A Software Tool for Recognizing Rule-based Ontology Languages

Ming Yuan

Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2017

Abstract

We developed a software tool that can automatically identify the ontology languages used in the given ontology. An ontology, which is a logical description that is used to define classes, relations and functions of entities or concepts that exist in the real world, can be modelled in a simple and natural way using existential rules. Such rule-based ontology languages have been playing an increasingly important role in databases theory. In recent years, the union of the ontology theory and the relational database system has given birth to advanced approaches for reasoning and query processing tasks. To ensure the decidability of those tasks, several classes of TGDs have been come up with. The software tool that we have designed can save researchers enormous time that used to be taken to manually check the ontology that could contain hundreds or even thousands of existential rules.

Acknowledgements

Firstly, I would like to express my sincere gratitude towards my supervisor Doctor Andreas Pieris for his guidance, continuous support and trust on my decisions in this project.

I would like to thank my colleague Chuqi Yang. We have worked together for this project, and he helped me a lot.

I am grateful to my friends Dongqin Liao, Zechao Hu, Zhicheng Suo and my uncle Anzhen Zhao for their helpful feedbacks.

The last by not least, my whole study would not have been possible without the support of my parents. Many thanks to them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Ming Yuan)

viii

Table of Contents

Chapter 1: Introduction

An ontology is a logical description that is used to define classes, relations and functions of entities or concepts that exist in the real world. Ontologies are widely used in many fields for both researching and commercial purposes. In the fields of artificial intelligence and biomedical informatics, ontologies are utilised to sort information and limit the growth of the complexity of the system. Ontologies can also be applied to improving the classic relational database system.

An ontology can be modelled in a simple and natural way using tuple-generating dependencies (TGD[1], also known as $Datalog^{\pm}$ rules[2] or existential rules). Such rule-based ontology languages(e.g.[3]) have been playing an increasingly important role in many fields. Due to the rule-based ontology's excellent expressivity, in recent years, the union of it and the relational database theory has given birth to a new technology, known as Ontology-based Data Access(OBDA[4]), which can be used to construct database management systems that have advanced methods for reasoning and query processing tasks. Unlike query answering tasks in the classical relational database, given the ontology theory Σ, the database D and a conjunctive query q, there are two common methods to complete the task: chasing[5][6][7][8] and rewriting[9]. Chasing is to expand the given database D with Σ using an algorithm called chase, then to answer the query q against the chase-expansion of D. By contrast, in the rewriting algorithm, q and Σ are rewritten into a domain independent first-order query $q\Sigma$, then the database management system can answer $q\Sigma$ against D.

However, neither chasing nor rewriting can be applied to all kinds of rule-based ontology languages[10]. If the ontology Σ allows arbitrary tuple-generating dependencies to occur in it, the chase algorithm for that Σ and a database D may never terminate. It is also possible that the ontology Σ and the conjunctive query cannot be rewritten if rules in ontology have no restriction. To ensure the decidability of those tasks, several classes of TGDs has been come up with. This project focuses on seven of them: The Linear set of TGDs[2], the Guarded set of TGDs[10], the Weakly-Guarded set of TGDs[10], the Acyclic set of TGDs, the Weakly-Acyclic set of TGDs[7], the Sticky set of TGDs[8] and the Weakly-Sticky set of TGDs[8].

We developed a software tool that can automatically check if the given ontology complies with any of the above classes of TGDs. It can also help users to modify the ontology so it can belong to a selected class of TGDs by removing the least number of TGDs in the ontology. This system can save researchers enormous time for they used to manually check the ontology, which could contain hundreds or even thousands of existential rules.

In this paper, preliminaries and background notions are introduced in Chapter 2. In Chapter 3, details of the software's development are described, including the implementation of ontology

languages-recognizing functions and suggestion giving functions for classes of TGDs mentioned above and the transformation function that can be used to read, transform and check the syntax of the input ontology. Chapter 4 shows different ways that are used to evaluate the system after its development. There is also a summary pointing out what needs to be done in the future in Chapter 5. Please notice that two people are in charge of the project presented, so the paper will focus on the details of the author's work while describing the generic work of the other's.

Chapter 2: Background

2.1 Tuple-Generating Dependencies

A Tuple generating dependency (TGD) is also known as $Datalog^{\pm}$ rules or existential rules. It is a subclass of embedded dependency (ED) which can be used to describe semantic relationships between relations in relational database theory.

A TGD can be described in many ways. Expressions in example 2.1 are all can be recognised as TGDs.

Example 2.1. Different styles that can be used to write an ontology:

$$\forall x, y \ human(x), father(x, y) \rightarrow \exists z \ human(z), mother(z, y). \quad (1)$$
$$\exists z \ human(z), mother(z, y) :- \forall x, y \ human(x), father(x, y). \quad (2)$$
$$\forall x, y \ (human(x), father(x, y) \rightarrow \exists z \ human(z), mother(z, y)) \quad (3)$$

Each TGD shown in Example 2.1 has the same meaning: "For every human father x and every x's child y, there exists a human z who is y's mother". In this paper, we will use the first style in Example 2.1 to express a TGD. No matter in what way a TGD is written in, it often contains the following elements:

1) Universally quantified variables, which is often declared with a symbol $\forall$. This symbol is interpreted as "for all".

2) Existentially quantified variables, which is often declared with a symbol $\exists$. An existentially quantified variable x is interpreted as "there exists a variable x that…". It should be noted that although in example 2.1 the first rule in the ontology has no existentially quantified variable, it is also a TGD: A TGD has no existentially quantified variable is called a full TGD[11].

3) Atoms. An atom contains a predicate and its variables. $human(x)$ in example 2.1 is an atom, the predicate is $human$ and its variables is x.

4) A head and a body. Variables in the body of a TGD should be only universally quantified variables, while the head of the TGD can have both universally quantified variables and existentially quantified variables. In the following sections given a TGD σ, the head of σ and the body of σ will be expressed as $head(\sigma)$ and $body(\sigma)$ respectively. Normally a TGD is divided into a body and a head by a symbol like $\rightarrow$ or $:-$.

A variable's position in its predicate is called a predicate position, which will be called position in the rest of the section for continence. In example 2.1's rule, variable z occurs in the first position of predicate $human$ and the first position of predicate $mother$. These two positions can be respectively expressed as the position $human[1]$ and the position $mother[1]$.

A set of TGDs, like lots of rule-based ontology languages, can be used for reasoning and query answering in a database system. One way to do that is to use Chase. An extensional database(EDB) can be expanded by ontologies using chase algorithm. Thus, any conjunctive query for the chase-expansion of the database is answered against the union set of the database and the ontology. As an example, consider the database $D = \{ Integer(10), rational(0.51) \}$ that contains two facts: the number 10 is an integer and the number 0.51 is a rational number. Giving a single TGD Σ:

$$\forall x \ Integer(x) \rightarrow \ rational(x).$$

The ontology Σ describes that every integer is a rational number. The chase-expansion of D given Σ will be:

$$chase(D, \Sigma) \ = \ \{Integer(10), \ rational(0.51), \ rational(10)\}$$

The expanded database $chase(D, \Sigma)$ has the ability to answer questions like "is 10 a rational number?" that the original database D cannot answer.

Another approach is rewriting. Unlike chase algorithm, which expands the database D by ontology Σ to get $D \cup \Sigma$. the rewriting algorithm rewrites a pair $< q, \Sigma >$, where q is a conjunctive query, into a first-order query $q\Sigma$.The rewritten query $q\Sigma$ can be answered against the database D in a classical way.

In this paper, given a set of TGDs as an ontology, each TDG in it is called a rule or an ontology rule for convenience.

2.2 Specific Classes of TGDs for Reasoning Tasks

Unfortunately, if arbitrary rules are allowed in an ontology Σ, most basic reasoning and query answering problems towards Σ and a database D can be undecidable, i.e. very difficult to solve. One of the reasons is that the chase-expansion of a database could be infinite. This is because there are always new elements generated by the ontology.

Example 2.2

As an extension of the example shown above, considering the database $D = \{\ Integer(10),\ rational(0.51)\ \}$ and a set of TGDs Σ:

$$\forall x\ \ Integer(x) \rightarrow\ \ rational(x).$$
$$\forall x, y\quad rational(x), rational(y) \rightarrow\ \ \exists z\ \ rational(z), is\_sum(z, x, y).$$

which means every integer is a rational number, and the summation of every two rational numbers is also a rational number. The chase-expansion of D given Σ will be infinite. To avoid that situation, ontology rules should be constrained. For example, if the ontology is constrained by weakly-acyclicity, the chase-expansion of the database will always be finite.

While rewriting is another way to solve the reasoning problems, an ontology that consists arbitrary rules may not be first-order rewritable, i.e. some pair of $q\Sigma$. cannot be rewritten in $q\Sigma$. Although the first-order rewritability is impossible to recognise in general[12], there are some ontology languages that ensure its ontologies are first-order rewritable.

In this section, seven ontology languages that have been mentioned in Chapter one will be introduced. Three of them will be emphatically described: weakly acyclic, sticky and weakly sticky. These classes of TGDs ensure the decidability of basic rezoning tasks like query answering. Those ontology languages can be automatically checked by the software tool we developed, and they are described in this paper.

2.2.1 Weakly-Acyclic Sets of TGDs

A weakly-acyclic set of TGDs guarantees the chase algorithm for any database can terminate in polynomial time[7]. So, any chase-expansion database expanded by weakly-acyclic set on TGDs is decidable. Over the recent years, the weakly-acyclic set of TGDs has been studied by many reaches and several extensions are coming up with(e.g.[13][14]). This paper only focuses on the original version of weak acyclicity.

To define Weakly-acyclicity, the notion of the dependency graph[7] must be introduced. The dependency graph visualised the relationship of positions. For each TGD σ in the ontology, and for each variable $V1$ that occurs in both $head(\sigma)$ and $body(\sigma)$: 1) draw an edge from the position in $body(\sigma)$, where $V1$ appears in, to the position in $head(\sigma)$, where $V1$ also appears in 2) draw a special edge from the position in body where $V1$ occurs to every position in $body(\sigma)$ where an existential quantified variable $V2$ occurs.

Example 2.3

Given a set of TGDs Σ:

$$\forall x \ A(x) \rightarrow \exists y \ B(x,y)$$
$$\forall x,y \ B(x,y) \rightarrow A(y)$$
$$\forall x \ C(x) \rightarrow A(x)$$
$$\forall x \ C(x) \rightarrow \exists y,z \ B(y,z)$$

The dependency graph for Σ is shown in part b in Figure 2.1. It should be noted that, in the fourth rule, even though y and z are all existentially quantified variables, no edge can be drawn in this rule because the variable in the body does not occur in any position in the head.

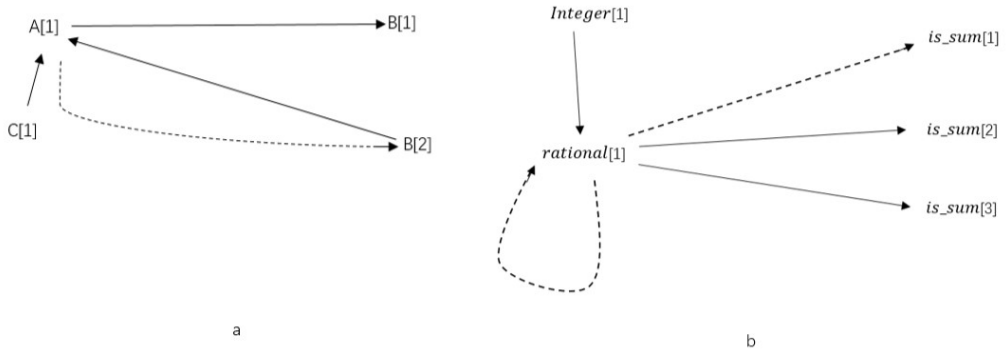

Figure 2-1 dependency graph for Example 2.3(a) and Example 2.2(b)

If at least one cycle in the dependency graph contains at least one special edge, then each position in that cycle belongs to a set called Π_∞. If a position π in the dependency graph can be reached by any position in Π_∞, then π also belongs to Π_∞. All positions that not in Π_∞ belongs to Π_F. In the dependency graph for the ontology in example 2.2 (Figure2.1 b), all positions but the position $Integer[1]$ are in Π_∞ because the special edge starts from position $rantional[1]$ points to the same position and forms a cycle.

According to the dependency graph in Example 2.3(Figure 2.1 a), it is obvious that position $A[1]$, $B[2]$ is in Π_∞ because they form a cycle with one special edge in it. $B[1]$ is also in Π_∞ because there is an edge from $A[1]$ to it. Since $C[1]$ is not reachable from any positions in Π_∞, it belongs to Π_F.

Given a set of TGDs, it is weakly-acyclic when Π_∞ is empty. In other words, there are no cycles that contain special edges in the dependency graph. Therefore, the ontology in Example 2.2 and 2.3 are all not weakly-acyclic.

2.2.2 Sticky Sets of TGDs

SMarking[15] is a preliminary procedure to check if a set of TGDs is sticky. given a set of TGDs Σ, the SMarked version of Σ, written as $SMark(\Sigma)$ for convince, is produced by following two steps:

1) Initial step: For each rule σ in Σ, for each universally quantified variable v in σ, if v does not appear in every one of atoms in the $head(\sigma)$, then mark each v's occurrence in the $body(\sigma)$.

2) Propagation step: For each pair of rules $<\sigma, \sigma'>$, for each atoms $\underline{a}$ in $head(\sigma)$ for each variable $v1$ that occurs in $\underline{a}$, and for each atom $\underline{b}$ which shares the same predicate name with $\underline{a}$ and appears in $body(\sigma')$, if the set of positions that a marked variable $v2$ occurred in $\underline{b}$ includes the set of positions that $v1$ occurs in $\underline{a}$, then each occurrence of $v1$ in $body(\sigma)$will be marked. i.e. mark v1 when $positions(head(\sigma), a, v1) \subseteq positions(body(\sigma'), b, v2)$. This step should be repeated until a fixpoint is reached.

Example 2.4:

Consider the following set of TGDs:

$$\sigma 1: \quad \forall x, y \;\; C(x, y) \rightarrow D(y, y)$$
$$\sigma 2: \quad \forall x \;\; D(x, x) \rightarrow \exists y \;\; A(y)$$

The way to mark the ontology is depicted in Figure 2.2.

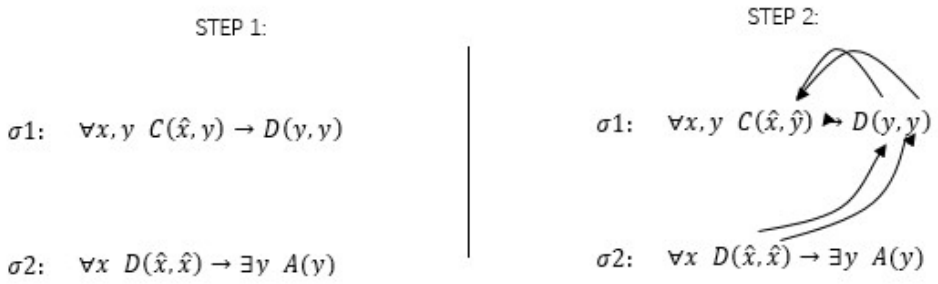

Figure 2.2 The SMarking procedure for example 2.4

After the ontology is SMarked, it is easy to check its stickiness by its definition:

Given an SMarked version of a set of TGDs Σ, it is sticky if, for each rule σ in Σ, every universal quantified variable that occurs more than once in $body(\sigma)$ is unmarked. So, the ontology in example2.4 is not sticky because in the body of the second rule, variable x appears more than once and is marked.

2.2.3 Weakly-Sticky Sets of TGDs

Sometimes the sticky ontology is not expressive enough, for example, rules like $\forall x \ A(x,x) \rightarrow \exists y \ B(x,y)$. Are not sticky. To extend the ontology's expressivity, one approach is to combine the stickiness and the weak acyclicity, which creates weakly stickiness.

Given a set of TGDs Σ, Σ is weakly sticky if, for each rule σ and each variable v that occurs more than once in the body(σ), v is not SMarked, or appears in at least one position in Π_F. It can be inferred that if any SMarked variables v does not appear in positions in Π_F and body(σ) exist, then the ontology is not weakly sticky, because if all positions π_1 that v occurs in body(σ) are in Π_∞, then all positions π_2 that v occurs in head(σ) are also in Π_∞ since the edge (π_1, π_2) exists in the dependency graph of the ontology. This deduction is used in the implementation of the judgement algorithm for weakly sticky.

It should be noted that if an ontology is sticky, it is also weakly sticky. furthermore, if an ontology is weakly acyclic, it is trivially weakly sticky as well, since, for any weakly acyclic ontology, the Π_F contains all the positions of the ontology so every variable occurs in Π_F.

2.2.4 Linear Sets of TGDs

Linearity is the simplest class of TGDs described in this paper. Given a set of TGDs, if each rule in the ontology has only one atom in its body, then the set of TGDs is linear. The ontology in Example 2.4 is linear.

2.2.5 Guarded Sets of TGDs

Given a set of TGDs Σ, it is guarded if, for each rule σ in Σ, there exists an atom in body(σ) that contains all universally quantified variables in σ. It is obvious that if the ontology is linear, it is guarded as well. Therefore, the ontology in Example 2.4 is also guarded.

2.2.6 Weakly- Guarded Sets of TGDs

Before giving a definition for the weakly-guarded set of TGDs, the notion of affected position should be introduced. For each rule σ in the ontology, and for each position π of predicates in σ, an existentially quantified variable occurs in π, then the position π is affected. After the initial step, all affected positions will be obtained by applying exhaustively on an inductive step: for each rule σ, and for each position π' of predicates in head(σ), if a variable V occurs in π' and V only occurs in bod y(σ) 's affected position, then π' is affected.

A set of TGDs Σ is weakly-guarded if the for each rule σ in Σ, there is an atom in the body(σ) and the atom has all the universally quantified variables of σ that appear only in affected positions.

Given the fact that the ontology is weakly guarded if it is guarded, it is obvious that the ontology in example 2.4 is weakly guarded.

2.2.7 Acyclic Sets of TGDs

Given a set of TGDs Σ, for each rule σ in Σ, for each pair of predicates<p1, p2> where p1 appears in body (σ) and p2 appears in head (σ), if there is no such rule σ' in Σ that p2 appears in body (σ') and p1 appears in head (σ'), then Σ is acyclic.

It should be noted that if an ontology is acyclic, then it is also weakly-acyclic.

Chapter 3: Software Design and Implementation

3.1 Initial Work

3.1.1 Environment Setup

The system is developed using Python 3.5. The finished system is tested on windows and the macOS operating systems.

The graphic user interface (GUI) of the system is designed and implemented by a library called Tkinter[16]. Tkinter is the most commonly used Python library for GUI programming. However, the GUI designed by Tkinter may vary across different operating systems[17], for example, the size of the button can be zoomed when the program is run on windows, but are constant on macOS. To ensure the system has a basic consistency, we have tested the performance of the GUI on both the windows system and the macOS.

The system needs to draw the dependency graph for the input ontology when checking the ontology's weak acyclicity. We used a python library called NetworkX[18] to construct the dependency graph. Furthermore, NetworkX also provides three different algorithms to find cycles in the constructed graph efficiently, which can be used to find all positions in Π_F and Π_∞ after the graph is constructed.

Some functions of Natural Language Toolkit(NLTK[19]) have also been used in the system. Although the main functions of the system have nothing to do with natural language processing, one of the transformation functions is to treat each rule in the input ontology as a natural language sentence and parse the rule by context- free grammar. Using NLTK can help to implement this transformation function in a more simple and efficient way.

3.1.2 Dataset's Obtaining and its Format

Testing the system during and after its development needs an ontology dataset. We used three different ways to obtain the data:

1) We have written some toy ontologies which can be used to check the result. Some of them are shown in Table 4.1 in section 4.2. These ontologies are mainly used to test the correctness of the system's outputs.

2) A program is written to automatically generate random ontologies. They are used to test the runtime and memory consumption. Unfortunately, these randomly generated ontologies can be very unrealistic, i.e. some of them are far too complicated than the ontology in the real world (for example, an ontology that consists 100 rules may contain millions of cycles which is impossible to be analysed in a short time). Therefore, these ontologies are not used to test judgment algorithms and suggestion algorithms.

3) We have used some ontology from relevant works and papers (e.g. [20]) to check if the tool can work well for the practical ontology.

The input of the ontology can be written in different styles as long as it can be accurately described by a context free grammar. But only ontologies written in the style below can be directly recognised by the system, the others need to be transformed:

$$1|||\ a(x), b(y) -> c(y, z).$$

In the rule shown above, "1" is the id, the id can be characters and numbers or both, it is indispensable. The id and the rule are separated by "|||". The body of the rule is on the left side of "−>" and the head of the rule is on the right.

Before the system run any of its main functions, the input ontology is transformed into a list of dictionary. For example, the ontology in example 2.4 will be transformed into the following data structure:

$$\begin{gathered}[\\ [\ [\{'name': 'C', 'variable': ['x', 'y']\}], [\{'name': 'D', 'variable': ['y', 'y']\}]\ ], \\ [\ [\{'name': 'D', 'variable': ['x', 'x']\}], [\{'name': 'A', 'variable': ['y']\}]\ ] \\ ]\end{gathered}$$

Each rule is stored in a list. In each rule's list, two sub-lists are stored, one of which contains information of its body, and the other has information of its head. In each of the sub-list of the rule, all atoms in the rule's body or head are stored as a Python dictionary. The dictionary of the atom has two keys: "name" and "variable", the key "name" maps to the predicate of the atom, and the key "variable" positions to a list that contains all of its variables in order.

3.2 Software Architecture

The system should have a graphic interface so users can use it more efficiently. It should also be able to read ontologies written in different styles because different users may have different ways to write their ontology. The main function of the system is to check whether the input ontology complies with any of the selected classes of TGDs, and give

suggestion to modify the ontology if it does not comply with the selected ontology languages. The structure of the system is shown in Figure 3.1.

Brief descriptions of the system's classes are shown in Table 3.1. In addition, some functions are crucial to the system and are not encapsulated into any classes so they can be called in an easier way. These functions are described in table 3.2. Please notice that, since this project is run by two people, the implementation of suggestion interface and recognizing classes of TGDs other than weak acyclicity, stickiness, and weak stickiness are not done by the author of this paper, and will not be discussed in details in this paper.

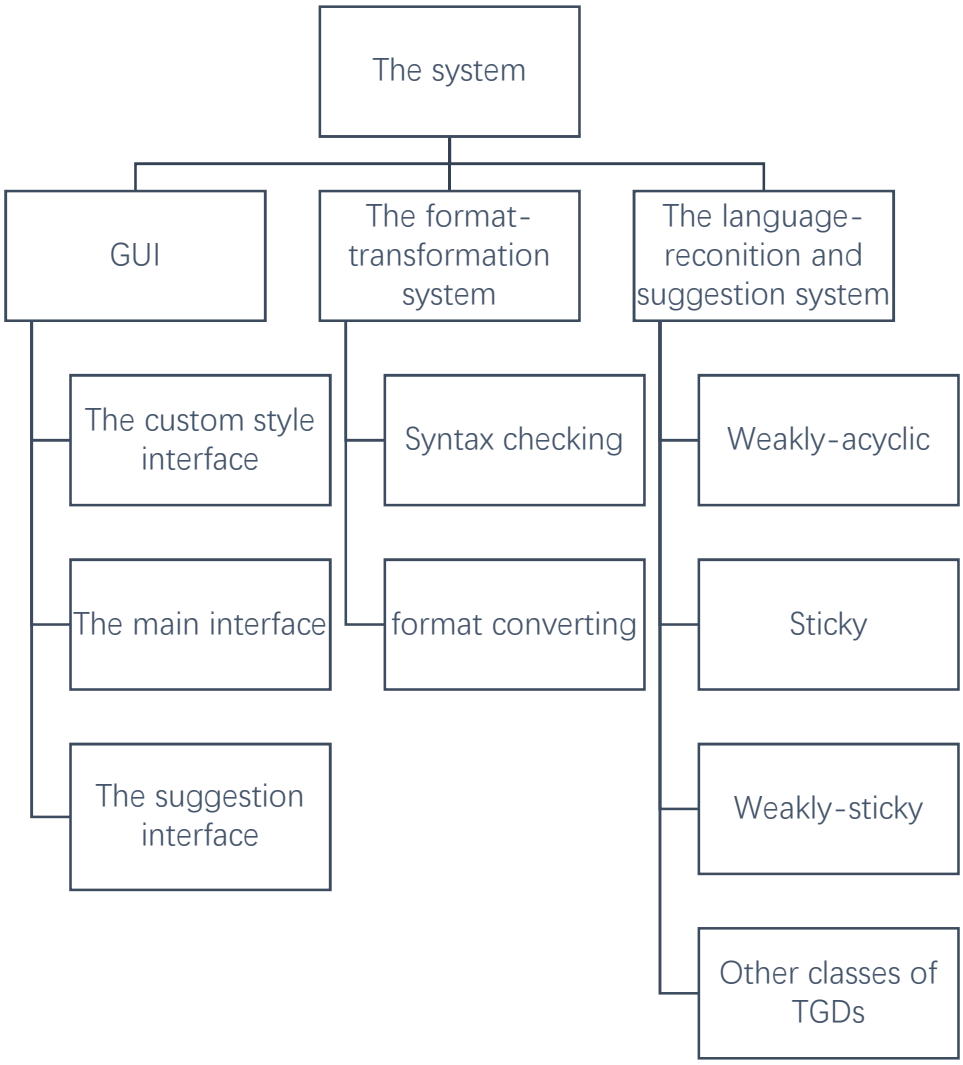

Figure 3.1 The architecture of the system

Class	Description
File weakly_acyclic.py	
Weakly_acyclic	This is a class that contains a method is_weakly_acyclic, the method is used to check if the ontology is weakly-acyclic and give suggestion if users need.
File sticky.py	
sticky	This class has a method is_sticky which check the input ontology's sickness and return a Boolean as a result.
File weakly_sticky.py	
Weakly_sticky	This class has a method called is_weakly_sticky, which checks if the ontology is weakly-sticky, and return a Boolean as a result.
File gui.py	
Home_gui	This is a class for the main interface. The main interface will appear if an instance is initialised.
File gui_grammar.py	
Gui_grammar	This is a class for the interface in which users can convert their ontology's format. The interface will appear if an instance is initialised. It provides a method that can be used to transform ontology using the tree-parsing method.
popupWindow	This is a pop-up window in which users can set the name of their ontology's style.

Table 3.1 Classes in the system.

Function	Description
File smarking.py	
smark	This function is used to proceed the SMarking procedure.
File dep_chart.py	
get_complete_pi_inf	This function is used to construct the dependency graph of the input ontology and get the Π_∞ and Π_F in it.
File transform.py	
transform0	This function provides a way to transform the input ontology using regular expressions.
quick_grammar_check	This function provides a way to check the syntax of the input ontology.
File write_formal_tgd.py	
write_formal_tgd	This function transforms the ontology format that the system can process (e.g. in 3.1.2) into the format that users can read.

Table 3.2 Some important functions in the system.

3.3 Recognizing Ontology Languages

One of the most important functions that our software tool should have is to give the verdict for ontologies. The tool should determine whether the input ontology matches any classes of TGDs selected by users. The result given by the tool should be sound.

In this section, details of algorithms that check the ontology will be described.

3.3.1 Recognizing the Weakly-Acyclic Set of TGDs

As it is mentioned in 2.2.1, to check if an ontology is weakly-acyclic, one way is to check whether the Π_∞ set for that ontology is empty. This is not necessary since Π_∞ is empty only when the dependency graph has no cycles containing special edges. In that case, the tool can give the verdict of the input ontology after finding all cycles in its dependency graph and check if any of them contains special edges.

A more efficient way is to look for special edges right after a cycle is found. when one cycle that contains special edges is found, it will return a negative result and stop the procedure immediately. However, this method cannot be used to help users modify the ontology because it cannot find all cycles that cause the ontology not weakly-acyclic.

The best option is to obtain all cycles that influent the ontology's weak acyclicity when users want to get suggestions from the tool. If the suggestion for modification is not needed, the system will only check if there is one cycle with a special edge in the dependency graph.

In this method, a position is described as a Python tuple containing information about the atom's predicate the and the ordinal location of the atom's variable. For example, given an atom $father(x, y)$, the position $father[1]$ where variable x occurs is described as ('father',0). Note that this position is not described as ('father',1) because in Python the index starts with 0 instead of 1.

The edge is also a tuple, storing the start position and the end position in the order. For example, a tuple $(('a', 0), ('b', 1))$ indicate an edge from $a[1]$ to $b[2]$. All edges will be added into a $networkX.DiGraph$ container denoted as G. This container is used to construct the directed graph. In addition, special edges will be put into a list called $special$ in order to convenience query.

The procedure of finding cycles containing special edges is as follows:

1) For each rule σ in the ontology, find and store all existentially quantified variables into a list ex_v, and store all variables that occur both in $the\ head(\sigma)$ and $body(\sigma)$ into a list $common\_v$.

2) For each position $\pi1$ in the head of σ, if a variable V that occurs in this position also occurs in a position $\pi2$ the body of the same rule, then store a tuple $(\pi2\ , \pi1)$ into G. otherwise if V appears in ex_v and position in a poison $\pi3$ in $head(\sigma)$, for each variable $V2$ in $common\_v$ and its each occurrence $\pi4$ in $body(\sigma)$, not only add tuple $(\pi4, \pi3)$ into G, but also store the same tuple into list $special$.

3) After parsing a whole rule, reset ex_v and $common\_v$.

4) When all rules have been parsed, the system uses one of the cycle finding

method called $simple\_cycles()$, which is provided by $networkX$ and is a very efficient algorithm to find cycles in a direct graph[21], to create a cycle finding a generator, then use the generator to find cycles in the dependency graph G. When a cycle is found, get all its edges. If any of them appears in the list $special$, then store the cycle into a list called $special\_cycles$. If users don't need a further suggestion to modify the ontology, then the procedure terminates and return false right after the list $special\_cycles$ is not empty.

After parsing all cycles, if the list $special\_cycles$ is empty, the method will return true which means the input ontology is weakly-acyclic, otherwise, the method will return false.

It is remarkable that the cycle finding function $simple\_cycles()$ returns a generator instead of a list of cycles. A generator can be used to find cycles when it is traversed. This means that when users do not need the suggestion for modification, most of the time the algorithm does not need to go through every cycle in the dependency graph. This mechanism can save lots of time in extreme conditions, which will be explained in detail in section 4.3.

3.3.2 Recognizing the Sticky Set of TGDs

The key of checking an ontology's stickiness is to find out whether all variables that occur more than once in the body of a rule in the ontology and are marked. However, it is impossible to check whether a variable is marked without proceeding SMarking for the entire ontology in advance. Because even if the variable is not marked during the initial step, it is still possibly be marked several steps later as in a propagation step.

The SMarking procedure is done by following instructions of SMarking shown in section 2.2.2 with a slight modification:

1) Creating a Python dictionary called marked, whose keys are the Sequence Number of each ontology rule, and each key maps to a list of marked variables in that rule. Note that items in the list are variables, not positions of predicates or atoms.

2) Executing the initial step for each ontology rule σ from the input: mark all variables that occur in $the\ body(\sigma)$ and do not occur in at least one atom in $the\ head(\sigma)$. Note that instead of marking the occurrence of the target variables in $the\ body(\sigma)$, the system simply marks the variable itself.

3) Repetitively executing the propagation step until no more new variables are marked. Likewise, this step marks variables instead of occurrences. Recall that for each pair of rules $< \sigma_1, \sigma_2 >$, σ_1 and σ_2 could be the same.

It should be noted that, while in the formal process of SMarking all marked variables are universally quantified variables, the algorithm described above sometimes marks existentially quantified variables in rules as well. Even so, this difference in the procedure does not affect the correctness of the result, because existentially quantified variables cannot occur in the body of the rule so they can be ignored in propagation steps even when they are marked.

3.3.3 Recognizing the Weakly-Sticky Set of TGDs

To check an ontology's weak stickiness, the tool has two options: 1) get both the Π_F and SMarked version of that ontology. Then check the ontology's weakly-stickiness by its original definition. Or 2) If the ontology is Sticky, then directly return true and terminate. Otherwise get all the variables that make the ontology non-sticky, put these variables into a set denoted as bad_var, and check if all variables that in bad_var occurs in positions in Π_F. If not, return false, otherwise true. The tool adopts the second option because it is faster most of the time.

Since Π_F is the complement of Π_∞, it can be acquired by finding all positions in Π_∞ for the input ontology. Therefore, the first step to check the input ontology's weak stickiness when it is not sticky is to get the Π_∞ of the ontology, which, by the definition, should be done when calling the judgement function for weak acyclicity, but the real algorithm has taken a shortcut so this step is skipped.

Getting the Π_∞ set can be carried out in three steps: The tool should first build the dependency graph for the input ontology, then find all cycles in the graph that contains special edges. The last step is a propagation step that recursively finds all positions that can be reached by any positions that currently in the Π_∞ set, and put those positions into Π_∞. The first two steps are the same as what the method that checks weakly-acyclicity does. The propagation step's procedure is as follows:

1) After finding all cycles that contain at least one special edges, put all positions that in those cycles into a set called pi_inf. The reason that the data structure of pi_inf is Python set instead of list or dictionary is that it is easy for a set to eradicate any duplicated elements and get the complement from itself[22].

2) Using complementary operation to get the initial pi_f set which is used to contain all positions in Π_F.

3) For each element (position) pi_1 in the set pi_f if there exists an element pi_2 in pi_inf, and the edge (tuple) (pi_2, pi_1) exists in the dependency graph for the input ontology, then put pi_1 into pi_inf and delete it in pi_f.

4) Repeat the third step until no more change for the set pi_f and pi_inf.

The bad_var set for the ontology is acquired using the procedure described in section 3.3.2.

When the bad_var is empty, which means that the ontology is sticky, then it is certain that the ontology is weakly sticky as well. If the length of the pi_inf is empty, then the ontology is also weakly sticky since the ontology is weakly acyclic. If the ontology is neither sticky nor weakly acyclic, check if each one of the variables in bad_var has at least one occurrence in a position in pi_f. If the answer is true, then the ontology is weakly sticky.

3.4 Modification Suggestion

If the input ontology does not belong to the selected ontology language(s), users may want to get some suggestion about how to modify that ontology, which is by deleting the least number of rules, so that the modified ontology can fall in the target rule-based ontology language.

Given a class of TGDs C and the input ontology O, the modified ontology O' should satisfy the following conditions:

1) $O' \subseteq O$.
2) O' falls in C.
3) There is no ontology O'' such that $O' \subset O'' \subseteq O$ and O'' falls in C.

In this section, details of the algorithm that generates the suggestion for weakly acyclic ontology are described. Unfortunately, suggestion function for sticky and weakly sticky is not yet implemented, but some ideas and attempts about these algorithms are come up with.

3.4.1 Suggestion Algorithm for Weakly-Acyclic

If the input ontology is not weakly acyclic, users can ask for suggestions to delete some of its rules so they can get a modified ontology that is weakly acyclic. It is required that the number of rules that the systems suggest deleting is minimal as it can be. Even so, it is possible that the suggestion that the system should return is more than one. For example, the system should give two suggestions for the ontology in Example 2.3: a) delete the first rule and b) delete the second rule.

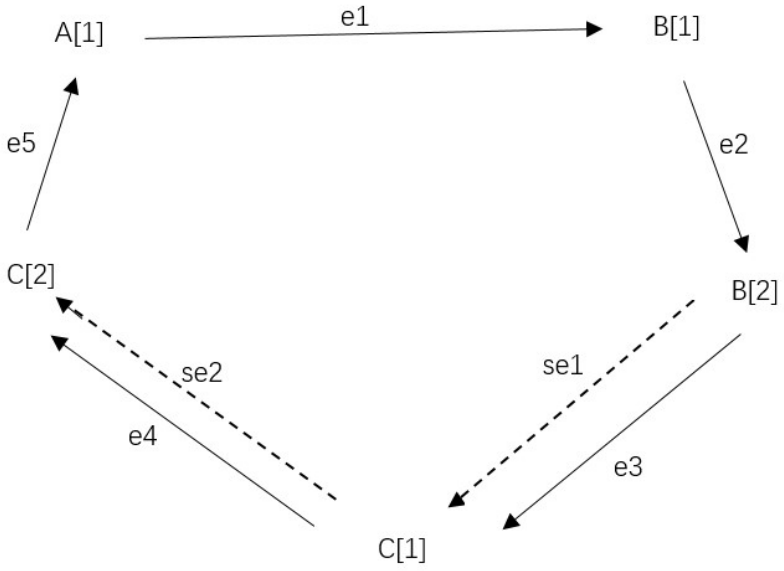

Figure 3.2 a typical cycle in the dependency graph of an ontology

To make the ontology weakly acyclic, all the cycles containing special edges should all be "normalised", i.e. for each cycle, either break its cyclicity by removing least number of edges or remove all its special edges. e.g. the cycle shown in Figure 3.2 has six ways to normalised:

a) Remove e1
b) Remove e2
c) Remove e3, se1
d) Remove e4, se2
e) Remove e5
f) Remove se1, se2

After obtaining all possible ways to normalise each cycle, the system picks one way from each cycles' normalizing way to get the global solution. When all global solutions are acquired, the system returns the set of solutions that deletes the least number of rules.

The logic of the algorithm is as follows:

1) Get the dictionary $special\_edge\_to\_tgd$ and $edge\_to\_tgd$ that maps from the edges to the rule(s) that creates them.

2) Find all possible ways to normalise the first cycle, for each way, get the set of rules that it deletes, and put that set into a list called $options\_for\_all$.

3) Find all possible ways to normalise another cycle for each way, get the set of rules that it deletes like step one, but instead, put that set into a list called $options\_for\_each$. Then update the list $options\_for\_all$: for each element e_1 in $options\_for\_each$ and each element e_2 in $options\_for\_all$, put $e_1 \cup e_2$ into a new list $updated$. After all elements are traversed, let $options\_for\_all = updated$. Then the list $options\_for\_all$ contains all the local solutions for now.

4) Repeat step three until all cycles are traversed.

5) Return the element(s) in $options\_for\_all$ that delete the least number of rules.

However, it is possible that the number of optimal solutions are too many, considering the following ontology Σ:

$$\sigma 1||| \ a(x) -> b(x,z).$$
$$\sigma 2||| \ b(x,x) -> c(x).$$
$$\sigma 3||| \ c(x) -> d(x).$$
$$\sigma 4||| \ d(x) -> e(x).$$
$$\sigma 5||| \ e(x) -> a(x).$$
$$6\sigma||| \ a1(x) -> b1(x,z).$$
$$\sigma 7||| \ b1(x,x) -> c1(x).$$
$$\sigma 8||| \ c1(x) -> d1(x).$$
$$\sigma 9||| \ d1(x) -> e1(x).$$
$$\sigma 10||| \ e1(x) -> a1(x).$$

The dependency graph of Σ contains two cycles, each of which consists five edges. The number of the optimal solutions, which suggests deleting one rule from $[\sigma 1 - \sigma 5]$ and one rule from $[\sigma 6 - \sigma 10]$, should be $5^2 = 25$. If an ontology contains 10 such cycles, which has high possibility occur in a practical ontology, there will be $5^{10} = 9765625$ best solutions. Showing all of the ontology will take a long time. To force the system work within the permitted time, during the process, if the total number of the local solutions (not only the least deletion) is more than ten thousand, then the system only keeps the first one hundred local optima. In this case, the suggestion given by the system could be suboptimal, so the system will inform the user by showing them a warning message.

3.4.2 Suggestion Algorithm for Sticky and Weakly-Sticky

Unfortunately, we have not yet come up with any solution that will give perfectly optimal suggestions for stickiness and weak stickiness.Compared with the suggestion function for weak acyclicity, giving suggestion to make the input ontology sticky and weakly sticky are more difficult to be implemented.

It is not hard to give suboptimal solutions for stickiness: deleting every rule that contains marked variables that occur more than once in its body. But this method will not necessarily lead to the optimal suggestion. For example, given a set of TGDs Σ:

$$\sigma 1|||\ a(x) -> b(x), b(y).$$
$$\sigma 2|||\ c(x,x) -> a(x).$$
$$\sigma 3|||\ d(x,x) -> a(x).$$

The best option is to only remove $\sigma 1$, but using the suboptimal method described above, $\sigma 2$ and $\sigma 3$ should all be removed.

Even though functions that gives perfectly optimal suggestions for stickiness and weak stickiness have not yet been implemented, we designed an algorithm that gives solutions close to optima, if are not the optima:

1) For each rule σ in the ontology Σ, if a variable v is marked and occurs more than once in the body(σ), then find all variables v' in Σ that make v marked. If v is marked only because of the initial step of the SMarking procedure, then $v' = v$, otherwise draw an edge from $each\ v'\ to\ v$.
2) For each v', get all variables v''that make it marked, draw an edge from each v'' to v' if $v' \neq v''$, repeat this step recursively until no more new edges are drawn.
3) Now each variable v that breaks the ontology's stickiness are linked by a set of chains. For convenience, this graph is called chain graph. v can be unmarked when all v' are unmarked. v' will be unmarked if all v'' are unmarked, and so on. Find all options that unmark all v. All options that delete the least number of ontology rules are optimal (most of the time globally).

Sometimes the algorithm above cannot get all global optima, or cannot get global optima at all. E.g. given a set of TGDs, its chain graph is shown in Figure 3.3. If the variable x need to be unmarked, one way is to unmark the variable z, because if z, is unmarked, then all the edges in the chain graph will disappear (step 4 won't exist if variable y is not marked). However, the algorithm above will be trapped in a loop: one way to unmark x is to unmark both y and z, one way to unmark y is to unmark x… it can be very difficult for the system to find out it is in a loop if the cycle in the chain graph contains more variables and more edges are added in the chain graph. Furthermore, using this method may produce too many options to filter and find optima in, it may cost a large amount of time and space.

The suggestion for weakly sticky should base on not only the SMarking procedure, but also the Π_F of the ontology. Therefore, this function cannot be implemented when the function to unmark variables with minimal deletion of rules is not implemented.

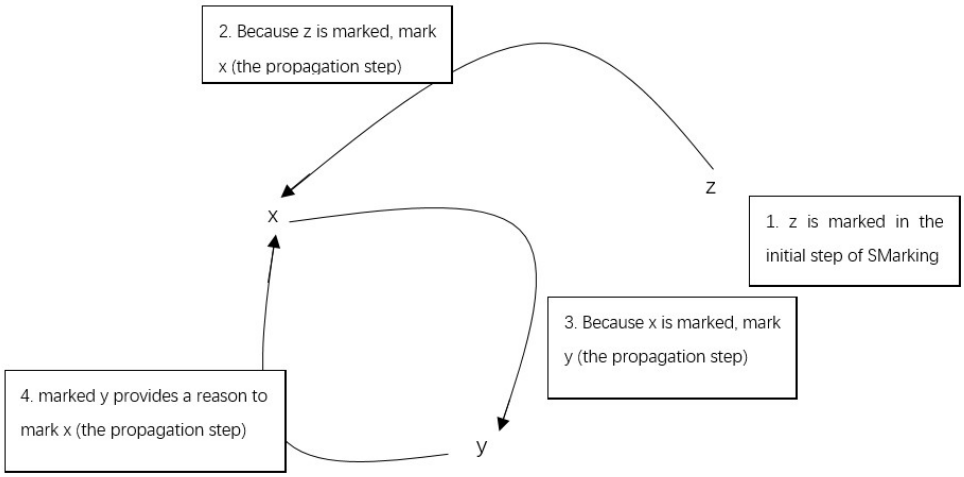

Figure 3.3 The chain graph of an ontology

3.5 Input Format Conversion and Syntax Check

Before checking if the given ontology belongs to any classes of TGDs the user selected, the ontology must be extracted from the target file, checked if contains any syntax error, and transformed into the data structure described in section 3.1.2. in this section, two different approaches for input transforming, which are tree parse method and regular expression method, are introduced. By using tree parse method, the system can transform any style of the ontology into the required format, while regular expression can have a higher execution speed toward a specific style.

3.5.1 Tree Parse

One way to parse the ontology written in custom style is to turn each ontology rule into a tree and parse the tree. Figure 3.4 shows how to transform a rule into a tree. The idea of turning a rule into a tree is to treat each rule as a natural language sentence. By using the NLTK library we can easily convert a sentence into a tree based on the context-free grammar that the user writes. We construct an instance called $nltk.ChartParser$ which can turn a string of context-free grammar into a tree parser. Then using the parser each ontology rule can be transformed in an instance called $nltk.tree$.

To parse each ontology rule, first, the system calls the $parse\_file$ function. This function constructs a $nltk.ChartParser$ then for each rule call the $parse\_rule$ function, which turns an ontology rule into a tree. After obtaining the tree, the $parse\_rule$ function finds a subtree in the rule's tree whose root is "HEAD", and a subtree whose root is "BODY", those two sub trees are then parsed by using $parse\_head$ and $parse\_body$ respectively. These two functions find all the sub trees

with the root named “ATOM”. At last, the function $parse\_atom$ is called for each “ATOM” subtrees to capture the information about each atom’s predicate and variable. When all the parse procedures are finished, the system can accurately extract all the atoms in each ontology rule’s head and body, so the ontology can be transformed into the data structure defined in section 3.1.2.

The tree parse method can transform the ontology in a reliable way as long as the grammar given by the user is correct. It can also be used to check if the input ontology has syntax errors: when parsing an ontology rule, if the $nltk.ChartParser$ returns a None type result instead of a $nltk.tree$, then it can be concluded that the rule has syntax errors.

Although the tree parse method is powerful, it takes a long time to parse the ontology if it contains many rules. Furthermore, if the length of each rule is too long, the system will spend even more time to parse the ontology. Due to the inefficiency of the tree parse method, it is not appropriate to use it in functions that users may use frequently. Therefore, we need another way to transform the input ontology.

Rule:	P1(x)-> P2(y,x).
Grammar:	RULE -> BODY '-' '>' HEAD '.' BODY -> ATOMS HEAD -> ATOMS ATOMS -> ATOM | ATOM COM ATOMS ATOM -> NAME "(" VARS ")" NAME -> CHARS VARS -> VAR | VAR COM VARS VAR -> CHARS CHARS -> CHAR | CHAR CHARS CHAR -> 'P'|'1'|'2'|'x'|'y' COM -> ','

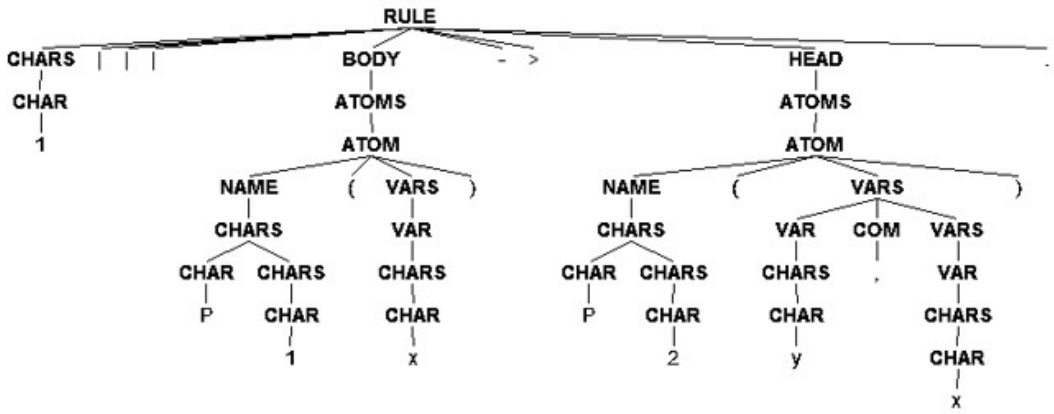

Figure 3.4 Transforming a set of TGDs into a tree

3.5.2 Regular Expression

Using the regular expression to parse and transform the ontology is the alternate way to the tree parse method. Although this method is less powerful so it cannot be used to transform all styles of the ontology (Python does not support the nesting regular expression likes "$\forall[a-z,]+ \ \ BODY\_ATOMS \rightarrow \exists * [a-z,] * \ \ HEAD\_ATOMS.$ ". Furthermore, using regular expressions may describe the ontology inaccurately), it can parse the ontology much faster than the tree parse method. In that case, the method is suitable for parsing ontologies written is one specific style. In this system, we use the regular expression to parse and transform the input ontology.

Although nesting regular expressions are not supported in python, the ontology rule can

be parsed in a similar way. Given an ontology rule:

$$1|||\ a(x), b(y) -> c(y, z).$$

The system ignores the id of the rule "1|||", then split the rule with the symbol "->" to get the body and the head of the rule. All atoms are found by matching the body or head of the rule with a pattern "[^ \t\(\)\,]+\([^\(\)]+\)" which means an atom should be a joint of a character string(predicate) and another character string(variables) surrounded by a pair of parentheses. For each atom that is found in this way, extract its predicate by extracting the character string before the left parenthesis "(", and all variables can be obtained by split the characters inside the parentheses by a comma ",".

It is possible that the regular expression method misread the ontology if it does not check its syntax in advance. For example, given an ontology rule is written in a wrong way:

$$1|||\ a(x,,y), b(y) ->\ ()$$

While the ontology rule shown above is definitely wrong and the system should return an error, if there is no syntax check for the rule, the system will still parse it in a normal way: the first atom in the body of the rule has three variables: *"x"*,*""*(a variable with no name) and *"y"*. The head contains one atom whose predicate has no name and has an "empty" as the only one variable.

We check the syntax of the ontology by simultaneously using the regular expression and checking the validity of the transformed ontology. Before parsing a rule, the system checks whether it coincides the pattern "$BODY -> HEAD.$". because nested regular expression cannot be used in python, this pattern is checked by four steps: 1) if the rule contains character string ',,' then return false immediately, which means atoms like $a(x,,y)$ or conjunction of atoms like $a(x,y),,b(x)$ cannot pass the syntax check 2) match all atoms and change each of them into a unusual symbol "♂", then 3) use "(♂S♂)" to replace each conjunction of atoms, i.e. "[\,\♂] +" in regular expression.4) the last step is to check whether the modified character string equals to "(♂S♂)−> (♂S♂)."

This procedure can filter out most of the syntax error. But if the rule is written as "(♂S♂)−> (♂S♂)." in the first place, it can pass the syntax check. To avoid that situation, after the transformation, the system will throw an error message if any of the following cases happens: 1) the body or the head has no atoms 2) some atom has empty predicate or variable.

Chapter 4: Result and Evaluation

4.1 The Finished System

The software tool that we developed has three interfaces: the main interface, an interface that allows users to transform their input ontologies (named as “custom style interface” for convenience), and an interface that shows the modification suggestion of the input ontology. Because in this joint project, the suggestion interface is not developed by the author of this paper, in this section we only discuss the main interface and the custom style interface.

Figure 4.1 and Figure 4.2 shows the appearance of the main interface and the custom style interface respectively. After the usability evaluation, these two interfaces are slightly modified. The appearances of these two interfaces after the usability evaluation are shown in Figure 4.6 and Figure 4.7 to aid comparisons.

Users can choose any files that store ontologies by clicking the “Browse” button. The ontology should be written in the way as required (described in section 3.1.2). After choosing the input ontology, users need to select a class of TGDs that they want to check for the ontology, then Click the ‘>>’ button to move it into the list on the right. If users want to get the suggestion to help them to modify the ontology, check the “Enable Suggestion” box.

Judgment functions for chosen classes of TGDs are executed by clicking the button “Check the Ontology”. Before checking the input ontology, the system will check if the input file is valid and commit a syntax check. Recall that details of syntax check that used in the main interface are described in section 3.5.2.

In the custom style interface, users can transform the format of their ontology file into the required format. The ontology that needs to be transformed can be written in any styles as long as they can be described by a context-free grammar(CFG). The custom style interface can be opened by clicking the button ‘Custom Style' in the main interface. In the text box, describe the input ontology's grammar in the form of CFG. The grammar should contain all non-terminal symbols as follows:

- ATOM: This non-terminal denotes an atom that contains predicate and variables, if atoms in body and head are different, users can use H_ATOM for head’s atom and B_ATOM for body’s atom.

- NAME: The Atom’s predicate, if predicates in body and head are different,

users can use H_NAME for head's predicate and B_NAME for body's predicate.

- VAR: Atom's variables, if variables in body and head are different, users can use H_VAR for head's variables and B_ VAR for body's variables.

- BODY: The body of the rule (contains only universal quantified variables)

- HEAD: the head of the rule (contains universal quantified variables and existentially quantified variables)

For example, rule "$a(x,y),b(x)->c(y)$." can be described as the following CFG:

```
RULE -> BODY '-' '>' HEAD '.'
BODY -> ATOMS
HEAD -> ATOMS
ATOMS -> ATOM | ATOM COM ATOMS
ATOM -> NAME "(" VARS ")"
NAME -> CHARS
VARS -> VAR | VAR COM VARS
VAR  -> CHARS
CHARS -> CHAR | CHAR CHARS
CHAR -> 'a'|'b'|'c'|'x'|'y'|'z'
COM -> ','
```

Every character in the description should be written separately. In other words, the first line of the example above cannot be written as:

```
RULE -> BODY '->' HEAD '.'
```

Users can save their CFG by clicking 'Save As New Grammar'.

It is remarkable that if the input ontology is space sensitive like the example in Figure 4.3, users can enable the 'Space Sensitive' function.

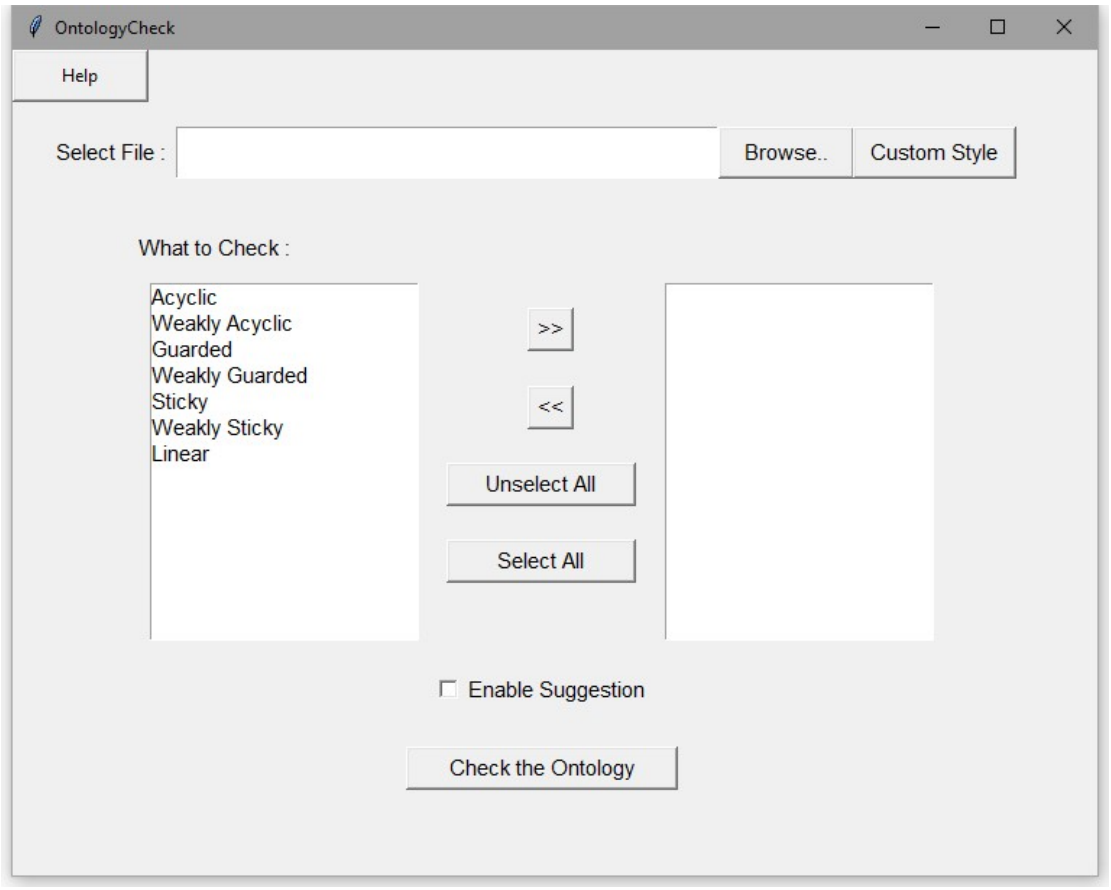

Figure 4.1 The main interface

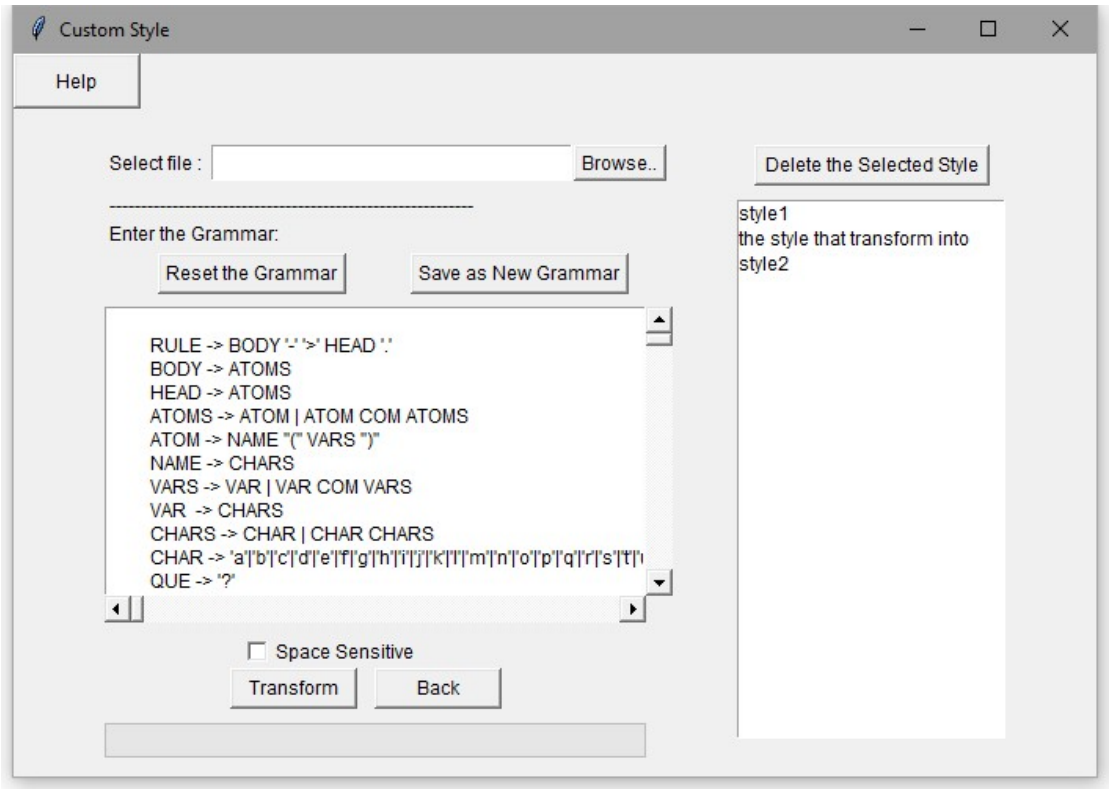

Figure 4.2 The custom style interface

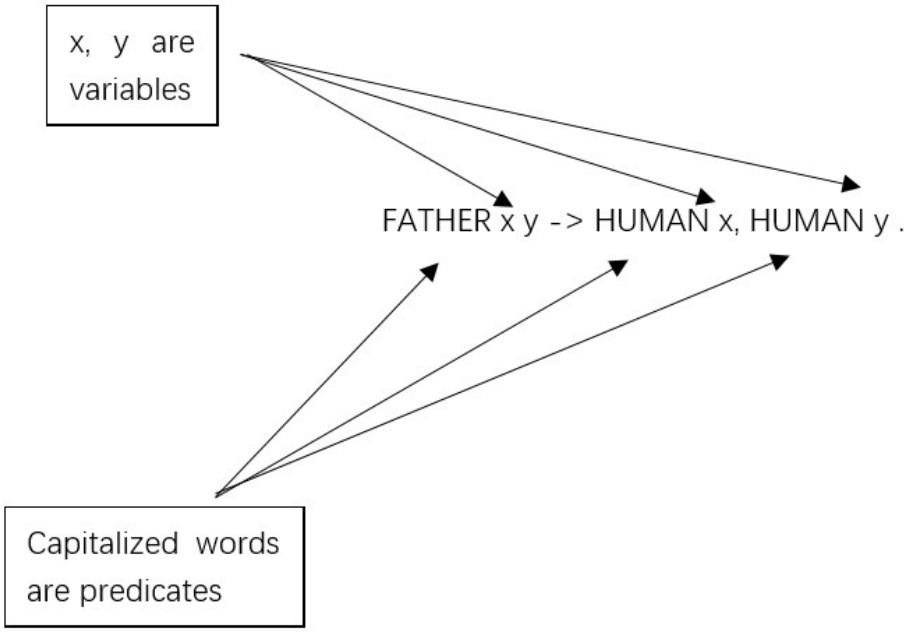

Figure 4.3 A special way to write an ontology rule

4.2 Correctness Evaluation

The result shown by the system should be sound, i.e. no mistake is tolerated. To ensure that the system can output the correct result every time, we regularly wrote several ontologies and check if the judgement given by the system corresponds our own judgement. Table 4.1 shows some of the ontologies for testing.

To make sure that the algorithm has no mistake, not only the result should be checked, some crucial procedures in the algorithm also need to be tested: When evaluating the judgement algorithm for weak acyclicity, all the cycles that the system thinks contains special edges will be shown and compared with manual work. Similarly, when evaluating judgement algorithm for stickiness, all the variables that are marked by the system in each step of SMarking procedure (initial step and propagation steps) will be checked as well.

The suggestion for weak acyclicity is a set of rules σ' that should be deleted from the original ontology σ. The number of rules in σ' should be the least as possible. This algorithm is also tested by comparing the human judgment and the systems.

It is also vital that the system can transform the format of users' ontology into the required format correctly. We have tested the transformation function and the syntax check function against a set of ontologies, the result is shown in table 4.2, where the capital letter A in the result means: normal operation, the system returns the correct result. B means: the system notices an error in advance and show informs to the user. C: an error occurs, but no message is given. D: the system gives an incorrect result.

ID	ontology	weakly-acyclic	sticky	weakly-sticky
1	1||| a(1,2)->a(1,3).	TRUE	TRUE	TRUE
2	1||| a(1,2,3)->a(3,1,2).	TRUE	TRUE	TRUE
3	1||| a(1,2),b(1,2)->c(1,2).	TRUE	TRUE	TRUE
4	1||| b(2)->c(3). 2||| a(1)->b(1). 3||| c(1)->a(1).	TRUE	TRUE	TRUE
5	1||| a(1),b(2)->c(1,2). 2||| a(1),b(1)->c(2,1). 3||| d(1),b(1)->b(3).	TRUE	FALSE	TRUE
6	1||| a(1,2)->b(1). 2||| a(2,3)->c(2).	TRUE	TRUE	TRUE
7	1||| a(1,2)->b(1,3). 2||| c(1),b(2,1)->a(1,2).	FALSE	FALSE	TRUE
8	1||| a(1,2)->b(1,3). 2||| c(2),b(2,1)->a(1,2).	FALSE	FALSE	TRUE
9	1||| dept(V ,W)-> emp(W, V , X, Y). 2||| emp(V ,W, X, Y)->dept(W, Z), runs(W, Y). 3||| runs(W, X), dept(W, Y)->project_mgr(Y , X).	FALSE	FALSE	TRUE
10	1||| a(x)->b(x,x,z). 2||| b(x,y,x)->a(x),a(y). 3||| c(x,y)->d(y,y). 4||| d(x,x)->a(y).	FALSE	FALSE	FALSE
11	1||| a(x)->b(x,y). 2||| b(x,y)->a(y),c(y). 3||| c(y)->d(y). 4||| d(y)->e(y). 5||| e(y),e(y)->f(z).	FALSE	FALSE	FALSE

Table 4.1: ontologies that are used to test judgment functions

Ontology	Syntax Check's Result	Transformation Result
1||| a(x,y),b(y)->b(x).	A	A
1||| a(x,y),b(y)->b(x)	B	B
1||| a(x,y),b(y)->b(x().	B	A
1||| a(x,y),b(y)->b(x,y).	B	A
1||| a(x,y)b(y)->b(x,y).	A	B
1||| a()b(y)->b(y).	B	B
1||| a(,)b(y)->b(y).	B	B
1||| a(!@#$%^&*)->b(x).	A	A
1||| a(->,x)b(y)->b(y).	B	B
a(x)b (x)->b(y).	B	B (expect A)
a(x)->b(y). a(x)b(x)->b(y).	B	B (expect A)

Table 4.2: ontologies that are used to test the transformation function

According to the information in table 4.2, the system will not work as expected when a TGD contains line separators (excluding the line separator at the end of the TGD) or when more than one TGD are in one line in the file. This is because the transformation function transforms one and only one ontology rule in each line. Since allowing an ontology rule containing space separator will dramatically raise the time complexity of the algorithm, and cause problems in situations like Example 3.1, we decided that space separators are not allowed in a rule. Similarly, because of situations like Example 3.2, the situation that more than one rules appear in one line should also be prohibited.

Example 3.1:

By allowing space separator appearing in the rule of the ontology, the system can recognise the ontology Σ in more than one way if Σ is written in the way as follows (in this case, no comma between atoms and no period in at the end of the rule):

$$A(x,y)B(x)-> C(x,z)$$
$$A(x,y)$$
$$B(y)-> C(z)$$

Recognition result:

Σ_1:

$$A(x,y)B(x)-> C(x,z)$$
$$A(x,y)B(y)-> C(z)$$

Σ_2:

$$A(x,y)B(x)-> C(x,z)A(x,y)$$
$$B(y)-> C(z)$$

Example 3.2:

By allowing more than one ontology rule appear in one line of the file, the system can recognise the ontology Σ in more than one way if Σ is written in the way as follows(in this case, no comma between atoms and no period in at the end of the rule):

$$A(x,y)B(x)-> C(x,z)A(x,y)B(y)-> C(z)$$

Recognition result:

Σ_1:

$$A(x,y)B(x)-> C(x,z)$$
$$A(x,y)B(y)-> C(z)$$

Σ_2:

$$A(x,y)B(x)-> C(x,z)A(x,y)$$

$$B(y) -> C(z)$$

4.3 Runtime Evaluation

The system should be able to give required judgement and suggestions in a reasonable time. In this section, we analyse the model of the runtime for all of the main function in the system: algorithms that recognises weakly acyclic, sticky and weakly sticky ontology, the algorithm that give suggestions for modifying none weakly acyclic ontology to be weakly acyclic, and the algorithm that converting the format of the input ontology.

Runtime evaluation for weakly acyclic ontology recognizing and suggestion-giving function: One of the factors that affect the runtime of weak acyclicity's judgement algorithm is the number of the cycles that contains special edges in the dependency graph. This factor is especially influential when users need the modification suggestion for the ontology, in which case all these 'bad cycles' are traversed. Therefore, the suggestion-giving function and the recognition function for the weakly-acyclic ontology can be tested at the same time. To find out the relation between the number of the cycles that have special edges and the runtime for the algorithm, we have created two rules that form one cycle containing one special edge:

$$1||| \ a(x) \ -> b(x,y).$$
$$2||| \ b(x,y) \ -> a(y).$$

Then those two rules are copied and given new names for each predicate in it:

$$1||| \ a1(x) \ -> b1(x,y).$$
$$2||| \ b1(x,y) \ -> a1(y).$$
$$3||| \ a2(x) \ -> b2(x,y).$$
$$4||| \ b2(x,y) \ -> a2(y).$$
$$\dots$$

By using this method, we can get the ontology that has the required number of cycles containing special edges.

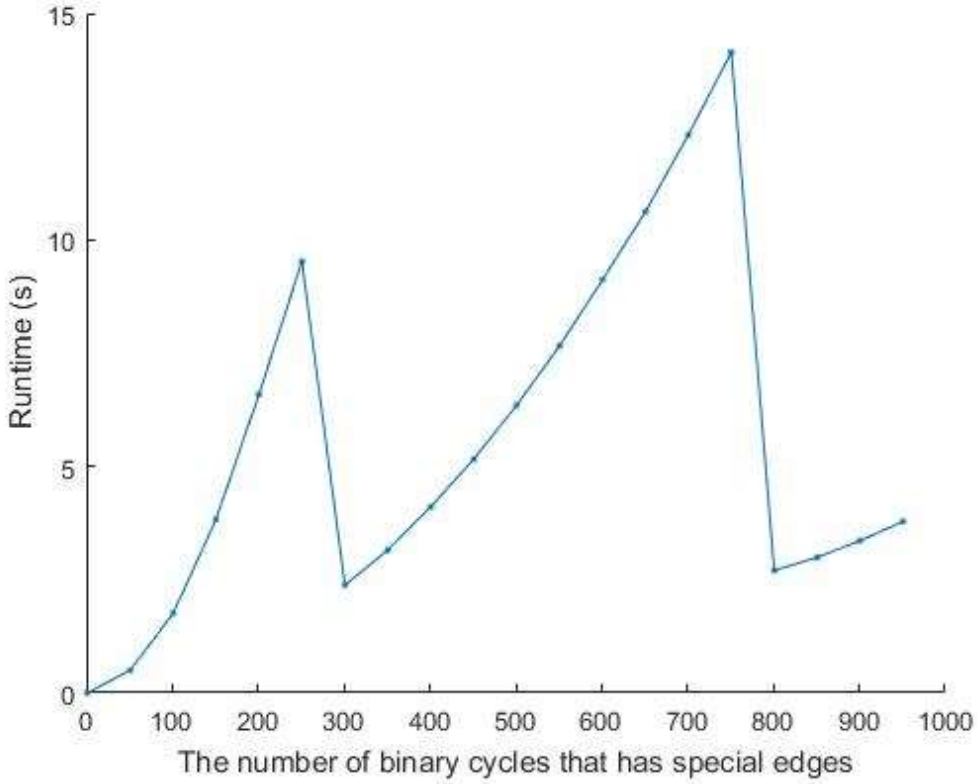

Figure 4.1 The correlation of the weakly-acyclic suggestion function's runtime and the number of cycles containing special edges

Figure 4.1 reveals the correlation of the runtime and the number of cycles containing special edges. The runtime in the figure dramatically drops twice because of the time optimization mechanism: when there are too many cycles or too many ways to break the cycles, the system will drop some of the local solutions in the procedure. As it is shown in Figure 4.1, even the worst time (the ontology has 750 cycles that make the ontology not weak acyclic) is less than 15 seconds and is still tolerable.

Runtime evaluation for weakly sticky and weakly sticky recognizing function: One of the factors that affect the length of the runtime for weak stickiness recognition and stickiness recognition functions is the number of steps in total that the SMarking procedure takes. The more propagations steps need to be marked, the longer time the algorithm takes. In addition to the number of propagation steps, the number of the rules in the ontology also affect the runtime of the system, because each rule in the ontology will be traversed during each propagation step. We first observed the relation between the number of propagation steps has with the system's runtime. After the first evaluation, we set the propagation steps number by a fixed value 10, and change the number of the ontology rules by duplicating the current ontology rules and changing the predicate's name. The evaluation results are shown in Figure 4.2 and Figure 4.3 respectively.

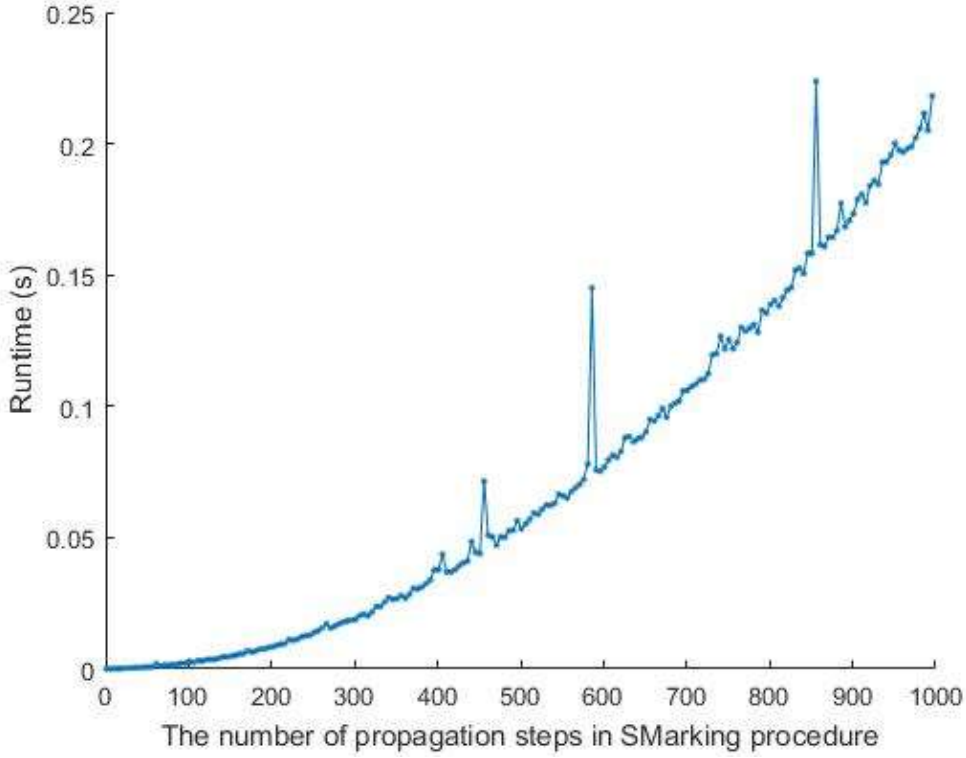

Figure 4.2 The relationship between the sticky ontology recognition function's runtime and the number propagation steps in the SMarking procedure.

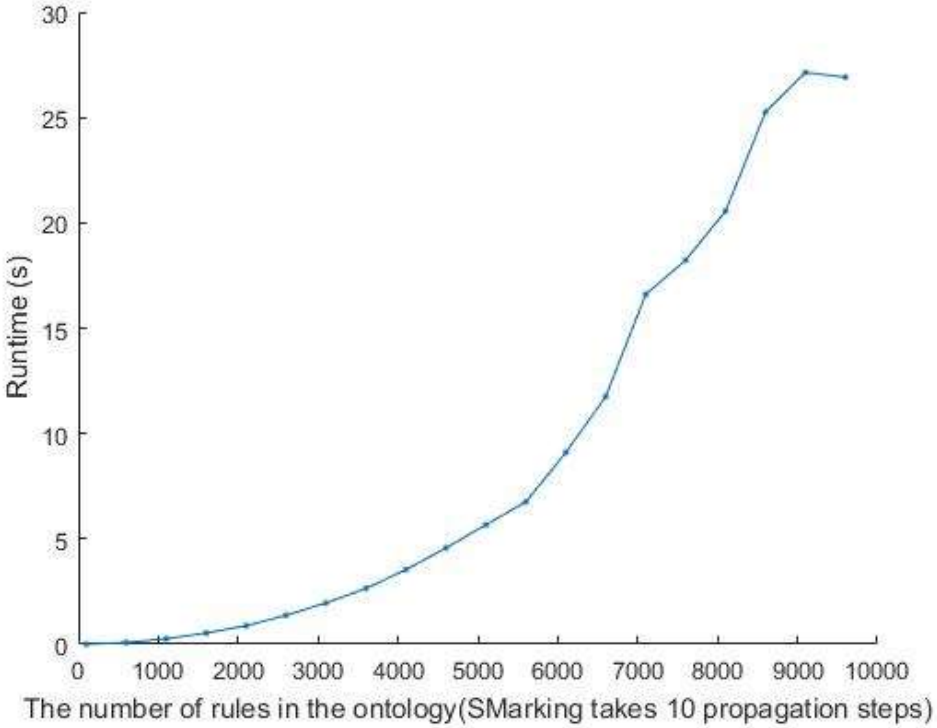

Figure 4.3 The relationship between the sticky ontology recognition function's runtime of the rules in the ontology when the SMarking takes 10 steps.

When the ontologies are simple, i.e. the number of rules is not large, the runtime for stickiness' recognition is very short even when the system needs to loop 1000 times in the SMarking procedure. If the ontology has 10000 rules, ten propagation steps in SMarking will take 30 seconds, which is acceptable.

Runtime evaluation of input transformation function: The transformation function parses each rule as a tree, and the leaves of the tree are characters. As the basic unit in the tree, the number of characters in an ontology plays a significant role in affecting the runtime of the system. We have tested the system by changing the number of the input's characters. At the same time, we observed the time that the system spends on the transformation function. The result is shown in Figure 4.4. Please notice that in test ontologies, variables have 3-4 characters and each predicate has 4-7 characters.

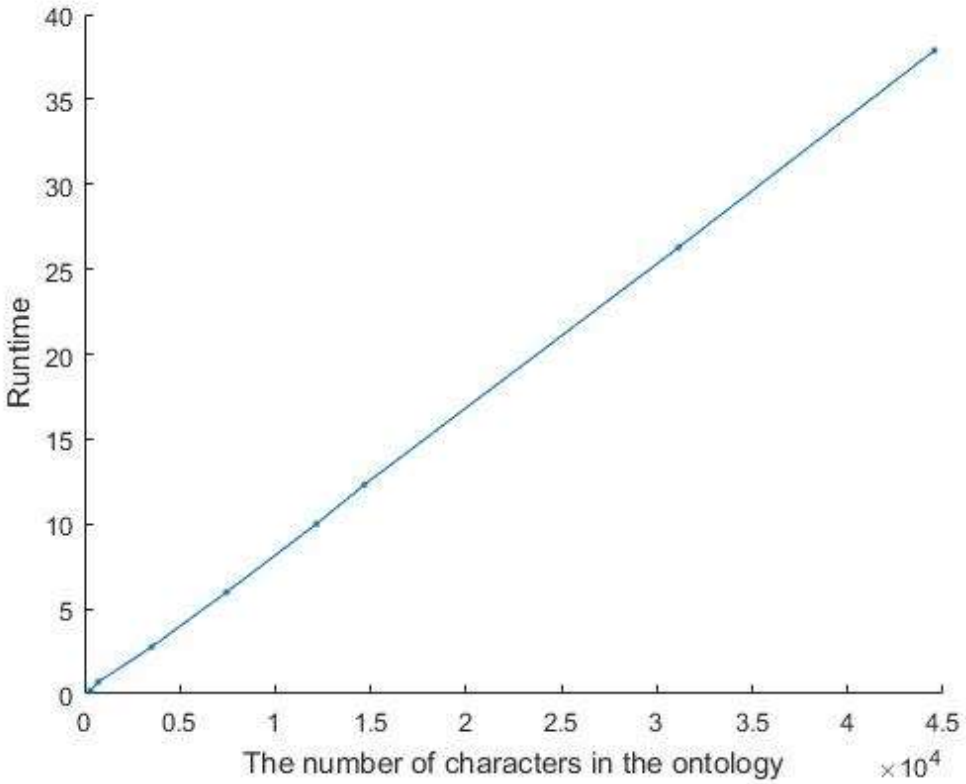

Figure 4.4 The correlation between the ontology's character's number and the runtime.

It is obvious that the correlation of the ontology's character's number and the length of the operating time is linear.

Furthermore, we have noticed that the length of the time to parse an ontology rule depends on the depth of the tree that the rule is transformed into. Because the depth of the tree relies on the length of the longest variable in the rule. It is also necessary to analyse the influence that the length of the variable has on the runtime. We have created

a single rule, the body and the head of the rule have only one atom with one variable. We have changed the variable's length in each rule's head and observed the time the transformation function used to transform the rule. Figure 4.5 shows that this relation is worse than linear. when the length of the variable's character reaches 400, parsing a single rule will spend 18 seconds. Fortunately, most variable of the ontology in the real world does not have that many characters. Recall that the system does not support using constant, i.e. only variables are allowed in the input ontology.

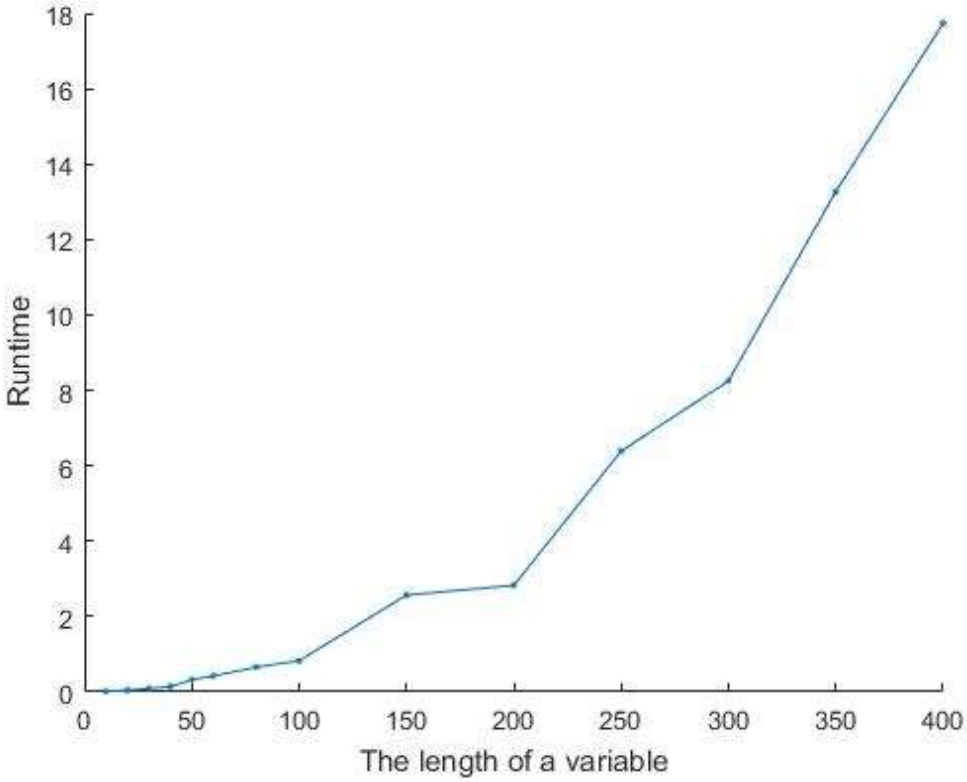

Figure 4.5 The correlation between the number of characters in the ontology's variables and the length of the operating time.

4.5 Usability Evaluation

4.5.1 Heuristic Evaluation

The software tool we developed was evaluated in usability by applying heuristic evaluation. heuristic evaluation is a usability evaluation technique where at least one evaluator tests an interface and judging it based on a set of recognised usability principles. Heuristic evaluation is simple and can be done by a single person, even developers themselves.

We used Nielsen's 10 Heuristics[23] as the heuristics to evaluate the usability of our system's interface. Nielsen's 10 Heuristics are come up with in 1994 and is still used today[24]. The conclusion of the evaluation is as follows:

1). Visibility of system status:

Demand: Users can always get information about what is going on from the interface of the system. For example, after saving an edited article, the system should let the user know that his article is saved.

The evaluation of the main interface:

what are good: a) When users select a file, the textbox on the right side of the 'Select File' will show the path of the chosen file. b) all the chosen classes of TGDs are shown in the right list box, while the unchosen ones are in the list box on the left. c) Whether the suggestion function is enabled are shown as a check box in the interface.

What needs to be improved: a) when checking an ontology takes a long time, users may want to know the current work progress of the system.

The evaluation of the Custom Style interface:

What are good: a) selected files can be viewed in the interface b) When a new grammar is saved, it will be shown in the list box on the right. c) When converting the format of an ontology file, users can view the current progress by checking the progress bar at the bottom.

What needs to be improved: When the system occurs into an ontology rule that cannot be converted, it skips the rule but without letting the user know immediately. Users might prefer to terminate the system and modify their ontologies when they realise that their ontologies contain mistakes.

2). Match between system and the real world:

Demand: Concepts and words in the interface should familiar to users. Information that shown in the interface should be logical and natural. Complicated information and system oriented terms are not encouraged to be used in the interface.

The evaluation of the main interface:

It is good that all the words and phrases in the interface are simple. But the button named "Custom Style" is a bit confusing. Clicking the button will open the Custom Style interface where users can convert their ontology written in custom ways. But it is hard for a beginner to the system to know its meaning immediately.

The evaluation of the Custom Style interface:

Most of phrases and sentences in this interface are easy to understand. The grammar shown in the text box is complicated and not written in natural languages. This is inevitable since the grammar has to be CFG so it can precisely describe the ontology.

3). User control and freedom:

Demand: the system should allow the user to make mistakes and correct them. Users can always cancel and reset their previous operation whenever they want to. Every action users take will not lead to major errors. Users should not be forced to do any actions.

The evaluation of the main interface:

When the judgement algorithm takes a long time, users cannot cancel the process. Users sometimes could choose a wrong ontology file and realised that mistake after executing the algorithm.

The evaluation of the Custom Style interface:

It is good that users can always reset the grammar into default.

However, if users delete a saved style by mistake, they do not have a chance to recover it. What's more, if users want to terminate the transformation process, they can only press "Back" button or close the window.

4). Consistency and standards

Demand: The system should have consistency, i.e. the same information should be expressed in the same way. By contrast, information that is different should appear to be different.

The evaluation of the main interface:

This interface has basic consistency: all the error messages will be shown in a message box with an error icon. All results are shown in a message box with an information icon.

The evaluation of the Custom Style interface:

The "Save as New Grammar" button should be "Save as New Style". Similarly, the sentence "Enter the grammar" in the interface should be "Describe the style of your ontology"

5). Error prevention:

Demand: The system should be able to prevent some errors in advanced. For example, reminding the users to check their input before executing any command, or informing the user potential problems they may get involved.

The evaluation of the main interface:

This interface prevents error by informing users their mistakes before the system calls the judgement function. Those mistakes include invalid file path, empty file path, invalid file and grammar errors. Most error messages are shown in a message box so users can correct immediately, while some complicated error messages like grammar error will be stored in a file, which is more or less inconvenient but currently, there is no better way.

The evaluation of the Custom Style interface:

the complicated error will be stored in a file in details. This is inconvenient but currently, there is no better way.

6). Recognition rather than recall:

Demand: Users do not need to recall “what’s it for” or “what I have done”. The system should show them the information they need rather than forcing them to remember. The system should also show users the instruction for the interface whenever they need it.

The evaluation of the main interface:

The system is designed for professional people who know the theory of ontology and ontology languages well. So most of the contents in the interface do not need to use icon or other methods to help users recognise what they mean. However, it would be nice if the help button has an icon so it can draw users’ attention.

The evaluation of the Custom Style interface:

The help button in this interface can also be an icon. A metaphor instead of a word can let users understand faster and better.

7). Flexibility and efficiency of use:

Demand: The system should have some shortcuts to allow users to interact with the Interface more efficiently. For example, in addition to mouse control, users can also

use a keyboard to operate the system. The system can also allow hot-keys so the experts can use the interface faster.

The evaluation of the main interface:

Users can select or unselect all classes of TGDs by clicking “Select All” button or the “Unselect All” button, which is efficient. It would be better if users can select/unselect a single class of TGDs by a double-click, which is much efficient that selecting an item then clicking the ‘>>’ or the ‘<<’ button.

The evaluation of the Custom Style interface:

This interface is fine, nothing needs to be improved in flexibility and efficiency.

8). Aesthetics and minimalist design:

Demand: The system should not give too much information to the user at one time. Any information shown in the interface should be as simple as possible. Otherwise, some information might be ignored by the user.

The evaluation of the main interface:

This interface is the simplest that it can be, every label and button has no more three words. Furthermore, Distances between widgets are wide enough so users can easily find what they look for.

The help button can be replaced by an icon, so the interface could be simpler.

The evaluation of the Custom Style interface:

The default size of the interface is too small so gaps between widgets are narrow. As it is mentioned in the main interface, the help button can be replaced by an icon. The last but not least, the label “Enter the Grammar” is not necessary because users know what is in the text box bellows.

9). Help users recognise, diagnose, and recover from errors:

Demand: When an error occurs, the system should explain the error in understandable language. If it is needed, the system should also provide the constructive advice on correction to the problem.

The evaluation of the main interface:

All error messages shown in this interface are clear and easy to understand. However, there are some grammar mistakes like “File does not exist” should be “The file does not exist” and “syntax check failed” should be “Failed to pass the syntax check”. The gap between the brief of the error and the detail should be wider.

The evaluation of the Custom Style interface:

Some error messages are shown as a warning message, which is the lack of consistency. It would be nice if error messages in all interfaces are shown as a warning, because users may be less panic when seeing a warning instead of an error.

10). Help and documentation:

Demand: unless the system is extremely simple and understandable, it should have proper documentation for explaining functions it provides and helps users to use them. The documentation needs to be easy to read and understand.

The evaluation of the main interface:

This interface has a help button which leads to a help interface. However, information provided in the help interface could be easier to understand.

The evaluation of the Custom Style interface:
The same as the main interface: the help interface for the Custom Style interface should be more clear and the help information should be more understandable.

After the evaluation, we have modified the system so some of the usability problems are fixed:

1) The help buttons in both the main interface and the custom style interface now have an icon instead of a word "help" on it. 2) Users can now select and unselect ontology languages from list boxes by double click. 3) When the transformation algorithm meets an inconvertible rule, the colour of the progress bar will turn red. 4) Users can now stop the transformation any time they want by clicking the “stop” button. 5) The label” Enter the Grammar” is removed. 6) A message box will appear when users want to delete a style. The selected style will be deleted only when users confirm. 7) The path of the transformed ontology will be shown in the main interface, so users can operate the system faster. 8) the default size of the custom style interface is zoomed.

Some usability problems cannot be corrected at the moment. One of the reasons is that certain functions are hard to be implemented when using Tkinter library to build the interface.

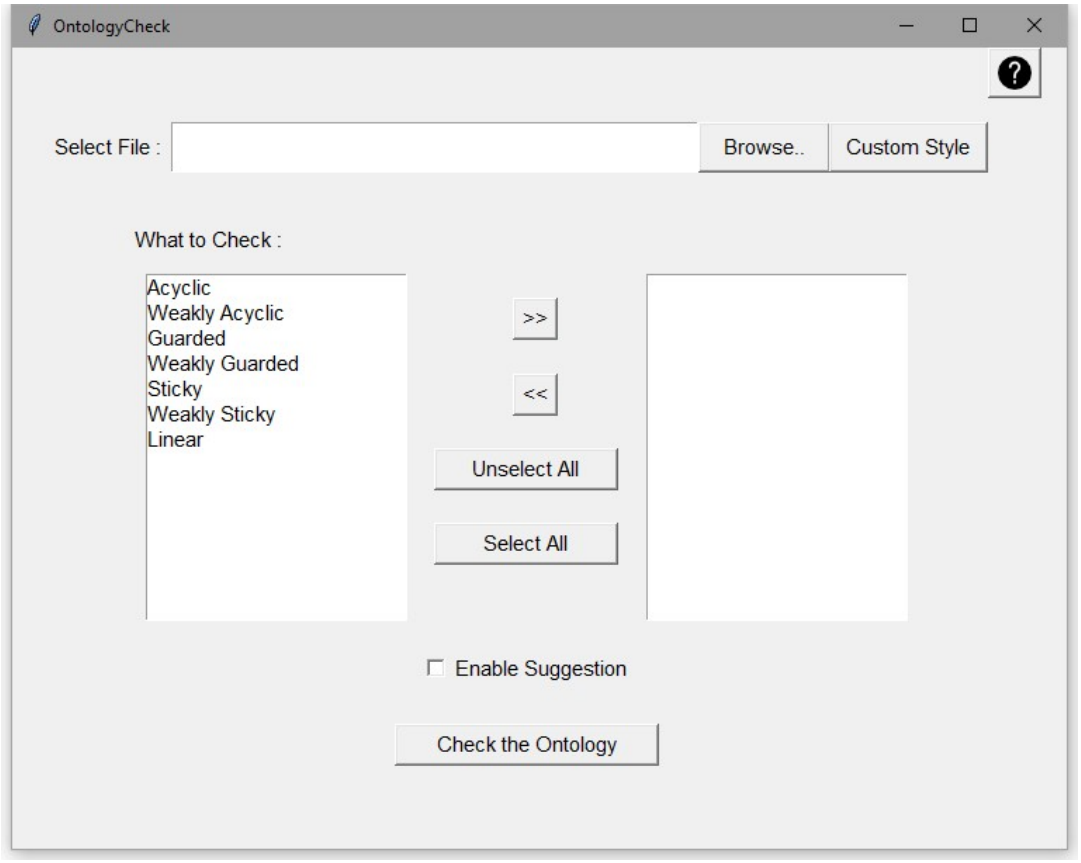

Figure 4.6 The main interface (After modification)

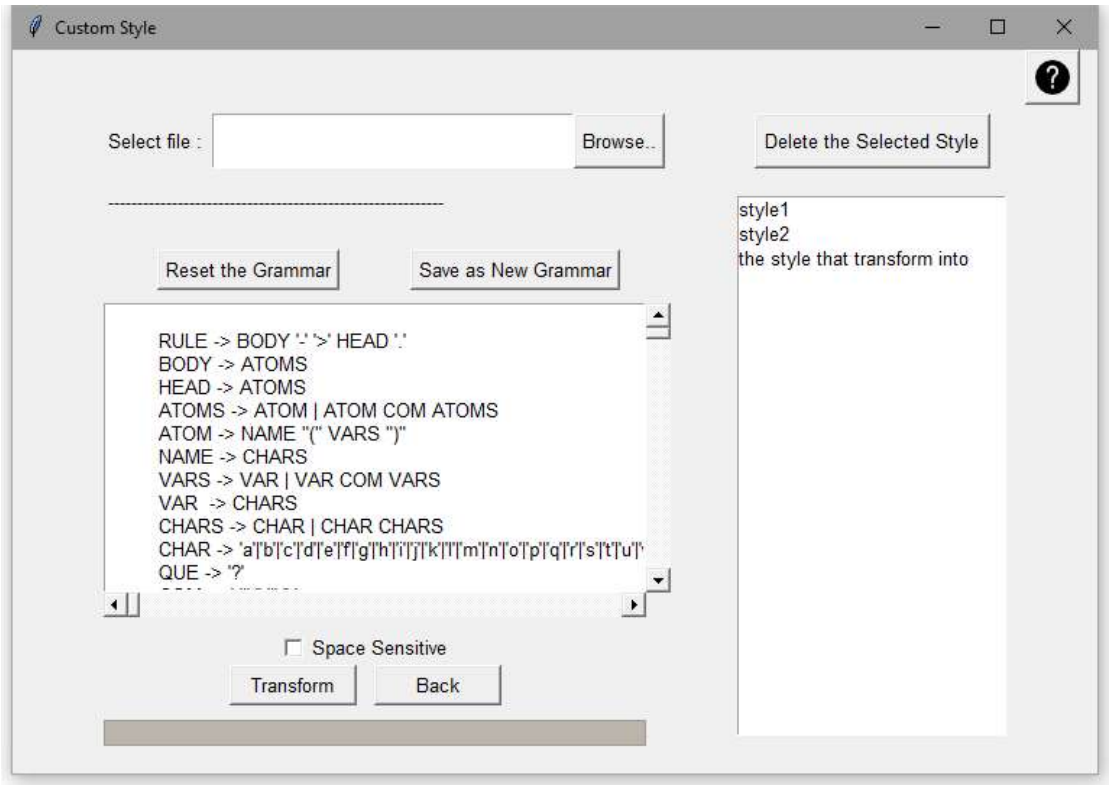

Figure 4.7 The custom style interface (After modification)

Chapter 5: Summary

5.1 Conclusion

The system has the function of converting the format of the input ontology, recognizing ontologies that are acyclic, weakly-acyclic, sticky, weakly-sticky, guarded, weakly-guarded or linear, giving a suggestion for making ontology weakly-acyclic, acyclic, linear and guarded. For common ontologies written as sets o TGDs, the system can analyse them and give results in a short time. And the results are sound.

5.2 Future work

The system still needs to be improved in the future. First, the system does not support constant as atoms' attributes, i.e. only variables are allowed to be used. Second, for an ontology that contains an extremely large number (millions or more) of variables, atoms, rules or cycles that in its dependency graph, the system could take a long time to analyse. The last but not least, the suggestion functions for weak guardedness, stickiness and weak stickiness have not yet been implemented. We have left the interfaces in the system and our opinion and suggestion in section 3.4.2.

Furthermore, it would be better if the system has the ability to check more ontology languages than the seven classes of TGDs that have already been implemented. The system does not have such extensibility yet, hopefully, it will be implemented in the nearest future.

Bibliography

[1] Beeri, C. and Vardi, M.Y., 1981, July. The implication problem for data dependencies. In International Colloquium on Automata, Languages, and Programming (pp. 73-85). Springer, Berlin, Heidelberg.

[2] Calì, A., Gottlob, G. and Lukasiewicz, T., 2012. A general datalog-based framework for tractable query answering over ontologies. Web Semantics: Science, Services and Agents on the World Wide Web, 14, pp.57-83.

[3] Poole, D., 1993. Probabilistic Horn abduction and Bayesian networks. Artificial intelligence, 64(1), pp.81-129.

[4] Wache, H., Voegele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H. and Hübner, S., 2001, August. Ontology-based integration of information-a survey of existing approaches. In IJCAI-01 workshop: ontologies and information sharing (Vol. 2001, pp. 108-117).

[5] Johnson, D.S. and Klug, A., 1984. Testing containment of conjunctive queries under functional and inclusion dependencies. Journal of Computer and system Sciences, 28(1), pp.167-189.

[6] Calì, A., Lembo, D. and Rosati, R., 2003, June. On the decidability and complexity of query answering over inconsistent and incomplete databases. In Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 260-271). ACM.

[7] Fagin, R., Kolaitis, P.G., Miller, R.J. and Popa, L., 2005. Data exchange: semantics and query answering. Theoretical Computer Science, 336(1), pp.89-124.

[8] Cali, A., Gottlob, G. and Pieris, A., 2012. Towards more expressive ontology languages: The query answering problem. Artificial Intelligence, 193, pp.87-128.

[9] Pérez-Urbina, H., Motik, B. and Horrocks, I., 2010. Tractable query answering and rewriting under description logic constraints. Journal of Applied Logic, 8(2), pp.186-209.

[10] A. Calì, G. Gottlob, M. Kifer, Taming the infinite chase: Query answering under expressive relational constraints, in: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, 2008, pp. 70–80.

[11] Refaeilzadeh, P., Tang, L. and Liu, H., 2009. Cross-validation. In Encyclopedia of database systems (pp. 532-538). Springer US.

[12] Gottlob, G., Orsi, G. and Pieris, A., 2011. Ontological query answering via rewriting. In Advances in Databases and Information Systems (pp. 1-18). Springer Berlin/Heidelberg.

[13] Marnette, B., 2009, June. Generalized schema-mappings: from termination to tractability. In Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 13-22). ACM.

[14] Deutsch, A., Nash, A. and Remmel, J., 2008, June. The chase revisited. In Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 149-158). ACM.

[15] Gottlob, G., Orsi, G. and Pieris, A., 2014. Query rewriting and optimization for ontological databases. ACM Transactions on Database Systems (TODS), 39(3), p.25.

[16] Lundh, F., 1999. An introduction to Tkinter. URL: www. pythonware. com/library/tkinter/introduction/index. htm.

[17] Shipman, J.W., 2013. Tkinter 8.4 reference: a GUI for Python. dostopno na http://infohost. nmt. edu/tcc/help/pubs/tkinter. pdf.

[18] Developers, N., 2010. NetworkX. networkx. lanl. gov.

[19] Bird, S., 2006, July. NLTK: the natural language toolkit. In Proceedings of the COLING/ACL on Interactive presentation sessions (pp. 69-72). Association for Computational Linguistics.

[20] Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D. and Tsamoura, E., 2017, May. Benchmarking the chase. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (pp. 37-52). ACM.

[21] Johnson, D.B., 1975. Finding all the elementary circuits of a directed graph. SIAM Journal on Computing, 4(1), pp.77-84.

[22] Van Rossum, G., 2005. Python documentation. Webová stránk a http://www. python. org/doc.

[23] Nielsen, J., 1994. Heuristic evaluation. Usability inspection methods, 17(1), pp.25-62.

[24] Nielsen, J., 2005. Ten usability heuristics.