level computer science students in the context of a formal logic course and a natural language processing course.

References

[1] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the minlog system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction, Vol. II*. Kluwer, 1998.

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001, 2001.

[3] Ronald J. Brachman, Alex Borgida, Deborah L. McGuiness, and Peter F. Patel-Schneider. "Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality. *Artificial Intelligence*, 114(1-2):203–237, 1999.

[4] Ronald J. Brachman, Deborah L. McGuiness, Peter F. Patel-Schneider, and Lori A. Resnick. Living with CLASSIC: when and how to use a KL-ONE-like language. In John Sowa, editor, *Principles of Semantic Networks*. Morgan Kaufmann, San Mateo, US, 1990.

[5] Ronald J. Brachman and J. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[6] Y.S. Chee. SMALLTALKER: A cognitive apprenticeship multimedia learning environment for learning Smalltalk programming. In *Proceedings of ED-MEDIA 94-World Conference on Educational Multimedia and Hypermedia*, pages 492–497, Vancouver, BC, Canada, 1994.

[7] Y.S. Chee and S. Xu. SIPLeS: supporting intermediate Smalltalk programming through goal-based learning scenarios. In *Proceedings of AI-ED 97: 8th World Conference on Artificial Intelligence in Education*, pages 95–102, Kobe, Japan, 1997.

[8] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in Description Logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, California, 1996.

[9] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Deborah L. McGuiness, and Peter F. Patel-Schneider. OIL: An Ontology Infrastructure for the Semantic Web. *IEEE Intelligent Systems*, 16(2), 2001.

[10] James Hendler and Deborah L. McGuinness. The DARPA Agent Markup Language. *IEEE Intelligent Systems*, 15(6), 2000.

[11] J. Herrington and R. Oliver. Using situated learning and multimedia to investigate higher-order thinking. *Journal of Interactive Learning Research*, 10(1), 1999.

[12] Robert MacGregor and Raymond Bates. *The Loom Knowledge Representation Language. Technical Report ISI-RS-87-188.* USC Information Sciences Institute, Marina del Rey, CA, 1987.

[13] D. L. McGuinness, Richard Fikes, James Hendler, and Lynn Andrea Stein. DAML+OIL: An Ontology Language for the Semantic Web. *IEEE Intelligent Systems*, 17(7), 2002.

[14] D.L. McGuinness. *Explaining Reasoning in Description Logics.* PhD thesis, Department of Computer Science, Rutgers University, 1996.

[15] Peter F. Patel-Schneider, Deborah L. McGuinness, Ronald J. Brachman, Lori Alperin Resnick, and Alex Borgida. The CLASSIC Knowledge Representation System: Guiding Principles and Implementation Rationale. *SIGART Bulletin*, 2(3):108–113, 1991.

[16] A. Simon, A. Martinez, M. Lpez, J.A. Maestro, J.M. Marquis, and C. Alonso. Learning computational logic with an intelligent tutoring system: SIAL. In *Proceedings of the First International Congress on Tools for Teaching Logic*, Salamanca, Spain, 2000.

[17] E. R. Sykes and F. Franek. A prototype for an intelligent tutoring system for students learning to program in JavaTM. In *IASTED International Conference on Computers and Advanced Technology in Education*, Rhodes, Greece, 2003.

[18] G. Weber and M. Specht. User modeling and adaptive navigation support in WWW-based tutoring systems. In C. Tasso A. Jameson, C. Paris, editor, *User Modeling*, pages 289–300. Springer-Verlag, 1997.